UDP

Hyang-Won Lee
Department of Computer Science and Engineering
Konkuk University
leehw@konkuk.ac.kr
https://sites.google.com/view/leehwko/

So Far

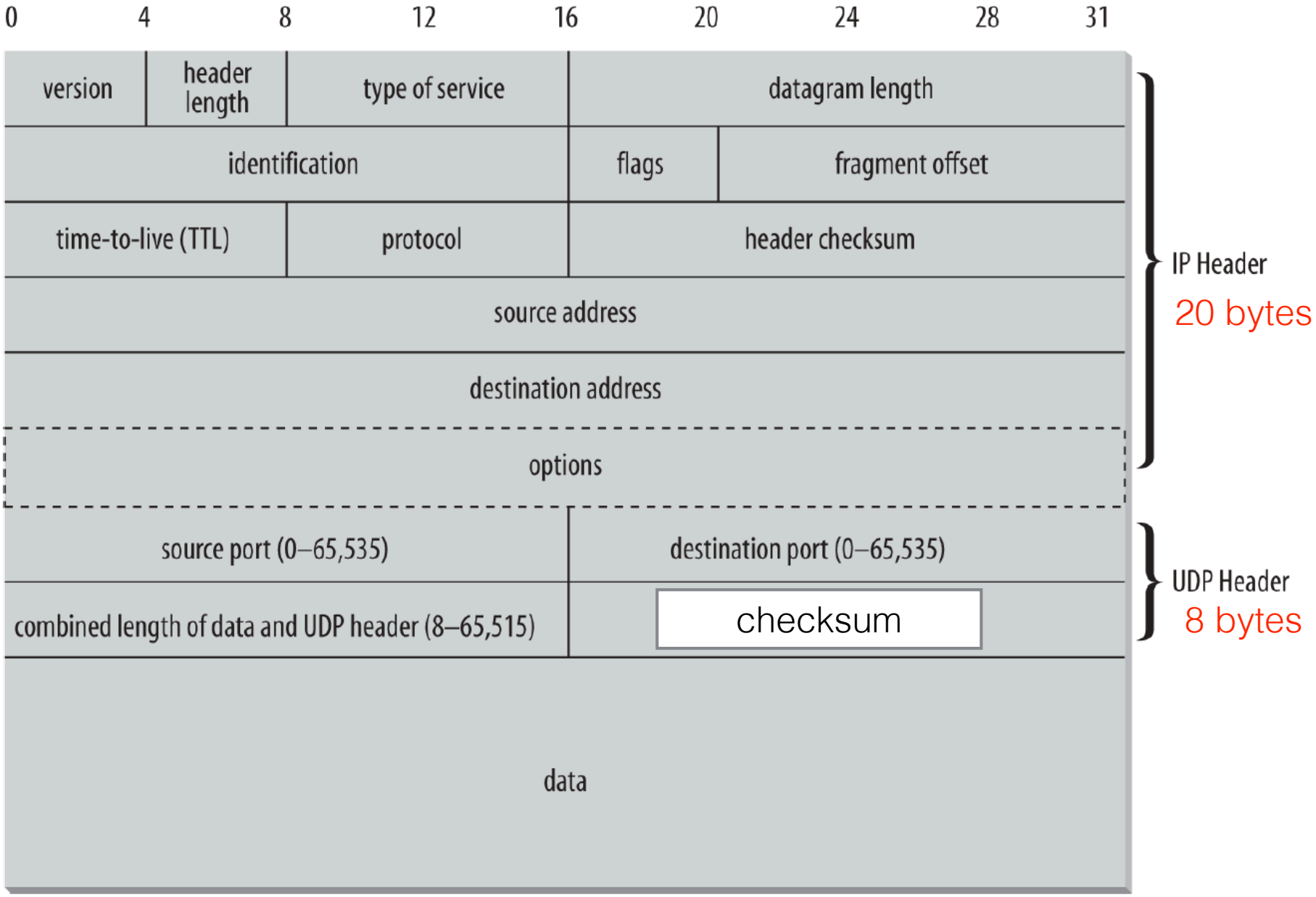
- We have discussed HTTP, Sockets, ServerSockets, SocketChannels,…
- All of these use TCP as a transport layer protocol

- Characteristics of TCP
  - reliable data transfer
  - congestion/flow control
  - three-way handshake for connection establishment
  - byte stream

User Datagram Protocol (UDP)

