CSE 508
Project #1 Report

Team name: SecurityHacker

Fangyu Deng, 109085117
Dandan Zheng, 109600563
Jing Yang, 107950330
Ying Luo , 108718935

1. Introduction

IP/port scanning is a very essential way to explore network layout and services running on the system. On the other hand, it's also a technique often used by hackers to attack target hosts. In this project, we follow the design of Nmap and implement a concurrent ip/port scanner, which is capable of checking a range of alive IPs and doing multi-modes port scanning. A main controller is also designed to communicate with distributed scanners, decompose tasks to multiple jobs, assign jobs to scanners with workload balance and collect reports. The controller accepts tasks and post reports through a user-friendly web UI. And we also add mechanisms to the whole distributed system to make it fault tolerant as possible as we can.

2. Overview

2.1. Dependency

- System:
  - Ubuntu 14.04
- Language:
  - Python 2.7
- Web:
  - Django 1.7.5
  - Bootstrap 3.3.4
  - jQuery 1.11.1
  - Google Chart
- Database:
  - MySQL 5.6
- Communication:
  - Protocol Buffer 3.0.0
  - gRPC (a RPC framework layered over HTTP/2 by Google)
- Network packet manipulation:
  - Scapy 2.3.1

2.2. Source Code

- Scanner:
  - src/scanner/scanning_func.py: functions of 5 scanning methods
  - src/scanner/scanner.py: scanner process
- Controller:
  - src/controller/controller.py: controller process
- Web:

src/web/*: web system

- Target server for scanning test

  src/target_server.py: target server process

- Others

  src/common/*: common classes

  src/config/*: configuration file, sql script

  src/log/*: log files generated during running time

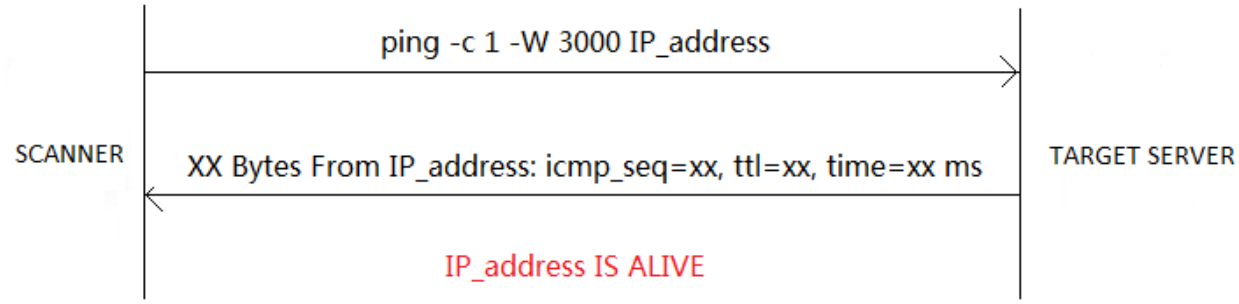
Total amount of work: 3000+ lines of code

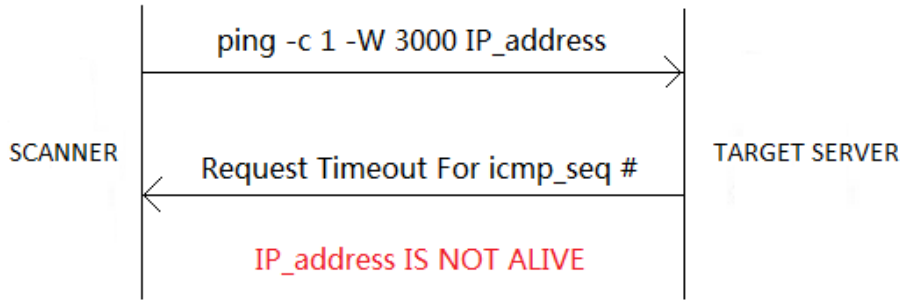
2.3. Installation, Configuration and Running

All the details are listed in src/Readme.txt

3. Scanning Theory

3.1. Check whether an IP address is alive



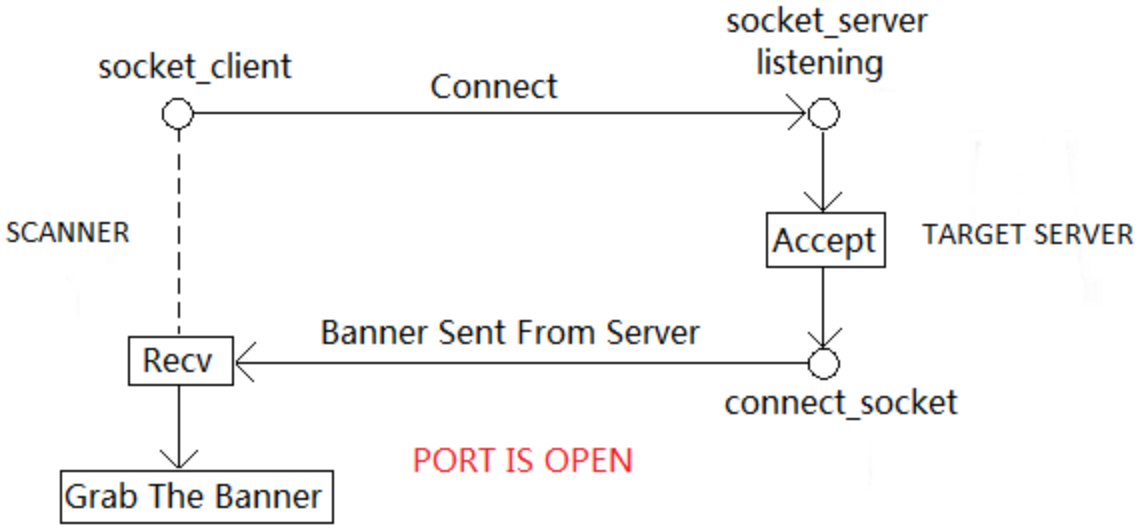


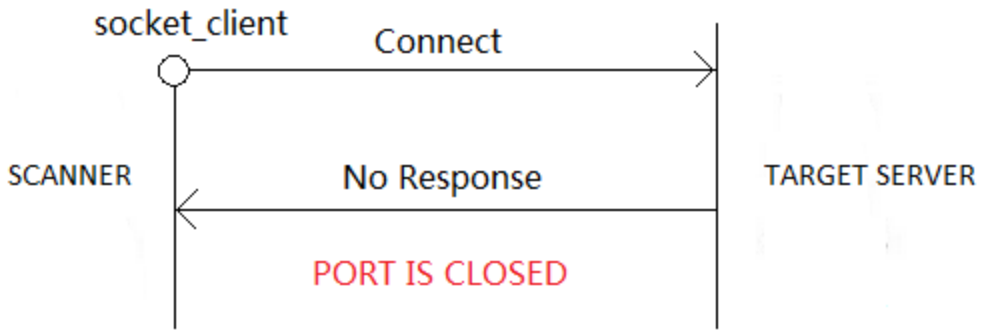
"Ping" command with a reasonable timeout (3000ms) is performed to check whether an IP address is alive. If a reply is received from the destination host, the IP address is alive, otherwise, not alive.

3.2. Check which IP addresses are alive in a block of IP addresses

Basic principle is same as above. User specifies the start and end of the IP address block. Scanners ping each IP in the designated IP range.

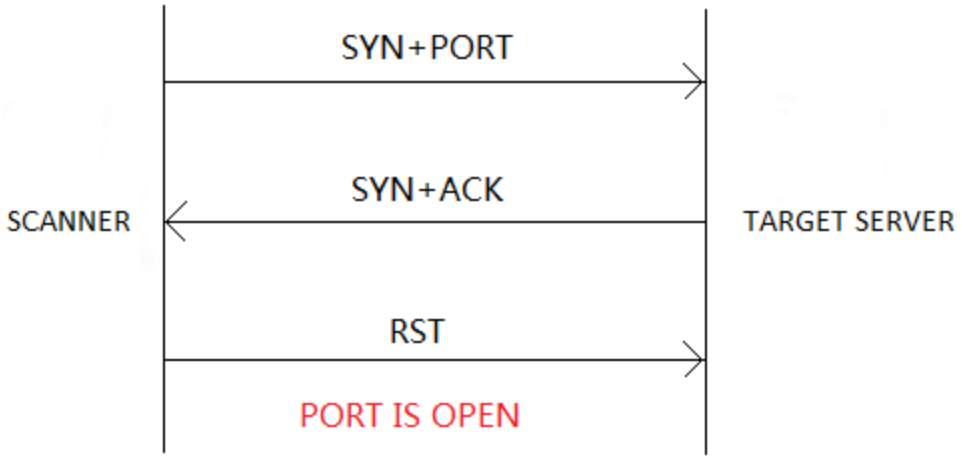
3.3. Normal Port Scanning

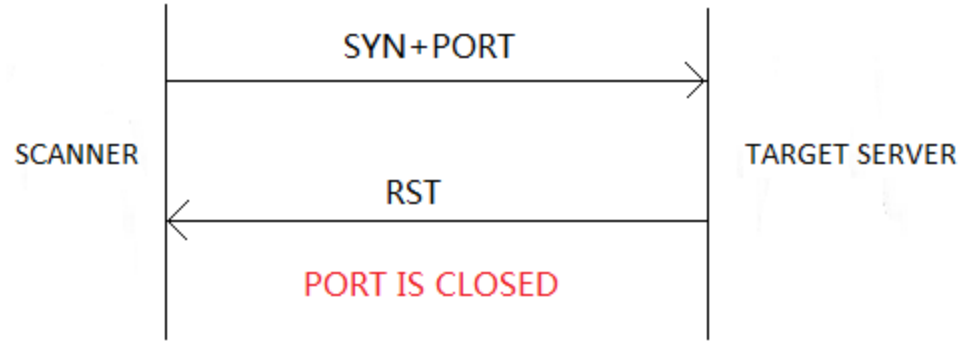




This is also known as TCP connect scanning. Scanner tries to establish connection with designated port on the target server using connect() function. If the port on server side is open, server will accept connection and create a new socket (connect socket). Later on, scanner will receive and grab the banner sent from the target server. If the port on server side is closed, there will be no response.

