Report Assignment-02
Concurrent and Distributed Programming

Muhamad Huseyn Salman

May 2, 2025

Contents

1 Async Programming

1.1 Design

The application is built around three Verticle classes:

1. `ClassAnalyserVerticle`: it reads and parse a java source file using Vert.x filesystem and Java Parser. It produces a `ClassDepsReport`.
2. `PackageAnalyserVerticle`: it lists java source files in a given package and it deploys a `ClassAnalyserVerticle` for each file. Then it awaits all class analysis tasks and aggregates the results into a `PackageDepsReport`.
3. `ProjectAnalyserVerticle`: it explores recursively the project directory structure, and for each folder deploys a `PackageAnalyserVerticle`. Then it awaits on all package analysis tasks and aggregates the results into a `ProjectDepsReport`.

Each verticle receives as input a promise, that is used by the caller to monitor the execution state and to await for the result. Internally, each verticle has different async methods that represents the various step it has to do to complete its task. These steps are chained in the `start` method.

All access to files or directories is done through the Vert.x filesystem to avoid blocking the event-loop. Moreover, parsing the file with Java Parser and navigating the AST is done using `executeBlocking`, in order to delegate the computation to a worker thread and not to the event-loop.

1.2 Behavior

This first Petri Net shows the high-level behavior of the library. It enfasizes that packages and files are analyzed in parallel using different verticles. Their futures are then combined at each step using `FutureAll`

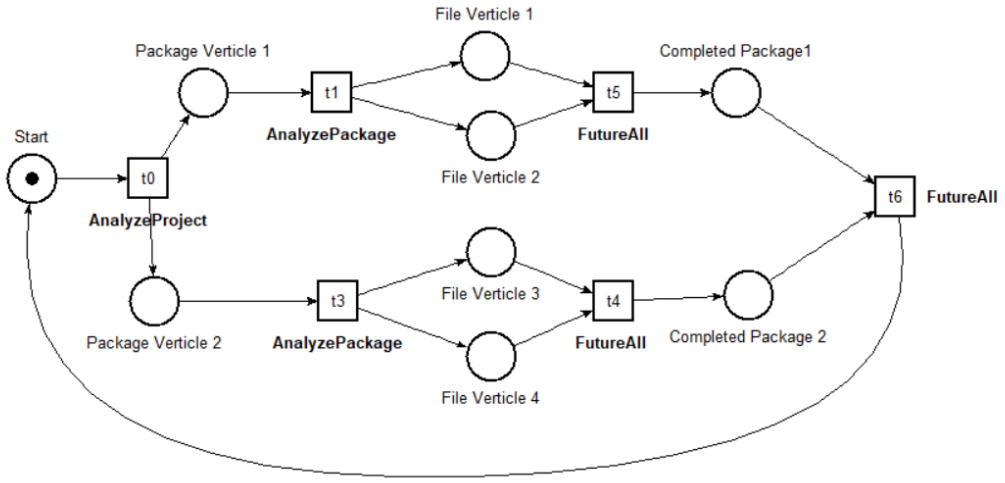

Figure 1: High-level Behavior

The next Petri Net shows the detailed async behavior of the PackageVerticle and FileVerticle. The former deploys a new FileVerticle for each file. Each FileVerticle asynchrnously parse the source file and completes its future returning the list of dependencies as a FileReport. These lists are then aggregated into a PackageReport by the PackageVerticle when all files' futures have completed.

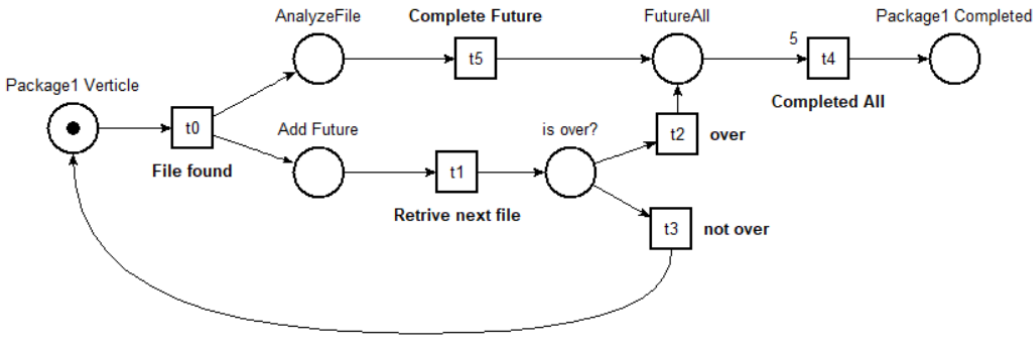

Figure 2: Detailed Async Behavior

2 Reactive Programming

2.1 Design

The backend is based on four main methods:

1. `extractDependencies`: it reads and parse a java source file, then it emits an item for each dependency found. It produces a `SingleDependencyResult` observable.

