Elaborate WebRTC Media with Artificial Intelligence

Let Janus, OpenCV and A.I. work together!

Paolo Saviano
Full Stack developer @Meetecho
psaviano@meetecho.com

FOSDEM

JANVS
WEBRTC SERVER

WebRTC and Artificial Intelligence?

- **Absolutely fascinating matter**
  Real Time Computer Vision & Sound Recognition
  Data collection and live elaboration
  Broadcasting experience customized by users

- **A lot of scenarios with different requirements**
  Massive broadcasting, small videorooms, device networks
  Device constraints, Time constraints...
  Must choose!

- **Make it simple, it’s a project so young!**
  Remove constraints
  Focus on video elaboration
  Explore the results

What we want from this project

- **Handle the video media in a comfortable way**
  Receive and elaborate the Janus RTP stream without (too much) manipulation
- **Minimize the client side effort**
  Avoid client-side elaboration
  Do not overload the client's bandwidth
- **Keep an eye on scalability**
  Put everything in a nutshell and replicate it, if needed
- **Play in the home pitch using known languages and environments**

What we want from this project

OPENCV

- **Handle the video media in a comfortable way**
  Receive and elaborate the Janus RTP stream without (too much) manipulation

- **Minimize the client side effort**
  Avoid client-side elaboration
  Do not overload the client's bandwidth

- **Keep an eye on scalability**
  Put everything in a nutshell and replicate it, if needed

- **Play in the home pitch using known languages and environments**

What we want from this project

OPENCV

- **Handle the video media in a comfortable way**
  Receive and elaborate the Janus RTP stream without (too much) manipulation
- **Minimize the client side effort**
  Avoid client-side elaboration
  Do not overload the client's bandwidth

SERVER-SIDE

- **Keep an eye on scalability**
  Put everything in a nutshell and replicate it, if needed
- **Play in the home pitch using known languages and environments**

What we want from this project

OPENCV

- **Handle the video media in a comfortable way**
  Receive and elaborate the Janus RTP stream without (too much) manipulation
- **Minimize the client side effort**
  Avoid client-side elaboration
  Do not overload the client’s bandwidth

SERVER-SIDE

- **Keep an eye on scalability**
  Put everything in a nutshell and replicate it, if needed

CONTAINERS :)

- **Play in the home pitch using known languages and environments**

What we want from this project

OPENCV

JAVASCRIPT-ISH

- **Handle the video media in a comfortable way**
  Receive and elaborate the Janus RTP stream without (too much) manipulation
- **Minimize the client side effort**
  Avoid client-side elaboration
  Do not overload the client's bandwidth

SERVER-SIDE

- **Keep an eye on scalability**
  Put everything in a nutshell and replicate it, if needed

CONTAINERS :)

- **Play in the home pitch using known languages and environments**

What we will use

- **Video Capture (WebRTC)**
  Janus WebRTC Server (http://github.com/meetecho/janus-gateway)

- **Video elaboration and Sampling**
  opencv4nodejs (https://github.com/justadudewhohacks/opencv4nodejs)

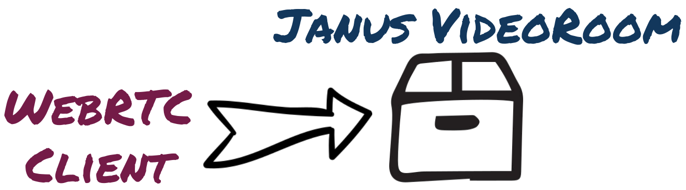
- **Server**
  NodeJS (https://nodejs.org)

- **Containerization**
  Docker (https://www.docker.com)

- **Language**
  TypeScript (https://www.typescriptlang.org)

Architecture

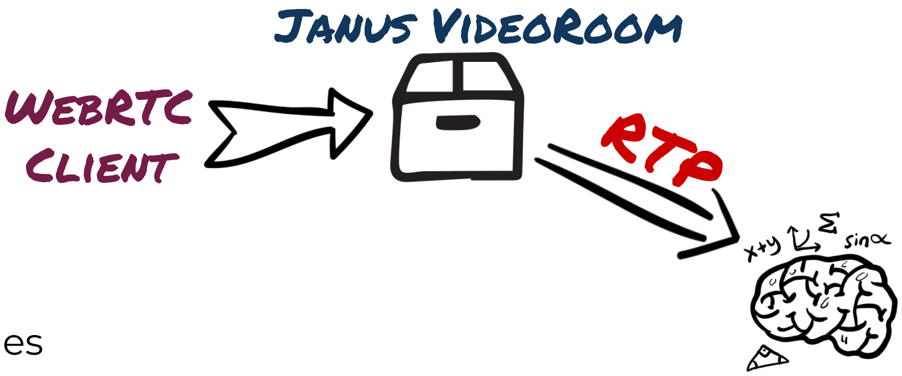
- **The Janus Videoroom Plugin**
  WebRTC entrypoint for media producers
  Forwards received stream as RTP stream



