BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2023
http://excel.fit.vutbr.cz

Optimizing KPI Processing of Smart Cities Using Multithreading in Node.js

Ondřej Šulc*

Abstract
This article deals with the optimization of the processing of key performance indicators (abbr. KPI) in smart cities. KPI is a special type of indicator that makes it possible to express the overall performance/success of the entire city as accurately as possible and to identify performance factors that are key to its future prosperity. For smart cities, the evaluation of these indicators is largely dependent on the data from the IoT sensors that are distributed around it. This means that it is necessary to process huge amounts of data at regular intervals. However, the procedure for such an evaluation consists of a large number of mutually independent calculations. Thanks to this, it is very effective to use multithreading for this evaluation. This article describes the concept of KPIs and their evaluation, as well as an example of implementation on the Node.js platform. The principles described here were practically demonstrated on the SmartCity project from the Logimic company, where the Node.js is also used.

*xsulco01@vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Key performance indicators (abbr. KPI) are an effective method for measuring the success of smart cities. They make it possible to monitor the quality of life of their residents (by measuring air quality or noise levels). The goal of KPI is to make the output as easy to read as possible – usually in the form of a control board containing the results of individual indicators. Examples of simple outputs can be seen in Figure 1. For example, with the smoke detector shown, it is immediately obvious that it has a charged battery, a stable signal, a stable ambient temperature and no alarm has been triggered. The user can therefore easily see whether everything is in order or whether a problem needs to be solved. One of the important sources for calculating the KPI of smart cities is the data coming from various IoT sensors distributed throughout the smart city. These sensors monitor the surrounding environment (for example by measuring CO_2, the noise level in dB, % of air pollution) and thus generate a large amount of valuable data that can be further processed. The simple principle of converting this data into KPI is illustrated in Figure 2. It can also be seen from this figure that the processing mostly consists of mutually independent steps. It is the independent processing that is the ideal property for using asynchronous and multi-threaded execution. These enable better use of available computing resources and increase the efficiency of the entire processing.

2. Asynchronous execution in Node.js

In order to take advantage of asynchronous execution, it is necessary to divide parts of the code into computationally independent blocks. According to [1] this can be achieved by using *Promise*, *async/await* or *callback* functions. Callback functions belong to the oldest of the methods and for the purposes of multithreading (which will be described later) they are unsatisfactory due to the readability of the code. A newer approach is offered by the mentioned data type `Promise`. This data type introduces a completely new concept to sequential code. It wraps an arbitrary action and directly "promises" that the action will be eventually executed – therefore breaking the sequential execution. An object of type `Promise` is always in one of the 3 control states (depending on the completion status of the wrapped action) that the programmer can work with. The latest versions of Node.js allow the use of *asynchronous functions*.

These are auxiliary syntactic constructions that only make it easier to work with objects of type `Promise`. The keywords *async* and *await* are used to work with asynchronous functions. The word `async` is always added to the header of the function definition, automatically wrapping the result of the function with the type `Promise`. The word `await` automatically removes the wrapping type `Promise` from calling of function with `async`. If the action awaited is not yet completed then waiting for its completion is automatically done. Asynchronous execution, however, is always executed on a single thread if no additional steps are made. This thread has the ability to switch between individual blocks during the execution of the program and thus use its time more efficiently (e.g. if a time-consuming operation such as communication with the database or reading from memory occurs in one of the blocks). But in order to use more processor cores, it is necessary to use multithreading.

3. Multithreading in Node.js

To create multiple threads, the *worker_threads* module is used in Node.js, which is available since version 10.5 and is described in [2]. More threads then allow the application to use more processor cores. Individual threads share source code, data from before the threads were created, and access to files. On the other hand each thread has its own registers and stack, which are important for its independent execution. This state is shown in Figure 3. The spawned threads therefore run in parallel with the main thread due to which no blocking occurs. The threads can then be managed individually (that is, the main thread directly determines which thread will perform which work) or a shared work queue can be created, whereby the required actions will then be evenly distributed between the individual threads. These two methods are shown in Figure 4. The method with a shared queue, requires the programmer to implement just that mechanism of equal distribution of work between threads. However, the advantage of this is easier future scaling with the improvement of the guest system, because then there is no need to manually add a new thread in the source code. It is also possible to use one of the already created libraries instead of your own implementation of the shared queue, like the *threads.js* library described in [3].

4. Implementation

An example of how it is possible to easily access multi-threaded KPI processing is on figure 5. It is possible to use the just mentioned library `threads.js`.

The source code is divided into two modules: upper *processKPImodule.ts* and lower *main.ts*.

The upper module describes the behavior of individual threads. It contains a *processKPI* function that receives the KPI to process – the processing logic itself is not essential now. Using the `expose` function, this function is then introduced to the `threads.js` library. Thanks to this, it will be possible to use this function to assign work to threads.

The bottom module defines the behavior of the main thread. At first it gets the KPIs and the data, necessary for its processing, from the database. It then creates individual threads using the *Pool* module from the `threads.js` library. During creation, the module from which the threads are to be created is specified (ie `processKPImodule.ts`). When the threads are ready, all KPIs are put into a shared queue – this happens in the first cycle. When inserting, the function to be executed (`processKPI`) by the thread is specified, and in addition individual `Promises` are stored so that the resulting values can be retrieved later. Insertion into the queue represents the "promise" that the given KPI will be processed by one of the threads. In the second cycle, the keyword `await` is used, which forces the main thread to wait for the completion of all processing (`Promises`) in a non-blocking way. The results are then stored in an array, which is finally stored in the database.

5. Results

The results achievable by multithreading are described by Amdahl's law. According to this law, the executed code can be divided into parallelizable and sequential part. The expected acceleration is then dependent on the ratio of time spent in these parts and the number of available cores. This dependency is shown in Figure 7.

6. Conclusion

In the KPI processing of smart cities , the share of the parallelizable part is very high, and despite the fact that the specific share is influenced by the resulting implementation and the number of measuring devices, the issue of optimization can be easily solved by multi-threaded processing.

Acknowledgments

I would like to thank the supervisor of my thesis Ing. Jiří Hynek, Ph.D., for his advices and motivation during the preparation of this article. At the same time, I would like to thank Ing. František Mikulu, CTO of Logimic for the opportunity to work on the SmartCity project and his advices around it.

References

[1] R.H. Jansen, V. Vane, and I.G. de Wolff. TypeScript: Modern JavaScript Development. Packt Publishing, 2016.

[2] Node.js. Node.js v20.0.0 documentation, 2023. online. Dostupné z: `https://nodejs.org/dist/latest-v20.x/docs/api/worker_threads.html`

[3] Threads.js. Threads.js documentation, 2023. online. Dostupné z: `https://threads.js.org/`