Packet Switching in Overlay Networks

Jon Turner
Computer Science & Engineering
Washington University

www.arl.wustl.edu/~jst
Switching at Different Levels

- Physical layer switching
  - optical wavelength switches, SONET cross-connects
- L2 switching and L3 routers
  - custom-built hardware implementations
  - but, growing use of multicore packet processors
- Application level switching
  - peer-to-peer networks, using PCs
- Overlay networks
  - experimentation, deployment of new network services
    - network testbeds, overlay hosting, cloud computing contexts
    - conventional servers, growing role of multicore
- Substantial similarities across different levels
ONL as Overlay Hosting Environment

- Multiple networks mapped onto shared infrastructure
  - configuration switches
  - implement virtual links
  - isolate user sessions from each other

- Can also use PCs to switch packets
  - lower performance, but core issues remain the same
  - easy to experiment with novel network services
    - useful first step before going to high performance platform
    - conventional multicore servers becoming competitive
Basic UDP Socket Programming

- Socket API used to support network programming
  » designed for multiple protocols (complicating things)
- Socket is a communications endpoint
  » identified by socket number & associated with a process
  » TCP & UDP port # fields can be associated with socket #
    • incoming packet sent to socket that is bound to dest-port #
    • outgoing packet carries socket’s port# in src-port field
- Socket address struct specifies (adr,port) pair

```c
struct sockaddr_in {
    short sin_family;  // AF_INET
    sin_port;          // port# in net byte order
    struct in_addr sin_addr; // adr in net byte order
    char sin_zero[8];  // padding
};
```
Simple UDP Forwarder

// usage:
//    forwarder myAddr myPort destAddr destPort ...
main(int argc, char* argv[]) {
    int s; // socket
    struct sockaddr_in srcAddr, destAddr;
    int myPort, destPort; char buf[MAXBUF];
    // process command line ...
    bzero(&srcAddr, sizeof(srcAddr));
    if (initAddr(myAddr.argv[1], myPort) fatal("can't read myAddr"));
    if (initAddr(destAddr.argv[3], destPort) fatal("can't read destPort"));
    if ((s = setupSock(srcAddr, destAddr, buf)) < 0) fatal("can't setup socket");
    while (1) {
        int n = getPacket(s, &srcAddr, buf);
        if (n < 0 ) fatal("receive error on socket");
        if (!putPacket(s, &destAddr, buf, n) fatal("send error on socket");
    }
}
receive packets at (myAddr,myPort) and forward to (destAddr,destPort)
initialize socket address structures
open socket and bind (adr,port) pair to it
get packet into buffer, sender (adr,port) pair into srcAddr
forward packet to destination (adr,port) pair
Utility Routines

```c
bool initSock(struct sockaddr_in *sa, char *ipAdr, int port) {
    // Initialize the socket address structure with the given IP address and port.
    bzero(sa, sizeof(*sa)); sa->sin_family = AF_INET; sa->sin_port = htons(port);
    return inet_pton(AF_INET, ipAdr, &sa->sin_addr) > 0 ? true : false;
}

int setupSock(struct sockaddr_in *sa) {
    // Setup a udp socket using the given address structure.
    int s;
    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ||
        bind(s, (struct sockaddr *) sa, sizeof(*sa)) < 0)
        return -1;
    return s;
}

int getPacket(int s, struct sockaddr_in* sa, char *buf) {
    // Get a UDP packet on the socket s and return the payload in buf.
    int n;
    socklen_t len = sizeof(*sa);
    n = recvfrom(s, buf, 1500, 0, (struct sockaddr *) sa, &len);
    return n;
}

bool putPacket(int s, struct sockaddr_in *sa, char *buf, int n) {
    // Send a UDP packet on socket s to the destination specified in *sa.
    int m = sendto(s, buf, n, 0, (struct sockaddr *) sa, sizeof(*sa));
    return m == n ? true : false;
}
```
Simple Traffic Generator

```
// usage:
//     sender myAdr myPort toAdr toPort n delta ...

main(int argc, char* argv[]) {
    int n; // number of bytes to send per packet
    int delta; // min number of us between packets
    struct sockaddr_in myAdr, toAdr; int myPort, toPort; char buf[HAXEBUF];
    // process command line, initialize address structures, setup socket ...
    struct timeval t, pt;
    if (gettimeofday(&t, NULL) < 0) fatal("can't read time");
    while (1) {
        if (gettimeofday(&t, NULL) < 0) fatal("can't read time");
        if (t.tv_sec == pt.tv_sec & t.tv_usec < pt.tv_usec + delta ||
            (t.tv_sec == pt.tv_sec + 1 &
             t.tv_usec - pt.tv_usec + delta = 1000000);
            continue;
        if (!putPacket(s, toAdr, buf, n)) fatal("send error on socket");
        pt.tv_usec += delta;
        delta = (pt.tv_usec - t.tv_usec) % 1000000;
        pt.tv_sec++; pt.tv_usec = 1000000;
    }
}
```
Forwarding Rate Measurements (ONL)