- User Datagram Protocol (UDP)
- Another transport layer protocol
- No reliable data transfer
  - reliability should be implemented in the application layer if needed
  - packet by packet send/receive
- No connection setup
- Used by DNS, NFS, …

UDP Header



Datagram Size

- Theoretical maximum amount of data in a UDP datagram
  - 65,507 bytes (limited by IP header)

- Many platforms limit the size to 8,192 bytes
  - not required to accept datagrams with more than 576 bytes, including data and headers
  - if datagram size exceeds the limit and the network truncates it, Java program won't be notified of the problem

- Rule of thumb
  - data portion of UDP packet should be kept to 512

Example

```
import java.io.*;
import java.net.*;

public class DaytimeUDPClient {

  private final static int PORT = 13;
  private static final String HOSTNAME = "time.nist.gov";

  public static void main(String[] args) {
    try (DatagramSocket socket = new DatagramSocket(0)) {
      socket.setSoTimeout(10000);
      InetAddress host = InetAddress.getByName(HOSTNAME);
      DatagramPacket request = new DatagramPacket(new byte[1], 1, host , PORT);
      DatagramPacket response = new DatagramPacket(new byte[1024], 1024);
      socket.send(request);
      socket.receive(response);
      String result = new String(response.getData(), 0, response.getLength(),
                                 "US-ASCII");
      System.out.println(result);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

Example

```
import java.net.*;
import java.util.Date;
import java.util.logging.*;
import java.io.*;

public class DaytimeUDPServer {

  private final static int PORT = 13;
  private final static Logger audit = Logger.getLogger("requests");
  private final static Logger errors = Logger.getLogger("errors");

  public static void main(String[] args) {
    try (DatagramSocket socket = new DatagramSocket(PORT)) {
      while (true) {
        try {
          DatagramPacket request = new DatagramPacket(new byte[1024], 1024);
          socket.receive(request);

          String daytime = new Date().toString();
          byte[] data = daytime.getBytes("US-ASCII");
          DatagramPacket response = new DatagramPacket(data, data.length,
              request.getAddress(), request.getPort());
          socket.send(response);
          audit.info(daytime + " " + request.getAddress());
        } catch (IOException | RuntimeException ex) {
          errors.log(Level.SEVERE, ex.getMessage(), ex);
        }
      }
    } catch (IOException ex) {
      errors.log(Level.SEVERE, ex.getMessage(), ex);
    }
  }
}
```

Multi-threading and UDP

- UDP servers tend not to be as multithreaded as TCP servers

- No acknowledgement of datagram reception
  - don't need to wait for the other end to respond

DatagramPacket Class

DatagramPacket

- Representation of UDP datagram
  - `public final class DatagramPacket extends Object`
- Constructors
  - use different constructors depending on whether the packet will be used to send data or to receive data
  - when receiving
    - byte array in input arguments in constructors buffers received data
  - when sending
    - byte array in input arguments holds the data to send

Constructors for Receiving Data

- Two constructors

```
public DatagramPacket(byte[] buffer, int length)
public DatagramPacket(byte[] buffer, int offset, int length)
```

- First constructor
  - socket receives datagram => it stores the datagram's data part in buffer beginning at buffer[0] and continuing until the packet is completely stored or until length bytes have been written into the buffer
  - example

```
byte[] buffer = new byte[8192];
DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
```

Constructors for Receiving Data

- Two constructors

```
public DatagramPacket(byte[] buffer, int length)
public DatagramPacket(byte[] buffer, int offset, int length)
```

- Second constructor
  - same as first constructor except that storage begins at buffer[offset]
  - length must be less than or equal to buffer.length - offset
    - otherwise, IllegalArgumentException is thrown

Constructors for Sending Data

- Four constructors

```
public DatagramPacket(byte[] data, int length,
    InetAddress destination, int port)
public DatagramPacket(byte[] data, int offset, int length,
    InetAddress destination, int port)
public DatagramPacket(byte[] data, int length,
    SocketAddress destination)
public DatagramPacket(byte[] data, int offset, int length,
    SocketAddress destination)
```

- Behavior
  - packet is filled with length bytes of data array
  - InetAddress or SocketAddress object points to the destination of the packet
  - port == port of remote host

Constructors for Sending Data

- Four constructors

```
public DatagramPacket(byte[] data, int length,
    InetAddress destination, int port)
public DatagramPacket(byte[] data, int offset, int length,
    InetAddress destination, int port)
public DatagramPacket(byte[] data, int length,
    SocketAddress destination)
public DatagramPacket(byte[] data, int offset, int length,
    SocketAddress destination)
```

- Behavior
  - data changed before sending => changed packet will be sent

Example

