4. Asynchronous data and event streams

📌 Since reactive applications deal with non-blocking I/O, efficient and correct stream processing is key.

📌 most events need to be processed as series rather than as isolated events.

Unified stream model

Vert.x offers a unified abstraction of streams across several types of resources such as files, network sockets, and more

- read stream
- write stream

For example, an HTTP request is a read stream, and an HTTP response is a write stream.

Streams in Vert.x span a wide range of sources and sinks

Vert.x common read and write streams

Aa Stream resource	☑ Read support	☑ Write support
TCP sockets	☑	☑
UDP datagrams	☑	☑
HTTP requests and responses	☑	☑
WebSockets	☑	☑
Files	☑	☑
SQL results	☑	☐
Kafka events	☑	☑
Periodic timers	☑	☐

Read and write stream are defined through the **ReadStream** and **WriteStream** interfaces of the **io.vertx.core.streams** package.

These interfaces can be seen as each having two parts:

1. Essential methods for reading or writing data
2. *Back-pressure* management methods 🤔

ReadStream essential methods

Aa Method	☰ Description
handler(Handler<T>)	Handles a new read value of type T (e.g., Buffer, byte[], JsonObject, etc.)
exceptionHandler(Handler<Throwable>)	Handles a read exception
endHandler(Handler<Void>)	Called when the stream has ended, either because all data has been read or because an exception has been raised

WriteStream essential methods

Aa Method	☰ Description
write(T)	Writes some data of type, T (e.g., Buffer, byte[], JsonObject, etc.)
exceptionHandler(Handler<Throwable>)	Handles a write exception
end()	Ends the stream
end(T)	Writes some data of type, T, and then ends the stream

▼ Ex: Reading a file using JDK I/O APIs (java.io.*)

```java
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class JdkStreams {

  public static void main(String[] args) {
    File file = new File("file");
    byte[] buffer = new byte[1024];
    try (FileInputStream in = new FileInputStream(file)) {
      int count = in.read(buffer);
      while (count != -1) {
        System.out.println(new String(buffer, 0, count));
        count = in.read(buffer);
      }
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      System.out.println("\n--- DONE");
    }
  }
}
```

▼ Reading a file using Vert.x streams

```java
import io.vertx.core.Vertx;
import io.vertx.core.file.AsyncFile;
import io.vertx.core.file.OpenOptions;

public class VertxStreams {

  public static void main(String[] args) {
    Vertx vertx = Vertx.vertx();
    OpenOptions opts = new OpenOptions().setRead(true);
    vertx.fileSystem().open("file", opts, ar -> {
```

```java
      if (ar.succeeded()) {
        AsyncFile file = ar.result();
        file.handler(System.out::println)
          .exceptionHandler(Throwable::printStackTrace)
          .endHandler(done -> {
            System.out.println("\n--- DONE");
            file.close();
          });
      } else {
        ar.cause().printStackTrace();
      }
    });
  }
}
```

The approach is declarative here, as we define handlers for the different types of events when reading the stream. We are being **pushed** data, instead **pulling** data from the stream.

This brings us to the notion of **back-pressure.**

What is back-pressure?

Back-pressure is a mechanism for a consumer of events to signal an event's producer that it is emitting events at a faster rate than the consumer can handle them

In reactive systems,
back-pressure is used to **pause** or **slow down** a producer so that consumers avoid accumulating unprocessed events in unbounded memory buffers, possibly exhausting resources.

E.g, File streaming server

The implementation of such a server would involve doing the following:

