Project Learning Docs

So we will be working on a Project Network Scanner here and I will be using go to scan the network . This docs is what i write while learning the fundamentals and building the scanner so it might come handy for you or not but you can check it if you want

— Upesh Bhujel AKA Cupid

Lesson 1 : What is a Port? (The Digital Doors)

So, let say there is a giant apartment building and that building has a single street address which is also called as IP address in the field of computer. It identify a specific computer on the network and in our example the apartment building where it is located.

Now, inside that paparment there are a total of 65,535 apartments(in this are ports), and each apartment has a unique number from 1 - 65,535 and these are what we called ports.

Now, as in apartment there live different family or person here different services live in these apartments.

- The web server lives in 80
- The secure web in 443
- The FTP (File Transfer Protocal) in 20/21 etc

Analogy: If we want to send a letter to our friends or even say our worst nightmare, who live in that building, we just can't write the building's street address but we also need to add the apartment number also, So that our mail reach to where it belongs.

IP address + PortNumber = A specific service on a computer

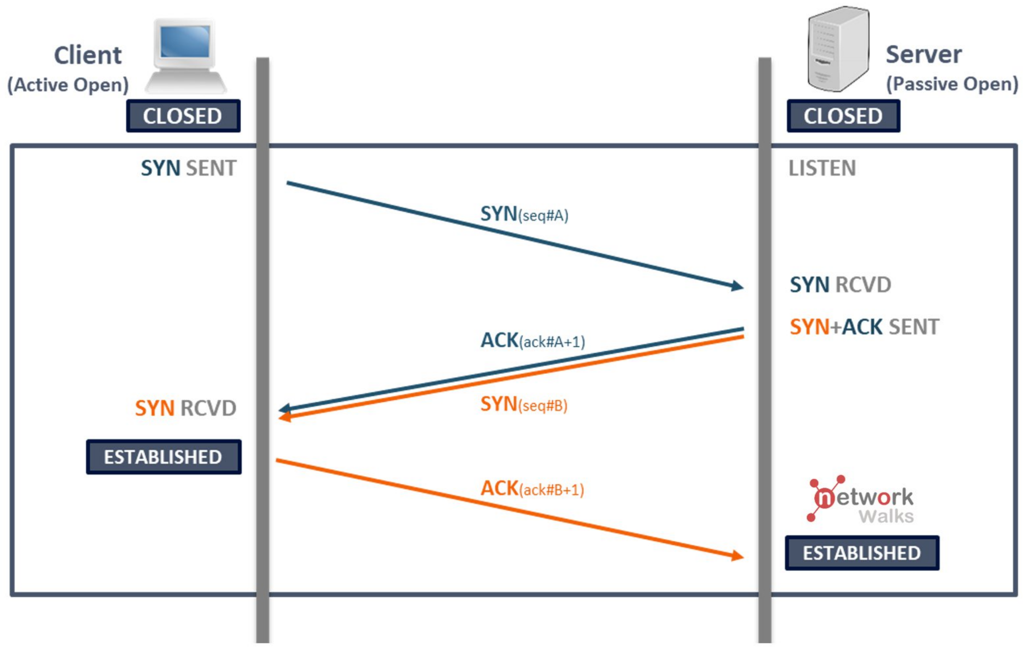
When Our Scanner checks if the Port is open or not, its basically checking each and every apartment door knocking if there is anyone or not. If there is someone it answers the port is open. If no one answer or the door is shut, the port is close

So what happen actually when we use internet ????
When we connect our devices with internet, it will have an IP address provided by our network and as we browse the web, our devices is making connection on port 443 on web server(Basiclly Google for example) and based on what request we send we get the responce we need.

Lesson 2 : How We "Knock" on the Doors (The TCP HandShake)

What we know up now is that we have to knock the doors to see if they are open or not. But how do we do that using our devices? Simple there is a process called Three-Way Handshake.

TCP 3-way Handshake Process



It's more like a very formal and quick phone call where there is 3 steps
Step 1 : SYN [Our device send the message called SYN packet. Its like dialing the contact and saying "Hey are you there to talk?"]

Step 2 : SYN/ACK [if the server is on and the port is also open it send back the SYN/ACK message. This is the server replying "Yes I am here and I acknoledge that you want to talk"]

Step 3 : ACK [Its the final message our devices send ACK packet to say Great I heared you lets start the communication]

After these 3 steps the connection is established and data can be transfer.

