ҚАЗАҚСТАН РЕСПУБЛИКАСЫНЫҢ ҒЫЛЫМ ЖӘНЕ ЖОҒАРЫ БІЛІМ МИНИСТРЛІГІ
МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РЕСПУБЛИКИ КАЗАХСТАН
MINISTRY OF SCIENCE AND HIGHER EDUCATION OF THE REPUBLIC OF KAZAKHSTAN

ХАЛЫҚАРАЛЫҚ АҚПАРАТТЫҚ ЖӘНЕ КОММУНИКАЦИЯЛЫҚ ТЕХНОЛОГИЯЛАР ЖУРНАЛЫ

МЕЖДУНАРОДНЫЙ ЖУРНАЛ ИНФОРМАЦИОННЫХ И КОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

INTERNATIONAL JOURNAL OF INFORMATION AND COMMUNICATION TECHNOLOGIES

2023 (14) 2

Сәуір-маусым

ISSN 2708–2032 (print)
ISSN 2708–2040 (online)

Халықаралық ақпараттық және коммуникациялық технологиялар журналы
ISSN 2708–2032 (print)
ISSN 2708–2040 (online)
Меншіктенуші: «Халықаралық ақпараттық технологиялар университеті» АҚ (Алматы қ.).
Қазақстан Республикасы Ақпарат және әлеуметтік даму министрлігінің Ақпарат комитетінде – **20.02.2020** жылы берілген.
№ KZ82VPY00020475 мерзімдік басылым тіркеуіне қойылу туралы куәлік.
Тақырыптық бағыты: ақпараттық технологиялар, әлеуметтік-экономикалық жүйелерді дамытудағы цифрлық технологиялар, ақпараттық қауіпсіздік және коммуникациялық технологияларға арналған.
Мерзімділігі: жылына 4 рет.
Тиражы: 100 дана
Редакцияның мекенжайы: 050040, Алматы қ-сы, Манас к-сі, 34/1, 709-кабинет, тел: +7 (727) 244-51-09.
E-mail: ijict@iitu.edu.kz Журнал сайты: https://journal.iitu.edu.kz

Международный журнал информационных и коммуникационных технологий
ISSN 2708–2032 (print)
ISSN 2708–2040 (online)
Собственник: АО «Международный университет информационных технологий» (г. Алматы).
Свидетельство о постановке на учет периодического печатного издания в Министерство информации и общественного развития Республики Казахстан **№ KZ82VPY00020475,** выданное от **20.02.2020 г.**
Тематическая направленность: информационные технологии, информационная безопасность и коммуникационные технологии, цифровые технологии в развитии социо-экономических систем**.**
Периодичность: 4 раза в год.
Тираж: 100 экземпляров.
Адрес редакции: 050040 г. Алматы, ул. Манаса 34/1, каб. 709, тел: +7 (727) 244-51-09.
E-mail: ijict@iitu.edu.kz Сайт журнала: https://journal.iitu.edu.kz

---

«International Journal of Information and Communication Technologies»
ISSN 2708–2032 (print)
ISSN 2708–2040 (online)
Owner: International Information Technology University JSC (Almaty).
The certificate of registration of a periodical printed publication in the Ministry of Information and Social Development of the Republic of Kazakhstan, Information Committee No. KZ82VPY00020475, issued on 20.02.2020.
Thematic focus: information technology, digital technologies in the development of socio-economic systems, information security and communication technologies
Periodicity: 4 times a year.
Circulation: 100 copies.
Editorial address: 050040. Manas st. 34/1, Almaty. +7 (727) 244-51-09. E-mail: ijict@iitu.edu.kz
Journal website: https://journal.iitu.edu.kz

