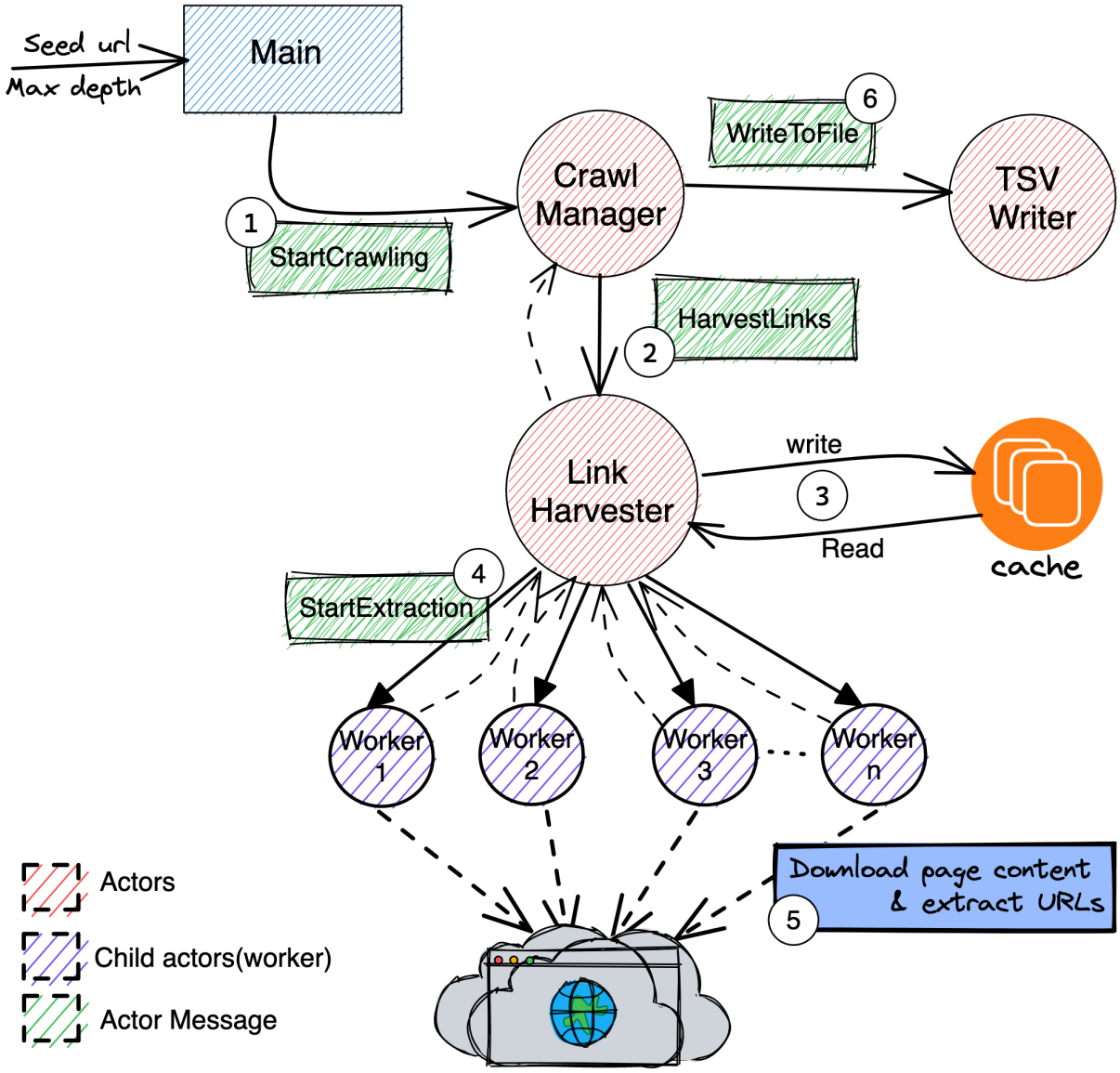
Web Crawler

A simple web crawler built using Scala and Akka Typed

Tech Stack:

- Scala
- Akka Typed
- Caffeine (In-memory cache)

Architecture



Various components of this web crawler are designed using akka actors to achieve maximum concurrency in a non-blocking fashion. The result obtained after crawling each webpage is written immediately to a TSV file.