How Our Scanner Uses These?

So, our scanner does what a brat does, it only does 2 steps :

1. It send the SYN (Hello?)
2. It wait for SYN/ACK (Yes , I am here)

As soon as it get “Yes I am here” it knows the port is open and move to another port, it doesn’t need to talk further to the open port. It mark that port as “Open” and move forward. This is called the “Connect Scan”.
If the scanner send the SYN and get nothing back or get conncection refused it mark that port as “CLOSED”.

Classic example :
Imagine you want to know if your friend is home, but you don't want to have a full conversation. You could call their phone. If they answer "Hello?", you could just hang up. You achieved your goal: you know they are home. This is exactly what our scanner does.

Lesson 3 : Building the first, Simple Scanner

Goal : To write a simple program that knock on a single digital door or port on a target and tell if its open or close

Tool : we will be using a build-in library called socket that give our code the power to interact with a network.

Part 1 : Python Version

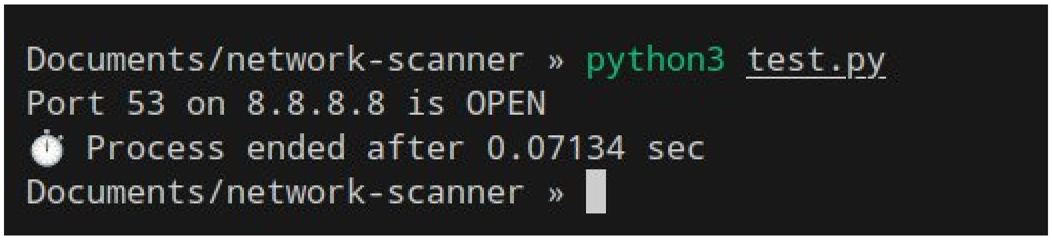
Code logic:

1. Import the socket library
2. Create a socket object
3. Specify the target ip address and the port we want to check
4. Use a function to try and connect. This function is design not to crash if connection is failed.
5. If the result of connection attempt is 0 the port is open other wise it’s close

Code :
```
import socket

# --- Configuration ---
target_ip = "8.8.8.8"  # Google's Public DNS server, a reliable target
port_to_scan = 53      # Port 53 is for DNS, it should be open on this server

# --- The Scanner Logic ---
try:
    # 1. Create a socket object
    # AF_INET means we are using IPv4 addresses (the standard x.x.x.x format)
    # SOCK_STREAM means we are using TCP (the formal handshake protocol)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Set a timeout so our program doesn't hang forever if a port is
unresponsive
```

```
    s.settimeout(1) # Wait for a maximum of 1 second

    # 2. Try to connect to the target IP and port
    # connect_ex returns an error indicator. 0 means success.
    result = s.connect_ex((target_ip, port_to_scan))

    # 3. Check the result
    if result == 0:
        print(f"Port {port_to_scan} on {target_ip} is OPEN")
    else:
        print(f"Port {port_to_scan} on {target_ip} is CLOSED")

    # 4. Close the socket to be tidy
    s.close()

except socket.error as e:
    print(f"An error occurred: {e}")
```

Output :



```
Documents/network-scanner » python3 test.py
Port 53 on 8.8.8.8 is OPEN
⏱ Process ended after 0.07134 sec
Documents/network-scanner »
```

Part 2 : Go Version

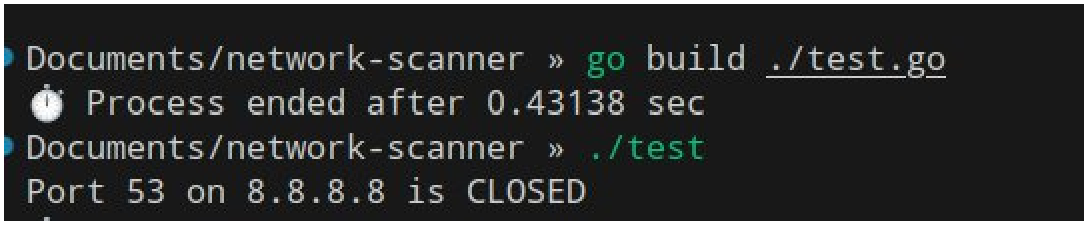
Code logic:

1. Import the net and fmt packages. Net is Go's powerful networking toolkit. Fmt is for printing
2. Specify the target and port, then combine them into a single address string like 8.8.8.8:53
3. Use the net.DialTimeout function. This is Go's way of knocking. It tries to connect but will give up after a certain timeout.
4. In Go, functions can return multiple values. DialTimeout returns a connection object and a potential error.
5. If the error object is nil, the connection was successful and the port is open otherwise it's closed.