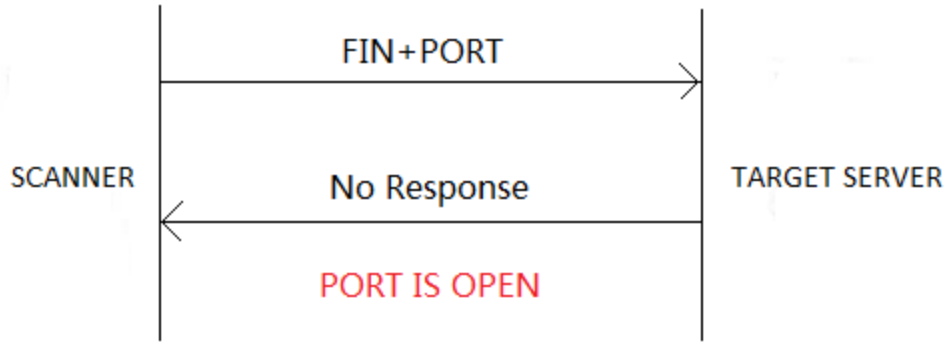
3.4. TCP SYN Scanning

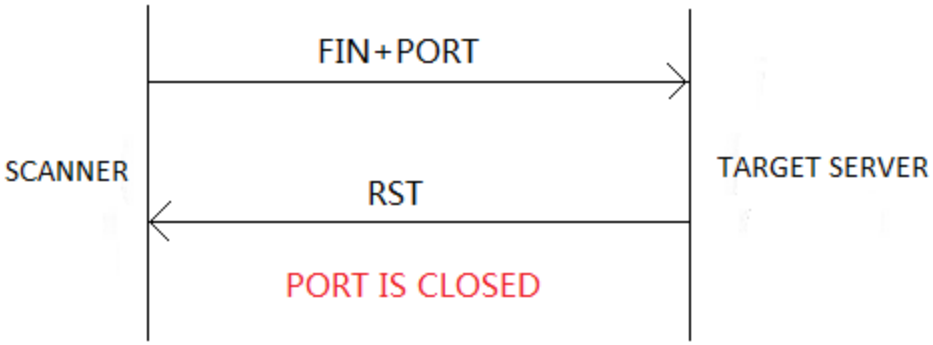




We build this scanning mode according to the working mechanism of Nmap and Python Scapy. Scanner sends a SYN packet at first. If the target server responds with a packet with both SYN and ACK bits set, it suggests that the port is open and is accepting connections. Then the scanner sends out a RST packet to close the connection. Otherwise if the server responds with a RST packet, it indicates that this particular port on server side is closed.

3.5. TCP FIN Scanning



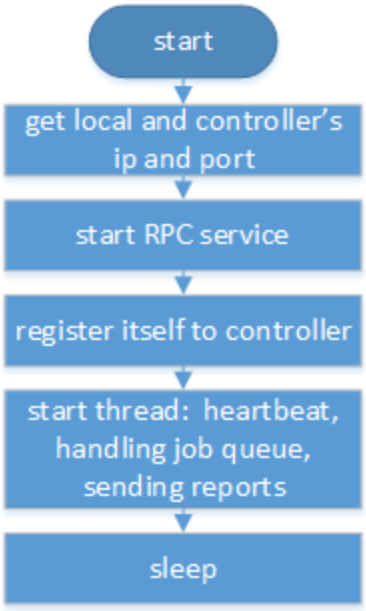


Similar to SYN scanning: scanner sends out a packet with FIN bit set in TCP layer. If the server does not respond, it means that the port is open and the server treats this as an anomalous packet, thus ignores the packet. Otherwise, if the server responds with a proper RST packet, it means the port on server side is closed.

4. Scanner

The scanner is implemented as an asynchronous multi-threaded process which includes a job queue and a report queue. Each queue has a producer and a consumer. The job queue's producer is RPC service which receives jobs from controller. And the job queue's consumer is Job Queue Thread which executes jobs. The producer of the report queue is the Job Queue Thread while the consumer is the Report Thread. The job queue is a priority queue based on the heap, using the task id as priority key (details see section 4.2.2). The report queue is a simple FIFO queue. Both queues are synchronized to ensure thread safety.

4.1. Main Thread



Main Thread Flow Diagram

The main thread reads controller's IP and port from the file configuration.ini. We suppose that the controller's IP is fixed. If the IP is assigned dynamically by DHCP, then configuration.ini needs to be modified manually.

The scanner's port is input by users with validity checks, but IP is dynamically selected. If the controller's IP is a loopback address, the scanner's IP is set to be 127.0.0.1. Otherwise, the scanner reads the IP address of "wlan0" interface (we just select WiFi environment for simplicity; for improvement, it should read all the outgoing interfaces and get an available IP list) and compares to the controller's IP. If same, the scanner is local and 127.0.0.1 will be used. If not, the scanner adopts the IP address of "wlan0" interface.

The scanner registers itself by sending its IP and port to controller. If the controller is not running, registration fails and scanner exits.

4.2. RPC Service (Remote Procedure Call)

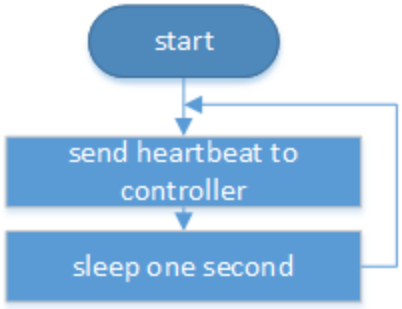
4.2.1. Alive scanner confirmation

When the controller starts, it gets a list of scanners from database of which the scanner status is "RUNNING". The reason to do so is that if the controller stops before scanners, the scanners' status in the database remains "RUNNING" and will never be updated until the controller restarts. For the scanners in the list, the controller sends messages to each of them to confirm whether they are still alive. The RPC service of alive scanner confirmation on the running scanner is to send back acknowledgement to controller.

4.2.2. Receive jobs

Each time when the RPC service of receiving jobs on the scanner side receives a job list, it adds the jobs to the job queue. Since job queue is a priority heap queue with the task id as the priority key, job with lowest task id is firstly executed and jobs with the same task id are served in FIFO order. It is desired that jobs belonged to earlier submitted tasks with lower task id could be finished earlier. Normally, jobs from early tasks reaches the scanner early, but there are exceptions that some jobs might be reassigned due to failed scanner (see section 5.3).

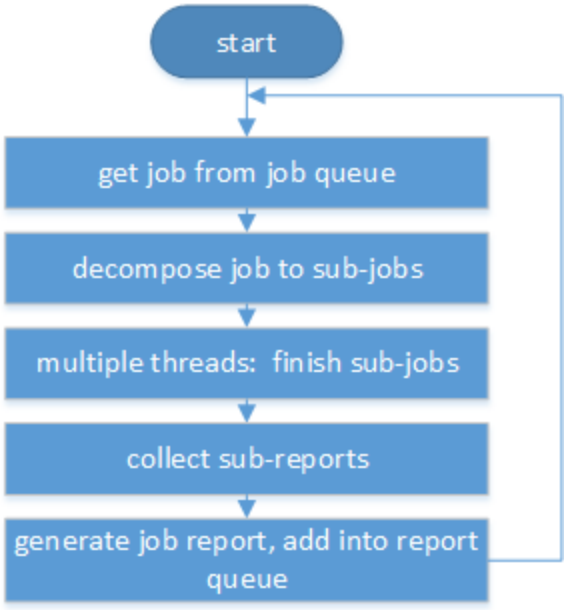
4.3. Heartbeat Thread



Heartbeat Thread Flow Diagram

The scanner sends heartbeat messages to the controller every second and receives error messages if there is any (eg. false alarms of failed scanners possibly raised at controller side which is mentioned in section 5.3)

