CS4411 Project 5: Link-State Routing

presented by Kai Mast
(Original slides by Soumya Basu)
Main Goals of this Assignment

- Learn the userspace side of networking
- Learn to work with an existing codebase
- Learn the standard C socket API
- Learn how to handle multiple connections in a single application.
Three Main Parts to the Project

- Setting up connections between nodes
- Broadcasting packets through the network
- Calculate shortest paths using Dijkstra’s Algorithm
Revisiting Stream Based Networking

- TPC has no notion of messages
- one send() might be split into multiple recv()’s on the other side
  - and vice versa...
- Application layer needs to convert stream into messages (if needed)
Setting up the socket

```c
int s = socket(AF_INET, SOCK_STREAM);
sockaddr_in addr;
addr.sin_port = htons(1337);
addr.sin_addr = INADDR_ANY;
bind(s, &addr, sizeof(addr));
```

Make sure you handle errors in your implementation!
TCP is connection-based

Bob

listen(s);
accept(s);

SYN

Alice

connect(s, &bob_addr, sizeof(bob_addr));

SYN, ACK

accept returns new socket for Alice

ACK

connect returns successfully
TCP is stream-based

The kernel allocates a send and a receive buffer for each socket.

- Buffers are FIFO
- `send()` appends to the local send buffer
- `receive()` takes data from the front of the local receive buffers
- OS takes care of emptying send buffer and filling receive buffer
epoll()
wait for different kinds of events on multiple file descriptors

Using a single blocking socket

Using multiple non-blocking sockets and epoll()

poll() is fine too
Creating an epoll() object

#include <sys/epoll.h>
int efd = epoll_create1();

Note: epoll only works on Linux
Setting up a socket for epoll()

// make socket nonblocking
fctnl(s, F_SETFL, SO_NONBLOCK);

// hand socket to epoll
struct epoll_event event;
event.data.fd = s;
event.events = EPOLLIN | EPOLLOUT | EPOLLHUP;
epoll_ctl (efd, EPOLL_CTL_ADD, s, &event);

Bitmask specifies which events to wait for
Handling events with epoll()

```c
struct event *events = calloc (MAXEVENTS, sizeof(event));

while (true) { // loop during the lifetime of the program
    int n = epoll_wait (efd, events, MAXEVENTS, -1);
    for (int i = 0; i < n; i++) {
        if ((events[i].events & EPOLLERR) {
            // error happened
        } else if (events[i].events & EPOLLIN) {
            // socket is ready to read
        } else if (events[i].events & EPOLLOUT) {
            // socket is ready to write
        }
    }
}
```
“Gossiping” messages

A has a message to send
“Gossiping” messages

It forwards to all its neighbors
“Gossiping” messages

They do the same
Optimization 1: Ignore neighbor that sent original message
“Gossiping” messages

D and E forward.
Optimization 2: Ignore messages you already saw.
“Gossiping” messages

System is stable: All parties have received the message.
Let’s use Gossip for Link State Routing!

• Each node in the network forwards its current configuration (i.e. list of neighbors using gossip)

• Once a node receives a new message, it runs Dijkstra to find optimal route

• Simplification: All edges have weight 1
  → But your Dijkstra implementation needs to work with other weights too
Message Handling

Two ways to receive messages:

- Through user input in a command prompt (i.e., you type the messages in the console)
- As a network package from another node

All messages use a simple plaintext protocol*

*That is probably not something you want to do for a serious project but sufficient for Prac
Strings in C

char *str = "cornell";

str points here
Strings in C

```c
char *str = "cornell";
char *str2 = str+5;
```

Note: Make sure you don't overflow your buffer. `strlen()` is your friend!
Array Semantics on C Strings

```c
char *str = "cornell";
char c1 = str[3];
char c2 = *(str+1);
```

str points here
Comparing string in C

char *str1 = "foo";
char *str2 = "foo";
char *str3 = "fo";

What is the difference? Which of those is true?
- str1 == str2
- strcmp(str1, str2, strlen(str1)) == 0
- strcmp(str1, str3, strlen(str3)) == 0
- strcmp(str1, str3, strlen(str1)) == 0
Telling a node to do stuff

C<addr>:<port>
- Connect to the specified address

S<dst_addr:port>/<TTL>/<payload>
- Send data over the network
- TTL specifies maximum number of hops
- Payload is the actual content of the message
Gossip protocol

G<src_addr>:<src_port>/<counter>/<payload>

- Counter is a message ID used to detect duplicate messages
- It should increase with every new gossip message

Where the payload is the list of neighbors:
;<addr1:port1>;<addr2:port2>;<addr3:port3>...
Dijkstra’s Algorithm
(from A’s point of view)

<table>
<thead>
<tr>
<th>Target</th>
<th>Distance</th>
<th>Route</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- A is the starting point.
- Connections and distances:
  - A to B: 2
  - A to C: 4
  - B to D: 1
  - D to E: 3
  - E to B: 1
Dijkstra’s Algorithm
(from A’s point of view)

<table>
<thead>
<tr>
<th>Target</th>
<th>Distance</th>
<th>Route</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>empty</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm
(from A’s point of view)

<table>
<thead>
<tr>
<th>Target</th>
<th>Distance</th>
<th>Route</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>empty</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
<td>B → D</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>B → E</td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm (from A’s point of view)

We found a shorter route to D!
Dijkstra’s Algorithm
(from A’s point of view)

No change!
Dijkstra’s Algorithm
(from A’s point of view)

No change and done!
How to approach the project

- Start by reading the skeleton code
  - You can modify it but it is a good point to start with

- Optional: Write some “unit tests” for your Dijkstra implementation

- Try to get connections between two peers to work.

- Then figure out the rest...
Don’t forget to test!

- Can you successfully connect a network of nodes?
  - They can all run on the same VM
- Do your routes reconfigure once a node (dis-)connects?
- Can you successfully send messages using the established routes?
Questions?

You Can Do It!