Code :

```
package main
import (
      "fmt"
      "net"
      "time"
)
func main() {
      // --- Configuration ---
```

```
	target_ip := "8.8.8.8"
	port_to_scan := "53" // In Go, it's common to handle the port as a
string
	// --- The Scanner Logic ---
	// 1. Combine host and port into a single address string
	address := net.JoinHostPort(target_ip, port_to_scan)

	// 2. Try to connect to the target address via TCP
	// We give it a timeout of 1 second.
	conn, err := net.DialTimeout("tcp", address, 1*time.Second)

	// 3. Check if an error occurred
	// 'nil' is Go's equivalent of 'None' or 'null'. It means "no error".
	if err != nil {
		fmt.Printf("Port %s on %s is CLOSED\n", port_to_scan, target_ip)
	} else {
		// If there was no error, the port is open.
		// We must close the connection we opened.
		conn.Close()
		fmt.Printf("Port %s on %s is OPEN\n", port_to_scan, target_ip)
	}
}
```
Output :



```
Documents/network-scanner » go build ./test.go
⏱ Process ended after 0.43138 sec
Documents/network-scanner » ./test
Port 53 on 8.8.8.8 is CLOSED
```

Lesson 4: The Need for Speed and the Problem with loops

The Goal: To modify our simple scanner to check a range of ports
The Method : We will use a for loop. A loop is just a way to tell the computer to report an action over and over again. In our case, the action is “run the port scan logic”. We will tell the program to do this for every number from 1 to 1024.

The Code's Logic:

1. Set the target IP.
2. Start a for loop from 1 to 1024.
3. Inside the loop, for each port, attempt the net.DialTimeout.
4. If there is no error, print that the port is open.

Code :

```
package main
```

```
import (
	"fmt"
	"net"
	"os"
	"time"
)

func main() {
	// --- Configuration ---
	var target_ip string
	if len(os.Args) > 1 {
		target_ip = os.Args[1]
	} else {
		target_ip = "scanme.nmap.org"
	}

	fmt.Printf("Scanning target: %s\n", target_ip)
	fmt.Println("==========================================
=========")

	// --- The Scanner Logic in a Loop ---
	// Loop through the first 1024 ports
	for port := 1; port <= 1024; port++ {
		// We have to convert the integer port to a string for
the address
		address := fmt.Sprintf("%s:%d", target_ip, port)
		// The logic is the same as before
		conn, err := net.DialTimeout("tcp", address,
1*time.Second)

		if err == nil {
			// If no error, the port is open
			conn.Close()
			fmt.Printf("Port %d is OPEN\n", port)
		}
		// We do nothing if the port is closed (err is not
nil)
	}
	fmt.Println("==========================================
=========")
	fmt.Println("Scan complete.")
}
```

Output :

The Big Problem: This is S-L-O-W

Let's do the math:

- We have a timeout of 1 second for each port.
- To scan 1024 ports, in the worst case (if all are closed or filtered), it could take up to 1024 seconds, which is about 17 minutes!
- To scan all 65,535 ports would take 65,535 seconds, which is over 18 hours!

This is unacceptable for a real tool. We are knocking on one door, waiting for an answer, and only then moving to the next door. This is not efficient.

What if you could hire 100 people and have them all knock on different doors at the same time? This is the core idea that will make our scanner fast.

This concept of "doing things at the same time" is called Concurrency. It's the most important topic for making our scanner powerful.