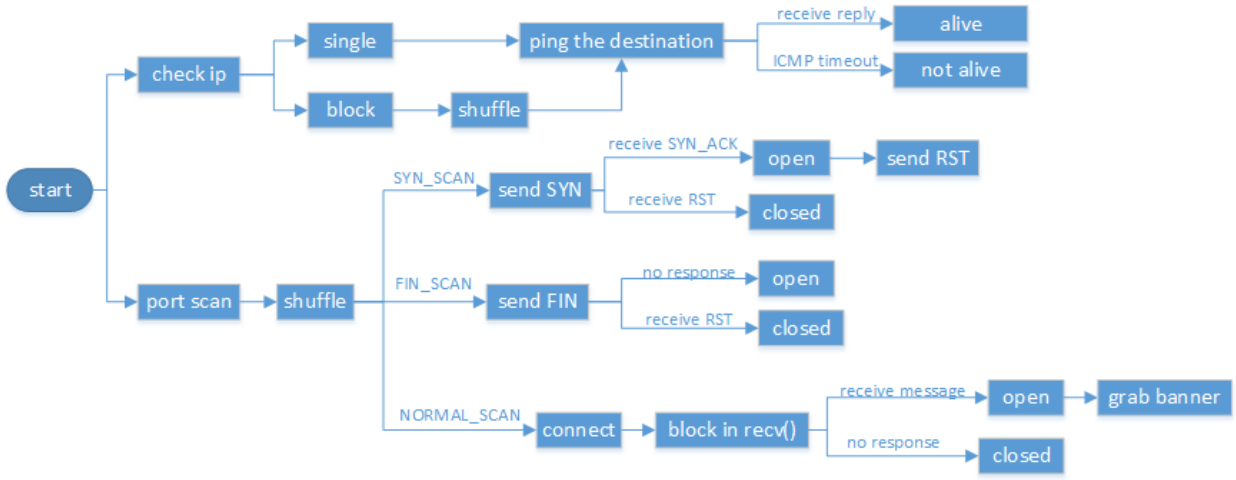
4.4. Job Queue Thread



Job Queue Thread Flow Diagram

Getting jobs from the job queue is blocking if job queue is empty. Each job is decomposed to multiple sub-jobs according to the number of scanning units of the job. One “unit” of a job is defined as scanning one IP or one port. Multiple Job Threads are spawned to finish the sub-jobs. The maximum number of Job Threads is set as five. The join() method in the class Thread is overridden in the inherited class JobThread so that all the sub-reports of sub-jobs are returned to the Job Queue Thread by join(). Finally a complete report of the job is generated from all the sub-reports and is added into the report queue.

4.5. Job Thread



Job Thread Flow Diagram

We take advantage of Python Scapy and socket programming packages to realize scanning functionality.

Scapy enables us to manually packetize the packet in different layers. In TCP SYN/FIN scanning, we make use of this characteristics to create packets with SYN/FIN bit set, send it out and wait for the response. From the response, it is easy to judge which bit is set and draw the conclusion according to the scanning theory.

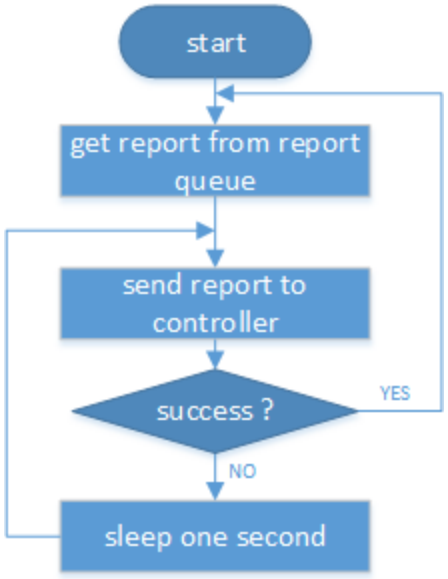
Socket programming in Python is similar to that in C language. In normal port scanning, we follow Nmap’s design to create socket, set socket options, connect to target server and grab banner.

As mentioned in the project requirement: any of the above IP/port scanning methods should be able to finish sequentially or in a random order. We calls random.shuffle() function (as shown below) in Python to make a block of IP addresses or ports order randomly.

```
def ip_order_shuffle(self, ip_start, ip_end, seq):
    ip_start = ip_start.split('.')
    ip_end = ip_end.split('.')
    ip_base = '.'.join(ip_start[0 : 3])
    s = list(range(int(ip_start[3]), int(ip_end[3]) + 1))
    if seq == const.SCAN_RAN:
        random.shuffle(s)
    return ip_base, s

def port_order_shuffle(self, port_start, port_end, seq):
    s = list(range(port_start, port_end + 1))
    if seq == const.SCAN_RAN:
        random.shuffle(s)
    return s
```

4.6. Report Thread

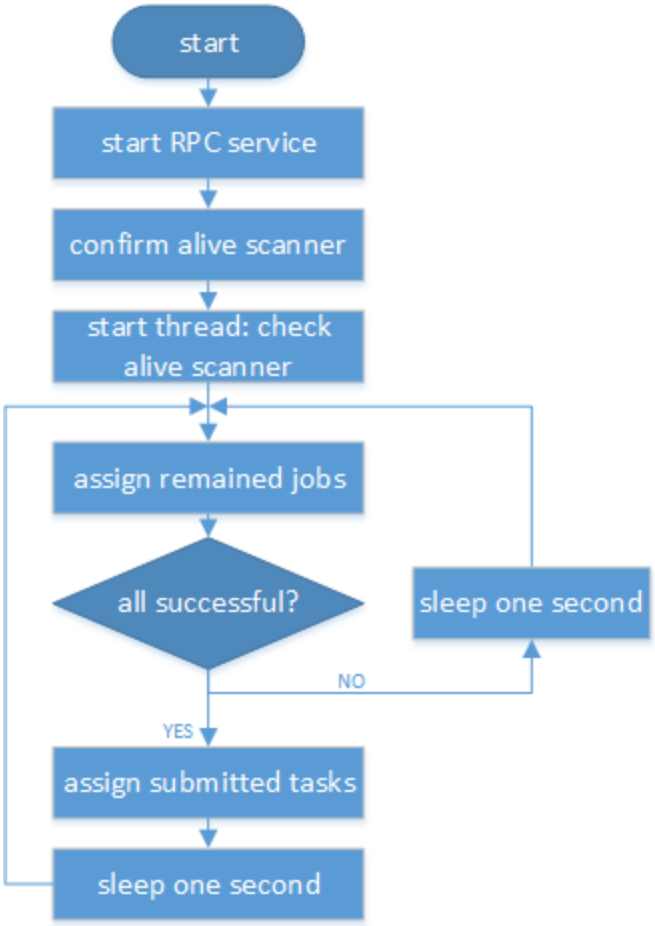


Report Thread Flow Diagram

Asynchrony between executing jobs and sending reports improves efficiency. Getting reports from the report queue is blocking if the report queue is empty. If sending fails (eg. controller shutdown, network issues), it retries every second until success. Since scanner is not persistent with all the data stored in the memory, if the scanner fails, all reports in report queue are lost. Controller then regards the jobs as unfinished and will reassign the jobs to other scanners to redo them (see section 5.3). After a report is sent, an ack from the controller will be received.

5. Controller

5.1. Main Thread



Main Thread Flow Diagram

Before entering the loop of assigning tasks and jobs, the controller firstly confirms which scanners are alive. For those scanners which do not reply acks, the controller updates their status as “STOPPED” and cleans the assignment information for all the unfinished jobs previously assigned to them.

The controller performs round polling in database at short intervals (one second) to get tasks. Each task is decomposed to multiple jobs and distributed to scanners. In the assigning loop, the remained jobs refer to the jobs sent unsuccessfully in the stage of assigning tasks or the jobs which need to be reassigned because scanners fail during the jobs running. All the remained jobs should be assigned completely before assigning newly submitted tasks in the loop, in order to make sure jobs from earlier submitted tasks are served earlier.

In the process of assigning tasks, the controller gets a list of newly submitted tasks, decomposes them to jobs with scanner assignment by a workload balance mechanism (see section 5.5) and sends the job lists to each corresponding scanner. Although a mechanism of checking alive scanners is performed, there is a short interval

between a scanner fails and the controller notices that. Thus, if a job list is sent successfully, update these jobs' assignment information in the database; if not, leave them to the stage of assigning remained jobs. As long as one job of a task has been assigned out, the task status is updated as "RUNNING", otherwise it remains "SUBMITTED" until at least one job is sent out.

In the process of assigning remained jobs, the workload balance mechanism (see section 5.5) is also performed because the workload of each scanner changes in real-time. If directly send a scanner with a job with a previously allocated size of scanning units, current workload balance between scanners might be destroyed. After a job is decomposed to multiple new jobs (maybe still one), the original job is deleted.

5.2. RPC Service (Remote Procedure Call)

5.2.1. Accept scanner register

The controller receives scanner register messages. Each scanner is identified by the composition of IP and port. If the scanner has registered before, the controller updates its status as "RUNNING" in the database and replies with its previous scanner ID. If not, the controller inserts the scanner's information into database and replies with a newly generated ID. The scanner ID is used in the heartbeat message. There is a special issue that possibly a register message is received from a scanner of which the status in the database is already "RUNNING". It occurs when a scanner stops and restarts immediately with an interval shorter than the time which the controller needs to judge a failed scanner. Under this condition, the scanner's status remains as "RUNNING" and the controller cleans the assignment information of all the unfinished jobs previously assigned to the scanner. These unfinished jobs will be reassigned in the Main Thread.