Architecture

- **The Janus Videoroom Plugin**
  WebRTC entrypoint for media producers
  Forwards received stream as RTP stream

- **The Video Stream Elaborator**
  Elaborates received RTP video streams
  Returns elaboration results as UDP messages

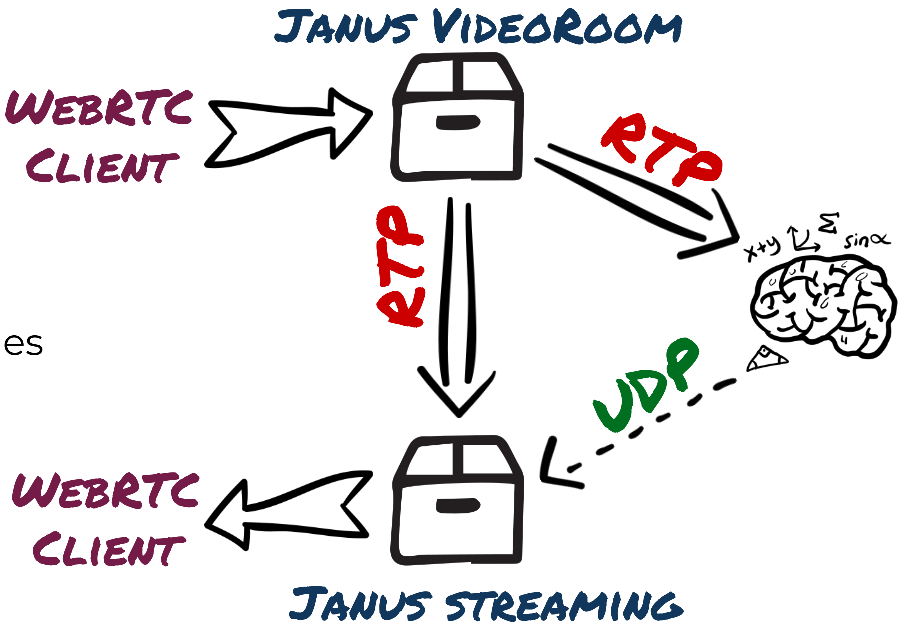


Architecture

- **The Janus Videoroom Plugin**
  WebRTC entrypoint for media producers
  Forwards received stream as RTP stream

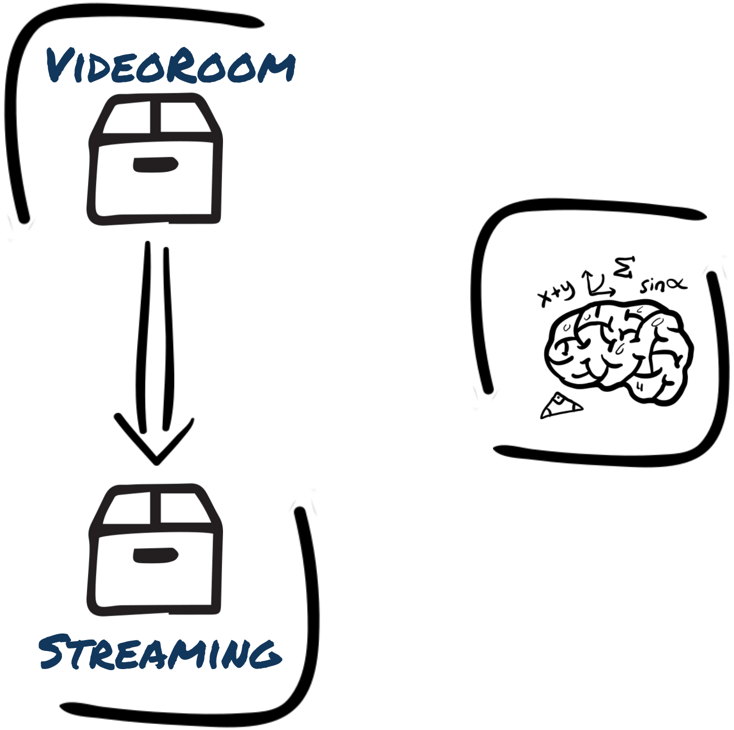
- **The Video Stream Elaborator**
  Elaborates received RTP video streams
  Returns elaboration results as UDP messages

