DOI: https://doi.org/10.33216/2222-3428-2021-20-14

УДК 004.4'236

Щербаков Є.В., Щербакова М.Є.

АЛГОРИТМИ ДИСПЕТЧЕРИЗАЦІЇ ПОДІЙ В NODE.JS

У традиційній клієнт-серверній архітектурі один потік не може обробляти кілька запитів одночасно через блокуючі операції введення-виведення. Операції роботи з мережею, а також читання і запису на диск занадто повільні в порівнянні зі швидкістю виконання інструкцій процесором комп'ютера. Тому диспетчер програм Node.js використовує орієнтовану на події модель і неблокуючу архітектуру введення-виведення, що робить його легким і ефективним за рахунок наступного: головний потік не блокується операціями введення-виведення; сервер під час операцій введення-виведення продовжує обслуговувати інші запити; сервер розробляється як набір асинхронних програм, керованих подіями, і не обов'язково потокобезпечних. Цикл обробки подій, який є центральною частиною диспетчера Node.js, реалізується відповідно до звичайного однопотокового підходу до асинхронного введення-виведення. Коли циклу обробки подій, що працює в основному потоці додатка, потрібно виконати операцію введення-виведення, для її асинхронного виконання він використовує інший потік з пулу потоків, а коли виконання операції завершується, функція зворотного виклику (колбек) ставиться в чергу подій для послідовної обробки основним потоком. Функції з черги подій (колбеки) виконуються тільки після того, як стек викликів основного потоку буде повністю очищений. Тільки після цього функції з черги подій поміщаються в стек викликів основного потоку для виконання. Якщо в стеку викликів основного потоку в даний момент знаходиться хоча б один елемент, то колбеки в стек викликів потоку потрапити не можуть. Якраз саме через це виклик функції по таймауту часто буває неточним за часом, оскільки функція не може потрапити з черги подій в стек потоку, поки в ньому виконується якась інша функція. Платформа Node.js знаходиться в фазі швидкого зростання, і багато хто розглядає її як переконливу альтернативу традиційним архітектурам веб-додатків.

Ключові слова: *веб-сервіс, Node.js, API, події, обробники подій, диспетчеризація, демультиплексор, движок V8.*

Вступ. Node.js - це середовище виконання JavaScript з відкритим початковим кодом, яке використовується, в основному, для написання веб-серверів, сайтів або RESTful (Representational State Transfer) API. Базова функціональність Node.js підтримується модулями, які базуються на API, спроектованому так, щоб зробити написання серверних додатків простішим. Node.js-додатки можна запускати на різних платформах: Linux, macOS і Windows; вони можуть бути написані на будь-якій мові, яка компілюється в JavaScript, таких, наприклад, як CoffeeScript, Dart або TypeScript. Node.js - це кросплатформне середовище виконання програм JavaScript з відкритим початковим кодом, засноване на движку Chrome V8 для JavaScript. Движок використовується для запуску коду JavaScript поза браузером для не клієнтських додатків.

Node.js добре підходить для створення додатків, які вимагають будь-якої форми взаємодії користувачів або спільної роботи в реальному часі - наприклад, сайтів-чатів або додатків, таких як CodeShare, в яких кілька користувачів можуть спостерігати в реальному часі за редагуванням документа кимось іншим. Node.js також добре підходить для створення API, які використовуються при обробці багатьох запитів, керованих введенням-виведенням (наприклад, які повинні виконувати операції з базою даних), або для сайтів, пов'язаних з потоковою передачею даних, оскільки Node.js дає можливість обробляти файли по ходу їх завантаження [1].

Модель функціонування багатопотокових серверів. Веб-додатки, розроблені у відповідності до традиційної клієнт-серверної архітектури [2], функціонують за такою схемою: клієнт запитує потрібний ресурс у сервера, а сервер знаходить і відправляє цей ресурс у відповідь, а потім розриває з'єднання. Обробка кожного запиту споживає комп'ютерні ресурси сервера (пам'ять, час CPU і т. д.). І тому, щоб обробляти наступні запити від клієнтів, сервер повинен звільняти ресурси, задіяні при обробці запитів, що завершились.