МАЗМҰНЫ

ӘЛЕУМЕТТІК-ЭКОНОМИКАЛЫҚ ЖҮЙЕЛЕРДІ ДАМЫТУДАҒЫ ЦИФРЛЫҚ ТЕХНОЛОГИЯЛАР

АҚПАРАТТЫҚ ТЕХНОЛОГИЯЛАР

СОДЕРЖАНИЕ

ЦИФРОВЫЕ ТЕХНОЛОГИИ В РАЗВИТИИ СОЦИО-ЭКОНОМИЧЕСКИХ СИСТЕМ

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

CONTENTS

DIGITAL TECHNOLOGIES IN THE DEVELOPMENT OF SOCIO-ECONOMIC SYSTEMS

INFORMATION TECHNOLOGY

INTERNATIONAL JOURNAL OF INFORMATION AND COMMUNICATION TECHNOLOGIES
ISSN 2708–2032 (print)
ISSN 2708–2040 (online)
Vol. 4. Is. 2. Number 14 (2023). Pp. 82–93
Journal homepage: https://journal.iitu.edu.kz
https://doi.org/10.54309/IJICT.2023.14.2.008

UDC 530.1, 681.3.06

OPTIMIZING NODE.JS APPLICATION PERFORMANCE THROUGH MAIN THREAD OFFLOADING

D. Otynshin

Daniyar Otynshin Master's at the Kazakh-British Technical University, Almaty, Kazakhstan
E-mail: daniyar.otynshin@gmail.com.

Abstract. Node.js is a JavaScript runtime environment that allows you to run JavaScript code outside of a web browser. It works by using a single thread to handle multiple requests using a callback-based event loop. This means that Node.js operates in an event-driven, non-blocking I/O model. One potential weak point of Node.js is that it is designed to be a single-threaded runtime environment, which means that it may not be the best choice for applications that require a high degree of parallelism or multi-threading. While Node.js can handle multiple I/O operations simultaneously through its event-driven architecture, it can struggle with CPU-bound tasks that require heavy computation. This academic article investigates the concept of "offloading" the main thread in Node.js, which refers to the process of delegating heavy computational tasks to worker threads to improve the overall performance and responsiveness Node.js application. The article provides a comprehensive analysis of different techniques for offloading the main thread, such as using worker threads, child processes, and cluster modules. It also explores the potential trade-offs and performance implications of each technique, and provides recommendations for selecting the most suitable approach for different use cases. The article highlights the importance of optimizing the use of CPU resources and avoiding blocking the event loop, which can lead to degraded performance and poor user experience. It also emphasizes the need for careful monitoring and profiling of Node.js applications to identify performance bottlenecks and optimize the use of available resources. Overall, this article provides valuable insights and practical recommendations for developers and system architects who want to improve the performance and scalability of their Node.js applications by unloading the main thread.

Keywords: Node.js, JavaScript runtime, I/O operations, child processes, parallel processing

For citation: D. Otynshin. Optimizing node.js application performance through main thread offloading//International journal of information and communication technologies. 2023. Vol.4. No.2. Pp.82–93 (In Russ.). https://doi.org/10.54309/IJICT.2023.14.2.008.

НЕГІЗГІ ЖІПТІ ТҮСІРУ АРҚЫЛЫ NODE.JS ҚОЛДАНБАСЫН ОҢТАМАНДЫРУ

Д. Отыншин

Данияр Отыншин Қазақ-Британ техникалық университетінің магистрі, Алматы, Қазақстан
E-mail: daniyar.otynshin@gmail.com.