- **The Janus Streaming Plugin**
  WebRTC entrypoint for media receivers
  Forwards received streams (RTP & UDP)
  to WebRTC clients through media stream
  or datachannels



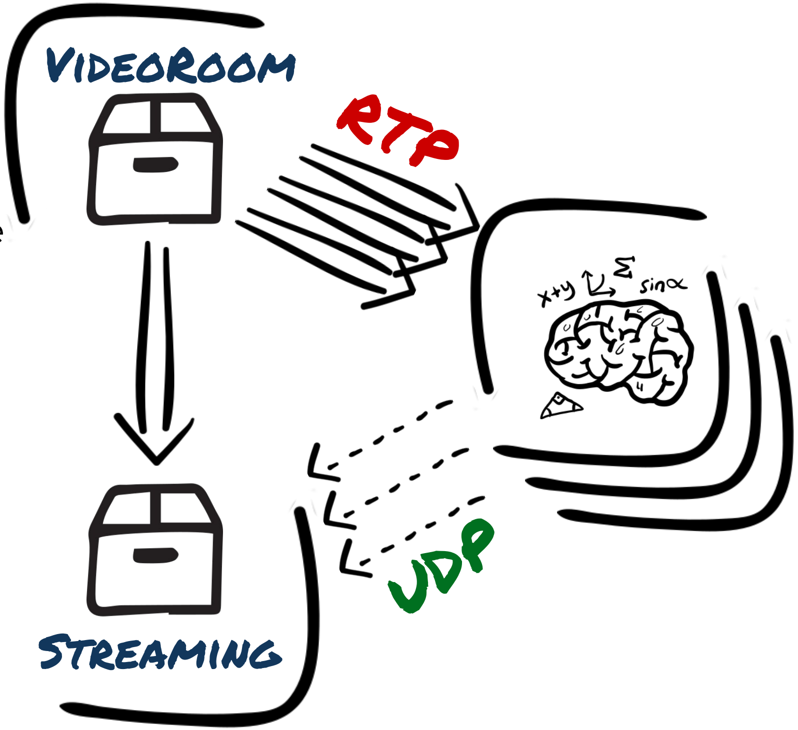
A container to rule them all

- **Two containers**
  Create a more flexible architecture



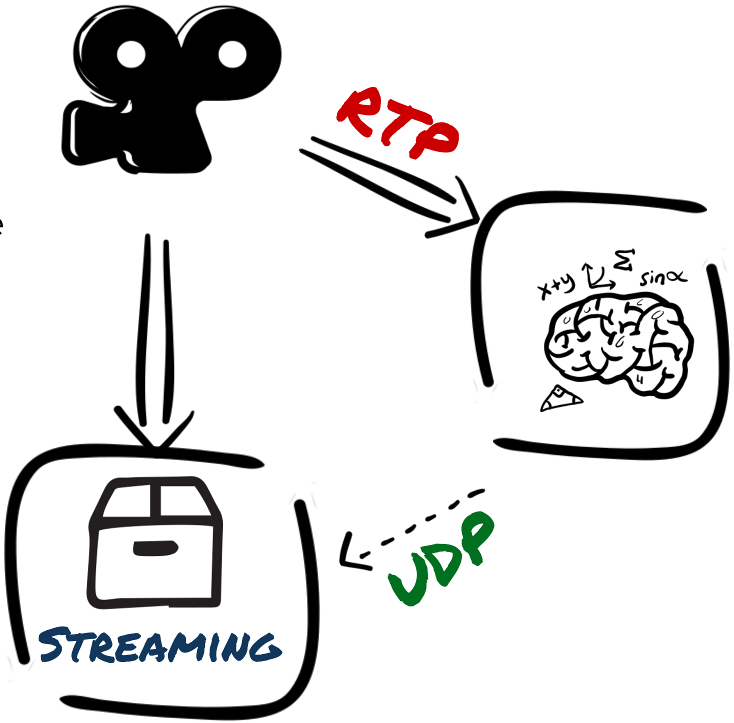
A container to rule them all

- **Two containers**
  Create a more flexible architecture
- **Multiple streams, same elaboration**
  Each of them elaborated by a dedicated instance
- **Single stream, multiple elaborations**
  Different elaborations on same stream in parallel
  Let users display results in a selective way