Коли сервер отримує новий запит, він створює окремий потік для його обробки. Потік - це час CPU і ресурси, які виділяються веб-сервером для послідовного виконання інструкцій обробки кожного запиту. Таким чином, сервер може обробляти кілька запитів одночасно, але тільки по одному на потік. Така модель обробки називатися моделлю потік-на-запит (thread-per-request model). Ця модель обробки графічно представлена на рис. 1.

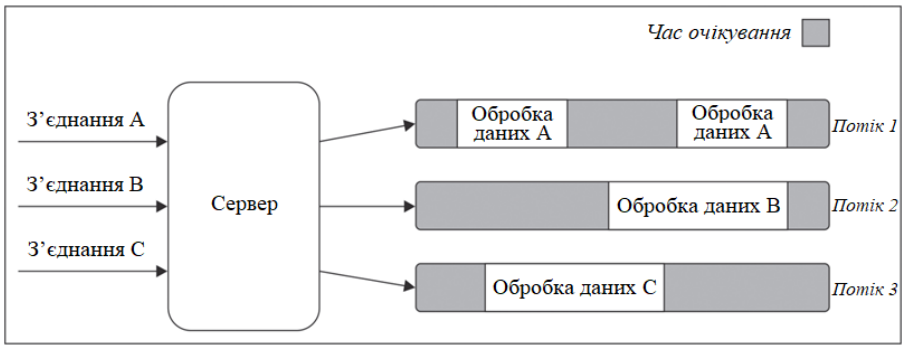

Рисунок 1 - Модель обробки потік-на-запит

Для обробки N запитів серверу потрібно N потоків. Якщо сервер отримує N + 1 запит, тоді він повинен чекати, поки один з потоків не стане доступним.

Як показано на рис. 1, сервер може обробляти до 4 запитів (потоків) одночасно, і коли він отримує наступні 3 запити, ці запити повинні чекати, поки будь-який з задіяних 4 потоків не стане доступним.

Один із способів позбавитися цих обмежень – це застосувати алгоритм демультиплексування подій.

Синхронне демультиплексування подій в Node.js. Багато сучасних операційних систем підтримують власний альтернативний механізм паралельної, неблокуючої роботи з ресурсами, включаючи і неблокуючі операції введення-виведення. Цей механізм називається синхронним демультиплексуванням подій або інтерфейсом сповіщення про події. Він реалізує наступне:

1. Приймає заявки на асинхронну роботу з ресурсами і додає їх в спеціальну структуру даних, пов'язуючи кожен з ресурсів з певною операцією і функцією зворотного виклику (обробником).

2. Механізм сповіщення про події демультиплексора налаштовується на стеження за набором затребуваних ресурсів.

3. Демультиплексор реагує на будь-яку подію, що відбувається при асинхронній роботі з затребуваними ресурсами. Обробники подій від цих ресурсів вишиковуються в чергу для послідовного синхронного виконання одним потоком. Після обробки всіх поточних подій основний потік блокується до приходу і доступності для обробки нових подій.

Як приклад, на рис. 2 представлена схема обробки однопоточним веб-сервером даних декількох з'єднань за допомогою синхронного демультиплексування подій.

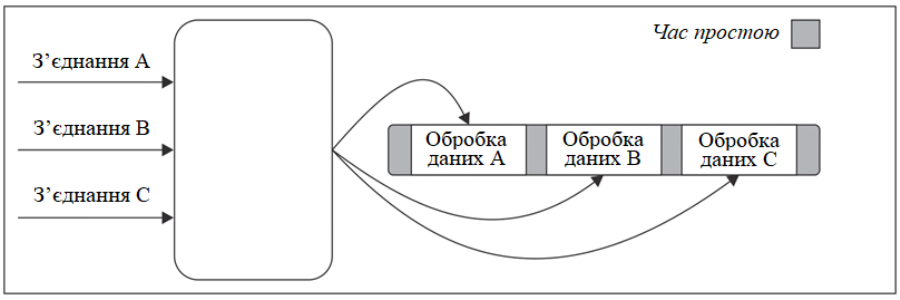

Рисунок 2 - Схема обробки даних декількох з'єднань за допомогою синхронного демультиплексування