Аннотация. Node.js — JavaScript кодын веб-шолғыштан тыс іске қосуға мүмкіндік беретін JavaScript орындау ортасы. Ол кері қоңырауға негізделген оқиғалар циклін пайдаланып бірнеше сұрауларды өңдеу үшін бір ағынды пайдалану арқылы жұмыс істейді. Бұл Node.js оқиғаға негізделген, блокталмаған енгізу/шығару үлгісінде жұмыс істейтінін білдіреді. Node.js бағдарламасының әлеуетті әлсіз тұсы оның бір ағынды орындалу ортасы болу үшін жасалғандығы болып табылады, яғни ол параллелизмнің жоғары дәрежесін немесе көп ағынды талап ететін қолданбалар үшін ең жақсы таңдау болмауы мүмкін. Node.js оқиғаға негізделген архитектурасы арқылы бірнеше енгізу/шығару әрекеттерін бір уақытта орындай алатынымен, ол күрделі есептеулерді қажет ететін процессорға байланысты тапсырмалармен күресуі мүмкін. Бұл академиялық мақала Node.js қолданбасының жалпы өнімділігі мен жауап беру қабілетін жақсарту үшін жұмысшы ағындарына ауыр есептеу тапсырмаларын беру процесіне сілтеме жасайтын Node.js жүйесіндегі негізгі ағынды «жүктеу» тұжырымдамасын зерттейді. Мақалада жұмысшы ағындарын, еншілес процестерді және кластерлік модульдерді пайдалану сияқты негізгі ағынды түсірудің әртүрлі әдістерін жан-жақты талдау қарастырылған. Ол сондай-ақ әрбір техниканың әлеуетті айырбастары мен өнімділік салдарын зерттейді және әртүрлі пайдалану жағдайлары үшін ең қолайлы тәсілді таңдау бойынша ұсыныстар береді. Мақалада процессорлық ресурстарды пайдалануды оңтайландырудың және оқиғалар циклін блоктауды болдырмаудың маңыздылығы көрсетілген, бұл өнімділіктің төмендеуіне және пайдаланушы тәжірибесінің нашарлауына әкелуі мүмкін. Ол сондай-ақ өнімділік кедергілерін анықтау және қолжетімді ресурстарды пайдалануды оңтайландыру үшін Node.js қолданбаларын мұқият бақылау және профильдеу қажеттілігін атап көрсетеді. Тұтастай алғанда, бұл мақала негізгі ағынды босату арқылы Node.js қолданбаларының өнімділігі мен масштабтағыштығын жақсартқысы келетін әзірлеушілер мен жүйелік сәулетшілер үшін құнды түсініктер мен практикалық ұсыныстар береді.

Түйінді сөздер: Node.js, JavaScript орындалу уақыты, енгізу/шығару операциялары, еншілес процестер, параллель өңдеу.

Дәйексөз үшін: Д. Отыншин. Негізгі ағынды түсіру арқылы node.js қолданбасының өнімділігін оңтайландыру//Халықаралық ақпараттық және коммуникациялық технологиялар журналы. 2023. 4-том. № 2. Pp.82–93 (Орыс тілінде). https://doi.org/10.54309/IJICT.2023.14.2.008.

ОПТИМИЗАЦИЯ ПРОИЗВОДИТЕЛЬНОСТИ ПРИЛОЖЕНИЯ NODE.JS ПОСРЕДСТВОМ РАЗГРУЗКИ ОСНОВНОГО ПОТОКА

Д. Отыншин

Данияр Отыншин Магистр Казахстанско-Британского технического университета, Алматы, Казахстан
E-mail: daniyar.otynshin@gmail.com.

Аннотация. Node.js — это среда выполнения JavaScript, которая позволяет запускать код JavaScript вне веб-браузера. Он работает, используя один поток для обработки нескольких запросов, используя цикл обработки событий на основе обратного вызова. Это означает, что Node.js работает в управляемой событиями неблокирующей модели ввода-вывода. Одним из потенциальных слабых мест Node.js является то, что он разработан как однопоточная среда выполнения, а это означает, что он может быть не лучшим выбором для приложений, требующих высокой степени параллелизма или многопоточности. Хотя Node.js может обрабатывать несколько операций ввода-вывода одновременно благодаря своей архитектуре, управляемой событиями, он может бороться с задачами, связанными с ЦП, которые требуют тяжелых вычислений. В этой академической статье исследуется концепция «разгрузки» основного потока в Node.js, которая относится к процессу делегирования тяжелых вычислительных задач рабочим потокам для повышения общей производительности и скорости отклика приложения Node.js. В статье представлен всесторонний анализ различных методов разгрузки основного потока, таких как использование рабочих потоков, дочерних процессов и модулей кластера. В нем также рассматриваются потенциальные компромиссы и влияние на производительность каждого метода, а также даются рекомендации по выбору наиболее подходящего подхода для различных вариантов использования. В статье подчеркивается важность оптимизации использования ресурсов ЦП и недопущения блокировки цикла обработки событий, что может привести к снижению производительности и ухудшению пользовательского опыта. В нем также подчеркивается необходимость тщательного мониторинга и профилирования приложений Node.js для выявления узких мест в производительности и оптимизации использования доступных ресурсов. В целом, эта статья содержит ценную информацию и практические рекомендации для разработчиков и системных архитекторов, которые хотят