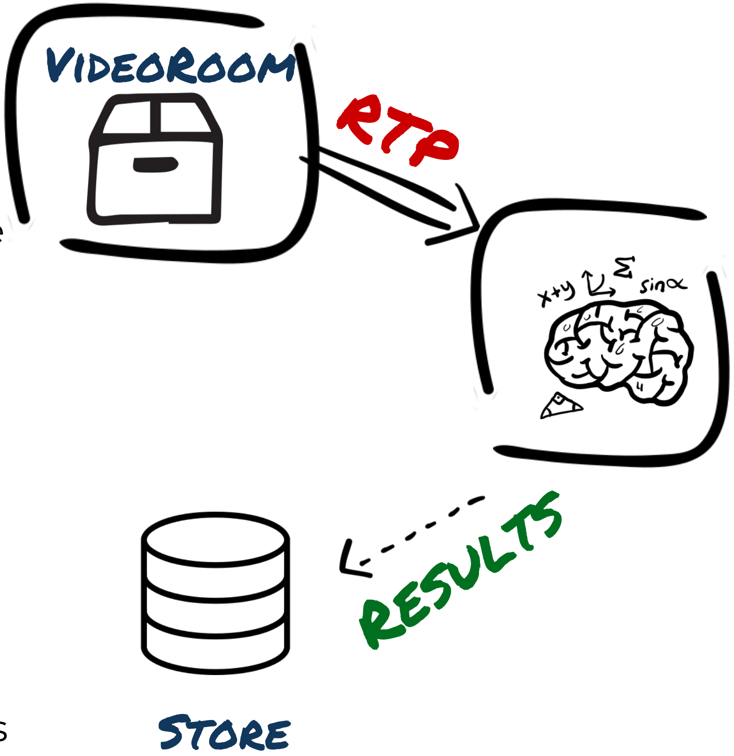
A container to rule them all

- **Two containers**
  Create a more flexible architecture
- **Multiple streams, same elaboration**
  Each of them elaborated by a dedicated instance
- **Single stream, multiple elaborations**
  Different elaborations on same stream in parallel
  Let users display results in a selective way
- **Reconfiguration**
  Replace containers according to our needs
  Use an external RTP source



A container to rule them all

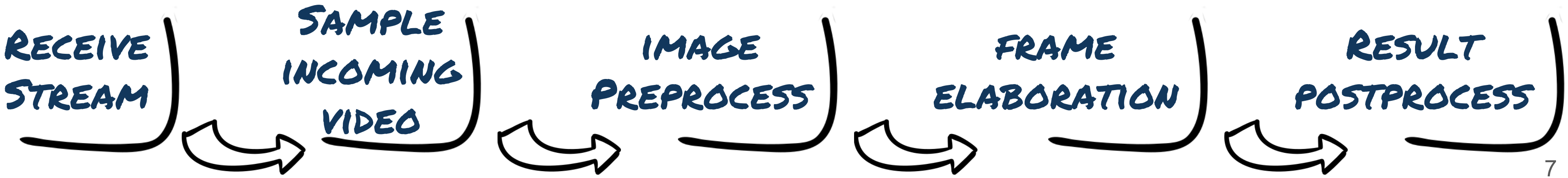
- **Two containers**
  Create a more flexible architecture
- **Multiple streams, same elaboration**
  Each of them elaborated by a dedicated instance
- **Single stream, multiple elaborations**
  Different elaborations on same stream in parallel
  Let users display results in a selective way
- **Reconfiguration**
  Replace containers according to our needs
  Use an external RTP source
  Save elaborated data without live UDP feedbacks



Let’s Code!

- **The idea**
  Break the elaboration flow in several steps
  Create a superclass that takes care of all the under the hood tasks
  Let programmers implement only the target-specific code using a inherited class

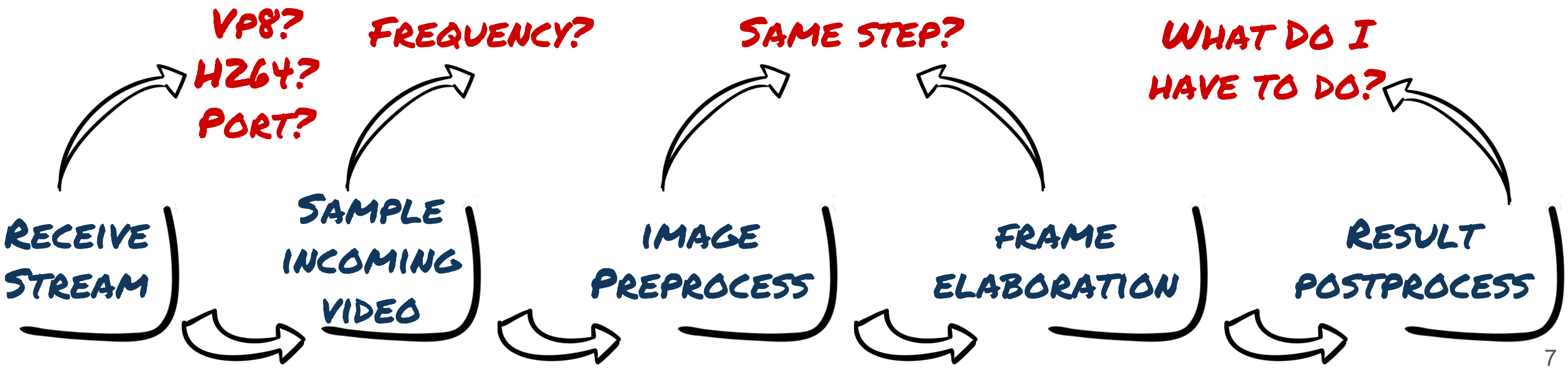
- **The code flow of our video elaboration**