2. `analyzeFile`: it subscribes to the dependencies streams for a file, it adds the file name and the package name to each item of the stream.

3. `analyzePackage`: for each java source file in the provided package it invokes the `analyzeFile` method. Then it flats all observables into a single stream of dependencies for that package.

4. `analyzeProject`: for each package in the project, it invokes the `analyzePackage` method. Then it flats all observables into a single stream of dependencies for that project.

To ensure that file I/O and parsing operations do not block the main thread, the production of items is scheduled on an I/O-optimized thread pool provided by RxJava using `subscribeOn` method.

Swing GUI: the GUI uses RxJava to handle `ActionEvents` reactively, subscribing to a `PublishSubject` object.
When the analysis is started, the gui subscribes to the stream returned by the `analyzeProject` method, and for each new dependency received it re-renders the panel, therefore the graph is displayed incrementally.
Source files nodes are colored with random colors, instead dependency file nodes are colored with gray shades. A legend panel shows each file name associated with its color, and it is updated incrementally for each new dependency. Nodes are drawn randomly on the panel.
On the bottom of the panel three counters show the number of file and packages analyzed, and the number of dependencies found. The dependency counter is updated incrementally.

2.2 Behavior

This first Petri Net shows the high-level behaviour of the application, it enfasizes how the streams of data are produces in parallel and then combined together until they reach the GUI. It has not been drawn here, but each observable should be intended as a cyclic petri net that produces a stream of tokens, one for each dependency (see next petri next for more details):

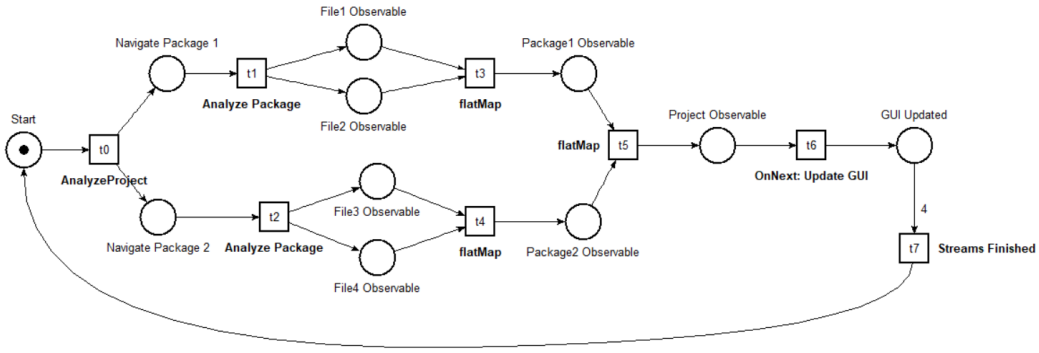

Figure 3: High-level Behavior

The next graph shows in details how the core stream is created with emitting parsed dependencies and how a stream of token is produce in the Petri Net. This stream of tokens will be processed also at the package anc project level until it reaches the GUI, which at that point update incrementally the graph.

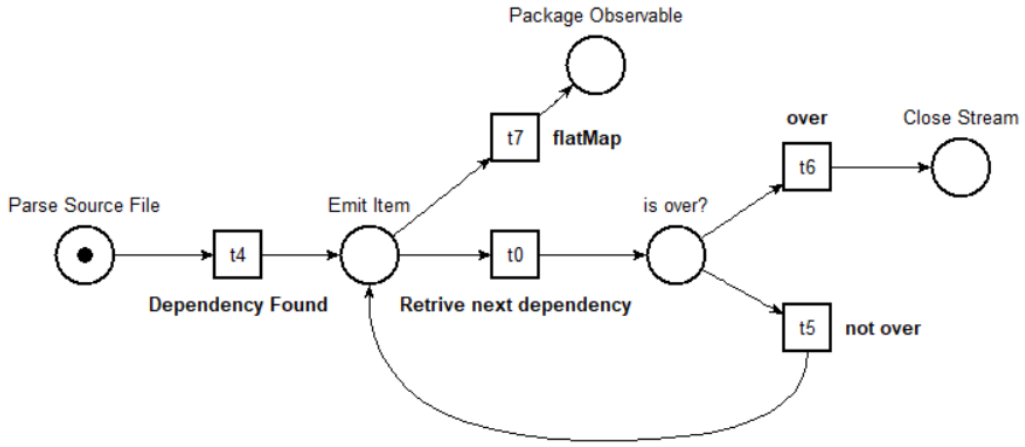

Figure 4: Detailed Item Emission