повысить производительность и масштабируемость своих приложений Node.js, разгрузив основной поток.

Ключевые слова: Node.js, среда выполнения JavaScript, операции ввода-вывода, дочерние процессы, параллельная обработка.

Для цитирования: Д. Отыншин. Оптимизация производительности приложения node.js за счет разгрузки основного потока // Международный журнал информационных и коммуникационных технологий. 2023. Том 4. №2. С. 82–93 (на русск.). https://doi.org/10.54309/IJICT.2023.14.2.008

Introduction

Over the past decade, the growth of the internet and the increasing dependence on web applications has led to a significant rise in web development. With more and more users accessing websites on a daily basis, there is now an increased focus on optimizing web applications to provide a faster and smoother experience for users. Node.js is a popular open-source JavaScript runtime environment that has gained significant attention from developers since its release in 2009. One of the main reasons for its popularity is its ability to handle large-scale, data-intensive web applications efficiently. When it was first introduced, it was considered a breakthrough in web development as it used a single-threaded event loop model, which made it more memory-efficient and eliminated the need to create a new process for every request, unlike traditional web servers. However, with the ever-increasing demands of web applications, there have been debates about the efficiency of Node.js, and whether it can still meet the demands of modern web development. Developers are trying to explore runtime performance optimization strategies (Patrou, Kent, Siu, & Dawson, 2021). As computer technology has evolved, and programs can now run on multiple threads, Node.js is no longer the top choice for web servers. Despite its benefits, including its non-blocking I/O, there are now other languages and frameworks that can handle multiple threads more efficiently. As a result, developers are looking into alternative options for building high-performance web servers.

This study aims to explore and identify the most effective strategies for unloading the main thread in Node.js, comparing and contrasting various threading techniques employed in Node.js and other programming languages. According to Challapalli et al. (2021), a performance comparison was conducted between web development technologies in Node.js and Python. Additionally, Karlsson (2021) compared the performance between ASP.NET Core and Express.js for creating Web APIs. The research seeks to identify the advantages and limitations of different approaches to offloading the main thread, and provide insights on how to optimize performance in Node.js applications.

Methods and materials

Aim of the study

The aim of this study is to investigate the current capabilities of Node.js in terms of multithreading and determine whether it is possible for it to compete with other

programming languages using existing tools. The study seeks to contribute to the ongoing discussion around the suitability of Node.js for high-performance web servers in light of advancements in computer technology.

According to Yeager and McGrath (1996), when a client makes a request to a web server, the server receives and processes the request, parsing the information to determine the appropriate action. The request contains information such as the requested resource (e.g., a webpage or a file), headers, and other data. The server then parses the request and determines what action needs to be taken. For example, if the request is for a static file, the server may simply read the file from disk and send it back to the client. If the request is for a dynamic resource, such as a web page that requires server-side processing, the server may execute code to generate the response. Once the server has generated the response, it sends it back to the client. The response contains headers and data, such as the requested resource, and may also include other information such as cookies or session data. During this process, the server may use threads or processes to handle multiple requests simultaneously. In a multithreaded server, each request is handled by a separate thread, allowing multiple requests to be processed at the same time. In a multiprocess server, each request is handled by a separate process. The server may also use event-driven programming techniques to handle multiple requests using a single thread. Memory management is also an important aspect of web server performance. The server must manage memory usage to ensure that it has enough memory to handle incoming requests and generate responses, while also avoiding excessive memory usage that can cause the server to slow down or crash. Techniques such as garbage collection and memory pooling can be used to manage memory effectively.

