1 2 UNIVERSIDADE D COIMBRA 9 0

Sistemas Distribuídos 2024/2025
Faculdade de Ciências e Tecnologia
Universidade de Coimbra

Exercises 3 - Distributed Web Indexing with RPC

This tutorial will guide you through building a distributed web indexing system similar to early search engines. You'll explore key distributed systems concepts including:

- Remote Procedure Calls (RPC)
- Exception handling
- Asynchronous approaches (Callback/Stream)
- Processing guarantees
- Thread-safe structures

Overview

You'll implement a distributed web indexing system with the following components:

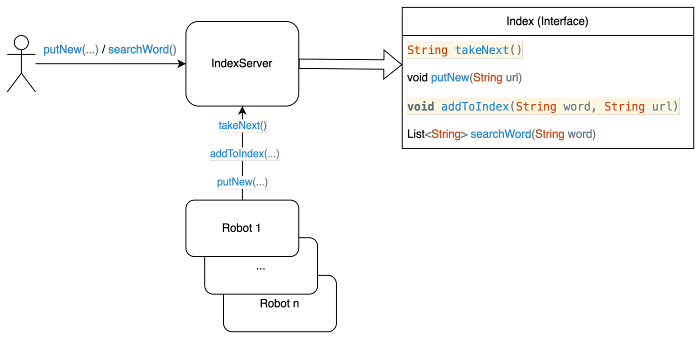

Figure 1: Tutorial 3 Architecture Overview

- IndexServer: Keeps Units of work to be distributed and maintains the global index
- Worker: Process web pages, extract keywords and update index

The final expected workflow is as follows:

1. The *IndexServer* receives a URL to be indexed (e.g. https://pt.wikip…) via a remote invocation to the `putNew(String Url)` call.
2. The *Robot* requests a URL to index by invoking `takeNext()` on the IndexServer.
3. The *Robot* simultaneously parses the webpage's words and URLs, updates the index and submits any found urls for further index, invoking `addToIndex(String word, String Url)` for each word found and `putNew(String url)` for each URL link found in the webpage
4. A User searches for a term by remotely invoking the `searchWord(String word)` and receives a collection of URLs were the word exists.

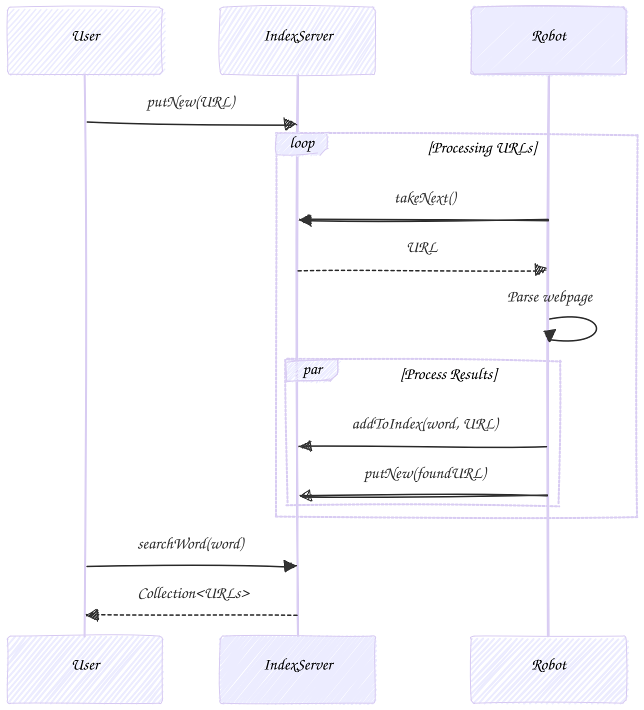

Figure 2: Tutorial 3 Flow Diagram

Initially, there is no user interaction, as we have "seeded" a first URL to be indexed in the implementation: https://pt.wikipedia.org/wiki/Wikipédia:Página_principal.

Building a Distributed Indexer

Part 1: Server Setup and RPC

1. Examine the provided interface in `Index.java` or `index.proto`

   These interfaces provide the basics for *Stub and Skeleton* generation. A *Stub* and *Skeleton* are generated code that allow for both the Client and Server to invoke each other without the need for **concrete implementation** details being shared. Only the methods and datatypes are used.

   For RMI, this generation is "transparent", as only the compiled *.class files are needed.

   For gRPC, we must generate the stub and skeleton code - For this, use the provided commands in `generate-gRPC-code.sh`.

2. Develop a Concrete implementation of the Indexe interface (`IndexServer.java` or `indexServer.python`) that:

   - Provides URLs on-demand, for processing, to any requesting workers. Start with the `takeNext()` function provided in the exercise.
   - Maintains the global index. For this, use the existing code in IndexServer and implement the `takeNext()` and

3. Implement a worker (`Robot.java` or `robot.py`) that:

   - Fetches and Processes URLs (invoking the `takeNext()` RPC on the server)
   - Extracts words from the webpage. The `Robot` file contains an initial example with `JSOUP` for Java and `BeautifulSoup` in python. Students should read the API and make use of tokenization and link extraction via the appropriate functions (e.g. `select("a[href]")`
   - Publishes each found word back to the Indexer using `addToIndex(String word, String url)`
   - Put any found links in the queue for future indexing using `putNew(String url)`

Part 2: Data structures selection:

Students should test simple scenarios first: 1 server, 1 worker, 1 url indexed. Most data structures will work in this simplified scenario. Problems might arise when we add more than one Robot. Students are encouraged to experiment with multiple robots, and assess what level of concurrecy will occur, specifically at the `putNew(url)` and `addToIndex(word, url)` functions.

