Computer Networks: Assignment 2

Nimitt (22110169) & Tapananshu Gandhi (22110270)

A. Connectivity Check

For Part A, we built the specified network topology using Mininet's Python API. We created a custom `Topo` class that adds all nine nodes: the four hosts (H1-H4), the four switches (S1-S4), and the dedicated `dns` host (`10.0.0.5`).

The most critical part was configuring the links. We used Mininet's `TCLink` interface, which allowed us to set the **bandwidth (`bw`)** and **delay** for each link precisely as shown in the assignment diagram (e.g., `bw=100`, `delay='5ms'`).

After starting the network, we ran `net.pingAll()` to confirm full connectivity between all nodes, which passed successfully.

```
*** Ping: testing ping reachability
dns → h1 h2 h3 h4 nat0
h1 → dns h2 h3 h4 nat0
h2 → dns h1 h3 h4 nat0
h3 → dns h1 h2 h4 nat0
h4 → dns h1 h2 h3 nat0
nat0 → dns h1 h2 h3 h4
*** Results: 0% dropped (30/30 received)
```

B. Default Resolver

Our goal was to use the Mininet VM's default DNS resolver for a baseline. However, we faced two core issues:

1. **Namespace Isolation:** Mininet hosts (like `h1`) run in an isolated network namespace and cannot directly access the VM's internal resolver at `127.0.0.53`.
2. **VM DNS Failure:** The VM's local DNS service (`systemd-resolved`) was "stuck," likely from a stale VM image configuration, and was unresponsive. Furthermore, `h1`'s overly broad routing table (`10.0.0.0/8`) incorrectly identified

the VM's *real* DNS server (`10.0.2.3`) as local, causing an ARP timeout instead of using the gateway.

We implemented a proxy bridge to solve the isolation and configuration issues.

1. **Resolver Discovery:** We manually tested the VM's network interfaces and discovered a working, non-public DNS server at `10.0.2.3`, provided by the VirtualBox "NAT" network.
2. **Proxy Bridge:** We used `socat` to create a proxy. This proxy was launched from our Python script *after* `net.start()` and was configured to:
   - **Listen** on the Mininet host's gateway IP (`10.0.0.6`).
   - **Forward** any DNS packets it received to the working resolver at `10.0.2.3`.
3. **Final Configuration:** We configured the Mininet hosts (`h1-h4`) to use the `socat` listener's IP (`10.0.0.6`) as their default nameserver. This successfully routed their queries across the namespace barrier to the VM's default resolver.

```
Starting DNS lookups for 100 do
mains...
Results for h1:
    - Successfully Resolved: 76
    - Failed Resolutions:    24
    - Average Lookup Latency: 116.
95 ms
    - Average Throughput:    3.94
queries/sec

Starting DNS lookups for 100 do
mains...
Results for h2:
    - Successfully Resolved: 71
    - Failed Resolutions:    29
    - Average Lookup Latency: 98.
89 ms
    - Average Throughput:    2.93
queries/sec
```

```
Starting DNS lookups for 100 do
mains...
Results for h3:
    - Successfully Resolved: 72
    - Failed Resolutions:    28
    - Average Lookup Latency: 17
9.28 ms
    - Average Throughput:    3.66
queries/sec

Starting DNS lookups for 100 do
mains...
Results for h4:
    - Successfully Resolved: 77
    - Failed Resolutions:    23
    - Average Lookup Latency: 17
5.94 ms
    - Average Throughput:    3.57
queries/sec
```

Reason for failed resolutions: We checked a few failed resolution urls by requesting a few urls but it prompted with ***site couldn't be reached.*** This is becuase either the site does not exit or the intermediate DNS servers are down.

