Video Surveillance over IP

CS418: Networked Entertainment

Project #2

Author: Eray Emin Ocak
ID: S040075

21 April 2024

1. Introduction

This is a project that provides surveillance over the internet for users, using ffmpeg to encode the stream with a DASH profile, to be served to the client using a Kotlin/Spring Boot backend.

A single backend program manages all processes, starting an embedded Tomcat server, hosting the player page, and encoding/serving the live stream.

Ffmpeg commands are executed using a custom API written specifically for this project, named ffexecutor. Ffexecutor API allows us to run encoding tasks by providing a set of option instances. Commands are run with a custom FallbackExecutor class, which incrementally retries tasks with mutated settings in case of failure. FallbackExecutor is configured with a FallbackProvider to try different camera inputs if ffmpeg fails to start the encoding.

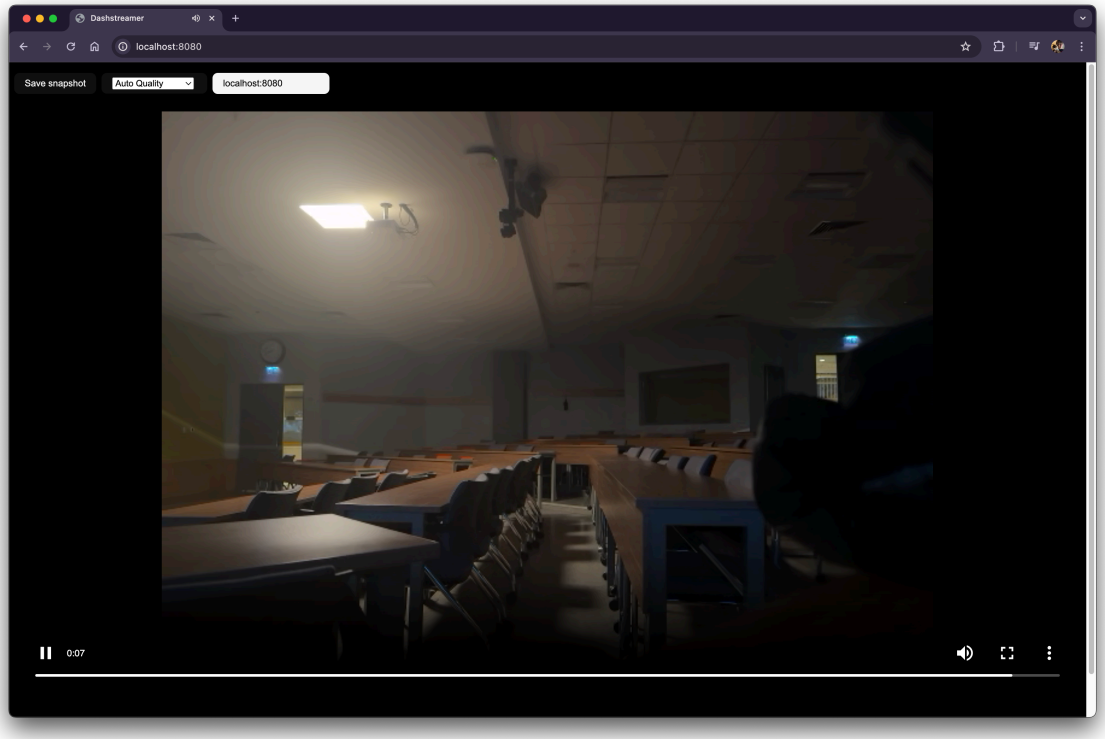

Figure 1: Screenshot of the home page

2. Features

- Player page
- Rewind or jump to a specific timestamp
- Play/pause

- Video quality control (Auto, 240p 1Mbit/s, or 480p 3Mbit/s)
- Automatic quality adjustment depending on network conditions
- Audio streaming from microphone
- Customizable host (for remote connection)
- Snapshot (Screenshot) capture
- Single executable (No separate backend/frontend applications)
- Use of temporary folders for minimal disk space consumption
- Fallback system with different cameras
- Multi-threaded
- Stream file API (Invulnerable to path traversal attacks)
- Scheduled file indexing component
- Custom Ffmpeg interface (Ffexecutor)

3. Dependencies

This project uses open-source codes and depends on a variety of libraries.

Spring Boot Backend Dependencies:

Oracle OpenJDK 17 by Oracle Corporation:

OpenJDK is required to run the backend application. OpenJDK is a runtime dependency for this project and is used to allow execution of JAR files via Java Virtual Machine.

Log4j (Core, SLF4J Adapter) 2.21.1 by Apache Software Foundation:

Log4j is used for logging, both by Spring internal tools and user code. Log4j is utilized to provide easy access to events, warnings, execution outputs, and errors. In the user code, Log4j is used in services, controller, and utility classes.

Kotlin Plugins (JVM, Spring) 1.9.23 by Kotlin Team

Kotlin Plugins are used during the build time for proper Kotlin class scanning and Kotlin functions. This dependency is defined in the project-level Gradle file (build.gradle.kts).

Spring Boot Gradle Plugin 3.2.4 by VMware

Spring Boot Gradle Plugin allows the packaging (JAR or WAR) and execution of this backend application. This dependency is defined in the project-level Gradle file (build.gradle.kts).

Spring Dependency Management Plugin 1.1.4 by VMware

Spring Dependency Management Plugin is used to automatically manage versioning of other Spring-related dependencies. This dependency is defined in the project-level Gradle file (build.gradle.kts).

Spring Boot Starter Web 3.2.4 by VMware

Spring Boot Starter Web is a dependency set that includes Apache Tomcat (HTTP server), Web MVC, JSON support, and Spring Boot Starter. Tomcat server is utilized to serve the API and home page over 8080 ports. Web MVC is used to serve the home page. This dependency is defined in the project-level Gradle file (build.gradle.kts).

