PENGEMBANGAN WEB (TEORI)

LAPORAN EKSPERIMEN: PERBANDINGAN EFISIENSI CONCURRENCY NODE.JS VS GO (GOLANG)

Laporan ini disusun untuk memenuhi tugas 1 mata kuliah Pengembangan Web (Teori)

Disusun oleh kelompok B4:

Asri Husnul Rosadi	221524035
Faris Abulkhoir	221524040
Mahardika Pratama	221524044
Muhamad Fahri Yuwan	221524047
Najib Alimudin Fajri	221524053
Septyana Agustina	221524058
Sarah	221524059

Dosen Pengampu:
Joe Lian Min, M.Eng.

JURUSAN TEKNIK KOMPUTER DAN INFORMATIKA
PROGRAM STUDI D4 TEKNIK INFORMATIKA
POLITEKNIK NEGERI BANDUNG
2024

DAFTAR ISI

A. **IDENTIFIKASI PROBLEM**

Efisiensi dalam menangani concurrency merupakan aspek krusial dalam pengembangan perangkat lunak modern. Node.js dan Go (Golang) adalah dua platform yang populer untuk pengembangan aplikasi yang memerlukan kemampuan concurrency tinggi. Node.js menggunakan event loop untuk menangani operasi asinkron, sedangkan Go menggunakan goroutines. Laporan ini bertujuan untuk membandingkan efisiensi kedua platform dalam hal throughput, latency, dan penggunaan CPU saat menangani banyak tugas secara bersamaan.

B. **DESKRIPSI PROBLEM**

Node.js dan Go (Golang) menawarkan solusi berbeda untuk menangani concurrency. Node.js mengandalkan event loop untuk menangani operasi non-blocking dan asinkron. Sebaliknya, Go menggunakan goroutines yang dikelola oleh scheduler internal untuk menjalankan banyak fungsi secara bersamaan. Dengan semakin banyaknya aplikasi yang membutuhkan pengelolaan concurrent tasks secara efisien, penting untuk mengevaluasi performa masing-masing platform dalam konteks ini.

C. **METODOLOGI EKSPERIMEN**

1) Tujuan

   Tujuan dari eksperimen ini adalah mengukur dan membandingkan throughput, latency, dan efisiensi penggunaan CPU antara Node.js dan Go (Golang).

2) Desain Eksperimen
   - Melakukan sejumlah tugas asinkron secara bersamaan untuk masing-masing platform.
   - Mengukur waktu total yang diperlukan untuk menyelesaikan semua tugas (latency).
   - Menghitung throughput sebagai jumlah tugas per detik.
   - Mencatat jumlah inti CPU untuk perbandingan.

3) Parameter Eksperimen
   - Jumlah Tugas: 100
   - Durasi Tugas: 100 ms

4) Metode Pengukuran

- enggunakan fungsi built-in untuk mengukur waktu (performance.now() di Node.js dan time.Now() di Go).
- Menghitung throughput dengan membagi jumlah tugas dengan waktu total.
- Menyertakan informasi jumlah inti CPU yang digunakan.

D. **PELAKSANAAN EKSPERIMEN**

1) Program Node.js

```
const { performance } = require('perf_hooks');
const os = require('os');

const numTasks = 100; // Jumlah tugas
const taskDuration = 100; // Durasi setiap tugas dalam
milidetik

const runExperiment = async () => {
    const startTime = performance.now();

    // Simulasi tugas asinkron
    const tasks = Array.from({ length: numTasks }, (_, index)
=> new Promise((resolve) => {
        setTimeout(() => {
            resolve(`Task ${index + 1} completed`);
        }, taskDuration);
    }));

    // Tunggu semua tugas selesai
    await Promise.all(tasks);

    const endTime = performance.now();
    const elapsedTime = endTime - startTime; // Latency dalam
milidetik

    // Hitung throughput
    const throughput = numTasks / (elapsedTime / 1000); //
Tugas per detik

    console.log(`Total Time: ${elapsedTime.toFixed(2)} ms`);
    console.log(`Throughput: ${throughput.toFixed(2)}
tasks/second`);
    console.log(`CPU Cores: ${os.cpus().length}`);
};

runExperiment();
```

2) Program Go (Golang)

```
package main
```

```
import (
    "fmt"
    "sync"
    "time"
    "runtime"
)

const (
    numTasks      = 100 // Jumlah tugas
    taskDuration  = 100 * time.Millisecond // Durasi setiap
tugas
)

func heavyOperation(id int, wg *sync.WaitGroup) {
    defer wg.Done()
    time.Sleep(taskDuration)
    fmt.Printf("Task %d completed\n", id)
}

func main() {
    startTime := time.Now()

    var wg sync.WaitGroup
    wg.Add(numTasks)

    // Jalankan goroutine untuk setiap tugas
    for i := 1; i <= numTasks; i++ {
        go heavyOperation(i, &wg)
    }

    // Tunggu hingga semua goroutine selesai
    wg.Wait()

    endTime := time.Now()
    elapsedTime := endTime.Sub(startTime) // Latency

    // Hitung throughput
    throughput := float64(numTasks) / elapsedTime.Seconds()

    fmt.Printf("Total Time: %v\n", elapsedTime)
    fmt.Printf("Throughput: %.2f tasks/second\n", throughput)
    fmt.Printf("CPU Cores: %d\n", runtime.NumCPU())
}
```

E. **ANALISIS HASIL EKSPERIMEN**

1) Hasil Percobaan

Percobaan	Platform	Total Time (ms)	Throughput (tasks/second)	CPU Cores
1	Node.js	114.19	875.74	4
	Go	101.2993	987.17	4

2	Node.js	117.07	854.22	4
	Go	101.5943	984.31	4
3	Node.js	110.28	906.82	4
	Go	101.2256	987.89	4

2) Observasi

- Latency: Go menunjukkan waktu total yang lebih rendah dibandingkan dengan Node.js pada semua percobaan. Ini menunjukkan bahwa Go lebih cepat dalam menyelesaikan tugas dalam pengaturan ini.
- Throughput: Go juga menunjukkan throughput yang lebih tinggi dibandingkan dengan Node.js pada semua percobaan. Ini menunjukkan bahwa Go dapat menyelesaikan lebih banyak tugas per detik.
- CPU Cores: Jumlah inti CPU yang digunakan sama untuk kedua platform dalam eksperimen ini.

F. **KESIMPULAN**

Berdasarkan hasil eksperimen:

- Go (Golang) menunjukkan performa yang lebih baik dibandingkan dengan Node.js dalam hal throughput dan latency. Go lebih efisien dalam menangani banyak tugas secara bersamaan berkat goroutines dan scheduler internalnya.
- Node.js meskipun efektif dalam menangani I/O-bound tasks dengan event loop-nya, dalam konteks concurrent tasks dengan banyak tugas, Go memberikan hasil yang lebih baik.