Як видно з рисунка, використання одного-єдиного потоку не виключає можливості одночасного виконання декількох задач, пов'язаних з введенням-виведенням. Виконання задач розподілено за часом, а не розділене на кілька потоків. Явна перевага полягає в зведенні до мінімуму загального часу простою потоку, як це видно на схемі. Але це не єдина перевага даної моделі. Насправді, наявність тільки одного потоку також благотворно впливає на весь підхід до програмування паралельної обробки в цілому, оскільки дозволяє уникнути конкуренції і синхронізації декількох паралельно працюючих потоків.

Послідовність операцій при обробці подій. Цикл обробки подій в Node.js реалізується за участю libuv - багатоплатформної бібліотеки підтримки Node.js з наголосом на асинхронні операції, зокрема, асинхронне введення-виведення.

Цикл обробки подій (або введення-виведення, як він часто називається) є центральною частиною функціональності libuv [3]. У цьому циклі встановлюється послідовність виконання для всіх обробників подій, в тому числі і завершення операцій введення-виведення, з тим, щоб всі обробники виконувалися в одному потоці. Можна запустити кілька циклів подій, але так, щоб кожен працював в окремому потоці. Цикл обробки подій libuv (або будь-який інший API, що включає подібний цикл або управління з тим же змістом) не є потокобезпечним.

Запуск на виконання обробників з черги подій проводиться тільки після того, як стек викликів основного потоку додатка повністю очищається, тобто, тільки після завершення виконання основної програми додатка або

ж після закінчення виконання чергового обробника з черги подій. Тільки після цього фрейм наступної функції-обробника з черги подій поміщається в стек викликів основного потоку для виконання. Якраз саме через це виклик функції-обробника по таймауту часто буває неточним за часом, оскільки функція не може потрапити з черги подій в стек потоку, якщо в момент закінчення заданого інтервалу часу в стеку потоку знаходиться хоча б один фрейм іншої функції.

З урахуванням описаного раніше механізму демультиплексування подій сучасних ОС, роботу циклу обробки подій в додатку Node.js можна описати такою послідовністю дій згідно з номерами цих дій біля стрілок, наведених на рис. 3.

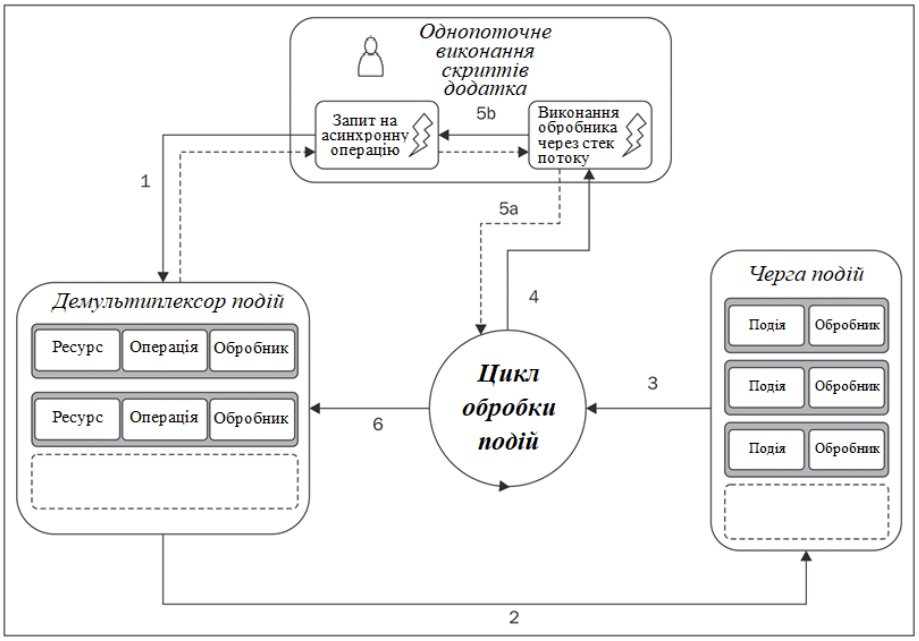

Рисунок 3 - Використання демультиплексора в циклі обробки подій

1. Додаток запитує нову асинхронну операцію, наприклад, операцію введення-виведення, передавши запит демультиплексору подій. При цьому додаток також задає функцію-обробника для його виклику після завершення операції. Відправлення нового запиту демультиплексору подій не призводить до блокування, управління негайно повертається додатку.