Spring Boot Starter Thymeleaf 3.2.4 by VMware

Spring Boot Starter Web is a dependency set that includes Thymeleaf Spring 6 and Spring Boot Starter. Thymeleaf is a template engine that is utilized during home page rendering. This dependency is defined in the project-level Gradle file (build.gradle.kts).

Spring Boot Starter Actuator 3.2.4 by VMware

Spring Boot Starter Actuator is a dependency set that includes Spring Actuator, Micrometer, and Spring Boot Starter. Spring Actuator allows debugging of the beans, mappings, and controllers within the backend application. This dependency is defined in the project-level Gradle file (build.gradle.kts).

JavaScript Dependencies

Dash.JS by DASH Industry Forum

Dash.JS is a reference player client for MPEG-DASH live profile. Dash.JS is defined and used in the home page (home.html in src/main/resources/templates/ folder). Dash.JS is used to load stream manifests and chunks into the video player. It also manages ABR (Adaptive bitrate) and video quality.

4. Running the Application

Make sure to install the Gradle build tool and JDK 17 first. Set the current directory to project root and execute the bootRun Gradle task by running:

```
./gradlew bootRun
```

After the initialization is completed, you should be able to open x in your browser to see the video stream.

5. Usage

User can visit 8080 port on the server to see the home page, where player client is located in. Stream files are served via the /api/v1/file/{FILENAME} API endpoint. For a local server you can access stream manifest on: localhost:8080/api/v1/file/stream.mpd.

6. Ffmpeg

Ffmpeg is used to capture and encode audio/video streams, write stream chunks to file system, and create DASH profile manifests. Ffmpeg tasks are executed using a custom written FfmpegExecutor class. A custom FallbackExecutor is used in combination with the FfmpegExecutor to try with different camera inputs if ffmpeg fails to execute.

```kotlin
private val kCustomOptions = arrayOf(
    "-ac", "2",
    "-ar", "44100",
    "-rtbufsize", "32M",
    "-hls_master_name", "stream.m3u8",
    "-hls_playlist", "1",
    "-adaptation_sets", "id=0,streams=v id=1,streams=a",
    "-f", "dash", "stream.mpd"
)

Eray Ocak *
private fun executeInternal(settings: CameraStreamTaskSettings, fallbackMessenger: FallbackMessenger) {
    FfmpegExecutor.run(
        FfmpegCommandTask(
            FfmpegBackendAPIOption(CameraUtil.getPlatformCameraBackendAPIFormat()),
            FfmpegFramerateOption(framerate = 30),
            FfmpegVideoSizeOption(width = 640, height = 480),
            FfmpegHWInputOption(camera = settings.camera, audio = "0"),
            FfmpegCustomOption(option = "-map 0:v:0 -map 0:a\\?:0 -map 0:v:0 -map 0:a\\?:0"),
            FfmpegVideoCodecOption(codec = FfmpegVideoCodec.H264),
            FfmpegAudioCodecOption(codec = FfmpegAudioCodec.AAC),
            FfmpegStreamBuilder(index = 0)
                .withVideoBitrate(megaBitPerSecond = 1)
                .withVideoScaleFilter(width = -2, height = 240)
                .build(),
            FfmpegStreamBuilder(index = 1)
                .withVideoBitrate(megaBitPerSecond = 3)
                .build(),
            FfmpegAudioChannelNumberOption(channels = 2),
            FfmpegAudioBitrateOption(kBitPerSecond = 128),
            FfmpegSegmentDurationOption(duration = 2),
            *kCustomOptions,
        ),
        settings.contextPath
    ).run { this: TaskResult
        fallbackMessenger.passTaskResult( taskResult: this)
    }
}
```

Figure 2: The Kotlin code fragment that creates and executes FfmpegCommandTask

FfmpegExecutor uses options of the provided command task. Although executor works with the list of tokens/arguments, due to its underlying usage of java.lang.Process, it also logs a joined command definition for easier debugging.

While the command can change depending on the platform (Different platforms use different camera backends) and the fallback state (Camera ID), following command has been logged on Darwin platform and can be provided as an example:

```
ffmpeg -f avfoundation -framerate 30 -video_size 640x480
-i 0:0 -map 0:v:0 -map 0:a\?:0 -map 0:v:0 -map 0:a\?:0 -c:v
libx264 -c:a aac -b:v:0 1M -filter:v:0 scale=-2:240 -b:v:1
3M -ac 2 -b:a 128k -seg_duration 2 -ac 2 -ar 44100
-rtbufsize 32M -hls_master_name stream.m3u8 -hls_playlist 1
-adaptation_sets id=0,streams=v id=1,streams=a -f dash
stream.mpd
```

Commands directory is set to context path (settings.contextPath from Figure 2). Context path is defined in the startup by the ContextPathComponent and normally is set to a folder with UUID name and the “/tmp/dashstreamer“ base. Base path can be configured through the dashstreamer.contextPathBase value defined in the application.properties file Stream chunks and manifests are stored in the context path.

7. Latency Measurements

Following table shows measured latencies depending on the stream timestamp and different segment duration settings.

Segment Duration	Elapsed Time (Stream Timestamp)		
	Initial	**15 seconds in**	**40 seconds in**
2	10.67s	11.07s	15.34s
4	13.13s	15.61s	21.38s
6	13.99s	17.06s	22.23s

Table 1: Measured latencies depending on segment duration and stream timestamp.

It can be seen that higher segment durations lead to more latency. Latency value gradually increases as the stream goes on, which may be a result of the low computing power of the encoder server.