```
String s = "This is a test";
byte[] data = s.getBytes("UTF-8");

try {
  InetAddress ia = InetAddress.getByName("www.ibiblio.org");
  int port = 7;
  DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
  // send the packet...
} catch (IOException ex)
}
```

- data can be changed and packet can be sent with changed data

Getters

- Six methods to retrieve different parts of a datagram
- public InetAddress getAddress()
  - return InetAddress object containing the address of remote host
  - datagram received => address of host that sent it
  - datagram locally created => address of destination host
  - useful when host want to reply to the host that sent the datagram
- public int getPort() and public SocketAddress getSocketAddress()
  - both similar to getAddress()

Getters

- public byte[] getData()
  - return a byte array containing the data from the datagram
  - examples

```
String s = new String(dp.getData(), "UTF-8");
```

  -

```
InputStream in = new ByteArrayInputStream(packet.getData(),
    packet.getOffset(), packet.getLength());
DataInputStream din = new DataInputStream(in);
```

can use DataInputStream's readInt(), readLong(),…

Getters

- public int getLength()
  - return the number of bytes of data in datagram
  - equivalent to getData().length
- public int getOffset()
  - return the point in the array returned by getData()

Example

```
public class DatagramExample {

  public static void main(String[] args) {

    String s = "This is a test.";

    try {
      byte[] data = s.getBytes("UTF-8");
      InetAddress ia = InetAddress.getByName("www.ibiblio.org");
      int port = 7;
      DatagramPacket dp
          = new DatagramPacket(data, data.length, ia, port);
      System.out.println("This packet is addressed to "
          + dp.getAddress() + " on port " + dp.getPort());
      System.out.println("There are " + dp.getLength()
          + " bytes of data in the packet");
      System.out.println(
          new String(dp.getData(), dp.getOffset(), dp.getLength(), "UTF-8"));
    } catch (UnknownHostException | UnsupportedEncodingException ex) {
      System.err.println(ex);
    }
  }
}
```

Setters

- Most of the time, constructors should be enough for creating datagrams

- Java also provides methods for changing (even after datagram has been created)
  - data
  - remote address
  - remote port

Setters

- public void setData(byte[] data)
  - change payload of UDP datagram
  - useful when sending a large file (chop the file and send repeatedly by changing data each time)
- public void setData(byte[] data, int offset, int length)
  - alternative to the above method

```
int offset = 0;
DatagramPacket dp = new DatagramPacket(bigarray, offset, 512);
int bytesSent = 0;
while (bytesSent < bigarray.length) {
  socket.send(dp);
  bytesSent += dp.getLength();
  int bytesToSend = bigarray.length - bytesSent;
  int size = (bytesToSend > 512) ? 512 : bytesToSend;
  dp.setData(bigarray, bytesSent, size);
}
```

Setters

- public void setAddress(InetAddress remote)
  - change the address a datagram packet is sent to

```
String s = "Really Important Message";
byte[] data = s.getBytes("UTF-8");
DatagramPacket dp = new DatagramPacket(data, data.length);
dp.setPort(2000);
int network = "128.238.5.";
for (int host = 1; host < 255; host++) {
  try {
    InetAddress remote = InetAddress.getByName(network + host);
    dp.setAddress(remote);
    socket.send(dp);
  } catch (IOException ex) {
    // skip it; continue with the next host
  }
}
```

Setters

- public void setPort(int port)
  - change the port a datagram is addressed to
- public void setAddress(SocketAddress remote)
  - change the address a port a datagram packet is sent to