Let's Code!

- **The idea**
  Break the elaboration flow in several steps
  Create a superclass that takes care of all the under the hood tasks
  Let programmers implement only the target-specific code using a inherited class

- **The code flow of our video elaboration**



The superclass interface

```
export interface ModelMethods<T extends Detection> {
  train(trainSet: any[], label?: any[]): void,
  classify(image: cv.Mat, frame_counter: number): Promise<any[]>,
  configure(config?: object): void,
  getVideoInput(pipe?: string): object,
  stopElaboration(): void,
  startElaboration(pipe?: string, elaborating_rate?: number): Promise<void>,
  elaborateVideo(sampling_rate: number): Promise<T[]>,
  elaborateResults(output: T[]): void,
}
```

The superclass interface

- **The under-the-hood stuff**
  Getting video input once pipe is defined
  Sample video every *sampling_rate* frame and provide image to the **classify** function
  Helpers function, like the **bootstrap** and the **teardown** of the component

- **What does a programmer have to provide?**

```
abstract train(trainSet?: any[], label: any[]): void; // if needed
abstract classify(object: cv.Mat, frame_counter: number): Promise<any[]>;
abstract configure(config?: object): void;
abstract elaborateResults(output: T[]): void;
```

The *configure* method

- **Sets the GStreamer receiving pipe used by openCV module**
  i.e. *udpsrc port=20010 caps=\"application/x-rtp, media=(string)video, clock-rate=(int)90000\" ! rtpvp8depay ! vp8dec ! videoconvert ! appsink*

- **Load here everything you need for elaboration phase**
  Import a Tensorflow/Caffe model
  Import dataset to train your own model
  Connect with an external cognitive service
  Define custom configurations

```
udp_client = dgram.createSocket('udp4');
pbFile = path.resolve('path/to/repo/', 'frozen_inference_graph.pb');
pbtxtFile = path.resolve('path/to/repo', 'ssd_mobilenet_v2_coco_2018_03_29.pbtxt');
net = Net.loadFromTensorFlow(pbFile, pbtxtFile);
```

The *elaborateResults* method

- **Receives an array of extended *Detection* object**
  Extending the Detection class allows programmers to handle elaboration results, marshalling the received data

- **Sends data to the Janus Streaming Plugin**
  Detection results could be sent to Streaming Plugin in order to be forwarded to receivers through datachannels

- **No boundaries**
  Store data
  Define a protocol between elaborator and clients to enrich the video they are receiving
  Involve external services

The *classify* method

A lot of things could be done!