C. DNS Configuration of Mininet Hosts

We need to update DNS resolver configuration files for each host as shown below:

```
# Update resolv.conf for hosts
    for h in ['h1', 'h2', 'h3', 'h4']:
        host = net.get(h)
        host.cmd('echo "nameserver 10.0.0.5" > /etc/resolv.conf')
```

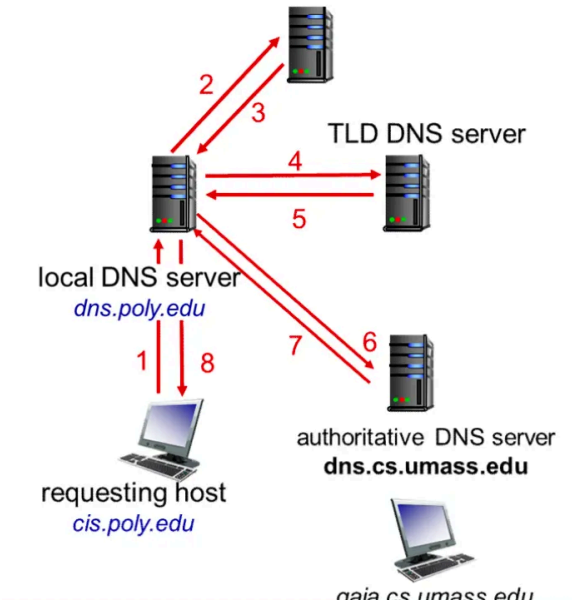
Updated configuration files for each host:

```
mininet> h1 cat /etc/resolv.conf
nameserver 10.0.0.5
mininet> h2 cat /etc/resolv.conf
nameserver 10.0.0.5
mininet> h3 cat/etc/resolv.conf
bash: cat/etc/resolv.conf: No such file or directory
mininet> h3 cat etc/resolv.conf
cat: etc/resolv.conf: No such file or directory
mininet> h3 cat /etc/resolv.conf
nameserver 10.0.0.5
mininet> h4 cat /etc/resolv.conf
nameserver 10.0.0.5
```

D. Custom DNS Resolver

Implementation

The implementation involves querying root servers, TLD servers and Authorative servers iteratively to get IP Address of lower-level DNS resolver till the final address is resolved.



Step Wise Execution:

1. Check Cache
2. Reach Root Server for TLD → TLD IP
3. Reach TLD for Authorative → Authorative IP
4. Reach Authorative for NS → NS IP/ NS URL
5. [If not glue]: Resolve NS URL
6. Reach NS → Final Address

Log timestamp, trace, and resolution data.

Execution

Step 1> Running the resolver `@127.0.0.1 port=53534`

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  JUPYTER

nimitt@Nimitts-MacBook-Air-10 A2 % python3 resolver.py
Starting custom DNS resolver with caching: True
Listening on 0.0.0.0:53534 with 50 workers...
```

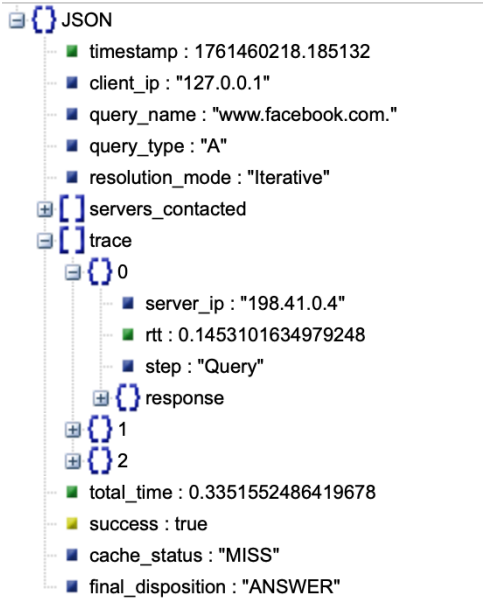
Step 2> Querying the resolver `www.facebook.com` with iterative mode. The resolver resolves the query and sends back the response and logs the corresponding resolution data.

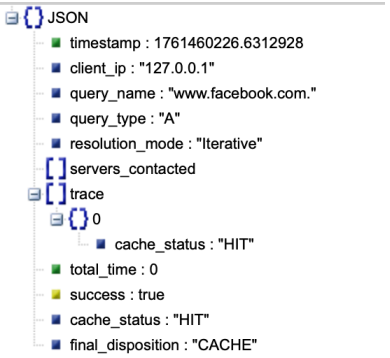
```
nimitt@Nimitts-MacBook-Air-10 A2 % dig @127.0.0.1 -p 53534 www.facebook.com +norecurse