Lesson 5: Juggling Tasks (The Theory of Concurrency)

The Goal: To understand the *concept* of doing many things at the same time so we can apply it to our scanner. We won't write code just yet; we'll focus on the idea.
Let's use a better analogy than just hiring people. Imagine you have a mountain of dirty dishes.
Method 1: The Sequential Way (What we did in Lesson 4) You, one person, do everything:

1. Pick up a dish.
2. Wash it.
3. Dry it.
4. Put it in the cupboard.
5. Repeat for the next dish.

This is our slow scanner. Each port is a dish, and it's processed completely, one after another.
Method 2: The Concurrent Way (What we will do now) You get a friend to help. You both agree on a system:

- **You are the "Producer"**: Your only job is to wash dishes and place them on a drying rack.

- **Your friend is the "Worker"**: Their only job is to take clean, wet dishes from the drying rack, dry them, and put them away.
- **The Drying Rack is the "Job Queue"**: This is the critical part. It's a shared space where you leave tasks for your friend. You don't have to hand each dish to them directly. You can wash a few and stack them up. They can grab one whenever they're ready.

You are both working *at the same time* on the overall "clean the kitchen" task. While you are washing the 5th dish, your friend might be drying the 3rd dish. This is **concurrency**.

Applying This to Our Port Scanner

- The **Dishes** are the port numbers we want to scan (1, 2, 3, ... 65535).
- The **Producer** is our main program function. Its new job is NOT to scan, but to quickly fill the "drying rack" with all the port numbers.
- The **Workers** are the functions that do the actual scanning. We can create 100s or 1000s of these workers.
- The **Drying Rack** is a special programming construct called a **Queue** or a **Channel**.

This new process will look like this:

1. Our main function creates a Job Queue.
2. The main function very quickly dumps all the ports to scan (1-1024) into the Job Queue.
3. The main function starts, for example, 100 Workers.
4. Each of the 100 workers immediately goes to the Job Queue, grabs a port number, and goes off to scan it. When it's done, it goes back to the queue to grab the next available port.
5. This continues until the Job Queue is empty and all workers are finished.

Because 100 ports are being scanned *simultaneously*, the total time is dramatically reduced. Instead of taking 1024 seconds, it might now only take a few seconds.

How Go handle this ?

Was built for this. It has a concept called a Goroutine. A goroutine is like a super lightweight, cheap, and efficient worker. You can create thousands of them without slowing down your system. The "drying rack" they use is called a Channel, and it's a core feature of the language. This is why the Go code we originally looked at was so fast.

Lesson 6: Building the Fast Scanner (Applying Concurrency)

The Goal: To rewrite our scanner using the "Producer/Worker" (concurrent) model to scan many ports simultaneously.

You will see that the worker function's core logic—the part that actually checks a single port—is almost identical to what we wrote in Lesson 3. The magic is in how we *organize* the work.

Part 1: The Fast Go Scanner (Using Goroutines)

This is where Go shines. Its goroutines and channels are built for this. This code is very similar to the first script I showed you, but now you will understand exactly why it's built this way.

The Code's Logic:

1. **The Worker:** A worker goroutine receives jobs from one channel (ports) and sends results to another channel (results).
2. **The Main Function (The Manager):**
   - Creates two channels: one for jobs (ports) and one for results (results).
   - Starts 100 goroutines (our workers).
   - Fills the ports channel with all the target port numbers in a separate goroutine.
   - Collects all the results from the results channel.
   - Uses a sync.WaitGroup to wait for all workers to signal they are finished before printing the results.

