REYKJAVIK UNIVERSITY

T-419-CADP PROJECT REPORT

Concurrent Port Scanner

Kristófer Fannar Björnsson
kristoferb21@ru.is

Logi Sigurðarson
logis21@ru.is

Teacher: Marcel Kyas

February 1, 2024

Introduction

In this report, the functionality and usage of the Concurrent Port Scanner from Group 5 will be discussed in detail. The Port Scanner was written in Go and utilized the net package heavily. Unit tests for utility functions were written to ensure their functionality.

The goal of the Port Scanner is to utilize Golang's concurrency model to scan ports concurrently, providing a better user experience to the end user in reduced execution time. The reduction of execution time can be attributed to the gains made by concurrently executing the Scan function in the port directory.

This is because the Scan function waits for 2 seconds for a TCP packet from the port it is trying to scan. Instead of waiting 2 seconds on each port sequentially, causing high execution times, other ports are scanned while this waiting proceeds. This is where concurrency shines. When a thread is blocked or waiting for some external resource like I/O or a server response.

```go
package port

import (
    "fmt"
    "net"
    "strings"
    "time"
)

// Scans TCP Port of a specified host
//
// If no packet is sent from the host within 2 seconds the connection times out
func Scan(ipAddr net.IP, port string) {
    address := net.JoinHostPort(ipAddr.String(), port)
    dialer := net.Dialer{Timeout: 2 * time.Second} // Set a timeout
    conn, err := dialer.Dial("tcp", address)

    if err == nil {
        fmt.Printf("%s open\n", address)
        conn.Close()
    } else if strings.Contains(err.Error(), "connection refused") {
        fmt.Printf("%s closed\n", address)
    }
}
```

Figure 1: Scan function

Data Flow

The project is separated into three packages, four program files and two test files. The main function resides in the file scanner.go in the root directory of the project.

The aim was to keep the length and complexity of the main function minimal and abstract the command line argument parsing. This can be seen in lines 34 and 35 of the scanner.go file. The arguments and flag parsing are delegated to the ConvertArgsToIPs and ConvertFlagToPorts functions which can be found in the utils directory in the ip_utils.go and port_utils.go respectively. These two functions are the only public functions from these files. Many other private functions can be found in each file which help in the validation and parsing of both IP addresses and ports. The rest of the function iterates over each IP and port and scans it. The concurrency model and resource management will be discussed in detail below.

Concurrency and Resources

The concurrency is modeled using a WaitGroup and a semaphore [1] made using channels. The purpose of the semaphore is to control the number of goroutines that can be executed at one time. This is done to prevent resource exhaustion. Each gorouting takes about 4KiB of memory [2]. This is why the number of concurrent goroutines is limited to 1.000, meaning a maximum of 1.000 go-routines will be executed at one time. This means that at maximum the program should need about 4-5MB of RAM respectively for allocating go-routines. Since the size of RAM is different for each user it was decided to keep the memory usage as low as possible, this allows the program to be executed on a host of different computers. Additionally, operating systems vary in configuration and some do not allow spawning of tens of thousands of threads. If the user is running other tasks in parallel it could affect the number of available threads the program is allowed to span. From our experience, the program crashed if more than 10.000 threads were spawned by the program, due to insufficient resources.

In the future, this could be a flag that the user sets himself. If a user has over 8Gb of RAM, he should be able to run 1 million go-routines (4GiB) at once without a problem (if the operating system allows that and no other tasks are running). This would mean significant performance gains but at the cost of more memory usage.

A channel of empty structs is utilized for keeping track of the number of go-routines running. It is initialized with a buffer of 1.000. Before each go routine is executed, the main thread sends an empty struct into the channel, reducing the buffer by one, then the go-routine scans the port, and finally releases an empty struct from the channel increasing the buffer by one. If the buffer hits zero, go-routines will be blocked until an empty struct is released by another go-routine. Thus the channel acts like a semaphore.

The command line argument parsing is not done concurrently, the need is simply not there since there are no long blocking calls in the parsing functions. Because of this and the fact that the output should be printed to stdout, there is simply no need for a shared resource like a slice of ports or a map. This is why race conditions should not arise in the program. No shared resource is being read or written to.

The program does not have deadlocks, the only section where a deadlock could arise is in lines 44 and 50 of the scanner.go where each go-routine acquires the semaphore and where the go-routine releases the semaphore. The reason why this does not cause a deadlock is because each go-routine is guaranteed to release the semaphore because port.Scan() always returns after 2 seconds. It is known that each go-routine finishes because the main thread only terminates after all threads that have been added to the WaitGroup have terminated. Each go-routine calls Wg.Done() after execution making sure it terminates.

```
37      var wg sync.WaitGroup
 1      // https://levelup.gitconnected.com/go-concurrency-pattern-semaphore-9587d45f058d
 2      sem := make(chan struct{}, maxGoroutines)
 3
 4      for _, ip := range ips {
 5          for _, po := range ports {
 6              wg.Add(1)
 7              sem <- struct{}{} // aquire semaphore
 8
 9              // Scan port concurrently
10              go func(host net.IP, portno string) {
11                  defer wg.Done()
12                  port.Scan(host, portno)
13                  <-sem // release semaphore
14
15              }(ip, po)
16          }
17      }
18
19      wg.Wait()
20
21      close(sem)
22
23      fmt.Println("Scanning complete.")
```

Figure 2: Concurrency model

References

[1] S. Okta, *Go concurrency pattern: Semaphore*. [Online]. Available: `https://levelup.gitconnected.com/go-concurrency-pattern-semaphore-9587d45f058d`.

[2] S. Overflow, *Go - max number of goroutines*. [Online]. Available: `https://stackoverflow.com/questions/8509152/max-number-of-goroutines`.