Journal of Science Research International (JSRI)
ISSN: 2456-6365 (Online)

OPEN ACCESS

LIGHTWEIGHT MEMORY-SAFE EVENT LOOP SCHEDULING FOR EMBEDDED WEB SERVERS

Smita Ketan Hadawale* and Prajapat Parabjeet Singh Gurudeep Singh

Department of Computer Science,

Pillai College of Arts, Commerce and Science (Empowered Autonomous), New Panvel

**Corresponding author E-mail: smitahadawale@mes.ac.in*

Received: 20 January 2026	Revised: 25 February 2026	Accepted: 02April 2026	Published: 16 April 2026

DOI: **https://doi.org/10.5281/zenodo.19610058**

Abstract:

This paper examines the relative merits of First-Come-First-Served request queueing for embedded web servers as opposed to the use of a priority order queue which may deem some requests to have higher priority than others and queue accordingly. After building each type of server using Node.js and Express (the priority server assigned a priority value to different types of requests), it was found that the use of priority order queueing does significantly reduce the wait time of high priority authentication requests (by around 85%), and there is virtually no memory or processing overhead (since the priority queue requires a single integer value to be assigned to each request and a single sort operation, equating to less than a hundred bytes of memory for the typical queue). These results therefore show the advantage of using even a minimal system of priorities to determine queue order for embedded web servers.

Keywords: *Event Loop Scheduling, Embedded Web Servers, Priority Queue, FIFO, Node.js, IoT, Authentication Latency,*

1. Introduction

Embedded web servers often have the capability to handle various types of requests at any time. An IoT device may have to deal with authentication requests, sensor data requests, firmware updates, and static file requests at the same time. The complexity arises since a simple First-In-First-Out (FIFO) queue does not differentiate between the types of requests. This means that a large static file transfer may be queued before an urgent authentication request that was made just before it. This results in a delay for the authentication request (1).

This is not just theoretical since authentication delays may cause session timeouts. Session timeouts may cause retry storms in devices that have limited CPU and memory resources. This is a significant threat to IoT devices. The key question this paper seeks to answer is whether a simple solution based on prioritization can be effective in mitigating this threat without increasing the complexity and resources used by the web server.

The answer to this question is affirmative. A simple data structure consisting of a single array and one extra integer for each task is sufficient to solve the priority inversion issue and reduce authentication time by 85.7% when compared to a simple FIFO queue. This paper outlines the two server implementations and the results obtained. It also discusses the implications for developers creating lightweight HTTP servers for constrained devices.

2. Literature review

2.1 Single threaded event loops

The Node.js engine utilizes a single-threaded event loop provided by libuv. This allows for I/O to be performed asynchronously and for JavaScript callbacks to be made when I/O operations are completed. This eliminates the possibility of memory consumption by thread-per-connection servers since each connection requires a stack allocation of 64 KB or more. For an embedded system where the total RAM space allocated is 256 KB, this difference alone dictates whether or not multithreading is even a possibility (2).

2.2 FCFS and its limitations

First-Come-First-Served (FCFS) is the primary scheduling mechanism for lightweight web servers because it is simple to implement and eliminates the possibility of starvation. However, this comes at the cost of the convoy phenomenon whereby the task at the head of the queue dictates the order in which tasks are serviced (3).

2.3 Rate monotonic scheduling

Rate Monotonic Scheduling was proposed by Liu and Layland (1973) as a framework for providing static priority to periodic real-time tasks (4). The key rule for this framework dictates that tasks which require to be serviced at a higher frequency should be allocated a higher priority than those which require less frequent servicing. This framework assumes preemption by the CPU. This is not possible in the JavaScript environment. However, the key principle remains: the allocation of priorities should be based upon the criticality of each task and should be adhered to at all times.

2.4 Memory constraints and GC behavior

The use of JavaScript's garbage collector eliminates the chance for memory leaks due to errors from manually allocating memory, but it also results in pauses that would be disruptive in any type of embedded system that is sensitive to latency. Non-limit-based queues can increase GC load. When an infinite queue accumulates data (1), the run time needs to pause collection until it has accumulated the value of a much larger quantity of this data. Thus, any good embedded scheduling methodology should also include measures to ensure that the queue is maintained at a low, consistent length.