5.2.2. Receive heartbeat

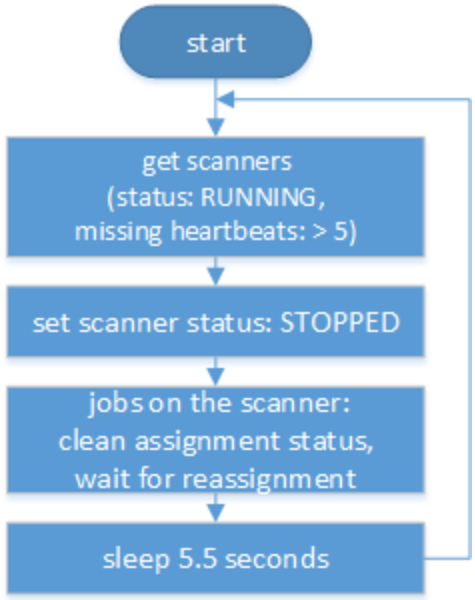
The controller receives heartbeat messages and checks database whether the scanner's status is "RUNNING". If so, update the scanner's heartbeat report time; if not, inform the scanner to restart manually and register again. The latter circumstance only takes place when the scanner was falsely judged as stopped (see section 5.3) and in our test environment it never happens.

5.2.3. Receive reports

The controller receives report messages, and check whether the report's job id exists in the database in case that the scanner was falsely judged as stopped and the job has been reassigned. If the job no longer exists, ignore the report; otherwise, insert the report into database and update the corresponding job's status as "FINISHED". Moreover, the controller checks the task from which the job is decomposed from. If all

the jobs of the task are finished, the task's status will be updated as "FINISHED". All these issues are done in one database transaction to guarantee data consistency. At last, an ack is replied to the scanner.

5.3. Check Alive Scanner Thread

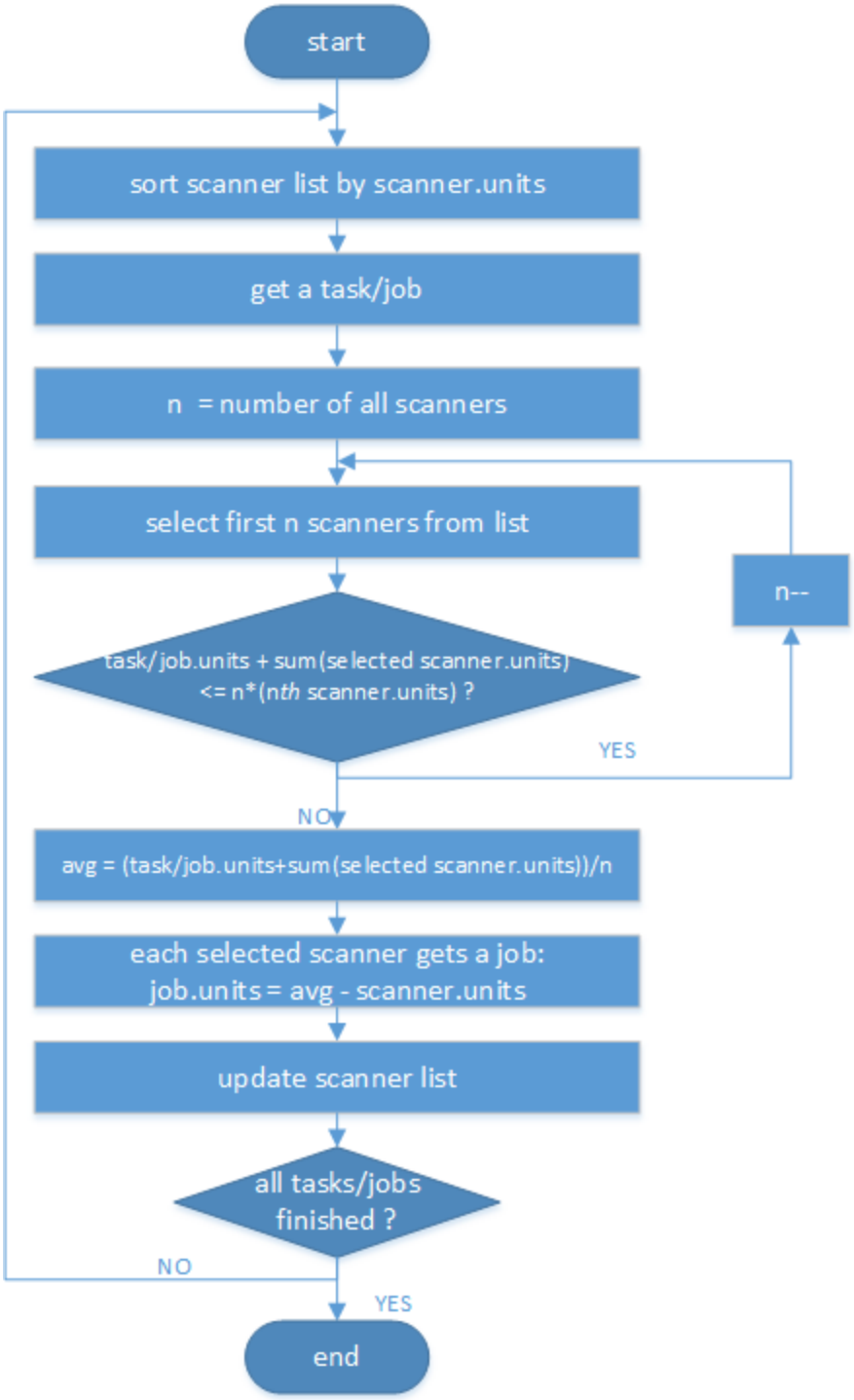


Check Alive Scanner Thread Flow Diagram

At intervals of 5.5 seconds, the controller checks heartbeat report time of all the running scanners in database. For a scanner whose latest report time is earlier than five seconds ago, the controller concludes the scanner has stopped. Thereby, the controller updates the scanner's status as "STOPPED" and cleans the assignment information of all the jobs currently running on that scanner. These unfinished jobs will be reassigned to other scanners in the Main Thread.

As our test environment does not involve wide-area networks or high network congestion, this mechanism works fine. But in reality, a false alarm of stopped scanner may be raised, which means probably a little time later, the "stopped" scanner gets in touch with the controller again. However, whenever a scanner is judged to be stopped, all the unfinished jobs on the scanner will be decomposed to new sub-jobs with new job IDs, and the original jobs are deleted from database. It's hard to track the current running jobs on the "stopped" scanner. When receiving heartbeats and reports, such exceptions are handled with simple solutions (see section 5.2.2 & 5.2.3), and a desired improved solution is proposed in section 9.2.

5.4. Workload Balance



Workload Balance Flow Diagram

In the flow diagram, scanner.units refers to current number of scanning units of all running jobs on that scanner; task/job.units refers to number of scanning units within the task/job.

The goal of workload balance is distributing jobs evenly in order to efficiently utilize scanners. For instance, currently there are 4 scanners: A with all running jobs of 10 scanning units, B with 20, C with 30 and D with 40. Then a task is submitted with 46 scanning units. The result would be 25 units assigned to A, 15 units assigned to B and last 6 units assigned to C. Now the scanners' workload is as follows: A with 35 units, B with 35, C with 36 and D with 40.

6. Database

6.1. Scanner Table

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI
id	INT(11)	☑	☑	☐	☐	☐	☐	☑
ip	VARCHAR(15)	☐	☑	☐	☐	☐	☐	☐
port	INT(11)	☐	☑	☐	☐	☐	☐	☐
status	ENUM('RUNNING','STOPPED')	☐	☑	☐	☐	☐	☐	☐
update_time	TIMESTAMP	☐	☐	☐	☐	☐	☐	☐

6.2. Task Table

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI
id	INT(11)	☑	☑	☐	☐	☐	☐	☑
type	ENUM('IP_SCAN','IP_BLOCK_SCAN','NORMAL_SCAN','SYN_SCAN','FIN_SCAN')	☐	☑	☐	☐	☐	☐	☐
status	ENUM('SUBMITTED','RUNNING','FINISHED')	☐	☑	☐	☐	☐	☐	☐
ip	VARCHAR(15)	☐	☐	☐	☐	☐	☐	☐
ip_end	VARCHAR(15)	☐	☐	☐	☐	☐	☐	☐
port_begin	INT(11)	☐	☐	☐	☐	☐	☐	☐
port_end	INT(11)	☐	☐	☐	☐	☐	☐	☐
seq_ran	ENUM('SEQ','RAN')	☐	☐	☐	☐	☐	☐	☐
submit_time	TIMESTAMP	☐	☐	☐	☐	☐	☐	☐
finish_time	TIMESTAMP	☐	☐	☐	☐	☐	☐	☐