2. Після завершення обробки набору операцій введення-виведення або настання подій іншого типу демультиплексор подій додає нові події в чергу подій.

3. У цьому місці цикл обробки подій виконує обхід елементів черги подій.

4. Для кожної події по черзі викликається відповідний їй обробник.

5. Обробник, що є частиною коду додатка, повертає управління циклу обробки подій (5a). Однак під час виконання обробника можуть замовлятися нові асинхронні операції (5b), що призводить до додавання нових операцій в демультиплексор подій (1) ще до повернення управління циклу обробки подій.

6. Після обробки всіх елементів черги подій цикл знову блокується демультиплексором подій і відновлює свою роботу при появі нової події.

Додаток на платформі Node.js автоматично завершується, коли в демультиплексорі подій не залишиться відкладених операцій і подій в черзі.

Обробка черги подій. Коли Node.js-додаток запускається на виконання, після ініціалізації циклу обробки подій починає оброблятися наданий на вхід движка V8 код JavaScript, який може виконувати обчислення, виклики асинхронного API, встановлювати таймери або викликати методи setImmediate() або process.nextTick(). Потім починається обробка циклу подій [4].

Представлена на рис. 4 схема спрощено показує послідовність виконання операцій (окремих фаз) в циклі обробки подій.

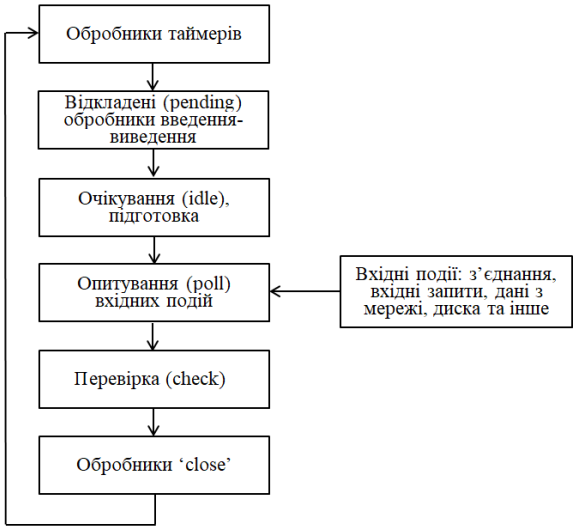

Рисунок 4 - Послідовність виконання операцій при обробці подій

Деякі фази формують свої FIFO-черги обробників (функцій зворотного виклику) для виконання. Хоча кожна фаза циклу є по-своєму особливою, зазвичай, коли цикл подій входить в дану фазу, він виконує специфічні операції, які стосуються цієї фази, а потім виконує обробники з черги цієї фази, поки черга не буде вичерпана, або не буде виконана максимальна кількість обробників, встановлених для цієї фази. Коли черга вичерпується або досягається максимальна кількість виконаних обробників, цикл обробки подій переміщується на наступну фазу і так далі.

Оскільки будь-яка з операцій кожної фази може запланувати на виконання інші операції, а нові події, оброблені в фазі опитування, поміщаються в чергу ядром ОС, нові події опитування можуть бути поставлені в чергу під час роботи в фазі опитування. В результаті, через тривале виконання функцій зворотних викликів фаза опитування може працювати набагато довше, ніж встановлений поріг по таймеру. Між кожною ітерацією циклу подій Node.js перевіряє, чи очікується завершення будь-яких асинхронних операцій введення-виведення або таймерів, і завершує роботу, якщо їх немає. Слід зазначити, що між реалізаціями циклу обробки для Windows і Unix/Linux існує невелика різниця. Крім того, насправді є сім чи вісім фаз обробки, але найбільш важливі з них, які фактично використовує Node.js, наведені на рис. 4 і детально описуються нижче.