2.5 Research gap

Network throughput and concurrency are the primary benchmarks used when comparing embedded servers (5,6), but when looking at how to schedule requests in a JavaScript Event loop, empirical evidence regarding how FIFO and priority access requests compare is limited. This paper provides a method for making a direct, reproducible comparison.

3. Methodology

3.1 Overview

Two servers were created that operate in precisely the same manner except for how they manage their queues; everything else is identical (HTTP framework, endpoints used, simulated processing delays, polling interval).

3.2 FCFS queue

The First-Come First-Serve (FCFS) Queue appends new tasks to the bottom of an in-memory JavaScript array and will work its way through the array front-to-back, processing one task at a time. AUTH requests simulate 15 milliseconds of process time and STATIC requests will take 35 milliseconds. Polling of the queue occurs every 5 milliseconds.

```
function enqueue(task) {
  queue.push(task);
}
```

3.3 Priority queue

The Priority Server queue is identical to the FCFS Server with one exception; when a new request comes into the Priority Queue, it will add the request onto the queue and then sort the whole queue in descending order of priority. AUTH have a priority of 20 and STATIC have a priority of 1.

```
function enqueue(task) {
  queue.push(task);
  queue.sort((a, b) =>
    b.priority - a.priority);
}
```

Both servers run on the same local machine and have /auth and /static as their endpoints. The Priority Server listens on port 3000 and the FCFS Server listens on port 3001.

3.4 Test conditions

Requests were sent in bursts of 10 — with five AUTH and five STATIC requests in each burst, split into two groups with a separation time of two milliseconds between the two request types. AUTH and STATIC Requests were sent in an interleaved manner, meaning that in about 50% of the cases, the STATIC requests would arrive before the AUTH requests did. Each server received ten request bursts per trial.

3.5 What was measured?

The following parameters were tracked: (1) processing order relative to the arrival order; (2) the rate at which priority inversion occurs; (3) the amount of overhead to enqueue a request recorded using high-resolution timers; (4) end-to-end latency for each request type.

4. Results

4.1 Processing order

The results showed that there was no difference between the processing order and the arrival order for any of the request types when using FCFS. When an AUTH request was sent after a STATIC request, the AUTH request had to wait 35ms for the STATIC request to be processed before it could be dequeued. The average delay for AUTH requests was around 21ms across all ten tests. All AUTH requests in the priority server move to the front of the request queue with every insert. The average wait for AUTH requests dropped to approximately 3 ms, an 85.7% reduction from previous wait times.

4.2 Inversion of priority

All FCFS queues had a priority inversion rate of 50%. Each time a STATIC request arrived before an AUTH request, it would cause the AUTH request to wait. An inversion is not a bug in the FCFS algorithm; it is part of how FCFS functions. In all 10 trials, the Priority Server had 0% priority inversions.

4.3 Use of overhead

The average processing time for the sort operation was 0.08 ms per queue entry when the queue depth was ten items. The only memory utilization increase per queued task is one integer — an 8-byte variable using V8's heap representation. Throughput was statistically equivalent between the two servers ($p > 0.05$).

4.4 Summary Table

Table 1: Performance comparison across ten trials per server.

Metric	FCFS	Priority
AUTH avg. wait	~21 ms	~3 ms
Inversion rate	50%	0%
Throughput	Baseline	Equivalent
Sort overhead	None	~0.08 ms
Extra memory	None	+8 bytes

5. Discussion

5.1 What the numbers mean

A reduction in wait time for an authenticated request (85.7%) is much more than a slight improvement and as such, this is significant enough not only to help reduce costs for the end user, but also for Web and Embedded Systems requiring a device performing multiple actions per session. A device will experience a long delay in receiving a session token, and thus experience a long delay before being able to call an API and may be forced to retry or time out waiting for a successful API response if there is an excessive queue to authenticate users.