6.3. Job Table

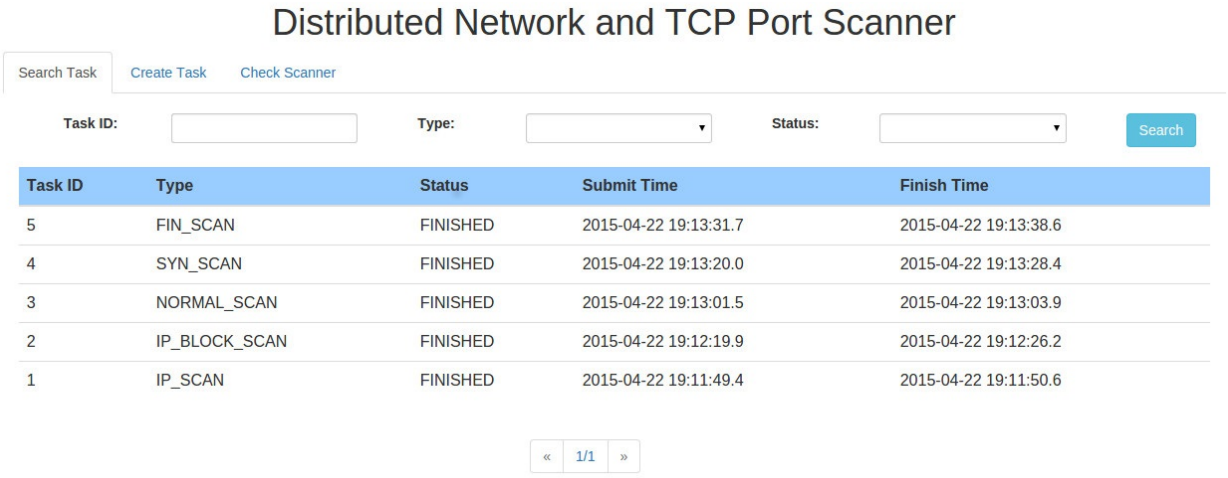
Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI
id	INT(11)	☑	☑	☐	☐	☐	☐	☑
task_id	INT(11)	☐	☑	☐	☐	☐	☐	☐
scanner_id	INT(11)	☐	☐	☐	☐	☐	☐	☐
type	ENUM('IP_SCAN','IP_BLOCK_SCAN','NORMAL_SCAN','SYN_SCAN','FIN_SCAN')	☐	☐	☐	☐	☐	☐	☐
status	ENUM('RUNNING','FINISHED')	☐	☐	☐	☐	☐	☐	☐
ip	VARCHAR(15)	☐	☐	☐	☐	☐	☐	☐
ip_end	VARCHAR(15)	☐	☐	☐	☐	☐	☐	☐
port_begin	INT(11)	☐	☐	☐	☐	☐	☐	☐
port_end	INT(11)	☐	☐	☐	☐	☐	☐	☐
seq_ran	ENUM('SEQ','RAN')	☐	☐	☐	☐	☐	☐	☐
num	INT(11)	☐	☐	☐	☐	☐	☐	☐
assign_time	TIMESTAMP	☐	☐	☐	☐	☐	☐	☐
finish_time	TIMESTAMP	☐	☐	☐	☐	☐	☐	☐

6.4. Report Table

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI
id	INT(11)	☑	☑	☐	☐	☐	☐	☑
task_id	INT(11)	☐	☑	☐	☐	☐	☐	☐
job_id	INT(11)	☐	☑	☐	☐	☐	☐	☐
ip	VARCHAR(15)	☐	☐	☐	☐	☐	☐	☐
port	INT(11)	☐	☐	☐	☐	☐	☐	☐
res	ENUM('ON','OFF','FILTER')	☐	☐	☐	☐	☐	☐	☐
banner	TEXT	☐	☐	☐	☐	☐	☐	☐
update_time	TIMESTAMP	☐	☐	☐	☐	☐	☐	☐