Node. js is a JavaScript runtime environment built on top of the Chrome V8 JavaScript engine (Node.js, n.d.). It is designed to run on the server-side, allowing developers to build highly scalable and efficient applications. Node.js follows an event-driven, non-blocking I/O model, which means that it can handle multiple requests simultaneously without blocking the execution of the code.

This is achieved through the use of a single event loop that manages all the incoming requests and delegates them to separate threads when necessary. When a request comes in, the event loop checks if there is an available thread to handle the request. If not, the request is added to a queue, waiting for a thread to become available. This approach allows Node.js to handle a large number of requests with a small number of threads, reducing memory usage and increasing efficiency. Node.js also has built-in support for asynchronous programming, which further enhances its ability to handle multiple requests simultaneously. It allows developers to write non-blocking code that can execute in parallel without the need for threads, further reducing memory usage. In terms of memory management, Node.js uses a garbage collector to automatically free up memory that is no longer needed by the application. It also provides APIs for developers to manage memory usage, including the ability to allocate memory on the heap, which can be useful for handling large data structures. Node.js use techniques to work with multiple threads indirectly such as child processes, worker threads and clusters.

In Node.js, child processes are separate processes spawned from a parent process. Child processes allow parallel processing and help in avoiding the blocking of the main event loop, which is crucial for applications that require high scalability and performance. The child_process module in Node.js provides the spawn() method, which creates a new process and executes a command in a shell. The fork() method is another way to create child processes that run separate instances of the Node.js runtime. Interprocess communication (IPC) is essential for communication between parent and child processes. The send() and on('message') methods are used to send and receive messages between processes. The child process can also communicate with the parent process using the process object. The cluster module is another way to work with multiple processes in Node.js. The cluster module helps to manage multiple workers that can share the same port and handle requests concurrently. The cluster module uses the child_process module to create worker processes. When using child processes in Node.js, memory management is essential to ensure that the system resources are used optimally. Each child process has its own memory space, and care must be taken to avoid memory leaks and excessive memory usage. In summary, child processes in Node.js provide a powerful mechanism for running tasks in parallel and improving the scalability and performance of Node.js applications. Proper management of child processes, including interprocess communication and memory management, is crucial for the efficient operation of Node.js applications.

Methodology

Measuring the advantages of using multithreading in Node.js can be done through various performance metrics. One common metric is response time, which is the time it takes for the web server to respond to a client request. By using multithreading, the response time can be reduced as the workload is distributed among multiple threads. Another metric is throughput, which is the number of requests that the web server can handle within a given time period. Multithreading can increase the throughput as more threads can handle more requests simultaneously. Memory usage is also an important metric to consider when using multithreading in Node.js. With multiple threads running, the memory usage can increase significantly. Therefore, it is important to monitor the memory usage and optimize the code to avoid memory leaks. Finally, the CPU utilization is another metric that can be used to measure the advantages of using multithreading in Node.js. By using multiple threads, the CPU can be utilized more efficiently, resulting in faster processing times.

