UDP Sockets

ComS 587X
Fall, 2002

UDP

- Unreliable datagram protocol
- Layer 4 (Transport)
- Add port identification numbers and payload checksum to IP
  - Ports allow multiplexing of data streams
- Low overhead
- Typically used for latency-sensitive or low-overhead applications
  - Video
  - Time
  - DNS

UDP Classes

- java.net.DatagramPacket
  - Remote IP address, port, and payload
- java.net.DatagramSocket
  - Local IP address and port

DatagramPacket

- Constructors
  - DatagramPacket(byte[] buffer, int length)
  - DatagramPacket(byte[] buffer, int length, InetAddress remoteAddr, int remotePort)
- Methods
  - InetAddress getAddress()
  - byte[] getData()
  - int getLength()
  - int getPort()
  - void setAddress(InetAddress remoteAddr)
  - void setData(byte[] buffer)
  - void setLength(int length)
  - void setPort(int remotePort)

DatagramSocket

- Constructors
  - DatagramSocket(int port)
  - DatagramSocket()
  - DatagramSocket(int port, InetAddress localAddr)
- Methods
  - void close()
  - void connect(InetAddress remoteAddr, int remotePort)
  - void disconnet()
  - InetAddress getInetAddress()
  - int getPort()
  - InetAddress getLocalAddress()
  - int getLocalPort()
  - int getReceiveBufferSize()

DatagramSocket Methods

- int getSendBufferSize()
- int getSoTimeout()
- void receive(DatagramPacket pkt)
- void send(DatagramPacket pkt)
- void setReceiveBufferSize(int length)
- void setSendBufferSize(int length)
- void setSoTimeout(int duration)

Receiving

```
DatagramPacket pkt = new DatagramPacket(new byte[256],
    256);
DatagramSocket sock = new DatagramSocket(2000);
boolean done = false;

while (!done)
{
  sock.receive(pkt);
  // Process packet
}
sock.close();
```

Packet Processing

- The handy ByteArrayInputStream:

```
ByteArrayInputStream bin = new
    ByteArrayInputStream(packet.getData());
DataInputStream din = new
    DataInputStream(bin);
// Use the read* methods on din to read the
// contents of the packet
```

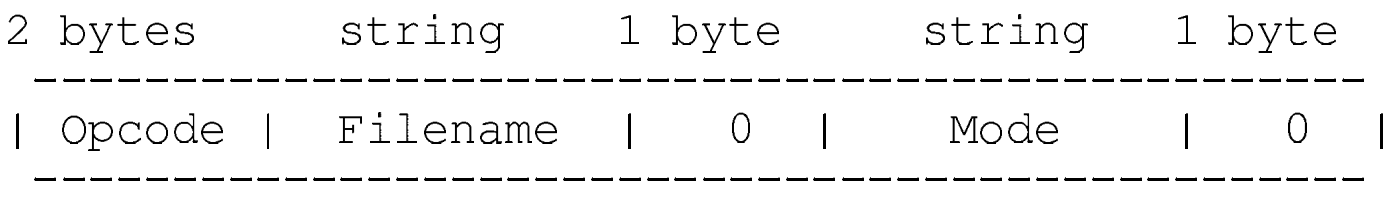
Sending

```
DatagramPacket pkt = new DatagramPacket(new byte[256],
    256);
DatagramSocket sock = new DatagramSocket();
boolean done = false;

while (!done) {
  pkt.setAddress(InetAddress.getByName("my.host.com");
  pkt.setPort(2000);
  // Fill in pkt's buffer and length
  sock.send(pkt);
  // Receive a response...
  //
}
sock.close();
```

TFTP

- Trivial File Transfer Protocol
  - Unauthenticated
  - Often used for transfering boot images and configuration files
  - UDP Port 69
- http://www.ietf.org/rfc/rfc1350.txt
- Read & write requests:



- Opcode: 1=Read, 2=Write
- Mode: “netascii”, “octet”, “mail”

TFTP (2)

- Data
  - Opcode = 3 (Data)
  - Block (sequentially increasing)

```
 2 bytes     2 bytes       n bytes
 ----------------------------------
| Opcode |   Block #  |   Data     |
 ----------------------------------
```

- Ack
  - Opcode = 4 (ACK)
  - Block (matched accepted block)

```
 2 bytes     2 bytes       n bytes
 ----------------------------------
| Opcode |   Block #  |   Data     |
 ----------------------------------
```