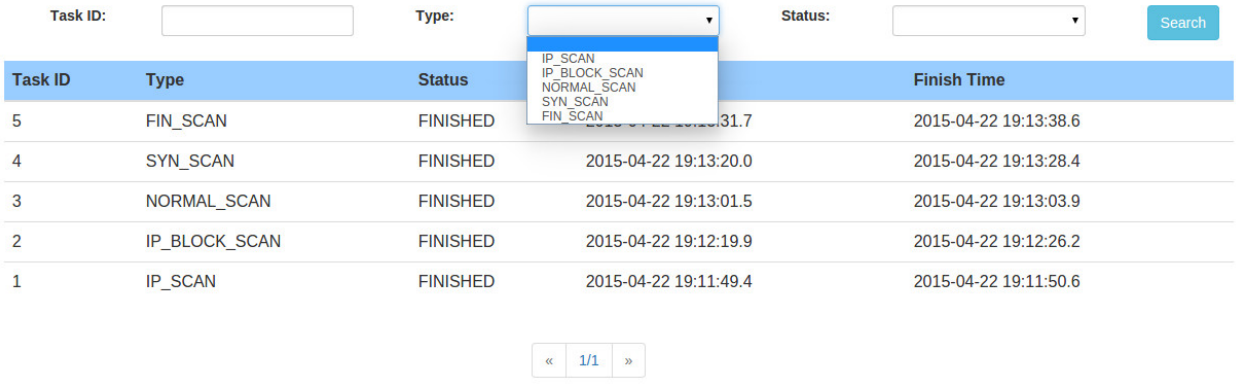
7. Web UI

7.1. Index

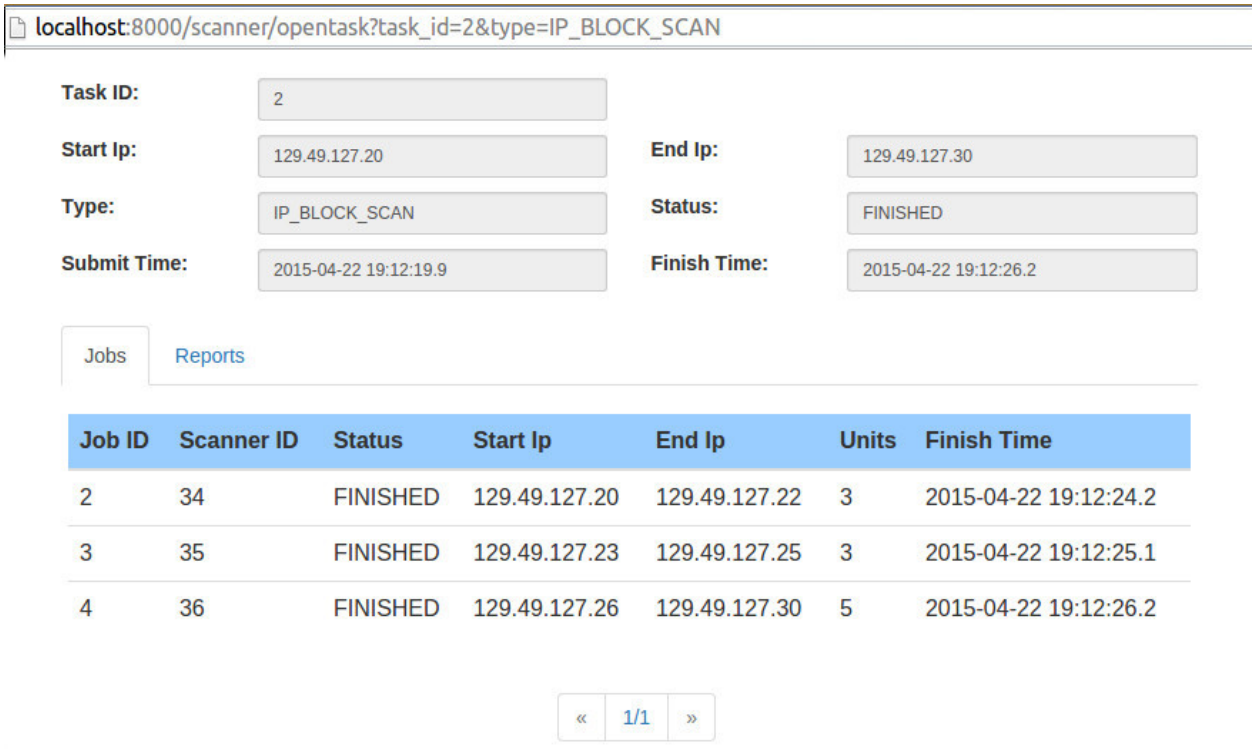


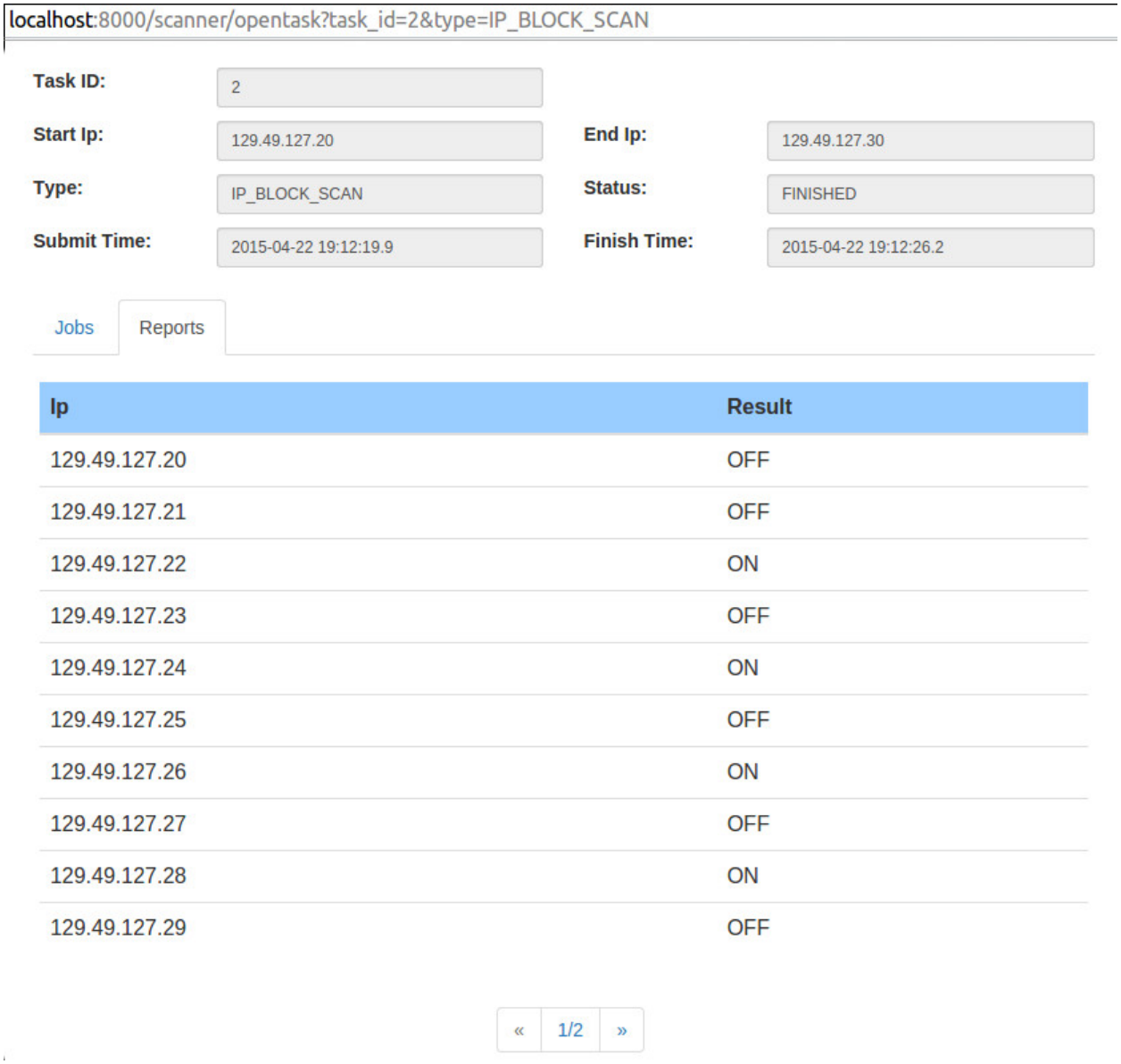
The web system consists of three modules: searching tasks, creating tasks and checking scanners' performance.

7.2. Search Task



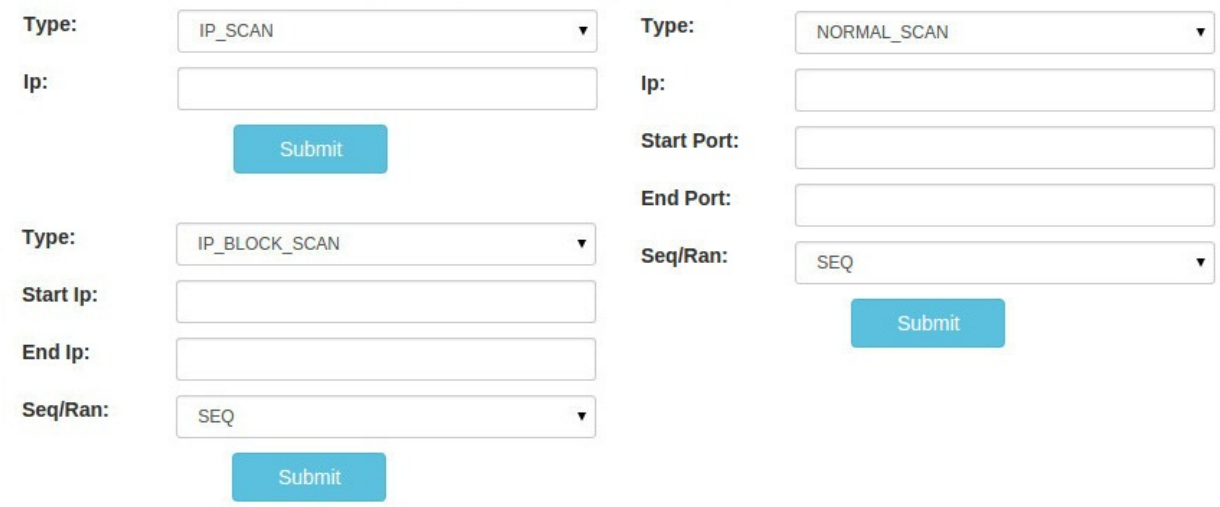
Tasks could be searched by task id, type and status. Each page of the task list contains ten entries. Double clicking each task entry opens a new window showing the task's details and report as the following two images.





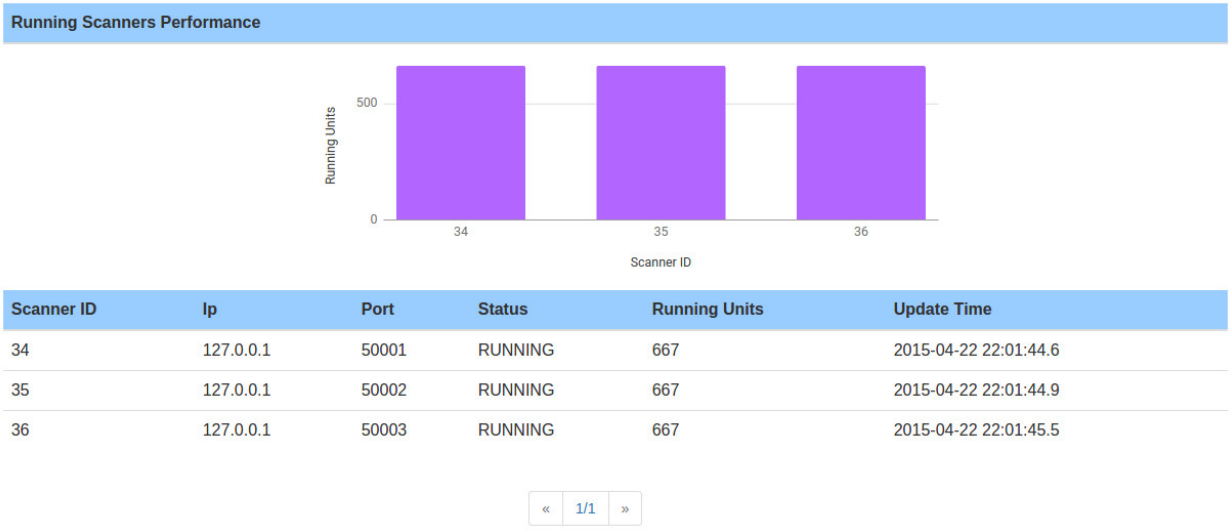
Take an IP block scanning task as an example. The job tab page presents a list of jobs decomposed from a task and the report tab page presents all scanning results for the task.

7.3. Create Task



The content of “create task” tab page dynamically changes with the user’s selection of task type. Page contents for normal port scan, syn scan and fin scan are the same. The validity checking of input IP and port is performed both at the front end with Javascript and the back end with Django, in case that users arbitrarily avoid javascript checking and submit the form. If checking fails, the form will not be submitted and corresponding alert window pops up.

7.4. Check Scanner



Scanner ID	Ip	Port	Status	Running Units	Update Time
34	127.0.0.1	50001	RUNNING	667	2015-04-22 22:01:44.6
35	127.0.0.1	50002	RUNNING	667	2015-04-22 22:01:44.9
36	127.0.0.1	50003	RUNNING	667	2015-04-22 22:01:45.5

« 1/1 »

In the “check scanner” tab page, all the registered scanners are listed with their current status and number of running job units. As mentioned earlier, one job unit refers to scanning one IP or one port. The bar chart displays the real-time performance of workload balance between running scanners.

8. Experiments

8.1. Target Server

We design a concurrent TCP server as a test case, which is able to bind multiple ports and listen on them (using select()). When receiving connect requests from scanners, it displays the scanner's IP address and port, and then responds to them with a welcome message "Hello + host IP + current time" as the banner.