; <<>> DiG 9.10.6 <<>> @127.0.0.1 -p 53534 www.facebook.com +norecurse
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 44199
;; flags: qr; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 8192
;; QUESTION SECTION:
;www.facebook.com.              IN      A

;; ANSWER SECTION:
www.facebook.com.       3600    IN      CNAME   star-mini.c10r.facebook.com.
star-mini.c10r.facebook.com. 60 IN      A       57.144.242.1

;; Query time: 339 msec
;; SERVER: 127.0.0.1#53534(127.0.0.1)
;; WHEN: Sun Oct 26 12:00:18 IST 2025
;; MSG SIZE  rcvd: 90
```

Step 3> **CACHING:** When queried again with `www.facebook.com` resolver finds it in cache and sends cached resolution as seen in query time and logs.

```
nimitt@Nimitts-MacBook-Air-10 A2 % dig @127.0.0.1 -p 53534 www.facebook.com +norecurse

; <<>> DiG 9.10.6 <<>> @127.0.0.1 -p 53534 www.facebook.com +norecurse
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 2893
;; flags: qr; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 8192
;; QUESTION SECTION:
;www.facebook.com.              IN      A

;; ANSWER SECTION:
www.facebook.com.       3600    IN      CNAME   star-mini.c10r.facebook.com.
star-mini.c10r.facebook.com. 60 IN      A       57.144.242.1

;; Query time: 0 msec
;; SERVER: 127.0.0.1#53534(127.0.0.1)
;; WHEN: Sun Oct 26 12:00:26 IST 2025
;; MSG SIZE  rcvd: 90
```

LOGGING: Resolver logs the required resolution details as JSON object for each query. Resolver's log for 1st and 2nd (Cached) query.





Execution with DNS in Mininet as our resolver

Now we run our custom resolver from the dns node in mininet topology to resolver `H{i}` 's queries with `CACHE = FALSE` .

Step 1> Configure hosts (Part C).

Step 2> Run `resolver.py` from dns node and start querying the dns sequentially.

Resolution Details:

Link to Resolver's logs: Link

```
Starting custom DNS resolver on dns node...

Starting DNS lookups for 100 do main
Results for h1:
  - Successfully Resolved: 70
  - Failed Resolutions:    30
  - Average Lookup Latency: 827 ms
  - Average Throughput:    0.79 q/s
```

```
Starting DNS lookups for 100 do main
Results for h3:
  - Successfully Resolved: 71
  - Failed Resolutions:    29
  - Avg Lookup Latency: 1013.65 ms
  - Average Throughput:    0.73 q/s

Starting DNS lookups for 100 do main
```

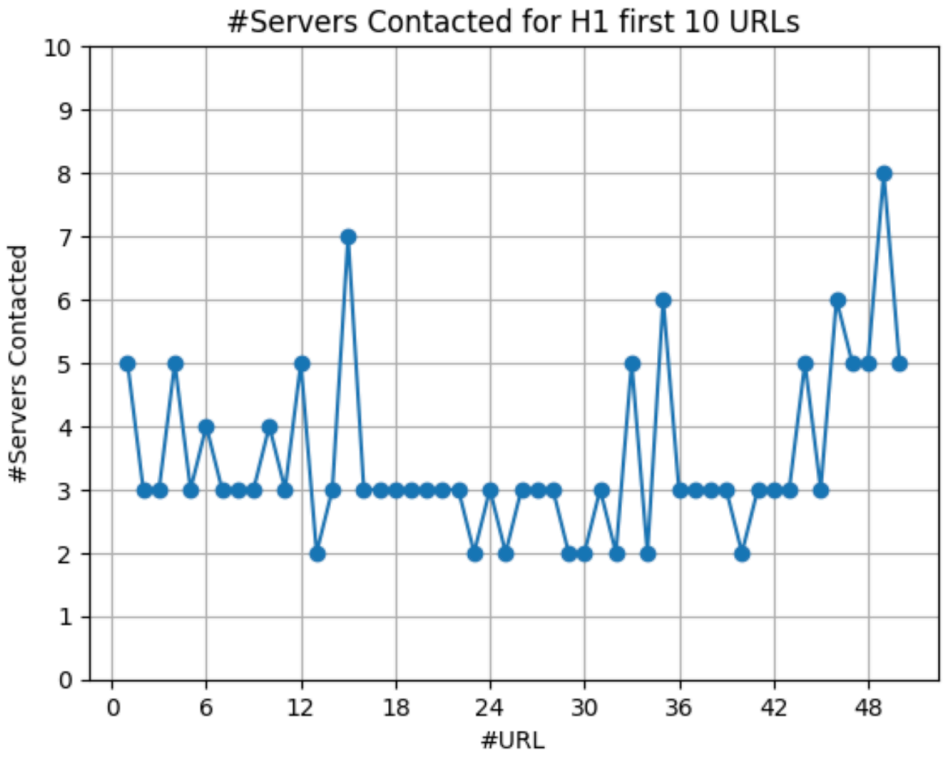
```
Starting DNS lookups for 100 do
main
Results for h2:
  - Successfully Resolved: 66
  - Failed Resolutions:    34
  - Average Lookup Latency: 839.
4 m
  - Average Throughput:    0.87 q/
s
```