TFTP (3)

- Error
  - Opcode = 5 (ERROR)
  - ErrorCode = 1 (FileNotFound), 2 (AccessDenied), 3 (DiskFull), 4 (IllegalOp), 5 (UnknownTransferID), 6 (FileAlreadyExists), 7 (NoSuchUser)

```
 2 bytes  2 bytes        string    1 byte
 ----------------------------------------
| 05    |  ErrorCode |   ErrMsg   |   0  |
 ----------------------------------------
```

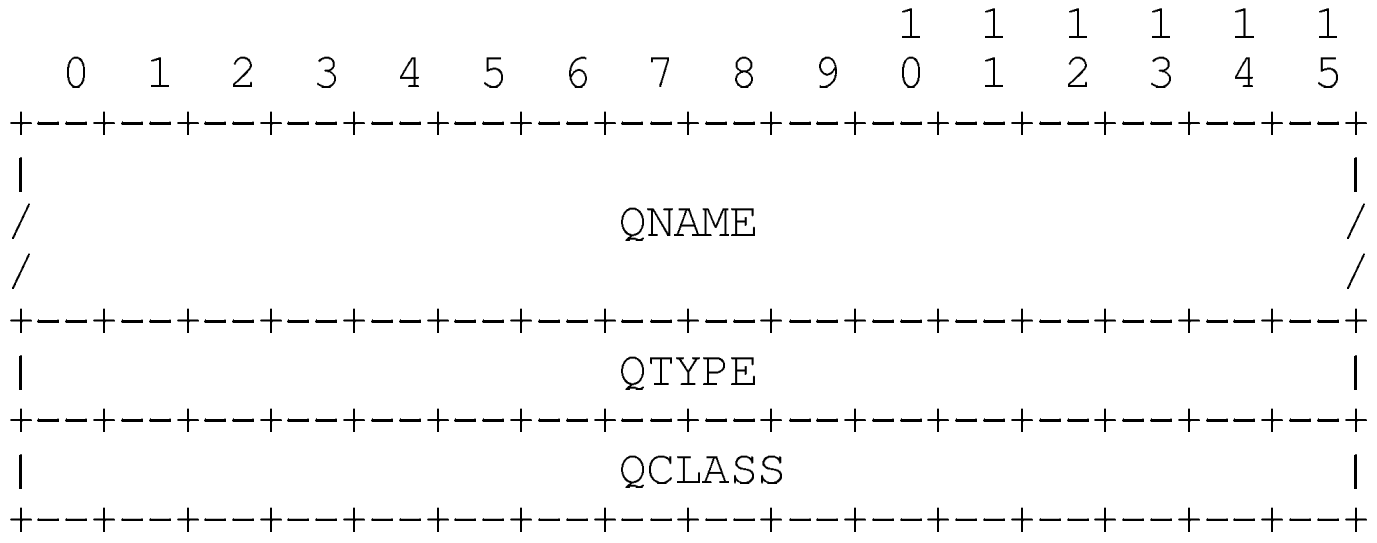
DNS

- DNS is one of the most common users of UDP
- DNS packets contain:
  - Header
    - Query or response
    - Number of each type of following record
  - Query record
  - Answer record(s)
  - Authority records(s)
  - Additional record(s)
- http://www.ietf.org/rfc/rfc1035.txt

DNS Header Members

- ID: 2 bytes: Uniquely identifies request / response
- QR: 1 bit: 0=query, 1=response
- Opcode: 4 bits: 0=standard query, 1=reverse query, 2=server status
- AA: 1 bit: 1=authoritative
- TC: 1 bit: 1=truncated
- RD: 1 bit: recursion desired
- RA: 1 bit: recursion available
- Z: reserved
- RCODE: 0=no error
- QDCOUNT: number of query sections
- ANCOUNT: number of answer records
- NSCOUNT: number of name server records
- ARCOUNT: number of additional records