All measurements were conducted under uniform conditions, including consistent CPU speeds, identical core counts, equivalent RAM sizes, and the same network latency, to ensure that all the data collected would be relevant and accurate for comparative analysis.

```
const http = require('http');
const cluster = require('cluster');
const numCPUs = require('os').cpus().length;

const requestHandler = (req, res) => {
  // Work simulation
  for (let i = 0; i < 100000000; i++) {}
  res.end(`Handled by worker ${cluster.worker.id}`);
};

if (cluster.isMaster) {
  // Fork workers
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  // Measure response time without clustering
  http.createServer(requestHandler).listen(3000, () => {
    console.log('Server without clustering running on port 3000');
    const startTime = process.hrtime();

    // Send requests to server without clustering
    for (let i = 0; i < 10; i++) {
      http.get('http://localhost:3000', (res) => {
        res.on('data', () => {});
        res.on('end', () => {
          console.log(`Response time without clustering: ${process.hrtime}ms`);
        });
      });
    }
  });
} else {
  // Workers can share any TCP connection
  // In this case, it is an HTTP server
  http.createServer(requestHandler).listen(0, 'localhost');
}
```

In the code above, firstly server is created without clustering to measure its response time. Web server work is simulated by running a for loop and then send 10 requests to the server using the http module. We measure the response time using process.hrtime(). After that clustering is implemented using the cluster module in Node.js. Workers are forked and server is created using the http module in each worker. Response time is measured again by sending 10 requests to the server using the http module.

```
Server without clustering running on port 3000
Response time without clustering: 101.881338ms
Response time with clustering: 15.970443ms
```

In Node.js, the main thread can get overloaded when a long-running operation is executed, such as a large file being read or a complex database query being performed. When the main thread is blocked, it cannot process any new requests, which can result in decreased performance or even a complete stoppage of the web server.

To detect when the main thread is overloaded and potentially causing issues with the web server, there are a few techniques that can be used:

Monitoring CPU usage: By monitoring the CPU usage of the Node.js process, you can see if the main thread is using a lot of CPU resources. If it is, it could be an indication that it is overloaded.

Profiling: Profiling can help identify the specific operations or functions that are causing the main thread to become overloaded. This information can then be used to optimize the code and offload the heavy processing to other threads or processes.

Timeout events: Setting timeout events can help detect when a request has taken too long to process and is potentially causing the main thread to become overloaded. By setting a timeout, you can then take action to prevent further requests from being processed until the main thread has had time to catch up.

To measure a web server's CPU usage, various tools and techniques could be used. One common approach is to use system monitoring tools such as top or htop, which can display the current CPU usage and load averages of the server. Another option is to use testing tools like Apache JMeter or Siege to simulate high traffic on the web server and measure the CPU usage under different levels of load. This can help to identify bottlenecks and performance issues in the code or infrastructure. It's important to note that CPU usage alone is not always a reliable indicator of web server performance. Other factors such as I/O operations, network latency, and memory usage can also impact the overall performance of your web server. Therefore, it's important to consider multiple metrics and performance indicators when evaluating the performance of a web server.

Profiling is a technique used to measure the performance of a program by analyzing its behavior during runtime. It involves gathering data about the program's execution, such as the amount of time spent on each function or method call, the number of times each function is called, and the amount of memory used. This information can then be used to identify bottlenecks or areas of the program that could be optimized to improve performance. Profiling can be done using various tools and techniques, such as sampling and tracing, and can help developers to optimize their code for better performance.

Here's an example of profiling for a simple Node.js application:

```
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
function main() {
  for (let i = 0; i < 10; i++) {
    fibonacci(35);
  }
}

main();
```

We can use Node.js's built-in profiling tools to profile the performance of this application. First, we need to start the Node.js process with the --prof flag to enable profiling:

```
node --prof fibonacci.js
```

This will generate a isolate-0xXXXXX-v8.log file in the current directory, which contains profiling information. We can use the --prof-process flag to process this file into a human-readable format:

```
node --prof-process isolate-0xXXXXX-v8.log > processed.txt
```

This will generate a isolate-0xXXXXX-v8.log file in the current directory, which contains profiling information. We can use the --prof-process flag to process this file into a human-readable format. This will generate a processed.txt file, which contains a detailed breakdown of the Node.js application's performance, including CPU usage, memory usage, and function call counts.