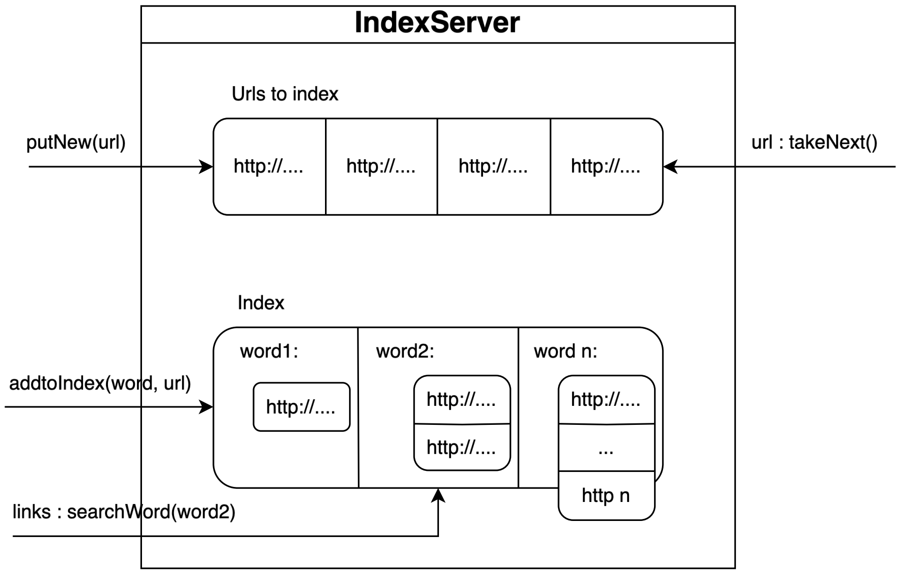

Figure 3: Tutorial 3 Data Structures Overview

4. What happens if we have a *Urls to Index* data structure of fixed size? What impact can that have? Students are advised to prefer dynamic data structures: Data structures that don't require a fixed size on creation and allow for operations analagous to `add(element)` and `removeFirst(element)`
5. Declare *Urls to index* as a concrete implementation of `Queue` (or similar). Make the appropriate changes to your code.
6. Declare *Index* as a concrete implementation of `Map` (or similar). Make the appropriate changes to your code.
7. Are the *Index* and *Urls to Index* thread-safe? Students are encouraged to start several robot workers and analyse the behaviour. Can Thread-Safety be guaranteed?

Part 3: Separation of Concerns (UI), Resilience and Fault Tolerance

User Interaction - via `putNew(String URL)` and `searchWord(String word)` - can initially be conducted directly on the server, via Stdin. Users should be able to input urls for indexing and search terms for retrieval, as exemplified below:

```
##################################################
### Add urls for indexing: Start with "http"   ###
### Search keywords: start with anything else  ###
##################################################
Input: https://pt.wikipedia.org/wiki/Wikipéd...
-> Server: added https://pt.wikipedia.org/wi...
Input: Educador
-> Server: {'https://pt.wikipedia.org/wiki/W...
```

8. Make the service interactive. For this implement the following loop in the IndexServer:
   - receive an input string from stdin
   - if string begins with http: or https: add to *Urls to Index* queue (`putNew(..)`)
   - if string doesn't begin with http: or https: retrieve the list of URLs where the search word exists (`searchWord(...)`)
9. A few client features are being guaranteed by the Indexer. This not only adds the potential for blocking the server, as it is bad Separation of Concerns. Create a Client that can:
   - Submit URLs for indexing
   - Search for keywords
   - Monitor indexing progress and statistics Students are encouraged to try and run the client on separate computers. This can be achieved by changing the following lines appropriately (192.168.0.3 is an example IP):

   ```
   LocateRegistry.getRegistry("192.168.0.3",
   8183).lookup("index");
   grpc.insecure_channel('192.168.0.3:8183')
   ```
10. Server stats. As we progressively process more and more data, it is important to keep track of our program's indexing "speed" and memory usage. Develop a function that:
    - Prints time taken between each print of Server stats. Students are encouraged to divide by 10 to get an average of time taken to parse each page
    - Prints Memory usage. Simple memory stats suffice (e.g. used/free), although students are encouraged to be more granular (e.g. process, data structure memory usage)
    - Prints Server stats every 10 worker requests - on `takeNext()` Useful functions:

    ```
    System.currentTimeMillis()
    ```

```
Runtime.getRuntime().totalMemory()
Runtime.getRuntime().freeMemory()
time.Time()
psutil.Process().memory_info().rss
sys.getsizeof(self.indexItems)
```

11. (Optional) Modify the IndexServer, adding a function called `String WorkerStatsCallback()` that will allow the Server to have a callback (RMI)/client-streaming (gRPC) functionality. This function, implemented on each *Robot* when invoked(RMI)/triggered(gRPC) should return a String containing the number of urls processed (number of times `takeNext()` retrieved an url and was sucessfuly indexed).
12. (Optional) Add persistence to the global index and statistics
13. (Optional) Implement work redistribution when a worker takes a URL but fails to add to index

Extra Questions:

1. How does the system handle network partitions?
2. What happens if the Indexer fails? How could you make it fault-tolerant?
3. How would you scale this system to handle millions of URLs?