Code :

```
package main
import (
	"fmt"
	"net"
	"os"
	"sort"
	"sync"
	"time"
)
// --- The Worker ---
// It receives work from the 'ports' channel and sends results to
the 'results' channel.
// The WaitGroup is used to signal that this worker is done.
func worker(ports chan int, results chan int, target string, wg
*sync.WaitGroup) {
	// Signal that we are done when this function returns
	defer wg.Done()
	// Keep taking work from the 'ports' channel until it's
closed
	for p := range ports {
		address := fmt.Sprintf("%s:%d", target, p)
		conn, err := net.DialTimeout("tcp", address,
500*time.Millisecond) // 0.5 sec timeout
		if err != nil {
```

```
                // Port is closed, do nothing
                continue
            }
            conn.Close()
            // Port is open, send the port number to the results
channel
            results <- p
        }}
// --- The Main Function (The Manager) ---
func main() {
        target_ip := "scanme.nmap.org"
        if len(os.Args) > 1 {
                target_ip = os.Args[1]
        }
        var wg sync.WaitGroup // To wait for all goroutines to
finish
        ports := make(chan int, 100) // The job queue (buffered)
        results := make(chan int)   // The results collector
        var openPorts []int
        fmt.Printf("Scanning %s...\n", target_ip)
        // Start 100 worker goroutines
        for i := 0; i < 100; i++ {
                wg.Add(1) // Increment the WaitGroup counter
                go worker(ports, results, target_ip, &wg)
        }
        // Start a goroutine to send all the ports to the workers
        go func() {
                for i := 1; i <= 1024; i++ {
                        ports <- i
                }
                close(ports) // Close the channel when all jobs are
sent
        }()
        // Start a goroutine to wait for all workers to finish, then
close the results channel
        go func() {
                wg.Wait()
                close(results)
        }()

        // Read all the results from the results channel until it's
closed
        for port := range results {
                openPorts = append(openPorts, port)
        }
        // --- Print the Results ---
        sort.Ints(openPorts) // Sort the final list
        if len(openPorts) > 0 {
            fmt.Println("Open ports found:")
            for _, port := range openPorts {
                fmt.Println(port)
            }
        } else {
            fmt.Println("No open ports found.")
        }
    }
```

Lesson 7: Making It User-Friendly

The Goal: To add an interactive menu to our fast scanner that asks the user for the target IP and lets them choose between a default scan (common ports) or a full scan (all ports).

Part 1: The Final Go Scanner

We'll follow the same pattern in Go, using the fmt package to get the user's input.

The Code's Logic:

1. Use fmt.Scanln to read the target IP and menu choice from the user.
2. Use a switch statement (Go's version of an if/elif/else chain) to check the user's choice.
3. Set the startPort and endPort variables based on the choice.
4. Use these variables in the for loop that populates the ports channel.

Code :

```
package main
import (
    "fmt"
    "net"
    "sort"
    "sync"
    "time"
)

// The worker function remains exactly the same
func worker(ports chan int, results chan int, target string, wg
*sync.WaitGroup) {
    defer wg.Done()
    for p := range ports {
        address := fmt.Sprintf("%s:%d", target, p)
```

```
          conn, err := net.DialTimeout("tcp", address,
500*time.Millisecond)
          if err != nil {
               continue
          }
          conn.Close()
          results <- p
     }}
// --- The Main Program (The Manager) ---
func main() {
     // 1. GET USER INPUT
     var target_ip string
     fmt.Print("Enter the target IP address to scan: ")
     fmt.Scanln(&target_ip)

     fmt.Println("\nSelect scan type:")
     fmt.Println("1. Default Ports (1-1024)")
     fmt.Println("2. All Ports (1-65535)")

     var choice int
     fmt.Print("Enter your choice (1 or 2): ")
     fmt.Scanln(&choice)

     var startPort, endPort int

     switch choice {
     case 1:
          startPort = 1
          endPort = 1024
     case 2:
          startPort = 1
          endPort = 65535
     default:
          fmt.Println("Invalid choice. Exiting.")
          return // Exit the program
     }

     fmt.Println("------------------------------------------------
---")
     fmt.Printf("Scanning %s from port %d to %d...\n", target_ip,
startPort, endPort)
     fmt.Println("------------------------------------------------
---")

     // 2. SETUP CHANNELS AND WORKERS (Same as before)
```

```
	var wg sync.WaitGroup
	ports := make(chan int, 200) // Buffered channel for
performance
	results := make(chan int)
	var openPorts []int

	for i := 0; i < 100; i++ {
		wg.Add(1)
		go worker(ports, results, target_ip, &wg)
	}

	// Fill the ports channel with the chosen range
	go func() {
		for i := startPort; i <= endPort; i++ {
			ports <- i
		}
		close(ports)
	}()

	go func() {
		wg.Wait()
		close(results)
	}()

	for port := range results {
		openPorts = append(openPorts, port)
	}

	// 3. DISPLAY RESULTS
	sort.Ints(openPorts)
	if len(openPorts) > 0 {
		fmt.Println("Open ports found:")
		for _, port := range openPorts {
			fmt.Println(port)
		}
	} else {
		fmt.Println("No open ports found in the specified
range.")
	}
}
```