```
Results for h4:
  - Successfully Resolved: 71
  - Failed Resolutions:    29
  - Avg Lookup Latency: 1000.14
ms
  - Average Throughput:    0.71 q/
s
 Stopping network...

 (q/s: queries per second)
```

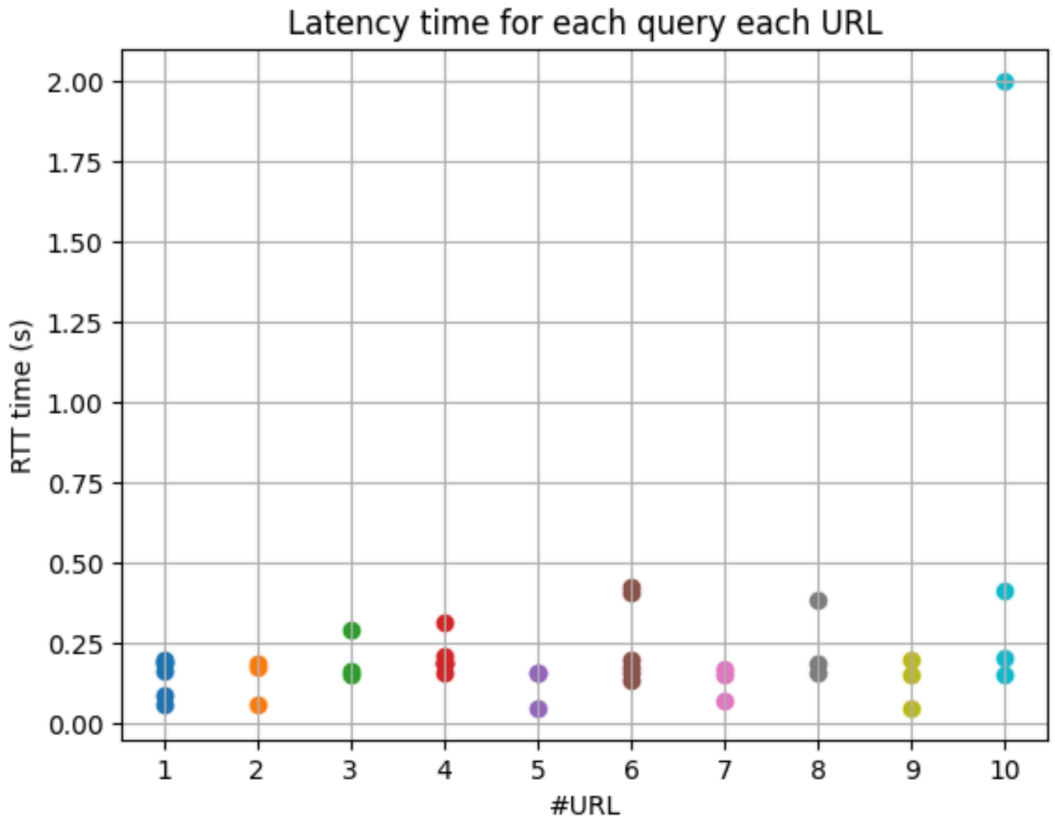
Visualization

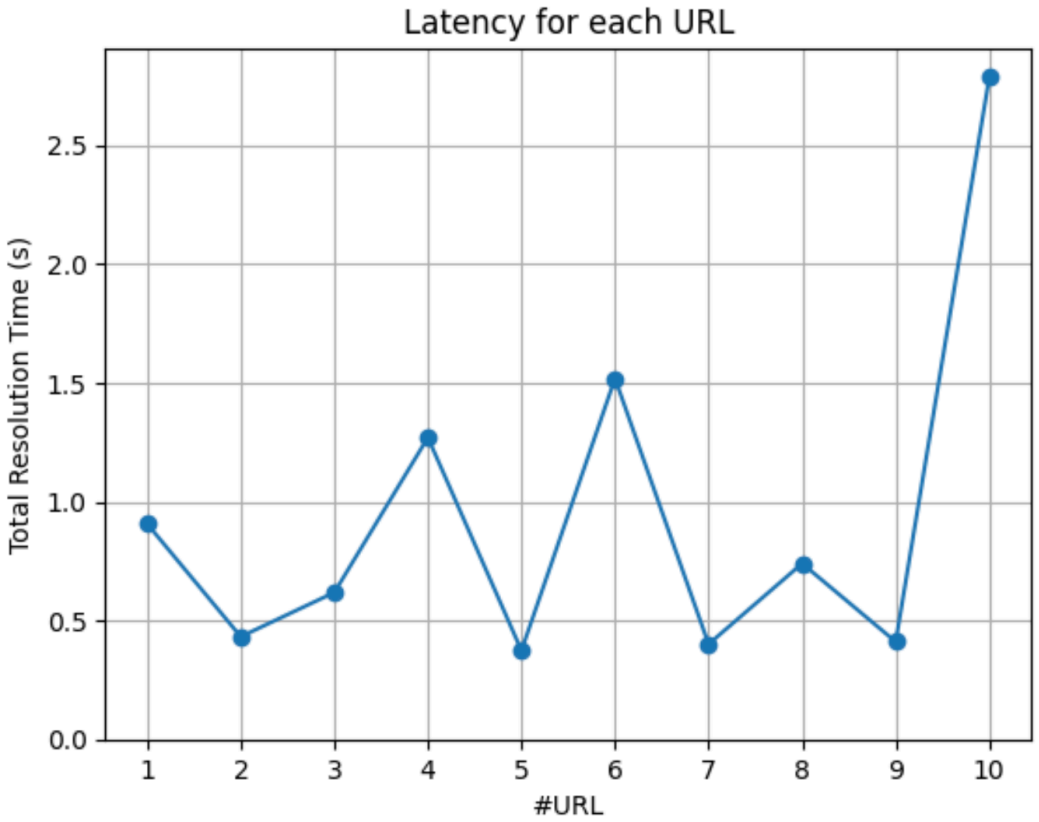
Number of Servers Contacted

Average #Servers Contacted for all URLs = `3.53`



Latency per query


Latency time for each query each URL
RTT time (s)
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
1
2
3
4
5
6
7
8
9
10
#URL


Latency for each URL
Total Resolution Time (s)
2.5
2.0
1.5
1.0
0.5
0.0
1
2
3
4
5
6
7
8
9
10
#URL

Comparison with Default Resolver

Our custom resolver is slower than the default resolver (as observed in Part B). The reasons for this is:

\> The custom resolver does not have cache unlike default resolver.