```
DatagramPacket input = new DatagramPacket(new byte[8192], 8192);
socket.receive(input);
DatagramPacket output = new DatagramPacket(
    "Hello there".getBytes("UTF-8"), 11);
SocketAddress address = input.getSocketAddress();
output.setAddress(address);
socket.send(output);
```

- public void setLength(int length)
  - change the number of bytes of data in the internal buffer

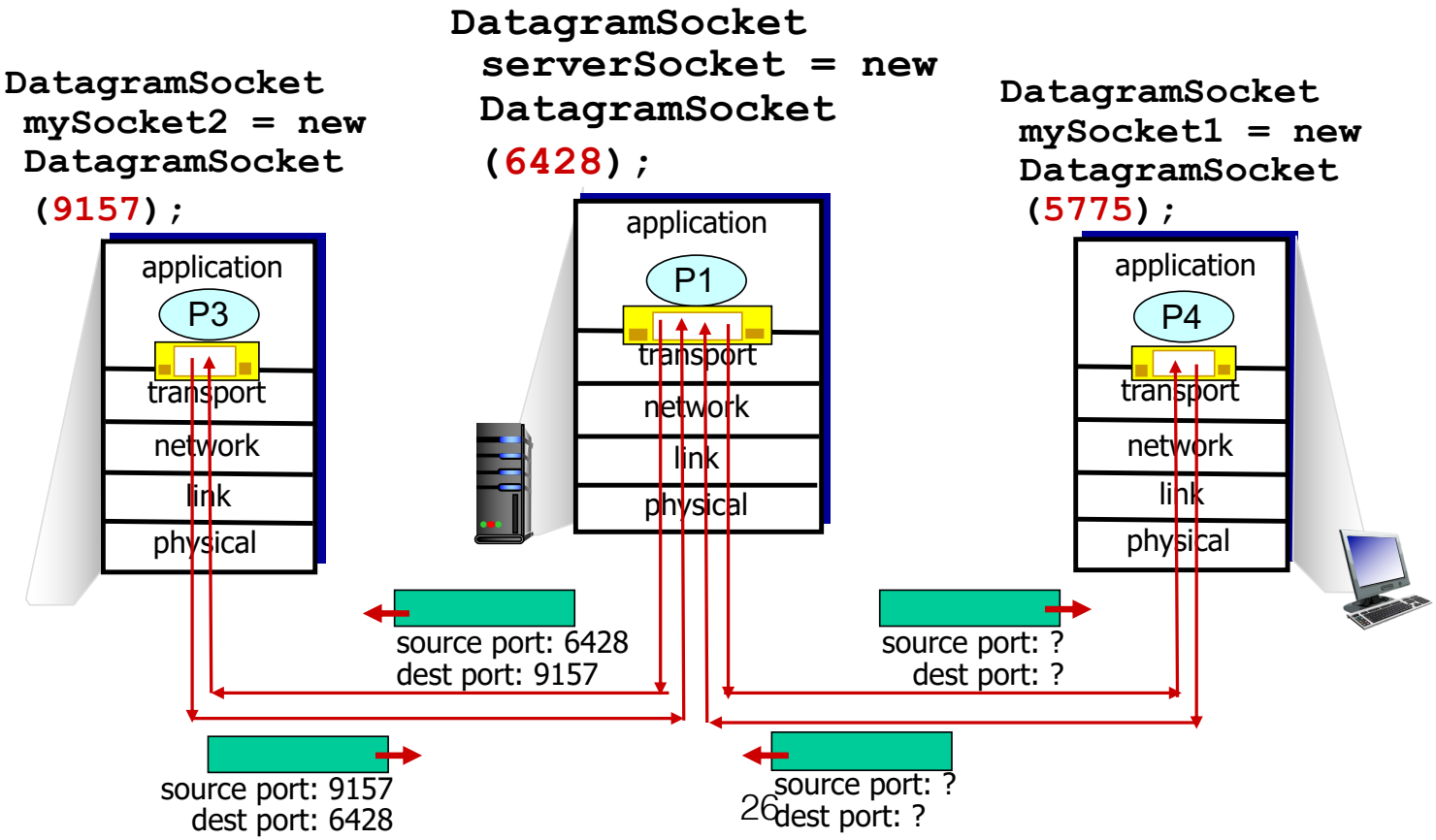
DatagramSocket Class

DatagramSocket

- Datagram socket must be opened to send or receive DatagramPacket
- Datagram socket is created and accessed through DatagramSocket class
  - `public class DatagramSocket extends Object`
- Both client and server use DatagramSocket

Notes

- Remote address and port are stored in DatagramPacket, not DatagramSocket
  - one DatagramSocket can send and receive datagrams from multiple hosts and ports



Constructors

when socket can't bound to a port

- public DatagramSocket() throws SocketException
  - create a socket bound to an anonymous port
  - example

```
try {
  DatagramSocket client = new DatagramSocket();
  // send packets...
} catch (SocketException ex) {
  System.err.println(ex);
}
```

  - use this constructor to write a client
  - this socket can both send and receive datagrams

Constructors

when socket can't be created

- public DatagramSocket(int port) throws SocketException
  - create a socket that listens for incoming datagrams on a particular port
  - use this constructor to write a server
  - this socket can both send and receive datagrams

- Note
  - two different programs can use the same port if one uses UDP and the other uses TCP (for example, Daytime protocol)

Example