The total number of requests processed by the web server did not change and is equally important. The usage of priority scheduling does not increase the performance of an authenticated server, only the order. Therefore, a priority server can be used as a complete upgrade to a FCFS server for all authenticated requests made to that server using multiple request types.

5.2 Why the overhead is negligible

V8 uses a sorting algorithm called TimSort for its implementation of Array.prototype.sort(). TimSort has a worst-case time complexity of O(n log n) but has a best-case time complexity of O(n) when processing nearly-sorted arrays. Since the queue is kept sorted after every insert, each new insertion finds the queue nearly sorted. Therefore, there is little to no increased cost for processing an insert until the queue is at least 50 items deep; most embedded web servers will never reach this threshold.

5.3 Limitations

Both server simulation's processing delays are emulated using setTimeout(), which does not accurately emulate real I/O. The timing characteristics of actual flash reads or SPI transactions vary significantly from simulated latency. Additionally, since both tests were executed on an x64 desktop computer instead of embedded hardware, the output overhead metrics should be regarded as such; V8's output metric will be very different when compared to Duktape or JerryScript running on a Cortex-M4. An unanswered starvation query relates to whether a Static request could remain indefinitely without response if there were an indefinite number of Auth requests arriving. The boundary rate for determining whether static Requests could wait indefinitely was not measured. Therefore, a max wait time and a time-based aging mechanism should be implemented to ensure that all lower priority pending requests are processed at some point.

5.4 What should change in practice

The actual implementation associated with Request prioritization consists of only two lines of code; thus there is no reason not to implement it. All Embedded HTTP Servers that handle multiple endpoints should assign priorities to those endpoints based upon how critical they are. Embedded HTTP Frameworks should also expose priority as a first class feature of their Routing API.

5.5 What to do next

There are three directions that seem worthwhile to follow. First, replacing the sorted array with a binary min-heap will reduce the cost of inserting an element from O(n log n) to O(log n); this starts to matter when you exceed approximately 50 items. Second, adding an aging mechanism that will cause the effective priority of a task to increase as time passes will mathematically bound the risk of starvation. Third, the priority queue needs to be tested on real, embedded hardware to confirm that the overhead values remain accurate outside of V8.

Conclusion

This paper set out to help answer whether a basic priority queue system should be implemented in an embedded web server. Based on the evidence presented in this experiment, the answer is a resounding yes. The use of a priority server resulted in an 85.7% reduction in the time for a user to authenticate to the web server; it eliminated all instances of priority inversion; and matched the FCFS server throughput in the number of requests served. The priority queue has a marginal cost of 8 bytes per task queued and 0.08 ms for each task insertion.

FIFO scheduling is simple and fair; however, it is not smart. In a FIFO scheduling system, every request is treated equally, regardless of type or urgency. In an embedded system where there are request types that differ from one another based on urgency and user needs, equality in this context is a fault, not a virtue. A priority queue provides a very cost-effective and simple solution to this issue; therefore, there is no valid justification for an embedded system to exclude the priority queue from their scheduling systems.

Acknowledgements

The authors would like to thank the faculty of Pillai College of Arts, Commerce and Science for their support and guidance throughout this research.

References

1. Atzori, L., Iera, A., & Morabito, G. (2010). The internet of things: A survey. *Computer Networks, 54*(15), 2787–2805.
2. Tilkov, S., & Vinoski, S. (2010). Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing, 14*(6), 80–83.
3. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating system concepts* (10th ed.). Wiley.
4. Liu, C. L., & Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM, 20*(1), 46–61.
5. Baccelli, E., Hahm, O., Wahlisch, M., Gunes, M., & Schmidt, T. (2013). RIOT OS: Towards an OS for the internet of things. *IEEE INFOCOM Workshops*, 79–80.
6. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., & Culler, D. (2004). TinyOS: An operating system for sensor networks. In W. Weber, J. Rabaey, & E. Aarts (Eds.), *Ambient intelligence* (pp. 115–148). Springer.