Evaluating back-end Languages for a Spotify Clone: A dot framework research paper
A Software Engineering Study Project

Yardi van Nimwegen
Complex software semester 6
Fontys University of applied sciences
2025-09-12

Document Version History

Version	Date	Author	Description
1	09-09-25	Yardi van Nimwegen	Initial draft created

Table of Contents

Contents

Abstract

This study investigates which programming languages are suitable for developing a scalable music streaming back-end, which is inspired by Spotify, capable of theoretically handling millions of concurrent streams. Using the **Development Oriented Triangulation (DOT) framework**, the research focuses on the application context, available work and innovation domains, for this specific paper the library, field and workshop strategies have been chosen. Simulated virtual streams are used to estimated the scalability of the system while remaining to be hostable on personal local hardware.

1. Introduction

A music streaming platform requires back-end systems capable of handling large-scale concurrent request while maintaining low latency.
This research uses the **DOT framework** to structure the investigation of programming languages for a high concurrency streaming back-end. The project simulates virtual streams to emulate millions of users, this addresses hardware constraints which are in place for this project, while still allowing evaluation of the performance, scalability and ecosystem support of multiple languages.

1.1 Why use not real streams. Developing a music streaming back-end capable of handling millions of concurrent users requires significant resources which can handle all this streaming traffic. Serving real audio streams to users requires high network bandwidth and memory resource, which scales directly with the number of concurrent streams that the application is serving. To keep the amount of memory being used small we can use the shared song model so that the song only has to be loaded into the ram once and not for every user. But the song also has to be streamed to every single user which scales logarithmically

Spotify itself has a maximum of 320kbps streaming to ensure “high” quality streaming, but when you make a calculation how much network traffic this requires makes it completely impossible to serve from a normal network.

$$N * \frac{B * 1000 * T}{8 * 1073741824} \text{ GB} = 186.2645149230957 \text{ GB}$$

Bitrate	Network traffic for 1 stream	Network traffic for 1M streams
256 kbps	$N * \frac{B * 1000 * T}{8 * 1073741824} \text{ GB} = 0.00003 \text{ GB}$	$N * \frac{B * 1000 * T}{8 * 1073741824} \text{ GB} = 29.80232 \text{ GB}$
192 kbps	$N * \frac{B * 1000 * T}{8 * 1073741824} \text{ GB} = 0.00002 \text{ GB}$	$N * \frac{B * 1000 * T}{8 * 1073741824} \text{ GB} = 22.35174 \text{ GB}$
128 kbps	$N * \frac{B * 1000 * T}{8 * 1073741824} \text{ GB} = 0.00001 \text{ GB}$	$N * \frac{B * 1000 * T}{8 * 1073741824} \text{ GB} = 14.90116 \text{ GB}$

Even at the lowest commonly used bitrate (128 kbps), streaming to **1 million users** requires nearly 15 GB/s of network throughput. A typical laptop network interface supports only about 1 Gbps ≈ 125 MB/s which is over 100 times less than what is needed to support 1 million streams. And even if the network could handle it, there are also limitations for the CPU for handling concurrent TCP/HTTP connections.