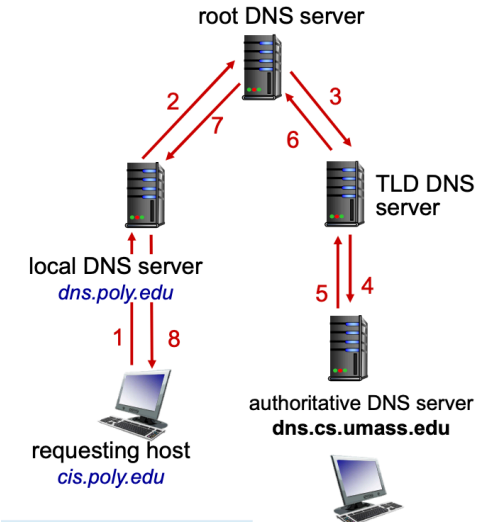
\> The custom resolver has to log all the resolution data for each query. This involves I/O and leads to latency.

\> The custom resolver does not have any optimisations to avoid dead and far DNS servers.

E. Recursive Resolver

Implementation

Recursive resolver sends recusirve query to root server, if it has recursion available it will directly send the resolved address otherwise if recusrion is not available it fallbacks to iterative mode until a server is found with recursion.



Execution

Link to servers logs: link

Now we run our custom resolver with `MODE="RECURSIVE"` from the dns node in mininet topology to resolver `H{i}` 's queries with.

Step 1> Configure hosts (Part C).

Step 2> Run `resolver.py` from dns node and start querying the dns sequentially.