We implemented a few examples found online

Emotion recognition using third party service

Streaming Controls

Status Info

Detections

ANGER	0
CONTEMPT	0.002
DISGUST	0
FEAR	0
HAPPINESS	0.977
NEUTRAL	0.022
SADNESS	0
SURPRISE	0

Use a trained LBPHFaceRecognizer

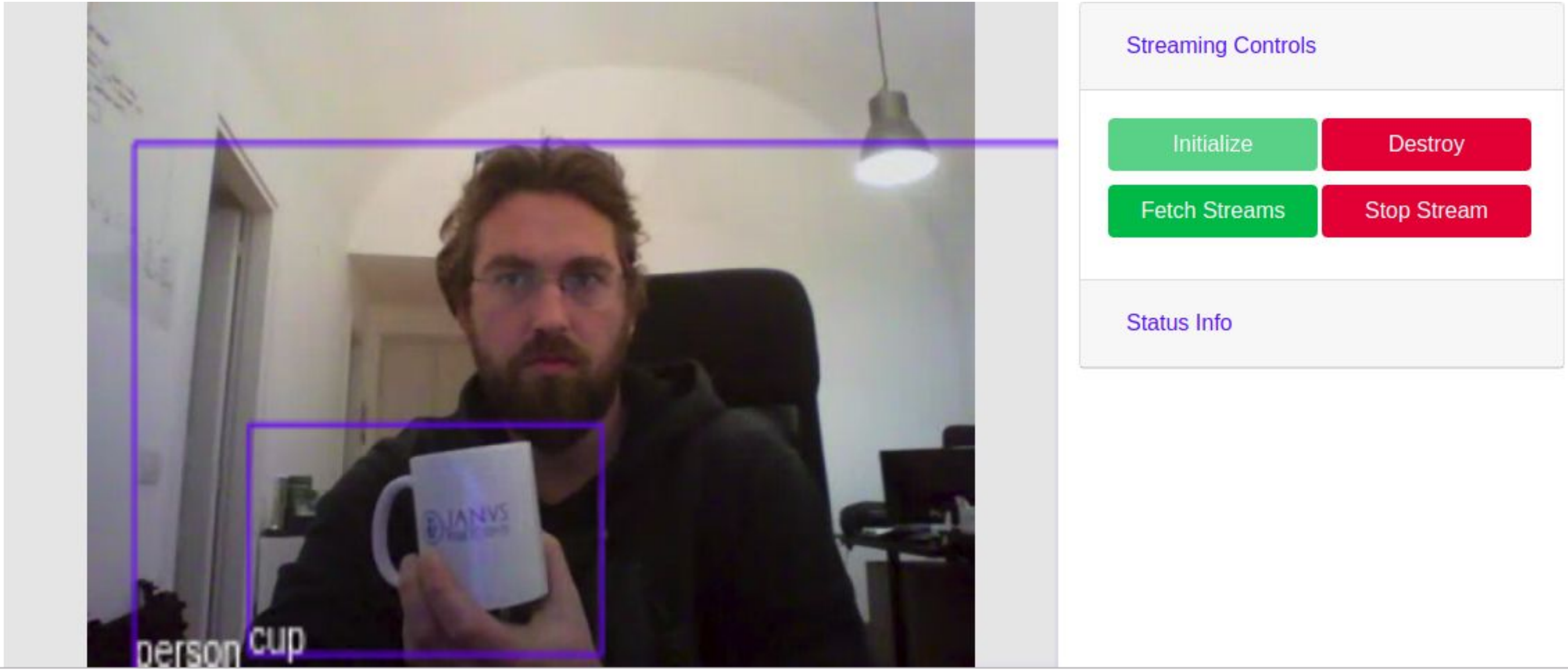
Streaming Controls

Initialize | Destroy

Fetch Streams | Stop Stream

Status Info

Object Detection loading a Tensorflow model



What's Next?

- **Measurements**
  Provide the capability to evaluate delays introduced by elaboration
- **Scalability**
  Make software ready-to-scale, taking advantages of Docker
- **Handle the audio and data media**
  Implement the necessary tools to treat other media
- **Human Intelligence**
  Improve our knowledge on A.I. and machine learning in order to match as many use cases as possible

Icon made by Freepik from www.flaticon.com


TIME
ALEX
ME
imgflip.com

Fuzzing the Janus WebRTC Server

And why you should fuzz too

Alessandro Toppi
Software Engineer @ Meetecho
<atoppi@meetecho.com>

The infamous Project Zero's post

- Natalie Silvanovich's post series [1] on Google Project Zero blog
- Aiming at RTC services, focusing on **End-To-End**, **RTP** testing
- Malicious endpoint generating randomized input
- *"fairly time intensive"* and *"required substantial tooling"* [2]

> *"Our research found a total of 11 bugs in WebRTC, FaceTime and WhatsApp. The majority of these were found through less than 15 minutes of mutation [...] We were surprised to find remote bugs so easily in code that is so widely distributed."*

- Is our WebRTC Server safe? [3]

