CSE-509 SYSTEM SECURITY PROJECT

Project 2: Distributed Network and TCP Port Scanner with Web UI

Team Script Buddies

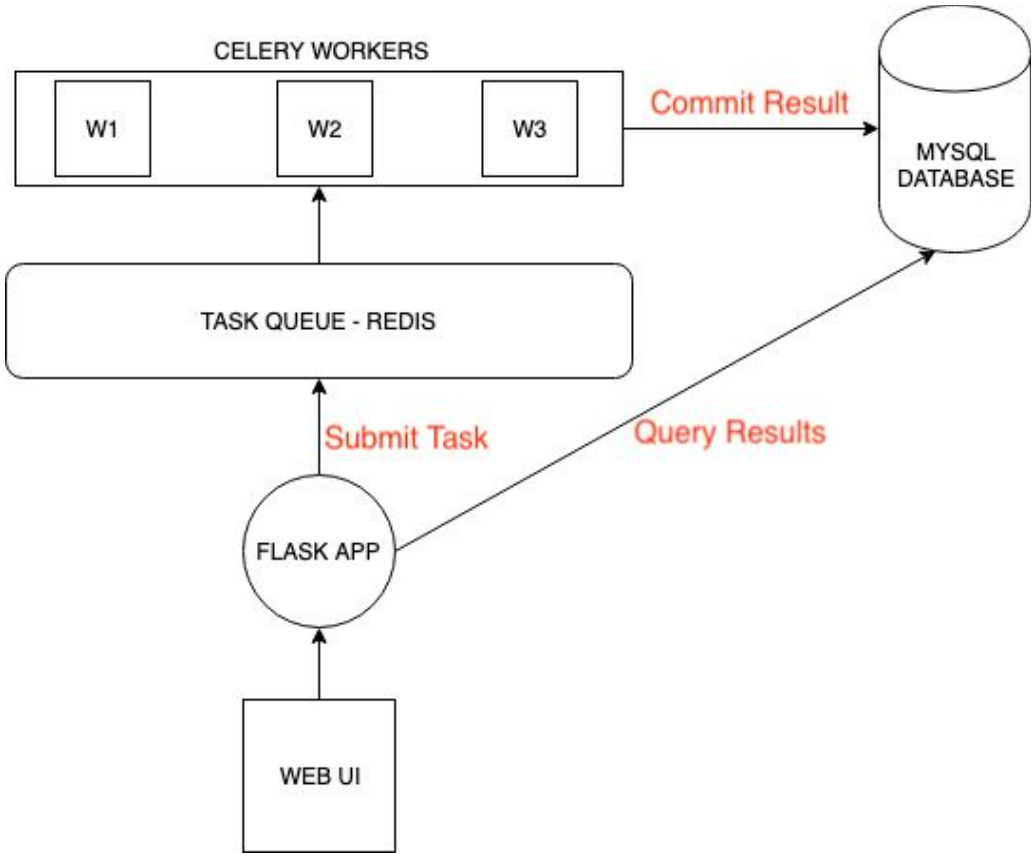
Team Members:

Raveendra Soori - 111498402 (raveendra.soori@stonybrook.edu)

Sanjay Thomas - 112026123 (sanjay.mathewthomas@stonybrook.edu)

Varun Hegde - 111986703 (varun.hegde@stonybrook.edu)

High Level Architecture



The port scanning process is a time consuming process and it is I/O bound. So it is important to decouple the producers of task from its consumers. Executing the task on the request thread is simply not an option for the following reasons:

- It can take a long amount of time and connections can time out, and the user is kept waiting.
- The thread serving the request could randomly crash.

Celery

So we chose Celery which is an Asynchronous Job Queue. The architecture of Celery consists of a client accepting a large request, breaking down the request into smaller tasks and distributing them among its worker processes. The client program distributes the tasks by loading them into a message queue. From here, free workers can retrieve tasks, process them and store the results into the database for later retrieval by the client. The use of a message queue allows for fault tolerance where if a worker is seen to be unresponsive for a given amount of time, the task assigned to it would be loaded back into the message queue for another worker

to take up. Celery uses heartbeats sent by the workers to determine It serves both our goal of decoupling the Producer from the Consumer as well and provides several fault tolerance mechanisms out of the box.