1. Open an HTTP server.

2. For each incoming HTTP request, find the corresponding file.

3. For each buffer read from the file, write it to the HTTP response body.

Reading, then writing data between streams without any back-pressure signaling:
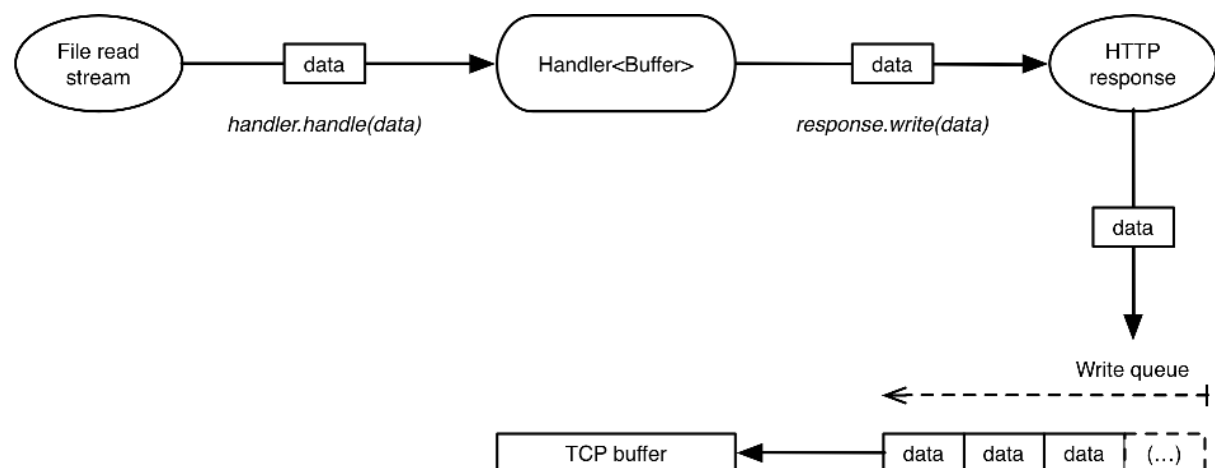
Reading from a filesystem is generally fast and low-latency, and given several read requests an operating system is likely to cache some pages into RAM. By contrast, writing to the network is much slower, and bandwidth depends on the weakest network link. Delays also occur.

The risk here is clearly that of **exhaustion**, either because the process eats all available physical memory, or because it runs in a memory-capped

environment such as a container. This raises the risk of consuming too much memory and even crashing.

📌 one solution is back-pressure signaling, which enables the read stream to adapt to the throughput of the write stream.

👉 Blocking I/O APIs have an implicit form of back-pressure by blocking execution threads until I/O operations complete. Write operations block when buffers are full, which prevents blocked threads from pulling more data until write operations have completed.

ReadStream back-pressure management methods

Aa Method	☰ Description
pause()	Pauses the stream, preventing further data from being sent to the handler.
resume()	Starts reading data again and sending it to the handler.
fetch(long)	Demands a number, n, of elements to be read (at most). The stream must have been paused before calling fetch(n).

WriteStream back-pressure management methods

Aa Method	☰ Description

Aa Method	☰ Description
setWriteQueueMaxSize(int)	Defines what the maximum write buffer queue size should be before being considered full. This is a size in terms of queued Vert.x buffers to be written, not a size in terms of actual bytes, because the queued buffers may be of different sizes.
boolean writeQueueFull()	Indicates when the write buffer queue size is full.
drainHandler(Handler<Void>)	Defines a callback indicating when the write buffer queue has been drained (typically when it is back to half of its maximum size).

the recipe for controlling the flow in our example of providing big files via HTTP:

1. For each read buffer, write it to the HTTP response stream.

2. Check if the write buffer queue is full.

3. If it is full

   Pause the file read stream.Install a drain handler that resumes the file read stream when it is called.

Note that this back-pressure management strategy is not always what you need:

- There may be cases where dropping data when a write queue is full is functionally correct and even desirable.