Обробники таймерів. Дана фаза запускається інструкцією uv__run_timers(loop) бібліотеки libuv, яка виконується відразу ж після поновлення часу циклу інструкцією uv__update_time(loop) на початку кожної ітерації циклу обробки подій. На цій фазі виконуються обробники, встановлені методами setTimeout() та setInterval().

Таймер визначає інтервал часу, при закінченні якого може виконуватися наданий обробник, а не точний час, коли користувач хоче його виконати. Обробники таймерів запускаються на виконання після закінчення заданого інтервалу часу так швидко, як це можливо, однак планові роботи операційної системи або виконання інших обробників можуть затримати їх виконання.

Слід зазначити, що виконання таймерів управляється фазою опитування. Припустимо, наприклад, що встановлений таймаут для виконання коду 100 мс, після чого скрипт задає асинхронне читання файлу, яке займе 95 мс:

```
const fs = require('fs');
function someAsyncOperation(callback) {
        // Передбачається, що до завершення це
        // потребує 95мс
    fs.readFile('/path/to/file', callback);
}
const timeoutScheduled = Date.now();
setTimeout(() => {
    const delay = Date.now() - timeoutScheduled;
    console.log(`${delay}ms пройшло з тих пір, як було заплановано`);
}, 100);
// Викликается someAsyncOperation(), на
// завершення якої треба 95мс
someAsyncOperation(() => {
    const startCallback = Date.now();
    // Робиться щось, що займає 10мс ...
    while (Date.now() - startCallback < 10) {
        // Нічого не робиться
```

```
    }
});
```

Коли цикл обробки подій входить у фазу опитування, у нього є порожня черга (метод fs.readFile() ще не завершений), тому він буде очікувати протягом часу, що залишився до вичерпання інтервалу найкоротшого таймера. Поки цикл чекає, щоб пройшло 95 мс, метод fs.readFile() закінчить читання файлу, а його обробник, виконання якого займає 10 мс, буде доданий в чергу опитування і почне виконуватися. Коли виконання цього обробника закінчиться, в черзі опитування більше не буде обробників, так що цикл обробки подій визначить, що вичерпався інтервал найшвидшого таймера, і тому повернеться назад до фази таймерів для виконання обробника таймера. У цьому прикладі можна побачити, що загальна затримка між установкою таймера і виконанням його обробника становить 105 мс. Щоб запобігти надмірним затримкам циклу обробки подій на фазі опитування, бібліотека libuv має залежний від системи жорсткий часовий максимум, після закінчення якого вона припиняє прийняття нових подій на фазі опитування.

Відкладені обробники введення-виведення. На цій фазі виконуються функції зворотного виклику введення-виведення, відкладені до наступної ітерації циклу. Коли демультиплексор подій виконує запит на операцію читання будь-якого файлу, він ставить обробник завершення в чергу фази відкладених обробників введення-виведення. У цій фазі виконуються обробники для всіх операцій неблокуючого введення-виведення, тобто, це саме ті функції зворотного виклику, які виконуються після запиту до бази даних, після завершення операцій з файлами або іншими ресурсами.

На цій же фазі виконуються функції зворотного виклику для деяких системних операцій, наприклад, при обробці помилок TCP: якщо TCP-сокет отримує помилку ECONNREFUSED під час спроби підключення, деякі *nix системи хочуть дочекатися повідомлення про помилку. Такий запит буде поставлений в чергу для виконання в фазі відкладених обробників введення-виведення.

Очікування, підготовка. На цій фазі виконуються внутрішні операції циклу обробки подій. Так як метод process.nextTick() виконується на будь-якій фазі, обробник, заданий в цьому методі, може бути виконаний, в тому числі, і на цій фазі. Якогось іншого способу запуску коду програми на фазі «Очікування, підготовка» в Node.js немає.

Опитування вхідних подій. Фаза опитування має дві основні функції:

1. В залежності від результатів виконання інших фаз обчислити час роботи (таймаут) фази опитування або зовсім заблокувати її роботу в поточному проході циклу обробки подій.

