Проект

по Системи за Паралелна Обработка

Архитектура на клиент-сървър приложение за паралелна обмяна на чат съобщения

Изготвил:

Георги Матеев ф. н. 80728

Ръководител:

ас. Христо Христов

Проверил: ........................

(ас. Христо Христов)

1. **Описание на задачата**

Да се реализира уеб приложение, което позволява обмяна на текстови съобщения между множество потребители. За целта трябва да се направят две независими една от друга програми – чат клиент от който може да се изпращат и визуализират съобшения и чат сървър, който да управлява връзките между клиентите.

Някои от по-важните характеристики на двете страни в тази архитектура са както следва:

Характеристики на Клиента:

* Подава заявки
* Изчаква отговор
* Взаимодейства си с крайните потребители чрез графичен интерфейс

Характеристики на Сървъра:

* Пасивност
* Чака за заявки от клиенти
* При получаване на заявки, ги обработва и след това отговаря
* Получава заявки от голямо количество клиенти

1. **Функционалност**

Текущата реализация поддържа следната функционалност:

* Потребителят навигира до уеб страница(клиента) в браузъра и въвежда потребителско име
* Наличен е списък с всички потребители, които използват приложението в момента
* Потребителят може да изпрати съобшение до всички на линия във вече наличната обша стая за чат
* При избиране на потребител от наличните в списъка се отваря нова чат стая в която може да се комуникира само и единствено с него
* Потребителят може да затваря вече отворени чат стаи

1. **Реализация на уеб сървъра**

Най-важните изисквания за сървъра са да поддържа множество конкурентни потребители и да може да достава голям брой съобшения в адекватен интервал от време.

**Платформата** която се използва е Node.js. Тя има следните отличителни черти:

* Притежава вграден HTTP сървър и като цяло е с малък обем. Написана е на C++.
* Всеки процес, който изпълнява Node приложение използва само една нишка. Заявките, които постъпват се обработват асинхронно в javascript event цикъл. Това е противоположно на приетия в Java, .NET и др. подход където за всяка заявка се създава нова нишка, която хаби допълнително оперативна памет.
* Всички входно-изходни операции като четене от файлова система, комуникиране през TCP и WebSockets са имплементирани така, че да не блокират изпълнението на главната нишка.

**Паралена обработка на заявки**

Тъй като процесът, който изпълнява уеб сървър програмата използва само една нишка, това означава, че всички заявки ще се обработват от един логически процесор. Node.js се справя с този проблем благодарение на вграден в библиотеката модул - Cluster. Той позволява паралелна обработкана данни. За всеки логически процесор ще се стартира процес на операционната система, който изпълнява една и съща уеб сървър програма. Тоест ако имаме двуядрен процесор с четири логически ядра, ще се стартират четири напълно еднакви уеб сървъра, които слушат на точно един споделен порт. Всяка пристигаща заявка ще се изпълни от един от сървърите и така може да се обработят четири заявки едновременно.

**Комуникация между клиент и сървър**

Осъществява се благодарение на javascript библиотеката Socket.IO. Ако клиентът поддържа WebSocket протокол се отваря двупосочна връзка със сървара. Така той може да изпраща съобшения до клиента, без последният да е инициирал питане. Ако не се поддържа този протокол, клиентът изпраща заявки през определен интервал от време, за да получи информация от сървъра.

Понеже за отварянето на WebSocket канал се изискват две http заявки наречени handshake, които зависят една от друга, сървърът имплементира Sticky session design pattern. IP-то на всеки клиент се асоциира посредством хеш функция с ID на някого от процесите. Тоест един клиент изпраша съобшения само до един процес. По този начин няма опасност заявка, която разчита на данни в паметта, да попадне при грешния процес.

**Синхронизация на процесите**

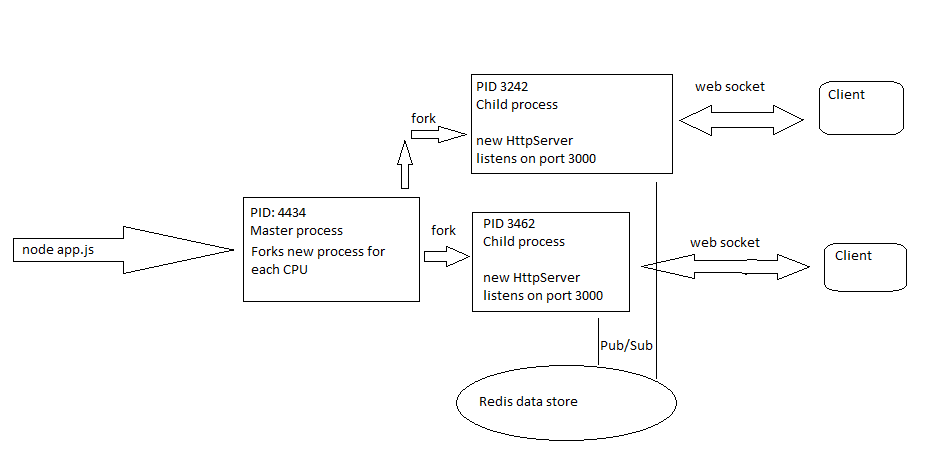
За споделяне на информация между Node програми в отделни процеси се ползва NoSql база с данни Redis. Socket.IO препраща съобщения между процеси благодарение на Publish/Subscribe функционалността на Redis. Ако в процес А пристигне съобщение за socket Б, базата ше уведоми всички процеси за новото събитие и този към когото е вързан socket Б ше изпрати съобшението.

В базата се пазят и всички потребители ползващи приложението в момента.

1. **Реализация на клиента**

Клиентът представлява Single page application и е достъпен от интернет адрес през браузъра. Библиотеката Angular JS спомага за интерактивността, а за комуникация със сървъра отново се ползва Socket.IO. Потребителят задължително трябва да изпрати име с което уникално да се разпознава. Може да се изпращат съобшения до определен потребител или всички потребители. Клиентът се абонира за сървърни събития като невалиден или съществуващ потребител, ново съобщение или прекъсване на потребителската сесия. За всяко събитие се извежда подходяща и разбираема нотификация.

**Цялостна архитектура на приложението**



1. **Тест на натоварването**

За да се провери дали описаната реализация може да понесе голямо натоварване,

ще разгледаме следната задача: За колко време могат да се обработят 100 000 заявки като имаме по 1000 заявки едновременно. Всяка заявка връща по 28 байта.

Тестът е направен с Apache Benchmark от независим сървър през интернет.

**Време за изпълнение в секунди**

На тази графика са изобразени стойностите

S(P) = T(1) / T(P)

E(P) = S(P) / P , където P е брой нишки.