```
Starting custom DNS resolver on dns node...
```

```
Starting DNS lookups for 100 do main
```

```
Starting DNS lookups for 100 do
main
Results for h1:
- Successfully Resolved: 69
- Failed Resolutions:    31
- Average Lookup Latency: 1300.
1 ms
- Average Throughput:    0.51 q/s

Starting DNS lookups for 100 do
main
Results for h2:
- Successfully Resolved: 67
- Failed Resolutions:    33
- Average Lookup Latency: 1643.
1 ms
- Average Throughput:    0.41 q/s
```

```
Results for h3:
- Successfully Resolved: 68
- Failed Resolutions:    32
- Average Lookup Latency: 1277.
3 ms
- Average Throughput:    0.6 q/s

Starting DNS lookups for 100 do
main
Results for h4:
- Successfully Resolved: 72
- Failed Resolutions:    28
- Average Lookup Latency: 1704.
2 ms
- Average Throughput:    0.50 q/s

 Stopping network...
```

Comparison

We note recusive resolver is slower than iteratitve as root servers are not replying to recursive queries as shown below, and fallback to iterative requires additional overhead.

```
nimitt@Nimitts-MacBook-Air-10 A2 % dig @202.12.27.33 -p 53 www.google.com

; <<>> DiG 9.10.6 <<>> @202.12.27.33 -p 53 www.google.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 50593
;; flags: qr rd; QUERY: 1, ANSWER: 0, AUTHORITY: 13, ADDITIONAL: 27
;; WARNING: recursion requested but not available
```

F. Cache

Implementation

We implement cache as a python dictionery storing already resolved servers. Whenever the resolution function is called it first checks cache if it is there it

responds with the cached record otherwise resolves using usual approach.

Execution

We run the topology.py with `CACHE_ENABLED = True` . We get the following resolution results.

Link to Resolver's logs: Link

```
Starting custom DNS resolver on dns node...

Starting DNS lookups for 100 do main
Results for h1:
- Successfully Resolved: 71
- Failed Resolutions:    29
- Average Lookup Latency: 800.1 3 ms
- Average Throughput:    0.86 q/s

Starting DNS lookups for 100 do main
Results for h2:
- Successfully Resolved: 67
- Failed Resolutions:    33
- Average Lookup Latency: 743.1 8 ms
- Average Throughput:    1.01 q/s
```

```
Starting DNS lookups for 100 do main
Results for h3:
- Successfully Resolved: 69
- Failed Resolutions:    31
- Average Lookup Latency: 777.3 5 ms
- Average Throughput:    0.89 q/s

Starting DNS lookups for 100 do main
Results for h4:
- Successfully Resolved: 72
- Failed Resolutions:    28
- Average Lookup Latency: 804.2 1 ms
- Average Throughput:    0.90 q/s

 Stopping network...
```

Cache Hit Rate

Total Number of Times Cache Searched = `400` URLs `+ 142` NS Resolutions = `542`

Number of Times Cache Hit = `21` Cache Hit Rate = `21/542` = `3.9 %`

Observations

\> Caching gives higher average througput as we are able to resolve already queried servers much quickly.