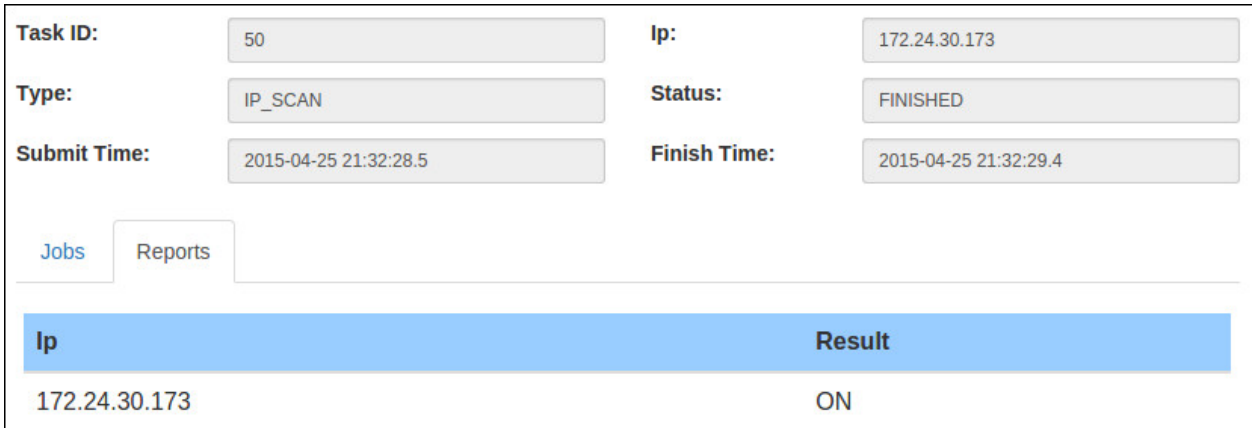
```
Fangyus-MacBook-Pro:cse508 fydeng$ python test_server.py 3 9811 9813 9815
Host ip address is 172.24.30.173
Host port #0 is 9811
Host port #1 is 9813
Host port #2 is 9815
Now listening
Connection from 172.24.30.200:57738
Now listening
Connection from 172.24.30.200:52088
Now listening
Connection from 172.24.30.200:52132
Now listening
```

As is shown above, the server listens on 3 ports: 9811, 9813, 9815. Each receives a connection from a scanner.

8.2. Results

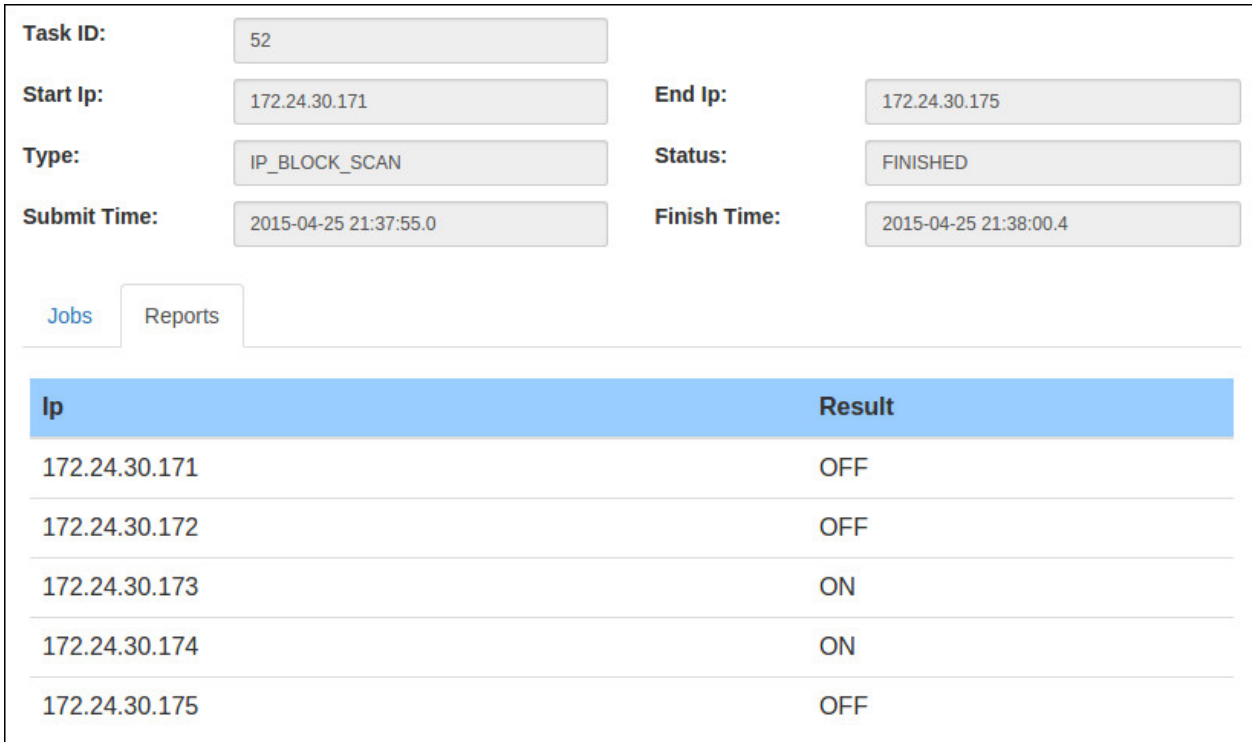
8.2.1. Check whether an IP address is alive:

Target IP: 172.24.30.173



8.2.2. Find which IP addresses are alive in a block of IP addresses

Target IP range: 172.24.30.171 - 172.24.30.175



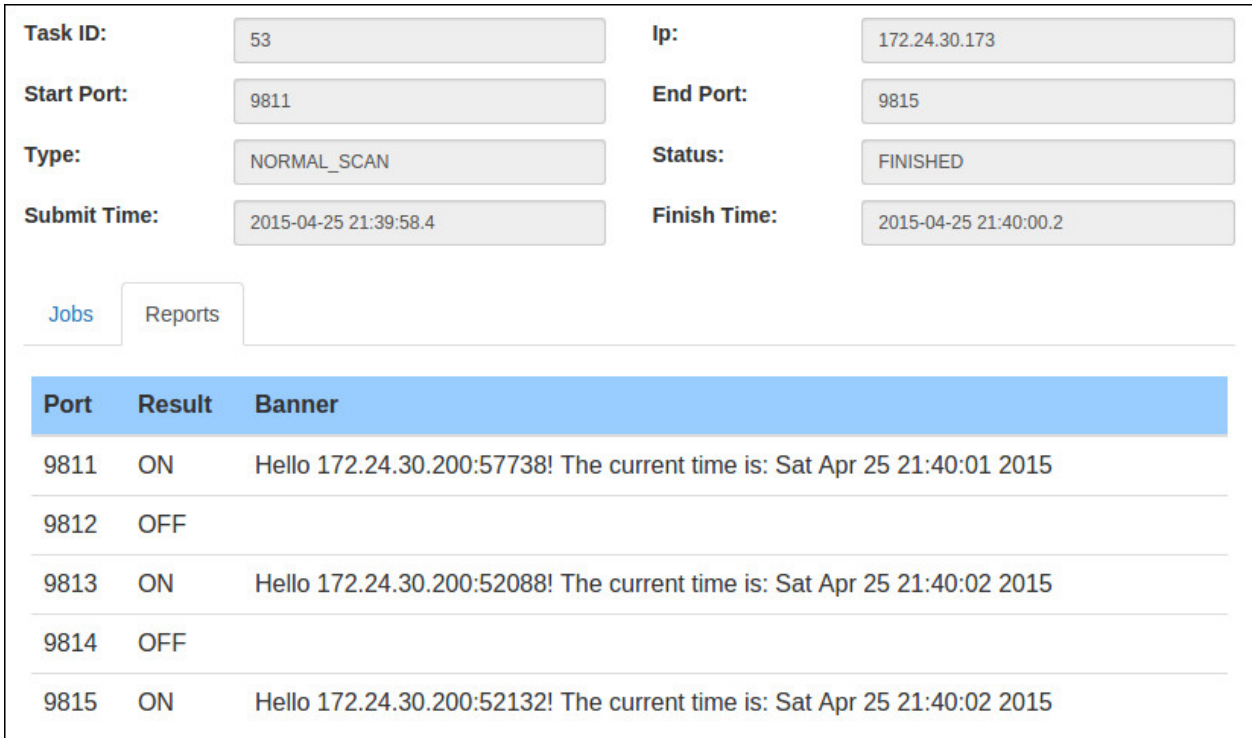
Task ID:	52		
Start Ip:	172.24.30.171	**End Ip:**	172.24.30.175
Type:	IP_BLOCK_SCAN	**Status:**	FINISHED
Submit Time:	2015-04-25 21:37:55.0	**Finish Time:**	2015-04-25 21:38:00.4

Jobs | Reports

Ip	Result
172.24.30.171	OFF
172.24.30.172	OFF
172.24.30.173	ON
172.24.30.174	ON
172.24.30.175	OFF