The application starts with reading two input parameters - seed url - The URL to start crawling at - max depth - Maximum depth until the crawler run recursively.

1. Crawl Manager

An actor manages the entire web crawling process. It creates a child actor called `LinkHarvester` and delegate work to it.

1. On receiving message `StartCrawling`, it is submitted to LinkHarvester
2. LinkHarvester responds with `HarvestedLinks` on successful URL download
3. LinkHarvester responds with `LinkHarvestFailed` on any failures
4. Calculate `page rank` on receiving successful response and send it to `TSVWriter`

This actor performs url downloads recursively by adjusting its behavior. Additionally, this actor keeps an internal state `Map[Depth, Number of requests]` to keep track every requests in-flight.

2. Link Harvester

This Actor coordinates url downloads and cache access. It creates `N` child actors (`LinkExtractionWorker`) based on the number of child Urls a page has. Each child actor will download and parse a single URL.

1. On receiving message `HarvestLinks` , it checks the cache for the URL.
    - If an entry is found in Cache, it will return the cached value to CrawlManager
    - If no entry is found in cache, it will spawn a worker and send a `StartExtraction` message to it.
2. When a response is obtained from a worker,
    - It will be written to cache
    - Return crawled urls to `CrawlManager`

3. Cache

An in-memory cache implementation backed by Caffeine It uses `Window TinyLfu` See: https://github.com/ben-manes/caffeine/wiki/Efficiency

For implementing caching within this application, an interface is provided using the trait `WebCrawlerCache` This way, introducing a new cache (eg: Redis) can be done without significant code changes. Just provide an instance of your custom cache implementation at application startup.

See: `Main.scala#RootBehavior.apply`

4. LinkExtractionWorker

These are child actors created on demand by the `LinkHarvester` actor. It performs the actual work of loading a webpage and extract child urls from it. It uses JSoup under the hood to perform URL scraping.

On successful URL download, - it will return `LinkExtractionSuccess` to LinkHarvester.

On any failure, - it will return `LinkExtractionFailed` to LinkHarvester.

> These are lightweight and short-lived actors. After completing the designated work, it stops and release memory. We can utilize the maximum available CPU cores on a

> machine this way. The current implementation is not distributed(runs on a single machine), but we can introduce an akka cluster and distribute these actors to multiple nodes for any future scalability requirements.

5. TSV Writer

This is a dedicated actor to perform write operations to a `TSV file`. - It will create the output directory if not present - The output directory path can be configured using the application configuration - It can also be configured using an environment variable `CRAWLER_OUTPUT_DIRECTORY` See: `application.conf`

Running application locally

Project structure

```
.
├── README.md
├── build.sbt  <- sbt build definition
├── docs       <- supporting readme assets
├── output     <- output directory
├── project    <- sbt specific settings
├── src
│   ├── main
├── resources
│   ├── application.conf <- the application configuration
│   └── logback.xml      <- logging configuration
└── scala
│   └── com
│       └── robinraju
│           ├── Main.scala   <- the main application entrypoint
│           ├── cache        <- cache implementations
│           ├── core         <- helpers for config and output result
│           ├── crawler      <- akka actors
│           ├── io           <- TSV file writer
│           └── util         <- Pagerank calculation function
│    └── test    <- unit tests
```

Prerequisites

- Java 11 or greater
- SBT 1.3.x or greater - https://www.scala-sbt.org/download.html

Verify tests and running crawler

Start a new sbt session from the project root `sbt` Run tests using the `test` command

```
sbt:web-crawler> test
```

Run crawler using the `run` command
`sbt:web-crawler> run https://crawler-test.com/ 2` This will start crawling the above url until depth 2.

Check the `output` directory for `TSV` file.

> For every run it creates a new `TSV` file with a filename containing system time stamp.

 Future Improvements

There is room for a lot of improvements on this crawler. Keeping the current crawling logic (just calculating pagerank), I would add the following

1. Metrics to monitor performance and behavior of different actors
    - Number of worker actors running
    - Crawl latency
    - URL fetch failure count
    - Cache hit/miss ratio
    - Cache size
2. Application packaging for deployment
    - A cli app with some command line argument parsing
3. Clustering, for distributed crawling
    - Akka cluster allows us to start crawler on different nodes/machines and join the existing cluster
4. More test coverage