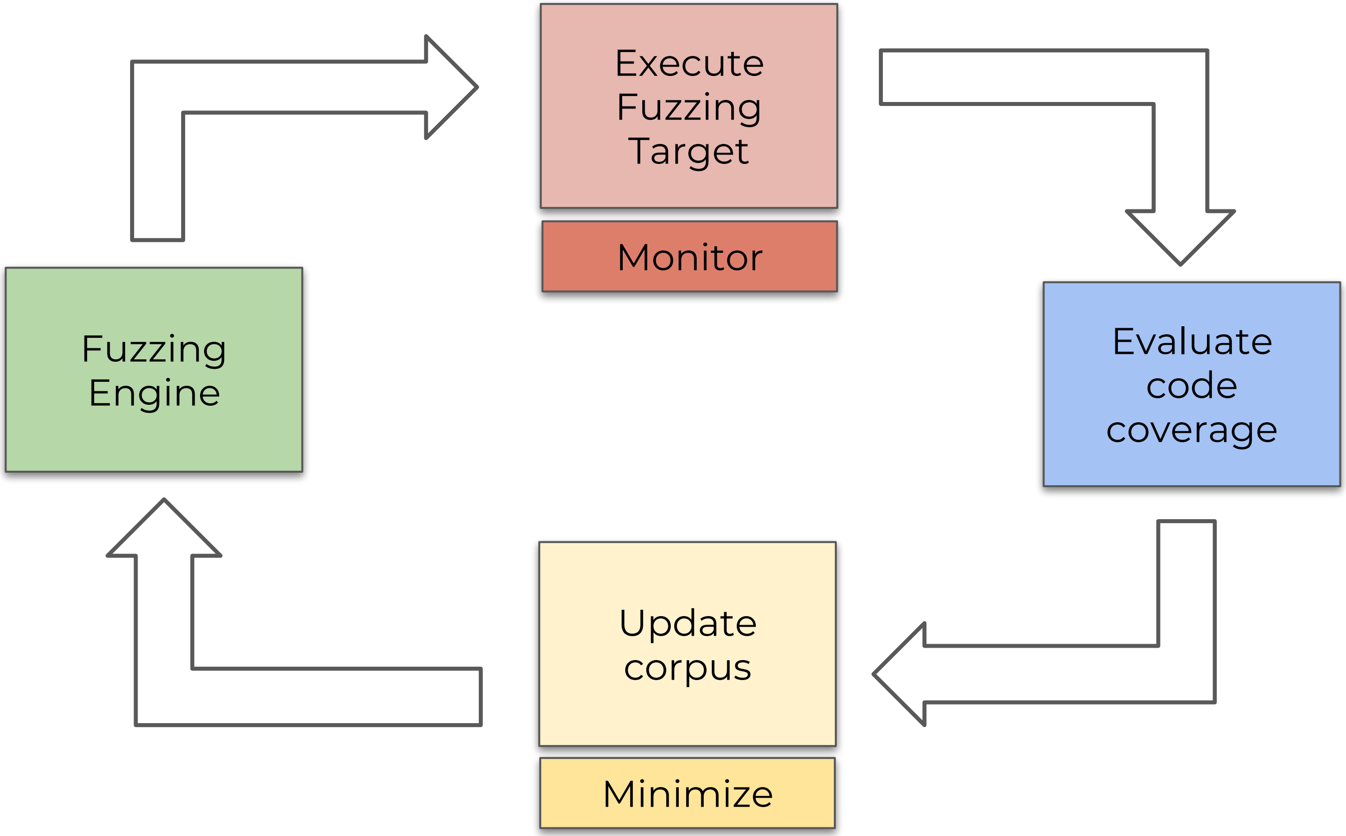
[1] https://googleprojectzero.blogspot.com/2018/12/adventures-in-video-conferencing-part-1.html
[2] https://github.com/googleprojectzero/Street-Party
[3] https://webrtchacks.com/lets-get-better-at-fuzzing-in-2019-heres-how/

Fuzz Testing

- Fuzzing is a software testing technique that consists of automatically submitting unexpected or invalid data to a program and modifying the input pattern according to a defined strategy
    - Coverage guided mutations

- Continuous checking of a Target function, input generated by an Engine
    - At every step, the Engine generates a mutated pattern depending upon
        - The current dataset known as Corpus
    - If new lines of code are covered while executing Target
        - The pattern is added to the Corpus
        - Coverage statistics get updated
    - Target execution is monitored through some tools (e.g. sanitizers)
    - Coverage data are occasionally minimized

Coverage guided fuzzing



libFuzzer

- Coverage-guided fuzzing engine [4]
- It is part of LLVM project, need to compile your sources with **Clang**
- Works **in-process** and linked with the library under test
- Feeds inputs to the target via a **fuzzing entrypoint** (target function)

- The execution of the target function is monitored with **sanitizers** tools
  - AddressSanitizer (ASan) [LLVM /GCC]
  - UndefinedBehaviorSanitizer (UBSan) [LLVM]
  - MemorySanitizer (MSan) [LLVM]

- Openssl, glibc, boringssl, SQLite, ffmpeg ...