8.2.3. Normal Port Scanning

Target IP: 172.24.30.173 Target port range: 9811 - 9815



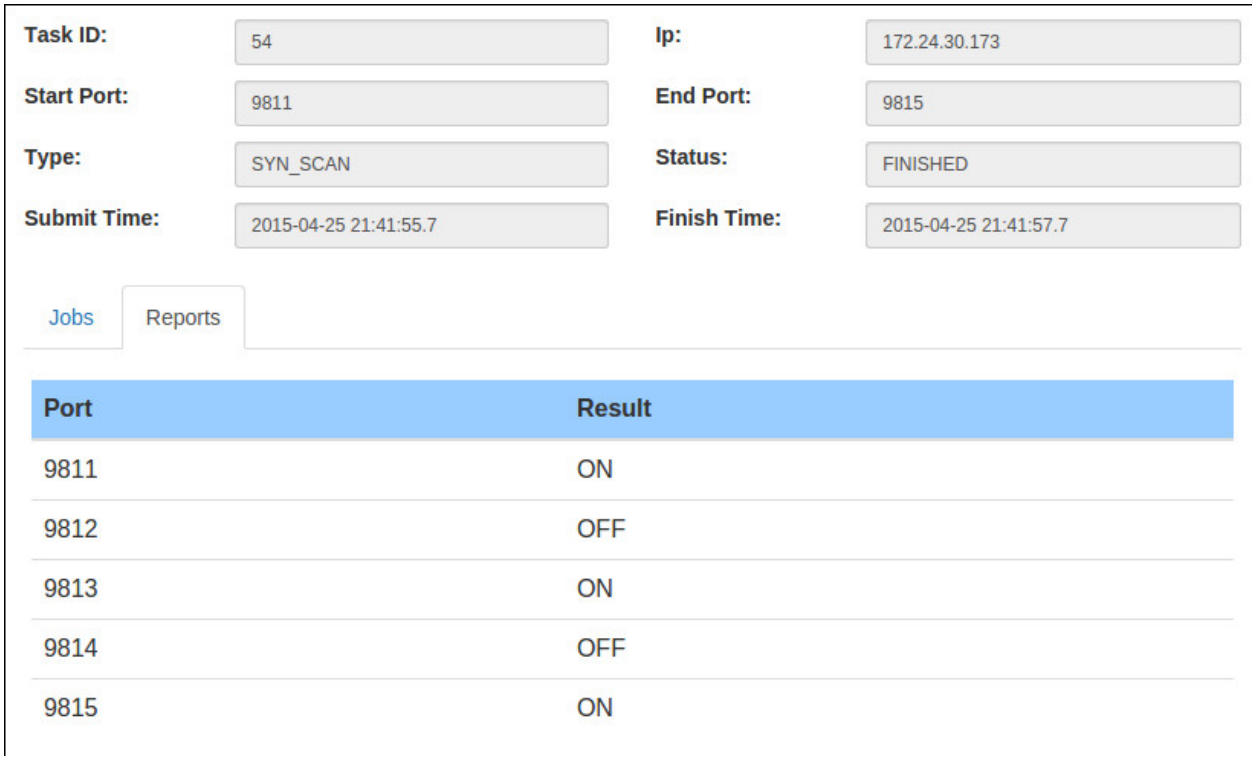
Task ID:	53	**Ip:**	172.24.30.173
Start Port:	9811	**End Port:**	9815
Type:	NORMAL_SCAN	**Status:**	FINISHED
Submit Time:	2015-04-25 21:39:58.4	**Finish Time:**	2015-04-25 21:40:00.2

Jobs | Reports

Port	Result	Banner
9811	ON	Hello 172.24.30.200:57738! The current time is: Sat Apr 25 21:40:01 2015
9812	OFF	
9813	ON	Hello 172.24.30.200:52088! The current time is: Sat Apr 25 21:40:02 2015
9814	OFF	
9815	ON	Hello 172.24.30.200:52132! The current time is: Sat Apr 25 21:40:02 2015

8.2.4. TCP SYN Scanning

Target IP: 172.24.30.173 Target port range: 9811 - 9815



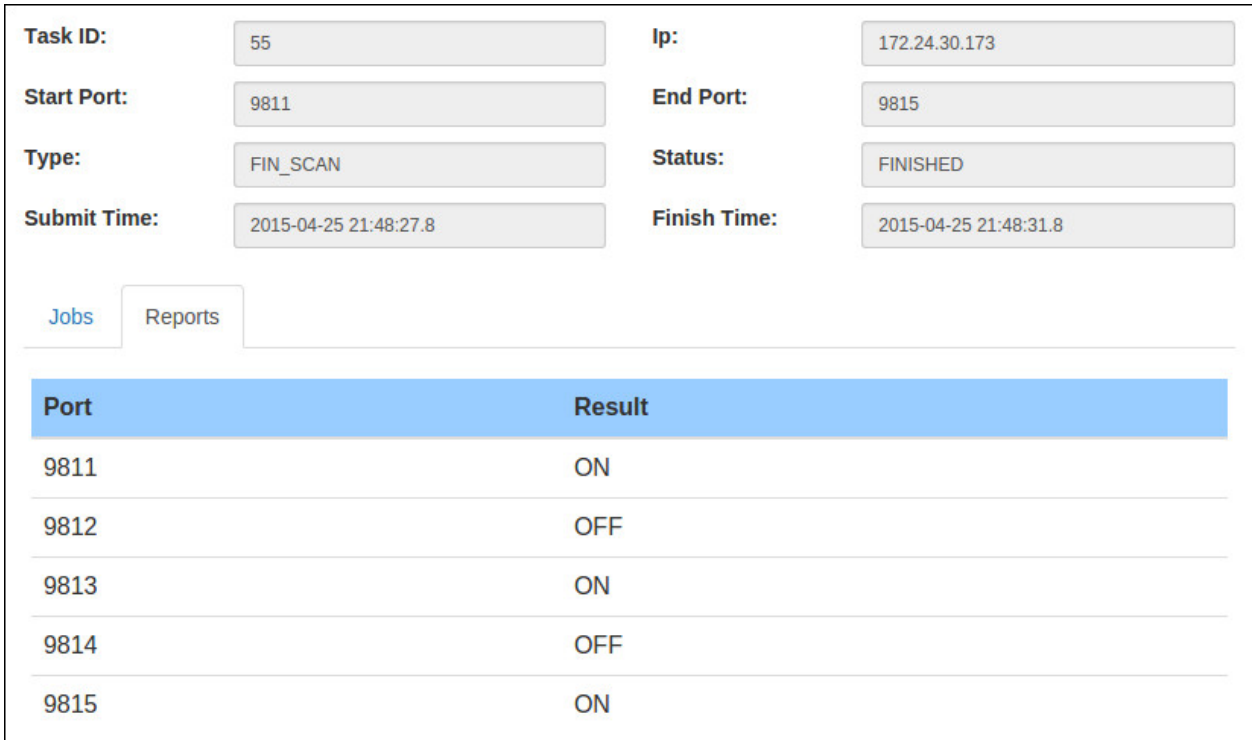
Task ID:	54	**Ip:**	172.24.30.173
Start Port:	9811	**End Port:**	9815
Type:	SYN_SCAN	**Status:**	FINISHED
Submit Time:	2015-04-25 21:41:55.7	**Finish Time:**	2015-04-25 21:41:57.7

Jobs Reports

Port	Result
9811	ON
9812	OFF
9813	ON
9814	OFF
9815	ON

8.2.5. TCP FIN Scanning

Target IP: 172.24.30.173 Target port range: 9811 - 9815



Task ID:	55	**Ip:**	172.24.30.173
Start Port:	9811	**End Port:**	9815
Type:	FIN_SCAN	**Status:**	FINISHED
Submit Time:	2015-04-25 21:48:27.8	**Finish Time:**	2015-04-25 21:48:31.8

Jobs Reports

Port	Result
9811	ON
9812	OFF
9813	ON
9814	OFF
9815	ON

9. Future Improvement

9.1 Dealing with firewalls/filter

This is a particular case in port scanning, especially in SYN/FIN scanning. In order to make our program robust, we take this issue into account in our SYN/FIN scanning program: In SYN scanning, if the server does not respond or the responded packet does not contain ICMP layer, currently we treat the scanning as being filtered; In FIN scanning, if server responds with a packet containing ICMP layer, we also treat the scanning as being filtered. This is an in-depth area in port scanning and is still incurring extensive research. Professor mentioned that it is not required in the project. We touch this field and have tried to search lots of resource just to make our program more complete. In the future, we plan to deal with filtered case more in detail, for example, deal with different ICMP type and code respectively.

9.2. Heartbeat Network

We make an assumption that the controller can tolerate five continuously missed heartbeats from a scanner, thus if the sixth missed heartbeat is detected, the scanner is diagnosed to be failed. However, it is possible that the unreachability is due to network communication problem but not the scanner failure, which leads to a false alarm.

A desired solution is as follows but not implemented:

If the controller receives heartbeats from a scanner whose status in the database is “STOPPED”, the controller should inform the scanner to stop the current running job (involve sub-jobs in multi-threads) and cancel all the waiting jobs in the job queue so that the scanner is free to accept new jobs. Thus, the falsely judged “STOPPED” scanner does not need to restart manually if the job cancellation mechanism is implemented. The mode of handling received reports with job IDs not existing in the database remains not changed.

9.3. Workload Rebalance

Currently, workload balance only takes place at the stage of assigning tasks/jobs. If a scanner just joins and there is no newly submitted tasks or remained jobs, the scanner does not share others’ workload. Besides, the time needed to finish one job unit is different because some scanning modes need to wait until timeout to get a result. So a better mechanism is needed to realize workload rebalance between scanners in real-time by monitoring their current workloads.

10. Reference

Nmap: http://nmap.org/
Scapy: http://www.secdev.org/projects/scapy/doc/usage.html
Python socket programming: https://docs.python.org/2/howto/sockets.html
gRPC: https://github.com/grpc/grpc
Protocol Buffer: https://developers.google.com/protocol-buffers/
Django: https://docs.djangoproject.com/en/1.7/