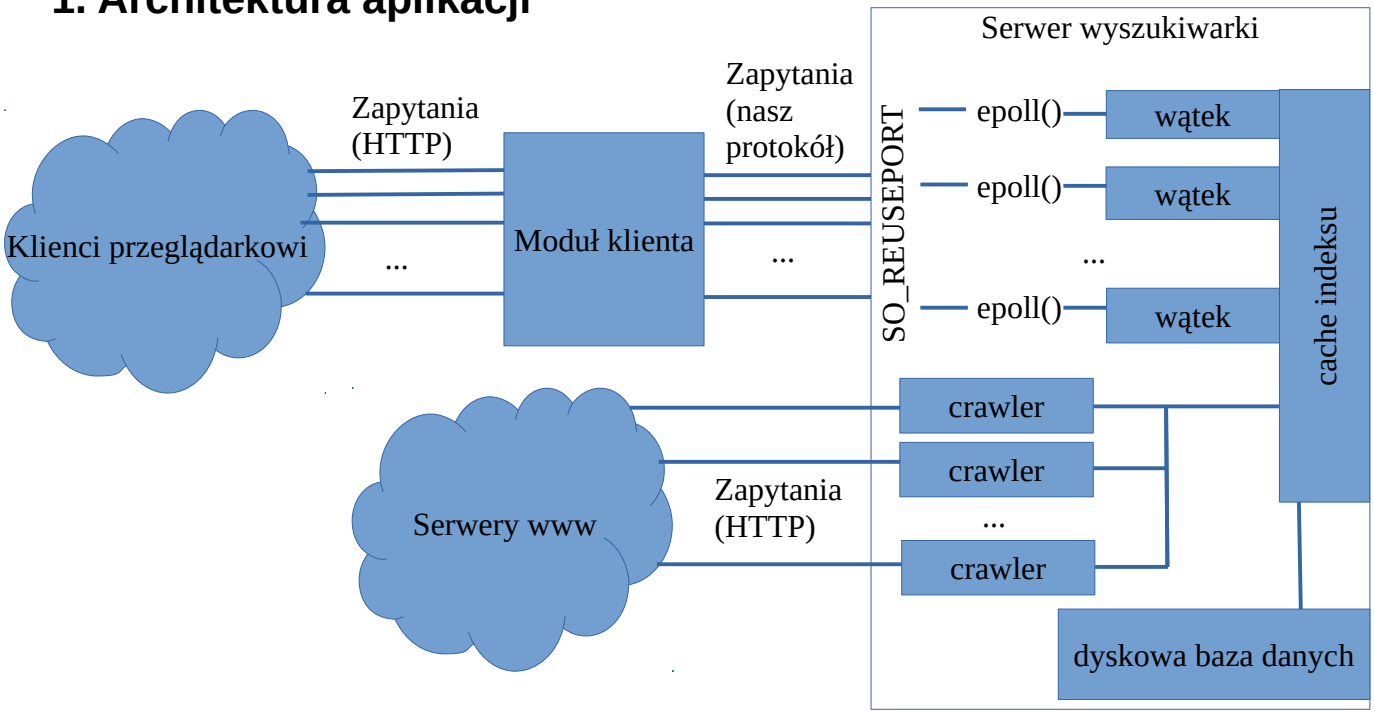
System robota indeksującego strony HTML wraz z aplikacją umożliwiającą wyszukiwanie stron na podstawie słów kluczowych

Daniel Nowak 136775, Kamil Piechowiak 136780

1. Architektura aplikacji



Rysunek 1: trzy moduły, działające w dwóch parach w architekturze klient-serwer. Odnosząc się dalej do klienta/serwera będziemy wspominać parę moduł klienta – serwer wyszukiwarki