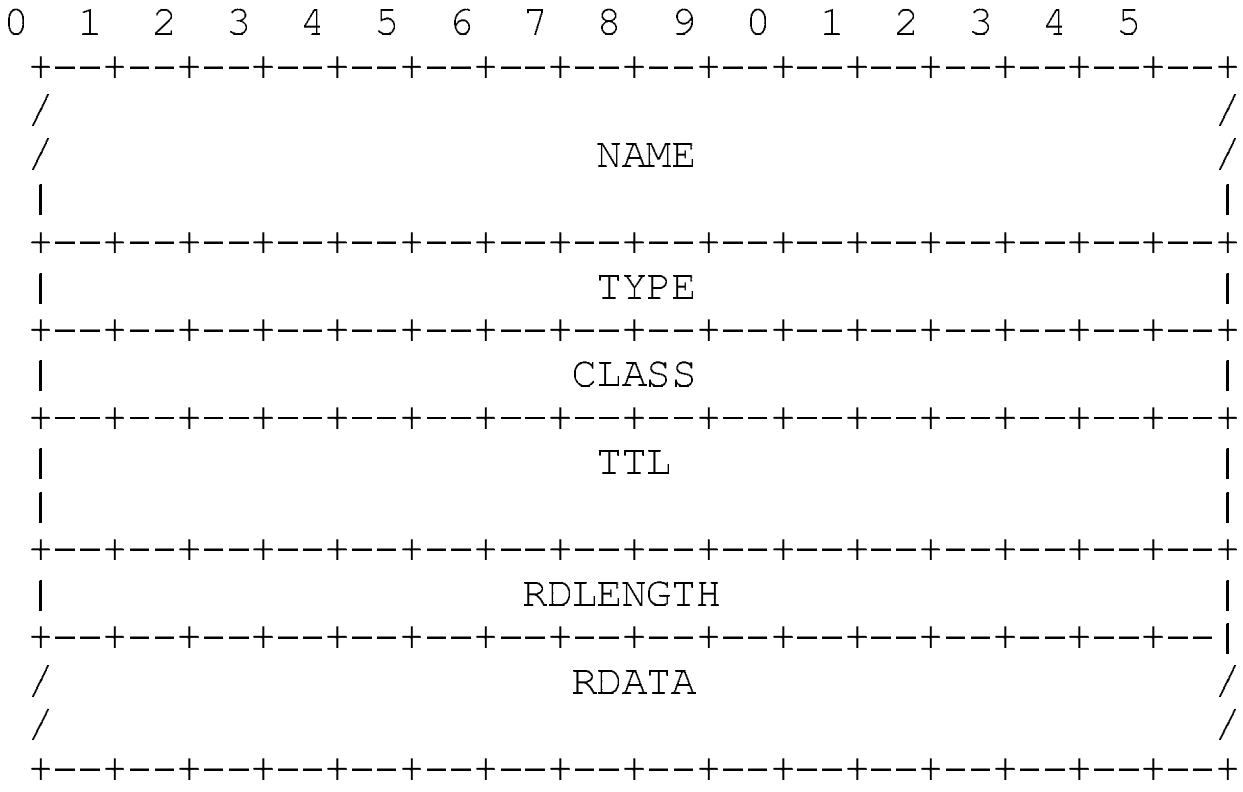
DNS Query Section



- QNAME: The (compressed) name for which to search
- QTYPE: Type of query: 1=address (A), 12=pointer (PTR), 255=All records
- QCLASS: Class of query: 1=Internet

DNS Records

- Each record contains a variable-length name, fixed header and variable-length data



DNS Record Members

- NAME: encoded for compression
  - Starts with a byte
    - Value < 64: Length
      - Followed by “length” bytes of characters
    - Value >= 192: Pointer
      - Jump to the offset given - 192
    - Value 0: End of name
  - E.g., 0x03|www|0x0b|palisadesys|0x03|com|0x00
- TTYPE (2 bytes): 1=address (A), 12=pointer (PTR), 255=All records
- Class (2 bytes): 1=Internet
- TTL (4 bytes): number of seconds that data can be cached
- RDLENGTH (2 bytes): Number of bytes in the following RDATA structure

DNS RDATA

- A (type = 1)
  - 4 bytes: Internet Address (version 4)
- PTR (type = 12)
  - Variable bytes: NAME
- Numerous other types: SOA, NS, MX, etc.