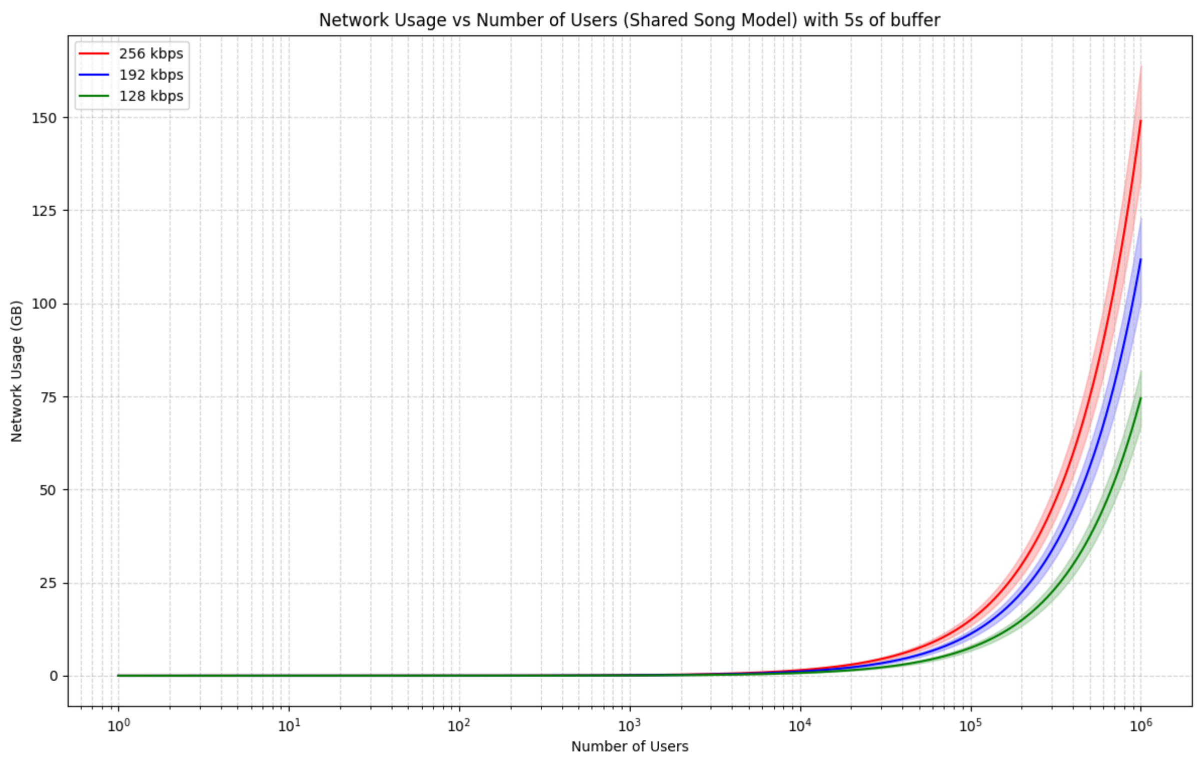

Figure 1: Network traffic with a 5 second buffer for audio streaming

Because of these reasons, **real audio streaming at scale is impossible** in a student project environment. To overcome this, the system uses simulated virtual streams, which simulate back-end workload without transmitting the actual audio. This makes sure that:

- Application can run on standard hardware
- Scalability testing for many more streams than regular ones
- Back-end focus on concurrency, throughput and latency
- Reproducibility across systems

This method follows the DOT framework

Methodology

This research will follow the rules of the **DOT Framework**, There are three domains that guide this research:

- **Application domain**: There is a need to create a back-end platform which is capable of scaling to millions of concurrent users, which reflects real-world music streaming platforms such as Spotify or Apple Music.
- **Available work**: Investigating prior studies on concurrency, back-end scaling and streaming architectures.
- **Innovation domain**: Implementing and evaluating back-end prototypes in different programming languages under simulated loads.

Research strategies applied in this research.

- **Library research**: Reviewing Spotify's engineering blog and scalability literature to inform design choices.
- **Lab research**: Running Controlled benchmarks with simulated streams.
- **Workshop research**: Iteratively refining prototypes and load simulation

Concurrency and Scalability

Before an analysis of specific languages can be done, it is required to get good understanding of what concurrency and scalability means. This project is a direct implementation of these concepts in a hard, real-world scenario.

Concurrency VS Parallelism. A common mistake in software engineering is the difference between concurrency and parallelism. Concurrency is that the system can handle multiple different tasks over a specific amount of time. While these tasks are running it might look like they are executing at the samen time, but they are not executing at the exact same time. A single core can execute concurrency by very quickly switch between tasks that have to be executed which is a technique known as time slicing.

Parallelism however, is when the CPU is handling executions specific tasks on separate processing units, these tasks are then executed by multiple CPU cores at once. This can of course be done on a laptop with only 4 cores but when more are available every single one of them can be used.

When looking at the difference between concurrency and parallelism, it becomes clear that for the Spotify clone project, that the way of handling a lot of connections in this project is a challenge of concurrency. To get to a high amount of concurrent connections we need to efficiently handle a lot of active connections and not do a lot of calculations at the same time, this is when parallelism would be required.

Which languages are commonly used for concurrency

For a project like this it is extremely important that the language that gets chosen has a great concurrency model, otherwise we will never be able to scale the project to the required amount of concurrent connections that have been set as a goal to strive for.

The Go Concurrency model. When talking about concurrency, Golang is usually one of the languages that pops up first since building concurrent systems is at the heart of Golang’s design. This is because of the very lightweight goroutines, which is a very lightweight thread which is managed by the Go runtime, and unlike thread in the OS GO goroutines are very efficient with memory, starting at a size of around 2KB. Because of this it becomes possible to spawn a huge amount of goroutines without running out of system resources.

The Java concurrency model (Virtual threads). Java, is a language with multiple decades of relevance and till this day still one of the most used languages in a corporate environment, has undergone massive modernization in recent years. In traditional Java concurrency still relies on a thread-per-request model which isn’t known for it’s high throughput. But in JDK21 Oracle has introduced a new way for developers to manage threads, with the new virtual threads Java underwent a huge upgrade in terms of concurrency. Since now the language is no longer reliant on OS thread and instead has lightweight threads like most modern languages.

The system behind these virtual threads is quite simple to understand. When one of the virtual threads makes a blocking call and has to wait on something (such as a database query or HTTP call), the runtime suspends the virtual thread, and the original OS thread that it is running on is freed which makes place for another virtual thread to run its task until it get blocked like the previous one. Just like the GO concurrency model can Java create a large number of virtual thread to a smaller amount of original threads.

Analysis of frameworks

The choice of which programming language should be used for this project is only half the question since there are many frameworks which are used. The decision on which framework should be used can significantly impact the performance and developer experience for creating the application.

Go’s Gin framework. Gin is one of Go’s most popular frameworks for creating web facing applications, this framework is known for its minimal footprint and fast performance which would make it a good consideration to keep in mind for this application since we are trying to reach for a lot of concurrent connections.

Java Quarkus. Quarkus is one of the newer guys on the block when considering Java frameworks. It is created for building “container first applications”, meaning that is optimized for fast startup times and low memory consumption, which are quite nice for containers, microservices and serverless environments.

Java Spring Boot framework. Out of these three options Spring Boot is the most mature one and widely adopted in the professional industry (Spotify itself also uses this framework). It provides the developer with a rich set of features, which significantly decrease the amount of boiletplate code and manual configuration that is required for regular Java. Even though Spring Boot is an older framework with the new **virtual threads** and **Spring webflux** you get a reactive non blocking framework which provides excellent performance with great developer experience