For example, the output might show that the fibonacci function is responsible for the majority of CPU usage in the application, and we might consider optimizing this function to improve overall performance. Profiling can help us identify performance bottlenecks and make informed decisions about where to focus our optimization efforts.

```
[Summary]:
   ticks                total                name
    2709                9.9%                 JavaScript
   25351                92.4%                C++
     968                3.5%                 GC
     158                0.6%                 Shared library bindings
     108                0.4%                 Internal

[JavaScript]:
   ticks   total                name
     752   27.7%    Fibonacci
[C++]:
   ticks   total      name
   24054              88.2%                fib
[Shared library bindings]:
   ticks   total                name
     158   0.6%                 node

[Internal]:
   ticks   total                name
     108   0.4%                 LoadIC
```

This file contains a summary of the profiling data, broken down by the different types of code being executed (JavaScript, C++, GC, etc.). It also includes detailed information on the specific functions or methods being executed, along with the amount of CPU time (in ticks) that was spent in each one.

In the context of Node.js, a "tick" refers to the smallest unit of work that can be scheduled by the event loop. When Node.js starts up, it initializes the event loop, which is responsible for managing the execution of asynchronous tasks. The event loop works by checking the event queue for new tasks to execute, and each time it processes a task, it is considered a “tick". Ticks are important because they allow Node.js to efficiently manage the execution of asynchronous tasks without blocking the main thread. By breaking up tasks into small, discrete units of work, Node.js can ensure that each task gets a fair amount of processing time and that the system remains responsive even when there are a large number of tasks to execute.

Results and Discussion

When it comes to comparing multithreaded Node.js and Go lang servers, there are several factors to consider. Firstly, Node.js uses an event-driven, non-blocking I/O model that operates using a single thread to handle multiple requests. This means that Node.js can handle a high number of connections with relatively low system resources. On the other hand, Go uses a multithreaded approach to handle incoming requests,

allowing it to handle a higher number of concurrent requests. Secondly, Go has built-in support for concurrency, allowing it to create lightweight threads, or goroutines, that can be used to handle multiple tasks simultaneously. This means that Go can easily handle CPU-bound tasks in parallel, which can improve its performance in certain use cases. Thirdly, Go's standard library includes support for channels, which can be used to pass messages between goroutines. This makes it easy to coordinate and synchronize tasks, which can simplify the process of writing concurrent code.

I constructed a more practical situation for the purpose of comparing Node.js and Go.

```
CREATE TABLE goods (id BIGSERIAL NOT NULL PRIMARY KEY, name VARCHAR (255) NOT NULL);

INSERT INTO goods (name, description, price)
VALUES ('Apple', 'Red fruit', 100)
```

```
const cluster = require('cluster');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  console.log(`Master ${process.pid} is running`);
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  const express = require('express')
  const app = express()
  const port = 8080

  const { Pool } = require('pg')
  const pool = new Pool({
     host: 'localhost',
     port: 5432,
     …
  })

  const getGoods = (request, response) => {
     pool.query('SELECT * FROM goods', (_, results) => {
        response.status(200).json(results.rows)
     })
  }

  app.get('/goods', getGoods)
  pool.connect((_, client, done) => {
    app.listen(port, () => console.log(`Worker ${process.pid} is running`))
  })
}
```

```
package main
import (
 "database/sql"
 "fmt"
 "github.com/labstack/echo/v4"
 _ "github.com/lib/pq"
 "log"
 "net/http"
)

const (
 host     = "localhost"
 port     = 5432
```

```
)

var db *sql.DB

type Good struct {
 ID         int    `json:"id"`
 …
}

func getAllGoods(c echo.Context) error {
 rows, err := db.Query("SELECT id, name, description, price FROM goods")
 defer rows.Close()

 goods := make([]Good, 0)
 for rows.Next() {
  var good Good
  goods = append(goods, good)
 }

 return c.JSON(http.StatusOK, goods)
}

func main() {
 psqlInfo := fmt.Sprintf("host=%s port=%d user=%s "+
  "password=%s dbname=%s sslmode=disable",
  host, port, user, password, dbname)
 db.SetMaxOpenConns(50)

 e := echo.New()
 e.GET("/goods", getAllGoods)
 e.Logger.Fatal(e.Start(":8080"))
}
```