2. Якщо фаза не блокується, обробити події в черзі опитування.

Коли цикл обробки подій входить у фазу опитування, і немає таймерів, які спрацювали, відбувається одне з двох:

2.1 Якщо черга опитування не порожня, цикл обробки подій буде ітерувати свою чергу обробників, виконуючи їх синхронно, поки не буде вичерпана черга або не буде досягнута системно-залежна межа.

2.2 Якщо черга опитування порожня, відбудеться ще раз одне з двох:

- якщо скрипти були заплановані за допомогою методу setImmediate(), цикл обробки подій завершить фазу опитування і перейде до фази перевірки для виконання таким чином запланованих скриптів;
- якщо скрипти не були заплановані за допомогою setImmediate(), цикл обробки подій буде очікувати на додавання обробників в чергу, а потім буде негайно їх виконувати.

Як тільки черга опитування стає порожньою, цикл подій перевіряє таймери, часові інтервали яких закінчилися. Якщо один або кілька інтервалів збігли, цикл обробки подій повертається до фази обробки таймерів для виконання обробників цих таймерів. Саме в фазі опитування виконується основний код додатка, написаного на JavaScript. Якщо сюди потрапить якась важка за обчисленнями операція, то на цьому етапі програма може просто зависнути і очікувати, поки не виконається ця довга операція.

Перевірка. Ця фаза дозволяє додатку виконувати обробники після завершення фази опитування. Якщо фаза опитування входить в стан очікування (becomes idle), а скрипти поміщаються в чергу за допомогою методу setImmediate(), обробка подій може тривати на фазі перевірки, а не очікувати фази опитування.

Метод setImmediate() насправді є спеціальним таймером, який запускається на виконання в окремій фазі циклу обробки подій. Він використовує API бібліотеки libuv, за допомогою якої обробники плануються для виконання після завершення фази опитування. Загалом, коли код виконується, цикл обробки подій в кінцевому підсумку потрапляє на фазу опитування, де він очікує вхідні з'єднання, запити та інше. Однак, якщо обробник був запланований за допомогою методу setImmediate(), а фаза опитування перейшла в стан очікування, цикл обробки подій закінчить цю фазу і перейде до фази перевірки, і не буде чекати подій опитування.

Обробники 'close'. Якщо сокет або дескриптор був раптово закритий (наприклад, за допомогою socket.destroy()), на цій фазі буде виконаний обробник події «close». Інакше він буде виконаний за допомогою методу process.nextTick().

Висновки. У той час, коли в багатьох програмних платформах потоки широко використовуються для масштабування додатків з метою максимального завантаження CPU, щоб компенсувати періоди очікування блокуючих операцій введення-виведення, Node.js уникає потоків через їх внутрішню складність, а також через великі накладні витрати при безкінечних переключеннях між ними. Вважається, що в однопотокових архітектурах, керованих подіями, споживаний обсяг пам'яті істотно нижче, загальна пропускна здатність вище, реактивність системи під навантаженням краще, а модель програмування простіше. Платформа Node.js

знаходиться в фазі швидкого зростання, і багато хто розглядає її як переконливу альтернативу традиційним архітектурам веб-додатків, що використовують Java, PHP, Python або Ruby on Rails.

Література

1. Griggs B. Node Cookbook. Fourth Edition / B. Griggs. - Birmingham : Packt Publishing Ltd., 2020. - 512 p.
2. Casciaro M. Node.js Design Patterns, Third Edition / M. Casciaro, L. Mammino. - Birmingham : Packt Publishing Ltd., 2020. - 661 p.
3. Node.js v15.10.0 Documentation [Електронний ресурс] / Node.js . – Режим доступу: https://nodejs.org/dist/latest-v15.x/docs/api/
4. Bojinov V. Node.js Complete Reference Guide / V. Bojinov, D. Herron, D. Resende. – Birmingham : Packt Publishing Ltd., 2018. - 1184 p.