- Sometimes the source of events does not support pausing like a Vert.x `ReadStream` does, and you will need to choose between dropping data or buffering even if it may cause memory exhaustion.
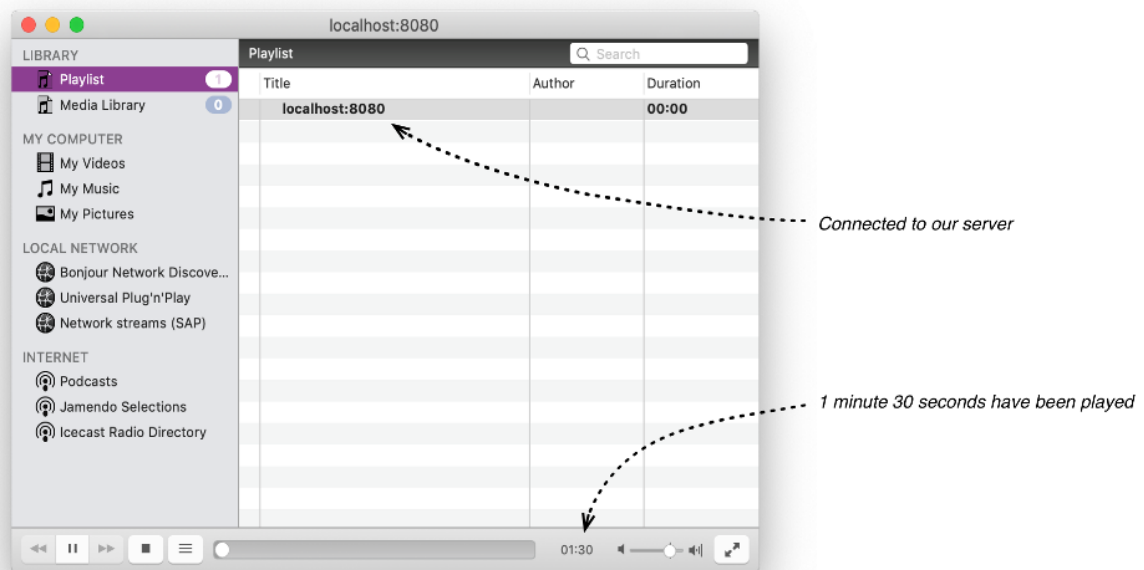
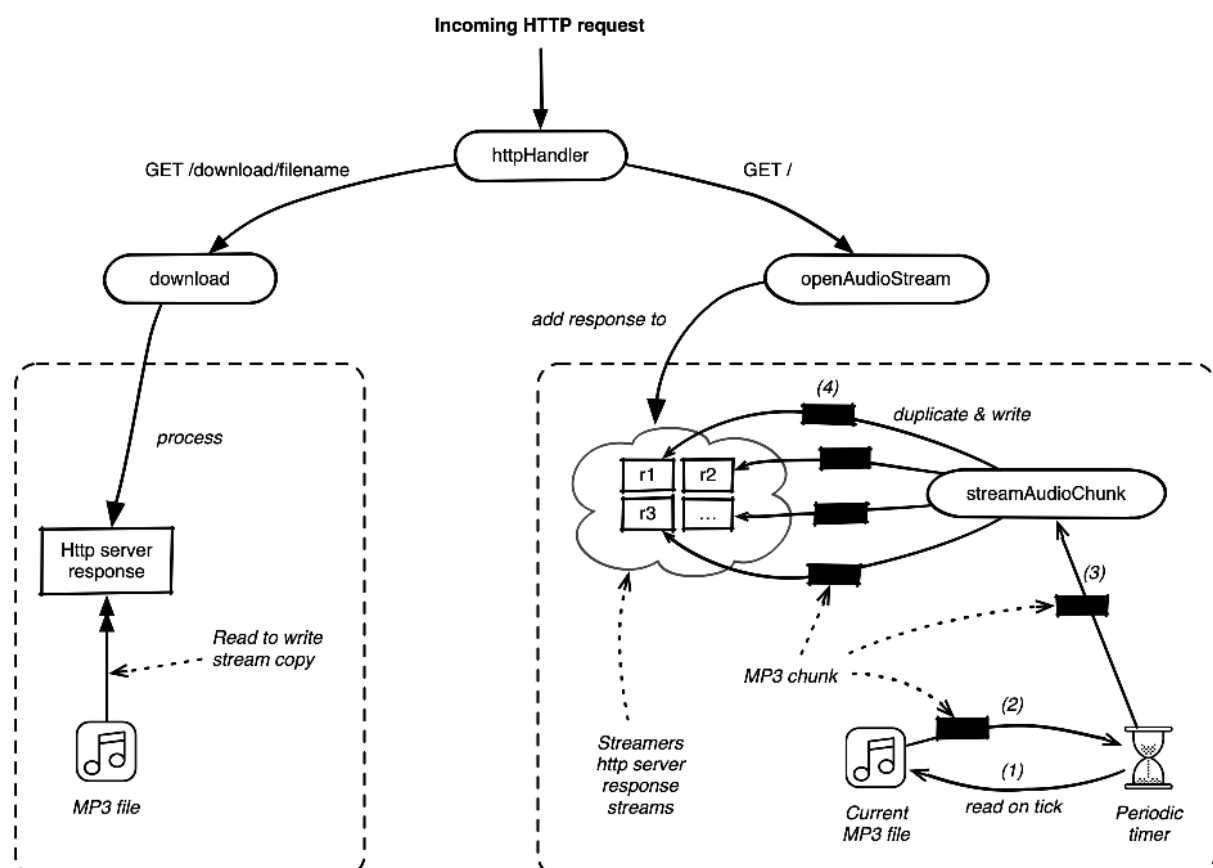**Making a music-streaming jukebox**