Aplikacja klienta została zbudowana za pomocą frameworka Spring Boot w javie. Na odpowiednich portach mapuje ona URL na dynamiczne dokumenty HTML, wraz z zasobami .js/.css, generowane przez template engine w postaci Thymeleafa. Dane, którymi wypełniamy strony, są pobierane z serwera wyszukiwarki. Na wystawionym porcie serwera funkcjonuje mechanizm load balancingu na poszczególne wątki wyszukiwarki, które kolejkują wiele połączeń równocześnie i obsługują je w sposób nieblokujący przy pomocy multipleksacji mechanizmem epoll. Wątki wyszukiwarki są zarządzane w naszej autorskiej dynamicznej puli, chociaż ostatecznie wykorzystujemy ich stałą liczbę (temat load balancingu, dzielenia surowców itd. nie był naszym głównym tematem). Zarówno wyszukiwarka jak i roboty posługują się tym samym mechanizmem przechowywania danych. Roboty przeszukujące sieć w celu indeksowania stron przetwarzają zwracany kod html i go odpowiednio indeksują po słowach kluczowych.

2. Źródła

W podkatalogu client naszego folderu znajdziemy projekt modułu klienta napisany w javie, wraz ze wzorcami HTML i częścią implementacji naszego protokołu klient-serwer. Logika komunikacji jest zaimplementowana w klasie net.controller.SearchService. W folderze server z kolei znajduje się implementacja serwera wyszukiwarki (@Rysunek 1). Rozważając foldery src i include w podfolderach:

- crawler znajdują się pliki .cpp zawierające implementację crawlerów oraz parsowania HTML

- multi_threading znajdują się klasy pomocnicze do obsługi wielowątkowości aplikacji, oraz stara dynamiczna pula wątków, której postanowiliśmy nie usuwać z repozytorium. Pierwotnie myśleliśmy o rozdawaniu requestów wątkom na poziomie naszej aplikacji, ale później poznaliśmy odpowiednie funkcjonalności systemu linux.

- service znajdują się implementacje serwerów przeznaczonych do przebywania w nowej puli, która również znajduje się w tym podfolderze.

- storage znajdują się klasy obsługujące przechowywanie zaindeksowanych danych.

Co ważne, ze względu na wysoką generyczność pisanego przez nas kodu, właściwa obsługa zapytań od klienta znajduje się w main.cpp.

3. Protokół komunikacyjny

1. Moduł klienta inicjuje połączenie do serwera przesyłając zapytanie w postaci

   “${PAGE} ${PAGE_SIZE} ${SEARCH_QUERY}\n”, gdzie PAGE i PAGE_SIZE dotyczą głębokości i liczby żądanych wyników, a właściwe zapytanie składa się ze słów kluczowych.

2. Serwer odpowiada zgodnie ze wzorcem, który składa się z nagłówka:

   “(?<status>#.+?)${HEADER_SEPARATOR}(?<totalPages>[0-9]+?)${HEADER_SEPARATOR}”

   oraz z ciała odpowiedzi:

   “((?<url>.+?)${BODY_SEPARATOR}(?<title>.+?)${BODY_SEPARATOR}(?<description>.+?)

   ${BODY_SEPARATOR}\0)*”.

3. Serwer kończy połączenie, przekazując tym samym informację o końcu przekazywanych wyników.

Stosowanego przez nas w innych częściach projektu protokołu HTTP nie będziemy opisywać.

4. Uruchamianie projektu

Przed uruchomieniem projektu należy mając zainstalowanego mavena (sudo apt install maven) wejść do folderu client i zainstalować w lokalnym repozytorium mavena zależności, np. ‘mvn clean install’. Następnie należy zainstalować/upewnić się o instalacji libcurl ‘sudo apt-get install libcurl4-openssl-dev’.

Mając zainstalowane zależności wystarczy skorzystać ze skryptów build-run.sh lub run.sh. W podfolderze server, run_populate.sh uruchamia web crawlery i mechanizm indeksowania.

GUI możemy odwiedzić na stronie localhost:8080 (chyba, że zmienialiśmy domyślny port).

Możliwości konfiguracyjne możemy odczytać w poszczególnych skryptach rozruchowych (są bardzo krótkie).