References

1. Griggs B. Node Cookbook. Fourth Edition / B. Griggs. - Birmingham : Packt Publishing Ltd., 2020. - 512 p.
2. Casciaro M. Node.js Design Patterns, Third Edition / M. Casciaro, L. Mammino. - Birmingham : Packt Publishing Ltd., 2020. - 661 p.
3. Node.js v15.10.0 Documentation [Electronic resource] / Node.js. - Access mode: https://nodejs.org/dist/latest-v15.x/docs/api/
4. Bojinov V. Node.js Complete Reference Guide / V. Bojinov, D. Herron, D. Resende. – Birmingham : Packt Publishing Ltd., 2018. - 1184 p.

В традиционной клиент-серверной архитектуре один поток не может обрабатывать несколько запросов одновременно из-за блокирующих операций ввода-вывода. Операции работы с сетью, а также чтения и записи на диск слишком медленные по сравнению со скоростью выполнения инструкций процессором компьютера. Поэтому диспетчер программ Node.js использует ориентированную на события модель и неблокирующую архитектуру ввода-вывода, что делает его легким и эффективным за счет следующего: главный поток не блокируется операциями ввода-вывода; сервер во время операций ввода-вывода продолжает обслуживать другие запросы; сервер разрабатывается как набор асинхронных программ, управляемых событиями, и не обязательно потокобезопасных. Цикл обработки событий, который является центральной частью диспетчера Node.js, реализуется в соответствии с обычным однопоточным подходом к асинхронному вводу-выводу. Когда циклу обработки событий, работающему в основном потоке приложения, нужно выполнить операцию ввода-вывода, для ее асинхронного выполнения он использует другой поток из пула потоков, а когда выполнение операции завершается, функция обратного вызова (колбек) ставится в очередь событий для последовательной обработки основным потоком. Функции из очереди событий (колбеки) выполняются только после того, как стек вызовов основного потока будет полностью очищен. Только после этого функции из очереди событий помещаются в стек вызовов основного потока для выполнения. Если в стеке вызовов основного потока в данный момент находится хотя бы один элемент, то колбеки в стек вызовов потока попасть не могут. Как раз именно из-за этого вызов функции по таймауту часто бывает неточным по времени, поскольку функция не может попасть из очереди событий в стек потока, пока в нем выполняется какая-то другая функция. Платформа Node.js находится в фазе быстрого роста, и многие рассматривают ее как убедительную альтернативу традиционным архитектурам веб-приложений.
Ключевые слова: *веб-сервис, Node.js, API, события, обработчики событий, диспетчеризация, демультиплексор, движок V8.*

In a traditional client-server architecture, a single thread cannot process multiple requests at the same time due to blocking I/O operations. Networking, reading and writing to disk are too slow compared to the speed at which the computer's processor can execute instructions. Therefore, the Node.js program manager uses an event-driven model and non-blocking I/O architecture, making it lightweight and efficient, because main thread is not blocked by I/O; server continues to serve other requests during I/O operations; server is designed as a set of event-driven asynchronous programs, and not necessarily thread safe. The event loop, which is the central part of the Node.js dispatcher, follows the usual single-threaded approach to asynchronous I/O. When the event loop, running in the main thread of the application, needs to perform an I/O operation, it uses another thread from the thread pool to execute it asynchronously, and when the operation completes, the callback function is queued for sequential processing by the main stream. Functions from the event queue (callbacks) are executed only after the call stack of the main thread has been completely cleared. Only after that functions from the event queue pushed onto the call stack of the main thread for execution. If, at least, one element is currently in the call stack of the main thread, then callbacks cannot get onto the call stack of the thread. Because of it function call by timeout is often inaccurate in time, because a function cannot get from the event queue onto the thread's stack while some other function is running on it. The Node.js platform is in a phase of rapid growth and is viewed by many as a compelling alternative to traditional web application architectures.
Keywords: *web service, Node.js, API, events, event handlers, dispatching, demultiplexer, V8 engine.*

Щербаков Є.В. – канд. тех. наук, доцент, доцент кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету ім. В. Даля, e-mail: gkvarc@gmail.com

Щербакова М.Є. – канд. тех. наук, доцент, доцент кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету ім. В. Даля, e-mail: m.shcherbakova432@gmail.com