Task Granularity

An important consideration is the granularity of the task assigned to each worker. At the smallest level, this could be a single IP and Port pair assigned to a worker while at the other end, we could assign ‘n’ IP, Port pairs to each worker. The former case would result in poor task distribution as the workers would have to more frequently interact with the Task Queue, but it would also mean the user is notified about the scans more frequently. Thus, the decision on the granularity is a tussle between optimizing the performance and keeping the user updated. In our current implementation, we assign 5 sets of IP, Port pairs to each worker.

Ordering of Tasks

By default, there is no determinism in the order of port scanning, because the workers pick it up on their own discretion. But Celery offers the concept of group where tasks will be applied one after another in the current process.

Result Backend and Task Associations

We have used MySQL database as our result backend. The Celery workers upon completing their unit of work commit the results to the result backend. The current task granularity we had set for Celery would result in the user request being split into a set of smaller tasks. It is paramount that we maintain an association between the user request(master task) with the smaller Celery tasks. This is useful for result aggregation once the entire task is completed. We use the @task_prerun hook, which is executed before the start of a Celery Task to perform the above-mentioned association.

Workflow

1. User Submits a Port Scanning Request through the Web Browser.
2. The request is sent to the Flask Application Server which places the request onto the task queue. In our case the task queue is Redis.
3. The Celery Workers pick up the request from the Task Queue.
4. The individual workers upon finishing their unit of work submits the result to a Result Backend, which is a MySQL database in our case.
5. The Flask Application can query the data from the database and pass it onto the client.

Port Scanning

We are using scapy for performing the port scanning. **Scapy** is a packet crafting library in Python.

Check Alive Hosts

To check if a host is alive, we start by issuing an ICMP request. If we get an ICMP response other than the type Destination Unreachable(type 3), then we know that the host is alive. Again if ICMP is turned off or if the ICMP packets filtered by the firewalls we wouldn't know for sure the real status of the host. So as a fallback mechanism we examine if there are any common services such as FTP, SSH, SMTP, DNS, HTTP, HTTPS, etc running on their designated ports. If we are successfully able to establish a connection to the sockets of these remote hosts we know that the host is alive.

Normal TCP Scan and Banner Grabbing

We use the connect() system call to open a connection to the desired port on the host machine. If the port is listening, connect() will succeed, otherwise, the port isn't reachable. Since this involves dealing with stream sockets there is no need for root privileges. This scan is easily detectable and filterable since the target hosts log the connection. **Banner grabbing** involves finding the services running on the identified open ports. We send random input on the open connection and capture the banner sent by the service. For HTTP service we send a get request on the open connection to capture the banner.

SYN Scan

This technique is often referred to as "half-open" scanning. Here we do not open a full TCP connection. You send an SYN packet and wait for a response. An SYN|ACK indicates the port is listening. An RST is indicative of a non- listener. If an SYN|ACK is received, you immediately send an RST to tear down the connection (actually the kernel does this for us). The primary advantage of this scanning technique is that fewer sites will log it. Since constructing SYN packets involves dealing with raw sockets, we need root privileges.

FIN Scan

Some firewalls and packet filters watch for SYNs to restricted ports. The idea is that closed ports tend to reply to your FIN packet with the proper RST. Open ports, on the other hand, tend to ignore the packet in question. They send RST's regardless of the port state, and thus they aren't vulnerable to this type of scan.

Instructions to set up the Project

Note: Please use a modern version of the Chrome browser to test the application. Older browsers do not support Javascript ES6 syntax.

1. Install **docker** and **docker-compose**.
2. git clone **repository**
3. cd **repository**
4. **docker-compose build** - Builds the images
5. **docker-compose up -d** - Start the services
6. **docker-compose down** - Stop the services

The application should be up on port **5001**.

We have hosted our application on this url http://18.222.174.193:5001/.

Task Distributions

- Raveendra Soori - UI , SYN Scan, FYN Scan
- Sanjay Mathew Thomas - UI, Celery Task Distribution, Fault Tolerance
- Varun Hegde - Port Alive, Banner Grabbing, Deployment, Celery Task Distribution

References

- **https://nmap.org/nmap_doc.html**
- **https://scapy.net/**
- **http://www.celeryproject.org/**
- **http://flask.pocoo.org/**
- **https://thepacketgeek.com/scapy-p-10-emulating-nmap-functions/**
- **https://github.com/vishnubob/wait-for-it**
- **https://stackoverflow.com/questions/4685217/parse-raw-http-headers**