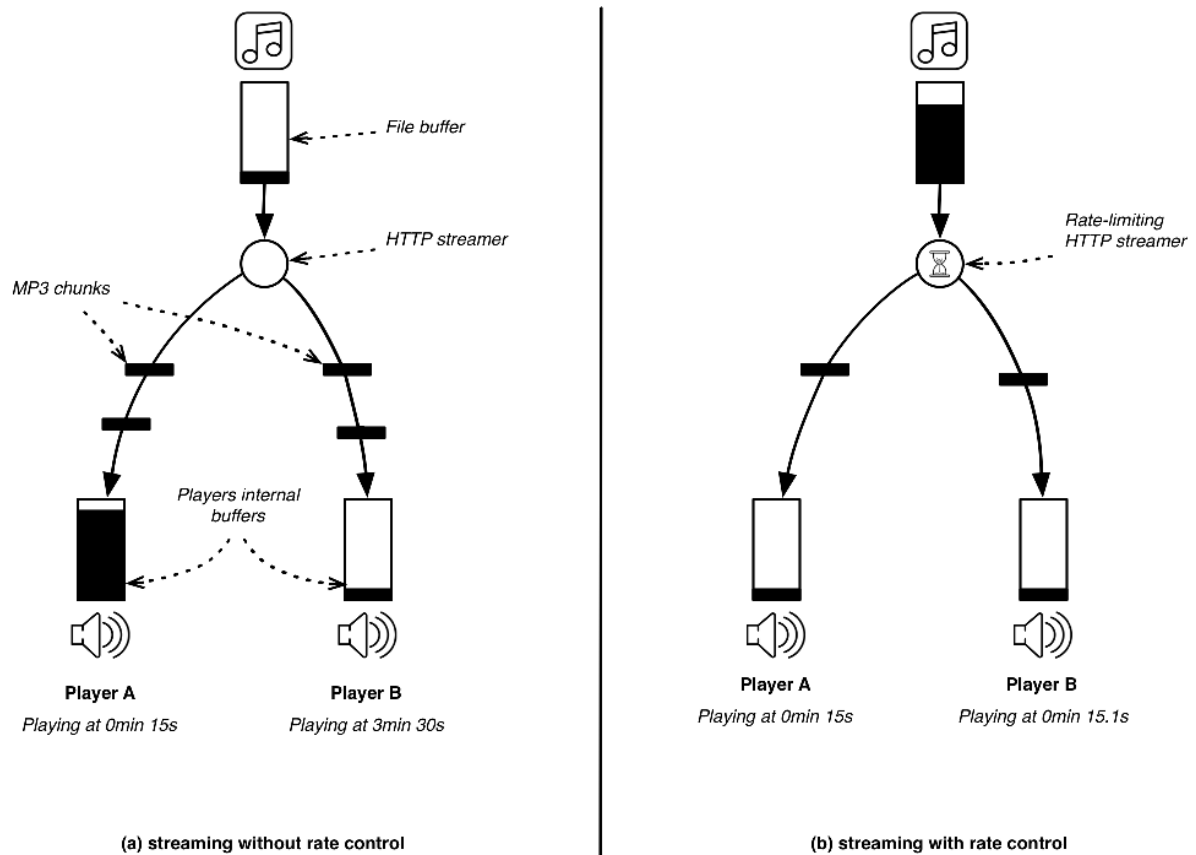## Jukebox application overview
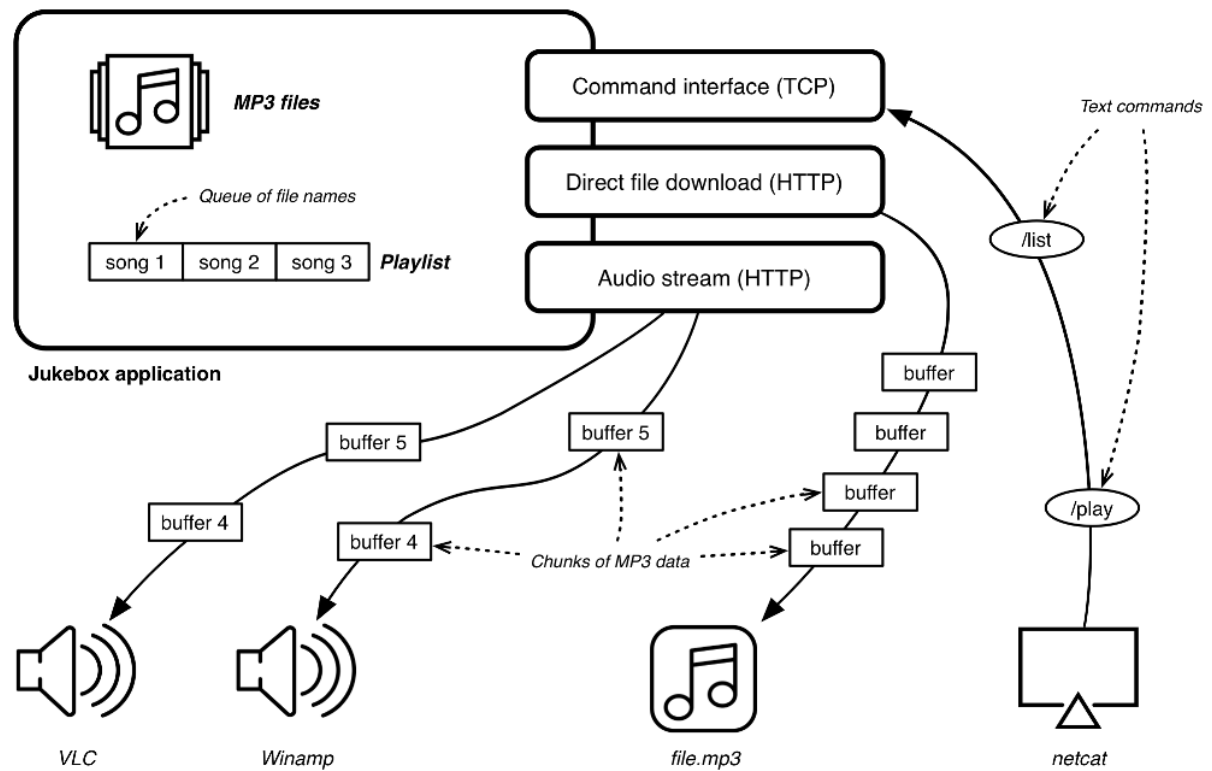
Features and usage

VLC connected to the jukebox

Connected to our server

1 minute 30 seconds have been played

HTTP processing: the big picture

Streaming without and with rate control

(a) streaming without rate control

(b) streaming with rate control

💻 - demo

Summary

- Vert.x streams model asynchronous event and data flows, and they can be used in both *push* and *pull/fetch* modes.

- Back-pressure management is essential for ensuring the coordinated exchange of events between asynchronous systems, and we illustrated this through MP3 audio streaming across multiple devices and direct downloads.