- Short packets limited by *forwarder* packet processing rate
- Long packets limited by link bandwidth
- 10 Gb/s link can carry about 13.6 million short packets each second
wunet-1 Overlay Network

- Version number (=1)
- Packet length (in bytes)
- Virtual network
  - identifies tree-structured subnet that packet "belongs to"
    - similar to Ethernet VLAN
- Address fields
  - numeric endpoint identifiers, no location information
- Simple packet forwarding
  - if dst address assigned to local endpoint, send it packet
  - else, send to all neighboring routers (except "upstream")
wunet Router Software Components

- **wuRouter** – main loop
  - if a packet has arrived
    - retrieve it, check it
    - determine outgoing link(s)
    - queue it
  - while packets ready to go
    - select packet to send
    - dequeue and send

- **IO processing**
  - socket handling, nonblocking input

- **Packet store** handles packets, buffers

- **Queue manager** queues and schedules packets

- **Tables** for link info, vnets, routes
### Router Configuration Tables

**Link Table**

<table>
<thead>
<tr>
<th>nIP</th>
<th>nPort</th>
<th>nAdr</th>
<th>nTyp</th>
<th>nRate</th>
<th>pRate</th>
<th>pRate</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.11231</td>
<td>x52</td>
<td>host</td>
<td>8200</td>
<td>4100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Vnet Table**

<table>
<thead>
<tr>
<th>Vnet</th>
<th>vAdr</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>0119...1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Route Table**

<table>
<thead>
<tr>
<th>Route</th>
<th>vNet</th>
<th>vAdr</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>22...5</td>
<td>14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Link table (indexed by link number)**
  - peer’s IP address, port number and wunet address
  - peer’s type (host or router)
  - max sending rate (Kb/s, packets/s)
- **Vnet table (indexed by vnet number)**
  - bit vector (31 bits) identifying links in vnet
- **Route table**
  - maps (vnet, dest adr) to outgoing link number
  - forward to all neighboring routers in vnet if no entry
Main Router Loop

```c
void wuRouter::run(uint32_t finishTime) {
    struct timeval ct, pt; // current and previous time values
    if (gettimeofday(&ct, NULL) < 0) fatal("...");
    new = 0;
    while (finishTime == 0 || now < finishTime) {
        int p = iop->receive();
        if (p != Null) {
            ps->unpack(p);
            if (pktCheck(p)) forward(p);
            else ps->free(p);
        }
        for (link = 1;
            ((int lnk = om->nextReady(now)) != Null) {
                p = qe->deq(lnk); iop->send(p, lnk);
            }
            pt = ct;
        if (gettimeofday(&ct, NULL) < 0) fatal("...");
        now += (ct.tv_sec == pt.tv_sec ?
            ct.tv_usec - pt.tv_usec :
            ct.tv_usec + (1000000 - pt.tv_usec ));
    }
}
```

- **Free-running microsecond clock - 32 bits**
- **Update time from system clock**
- **Run until finishTime or "forever"**
- **If a packet has arrived, check it and forward (enqueue on proper output link(s))**
- **Dequeue and send all overdue packets**
Queue Manager

- Maintains three data structures
  - set of packet queues (one queue per link)
  - "circular heap" with entry for each active link
    - link is active if its queue contains at least one packet
    - key in heap is time when link is due to send its next packet
    - keys compared circularly
      - \( x > y \) if \( x \) is clockwise from \( y \), over smaller distance
  - second heap for virtually active links
    - links whose queues recently became empty but whose packet due time has not yet arrived

- Methods
  - enq and deq add/remove packets, update heaps
  - nextReady returns link with most "overdue" next packet
    - and purges stale entries from heap of virtually active links