DNS Data Structures (in C)

```
struct domain_header
{
  u_short id;
  u_short flags_field;
  u_short qdcount;
  u_short ancount;
  u_short nscount;
  u_short arcount;
};

struct rr_record
{
  u_short type;
  u_short class;
  u_int32_t ttl;
  u_short rdlength;
};
```

DNS Header Code (in C)

```
printf("DNS: ID=%u %s QueryType=%s Authoritative=%s Trunc=%s",
        (int)ntohs(dh->id),
        ntohs(dh->flags_field) & 0x8000 ? "Response" :
   "Query",
        query_type,
        ntohs(dh->flags_field) & 0x0400 ? "Yes" : "No",
        ntohs(dh->flags_field) & 0x0200 ? "Yes" : "No");
printf(" RecursionDesired=%s RecursionAvailable=%s Error=%s",
        ntohs(dh->flags_field) & 0x0100 ? "Yes" : "No",
        ntohs(dh->flags_field) & 0x0080 ? "Yes" : "No",
        response_type);
printf(" Questions=%u Answers=%u NameServers=%u Additional=%
   u",
        (int)ntohs(dh->qdcount), (int)ntohs(dh->ancount),
        (int)ntohs(dh->nscount), (int)ntohs(dh->arcount));
```

DNS Query Code (in C)

```
for (i = 0; i < (int)ntohs(dh->qdcount) && len > 0; i++)
  {
    printf("  DNS Question: '");
    if ((j = print_label_sequence(message_start, p, len)) > 0)
      {
        p += j;
        len -= j;
        memcpy((void *) &short_temp, (void *) p,
   sizeof(u_short));
        printf("' QType=%d", (int) ntohs(short_temp));
        len -= 2;
        p += 2;
        memcpy((void *) &short_temp, (void *) p,
   sizeof(u_short));
        printf(" QClass=%d", ntohs(short_temp));
        len -= 2;
        p += 2;
      }
  }
```

DNS RR Code (in C)

```
for (i = 0; i < (int)ntohs(dh->ancount) && len > 0; i++)
  {
    j = print_resource_record("Answer", message_start, p,
   len);
    if (j > 0)
      {
        len -= j;
        p += j;
      }
  }
```

DNS RR Code (2)

```
static int print_resource_record(char *label, const u_char
   *message_start, const u_char *p, int len)
{
  int used, slen, i, j;
  struct rr_record rr;
  u_short short_temp;
  char temp[256];

  printf("     DNS %s: '", label);
  used = print_label_sequence(message_start, p, len);
  putc('\'');
  p += used; len -= used;
  memcpy((void *)&rr, (void *) p, 10 /*sizeof(struct
   rr_record)*/); /* Avoid alignment problems. */
  used += 10 /*sizeof(struct rr_record)*/;
  p += 10; len -= 10;
  printf(" Type=%u Class=%u TTL=%lu RDLength=%u",
          (int)ntohs(rr.type), (int)ntohs(rr.class),
   (u_int32_t)ntohl(rr.ttl),
          (int)ntohs(rr.rdlength));
```

DNS RR Code (3)

```
[ print_resource_record continued from prev slide ]
  switch (ntohs(r->type))
    {
    case 1:
      if (ntohs(r->rdlength) == 4 && ntohs(r->class) == 1)
         printf("\tHost Address %d.%d.%d.%d",
                  p[0], p[1], p[2], p[3]);
      break;
    case 12:
      printf("\tPTR ");
      print_label_sequence(message_start, p, len);
      break;
    }
  return used;
}
```

DNS NAME Decoding

```
static int print_label_sequence(const u_char *message_start,
   const u_char *p, int len)
{
  int i, index;
  for (index = 0; p[index] != 0; index++) {
      if ((p[index] & 0xc0)==0xc0){ /* Compressed DNS name */
          print_compressed_sequence(message_start,
   message_start + (((p[index] & 0x3f) << 16) + (p[index +
   1])));
          index++;
          break;
      } else {
          for (i = 1; i <= p[index]; i++)
              putc(p[index + i]);
          putc('.');
          index += (int)p[index];
      }
  }
  return index + 1;
}
```

DNS NAME Decoding (2)

```
static void print_compressed_sequence(const u_char
   *message_start, const u_char *p)
{
  int len;

  while (*p != '\0') {
      if ((*p & 0xc0) == 0xc0) {
          /* Follow further compression. */
          p = message_start + ((*p & 0x3f) << 16) + *(p + 1);
          continue;
      }
      for (len = *p++; len > 0; len--)
          putc(*p++);
      }
      putc('.');
  }
}
```