```
import java.net.*;

public class UDPPortScanner {

  public static void main(String[] args) {
    for (int port = 1024; port <= 65535; port++) {
      try {
        // the next line will fail and drop into the catch block if
        // there is already a server running on port i
        DatagramSocket server = new DatagramSocket(port);
        server.close();
      } catch (SocketException ex) {
        System.out.println("There is a server on port " + port + ".");
      }
    }
  }
}
```

Constructors

- public DatagramSocket(int port, InetAddress interface) throws SocketException
  - create a socket that listens for incoming datagrams on a particular port and network interface
- public DatagramSocket(SocketAddress interface) throws SocketException
  - create a socket that listens for incoming datagrams on the port and network interface read from SocketAddress
  - example

```
SocketAddress address = new InetSocketAddress("127.0.0.1", 9999);
DatagramSocket socket = new DatagramSocket(address);
```

- protected DatagramSocket(DatagramSocketImpl impl) throws SocketException
  - provide their own implementation of UDP

Sending Datagrams

- public void send(DatagramPacket dp) throws IOException
  - send a datagram in DatagramPacket object dp to remote host specified in DatagramPacket object dp
  - example

```
theSocket.send(theOutput);
```

DatagramSocket DatagramPacket

  - IOException is thrown if there's a problem sending datagram, but it's less likely because UDP doesn't care whether datagram has been successfully received by the destination

Example

```
import java.net.*;
import java.io.*;

public class UDPDiscardClient {

  public final static int PORT = 9;

  public static void main(String[] args) {

    String hostname = args.length > 0 ?  args[0] : "localhost";

    try (DatagramSocket theSocket = new DatagramSocket()) {
      InetAddress server = InetAddress.getByName(hostname);
      BufferedReader userInput
          = new BufferedReader(new InputStreamReader(System.in));
      while (true) {
        String theLine = userInput.readLine();
        if (theLine.equals(".")) break;
        byte[] data = theLine.getBytes();
        DatagramPacket theOutput
            = new DatagramPacket(data, data.length, server, PORT);
        theSocket.send(theOutput);
      } // end while
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

read lines of user input from System.in

↓

send them to a discard server (PORT 9)
which simply discards all the data

Discard Protocol

Receiving Datagrams

- public void receive(DatagramPacket dp) throws IOException
  - receive a single UDP datagram from network and store it in DatagramPacket object dp
  - this method blocks the calling thread until a datagram arrives
- Note on the buffer size in DatagramPacket
  - datagram's buffer should be large enough to hold the data received
  - if not, receive() places as much data as it can hold in the buffer, and the rest is lost
    - 65,507 should be enough in any case

Example

```
import java.net.*;
import java.io.*;

public class UDPDiscardServer {

  public final static int PORT = 9;
  public final static int MAX_PACKET_SIZE = 65507;

  public static void main(String[] args) {
  byte[] buffer = new byte[MAX_PACKET_SIZE];

  try (DatagramSocket server = new DatagramSocket(PORT)) {
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
    while (true) {
      try {
        server.receive(packet);
        String s = new String(packet.getData(), 0, packet.getLength(), "8859_1");
        System.out.println(packet.getAddress() + " at port "
            + packet.getPort() + " says " + s);
        // reset the length for the next packet
        packet.setLength(buffer.length);
      } catch (IOException ex) {
        System.err.println(ex);
      }
     } // end while
  } catch (SocketException  ex) {
    System.err.println(ex);
  }
 }
}
```

receive datagrams and do nothing

needed?

Exercise: set MAX_PACKET_SIZE to a small number and see what happens

Closing

- public void close()
  - free the port occupied by this socket
  - examples

```
DatagramSocket server = null
try {
  server = new DatagramSocket();
  // use the socket...
} catch (IOException ex) {
  System.err.println(ex);
} finally {
  try {
    if (server != null) server.close();
  } catch (IOException ex) {
  }
}
```

OR

```
try (DatagramSocket server = new DatagramSocket()) {
  // use the socket...
}
```

Getters (of local info)

- public int getLocalPort()
  - return local port on which socket is listening

```
DatagramSocket ds = new DatagramSocket();
System.out.println("The socket is using port " + ds.getLocalPort());
```

- public InetAddress getLocalAddress()
  - return InetAddress object representing local address to which socket is bound

- public SocketAddress getLocalSocketAddress()
  - return SocketAddress object representing local address and port to which socket is bound