[4] https://llvm.org/docs/LibFuzzer.html

libFuzzer

- Prepare a fuzzing target that accepts a bytes array and does something using the API under test

```
// api_fuzzer.c
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  TargetAPI(Data, Size);
  return 0;
}
```

- Build using **`-fsanitize=fuzzer`** flag
- Combine libFuzzer with ASan and/or UBSan!

```
> clang -g -O1 -fsanitize=fuzzer,address,undefined api_fuzzer.c
```

- `-fsanitize=fuzzer` links in the libFuzzer's main() symbol

libFuzzer

- Create a Corpus folder holding the initial **samples** (may be empty)

```
> ./my_fuzzer CORPUS_DIR
```

- Coverage incrementing test cases will be added to the corpus
- The fuzzing process will **stop** when a bug is found
- The input that triggered the bug will be **dumped** to the disk

- The fuzzer can be re-executed against the offending pattern
- Use to check if a bug has been fixed and for regression testing too!

```
> ./my_fuzzer crash-file-dump
```

Fuzzing integration in Janus

- The codebase has been checked for compilation with Clang
  - Some compilation flags have been updated to better support Clang
  - Clang generated useful warnings that led to some fixes!
- Started off from **RTCP**
- Wrote a meaningful fuzzing target (`fuzz-rtcp.c`)
  - Identify critical functions that handled raw pointers
- Added helper scripts to build (`build.sh`) and run (`run.sh`) the fuzzers
- Source [5]

[5] https://github.com/meetecho/janus-gateway/pull/1492

RTCP fuzzing target

```c
// fuzz-rtcp.c
#include "janus/rtcp.h"

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
	if (size < 8 || size > 1472) return 0;
	if (!janus_is_rtcp(data, size)) return 0;
	/* Initialize an empty RTCP context */
	janus_rtcp_context ctx;

	janus_rtcp_parse(ctx, (char *)data, size);
	GSList *list = janus_rtcp_get_nacks((char *)data, size);
	...
	if (list) g_slist_free(list);
	return 0;
}
```

Building the target

- Build needed Janus objects for the fuzzer target

```
> FUZZ_FLAGS="-g -O1 -fno-omit-frame-pointer -fsanitize=fuzzer,address ... "
> ./configure CC=clang CFLAGS="$FUZZ_FLAGS"
> make janus-log.o janus-utils.o janus-rtcp.o
> ar rcs janus-lib.a janus-log.o janus-utils.o janus-rtcp.o
```

- Collect needed Janus dependencies and build the fuzzer

```
> DEPS_CFLAGS="$(pkg-config --cflags glib-2.0)"
> DEPS_LIB="$(pkg-config --libs glib-2.0)"
> clang $FUZZ_FLAGS $DEPS_CFLAGS fuzz-rtcp.c -o fuzz-rtcp janus-lib.a $DEPS_LIB
```

Running the target

- Corpus: reused the *webrtc.org* RTCP corpus [6]
- Created **our own** too, with valid WebRTC RTCP packets and invalid patterns that crashed our server

```
> ./fuzz-rtcp fuzz-rtcp_corpus
```

- Reproduce a crash within the **debugger**

```
> ASAN_OPTIONS=abort_on_error=1 gdb --args ./fuzz-rtcp crash-file
```

[6] https://chromium.googlesource.com/external/webrtc/+/master/test/fuzzers/corpora/rtcp-corpus/

Meet OSS-Fuzz

- OSS-Fuzz [7] is Google's infrastructure dedicated to **continuous** fuzzing of critical Open Source Software (openssl, ffmpeg … Janus?)
    - Multiple fuzzing engines and sanitizers
    - Issue tracker and dashboard for collecting statistics

- To ask for a project integration you need to submit a PR and provide
    - **`Dockerfile`**: prepare the environment (fetch code, install packages)
    - **`build.sh`**: build your library and your fuzzing targets
    - **`project.yaml`**: your project descriptor

- Janus helper script `build.sh`
    - Can be seamlessly used for both local and OSS-Fuzz setup
    - Reads all the env vars defined in the OSSF building environment

[7] https://github.com/google/oss-fuzz

Results

- Fixed **dozens** of RTCP parsing bugs in Janus
    - Many of them were memory buffer overflows (DoS, security flaws)
    - Some of the bugs were confirmed in end-to-end testing with a malicious fuzzing browser
- Built useful tools for fuzzing and regression testing
    - Great opportunity to integrate in OSS-Fuzz for continuous fuzzing [8]
- Got in Clang compiler and some LLVM tools
- Contribute to make WebRTC a safer world [9]

[8] https://github.com/atoppi/oss-fuzz/tree/janus-gateway
[9] https://github.com/RTC-Cartel/webrtc-fuzzer-corpora

What’s next

- Extend to **other protocols** creating new fuzzing targets
  - RTP, SDP ...
- Submit a PR to OSS-Fuzz
- Investigate other fuzzing engines (AFL)
- Integrate the fuzzer building script in Janus GNU Autotools scripts
  - Create a specific target in Makefile for building fuzzers

Thank You!

janUScon

Naples, Italy

September 23 - 25

www.januscon.it