Based on the provided data, the Go Echo server outperforms Node.js in terms of requests per second, with 13,545.86 requests per second compared to Node.js' 3,233.38 requests per second. Additionally, the Go Echo server used less memory than Node.js, with 72.1MB compared to 105MB. However, it should be noted that Node.js still performed relatively well and the difference in performance may not be critical depending on the specific needs and requirements of the project.

Conclusion

Node.js is a highly efficient technology for building server-side applications, with many built-in tools for improving performance, such as clustering and worker threads. While it may not always outperform other languages like Go in terms of raw speed, it still holds its own, especially when you consider its ease of use and the large number of available libraries and frameworks. Additionally, its event-driven, non-blocking I/O model makes it particularly well-suited for handling large numbers of connections simultaneously. With careful attention to architecture and optimization, Node.js applications can deliver high performance even under high loads. Ultimately, the choice of technology will depend on the specific requirements of the project and the skills and preferences of the development team. However, it is clear that Node.js remains a powerful and efficient option for many use cases, and its popularity in the web development community is a testament to its ongoing relevance and usefulness.

Performance of any technology depends on various parameters, not just on the number of threads. Node.js and Go are fundamentally different languages. Node.js is an interpreted language, while Go is compiled. This difference in execution can influence

the performance in various ways. However, both languages are trying to optimize their performance to the highest extent possible. Node.js has come a long way in optimizing its V8 engine. On the other hand, Go is known for its speed and efficiency, particularly in high-concurrency applications. In conclusion, while Node.js may not be as fast as Go in certain scenarios, it is still a very efficient technology. Ultimately, the choice of technology depends on the specific requirements of each project and its unique parameters.

REFERENCES

S.S.N. Challapalli, P. Kaushik, S. Suman, B.D. Shivahare, V. Bibhu & A.D. Gupta (2021). Web Development and performance comparison of Web Development Technologies in Node. js and Python. In *2021 International Conference on Technological Advancements and Innovations. ICTAI.* Pp. 303–307. IEEE.

O. Karlsson, (2021). A Performance comparison Between ASP. NET Core and Express. js for creating Web APIs.

Node.js Foundation. (2021). Node.js. Retrieved from https://nodejs.org.

M. Patrou, K.B. Kent, J. Siu & M. Dawson (2021). Energy and runtime performance optimization of node. js web requests. In *2021 IEEE International Conference on Cloud Engineering. IC2E. Pp.* 71–82. IEEE.

N.J. Yeager & R.E. McGrath (1996). *Web server technology*. Morgan Kaufmann.

ХАЛЫҚАРАЛЫҚ АҚПАРАТТЫҚ ЖӘНЕ КОММУНИКАЦИЯЛЫҚ ТЕХНОЛОГИЯЛАР ЖУРНАЛЫ

МЕЖДУНАРОДНЫЙ ЖУРНАЛ ИНФОРМАЦИОННЫХ И КОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

INTERNATIONAL JOURNAL OF INFORMATION AND COMMUNICATION TECHNOLOGIES

Правила оформления статьи для публикации в журнале на сайте:
https://journal.iitu.edu.kz
ISSN 2708–2032 (print)
ISSN 2708–2040 (online)
Собственник: АО «Международный университет информационных технологий» (Казахстан, Алматы)

Подписано в печать 15.06.2023.
Формат 60x881/8. Бумага офсетная. Печать - ризограф.6,5 п.л. Тираж 100
050040 г. Алматы, ул. Манаса 34/1, каб. 709, тел: +7 (727) 244-51-09).

Издание Международного университета информационных технологий
Издательский центр КБТУ, Алматы, ул. Толе би, 59