Managing Connections

- UDP is connectionless, but in some situations, you may want to talk only to particular host
  - e.g., networked game
- There are five methods that let the program choose a which host you can send datagrams to and receive datagrams from, while rejecting all others' packets
- public void connect(InetAddress host, int port)
  - specify that DatagramSocket will only send datagrams and receive from the specified host on the specified remote port
  - IllegalArgumentException thrown on any attempt to send/receive on other host/port
  - doesn't really create connection

Managing Connections

- public void disconnect()
  - revert what connect() did, i.e., allow send/receive from any host and port
- public int getPort()
  - return remote port that it is connected to, if and only if DatagramSocket is connected
- public InetAddress getInetAddress()
  - return remote address of remote host that it is connected to
- public SocketAddress getRemoteSocketAddress()
  - return remote address of remote host that it is connected to

Socket Options

- Six options for UDP
  - **SO_TIMEOUT**
  - **SO_RCVBUF**
  - **SO_SNDBUF**
  - SO_REUSEADDR
  - SO_BROADCAST
  - IP_TOS

SO_TIMEOUT

- Amount of time (ms) that receive() waits for an incoming datagram before throwing an InterruptedIOException
  - default 0, meaning receive() never times out
- Two methods

```
public void setSoTimeout(int timeout) throws SocketException
public int getSoTimeout() throws IOException
```

  - when needed?
    - response required within a fixed time (e.g., secure protocol)
    - want to check the host you are communicating with is dead
-

Example

```
try {
  byte[] buffer = new byte[2056];
  DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
  DatagramSocket ds = new DatagramSocket(2048);
  ds.setSoTimeout(30000); // block for no more than 30 seconds
  try {
   ds.receive(dp);
    // process the packet...
  } catch (SocketTimeoutException ex) {
    ss.close();
    System.err.println("No connection within 30 seconds");
  }
} catch (SocketException ex) {
  System.err.println(ex);
} catch (IOException ex) {
  System.err.println("Unexpected IOException: " + ex);
}
```

```
public void printSoTimeout(DatagramSocket ds) {
  int timeout = ds.getSoTimeOut();
  if (timeout > 0) {
    System.out.println(ds + " will time out after "
     + timeout + "milliseconds.");
  } else if (timeout == 0) {
    System.out.println(ds + " will never time out.");
  } else {
    System.out.println("Something is seriously wrong with " + ds);
  }
}
```

SO_RCVBUF

- Determine the size of the buffer used for network I/O
  - large buffers tend to improve performance (can store incoming datagrams more before overflowing)
  - important for UDP because if buffer is full, any incoming datagram will be dropped (without notifying its sender)
    - TCP will retransmit the lost packet in this case
  - can be limited by OS
- Two methods

```
public void setReceiveBufferSize(int size) throws SocketException
public int getReceiveBufferSize() throws SocketException
```

- Exercise
  - check default SO_RCVBUF, and try to change it to arbitrary values

SO_SNDBUF

- Buffer size used for network output
  - also limited by OS
- Two methods

```
public void setSendBufferSize(int size)  throws SocketException
public int getSendBufferSize() throws SocketException
```

Some Useful Applications

UDPPoke

- send an empty UDP packet to a specified host and port
- read a response packet from the same host

```
import java.io.*;
import java.net.*;

public class UDPPoke {

  private int bufferSize; // in bytes
  private int timeout; // in milliseconds
  private InetAddress host;
  private int port;

  public UDPPoke(InetAddress host, int port, int bufferSize, int timeout) {
    this.bufferSize = bufferSize;
    this.host = host;
    if (port < 1 || port > 65535) {
      throw new IllegalArgumentException("Port out of range");
    }

    this.port = port;
    this.timeout = timeout;
  }

  public UDPPoke(InetAddress host, int port, int bufferSize) {
    this(host, port, bufferSize, 30000);
  }

  public UDPPoke(InetAddress host, int port) {
    this(host, port, 8192, 30000);
  }
```

UDPPoke

key method

```
public byte[] poke() {
  try (DatagramSocket socket = new DatagramSocket(0)) {
    DatagramPacket outgoing = new DatagramPacket(new byte[1], 1, host, port);
    socket.connect(host, port);
    socket.setSoTimeout(timeout);
    socket.send(outgoing);
    DatagramPacket incoming
        = new DatagramPacket(new byte[bufferSize], bufferSize);
    // next line blocks until the response is received
    socket.receive(incoming);
    int numBytes = incoming.getLength();
    byte[] response = new byte[numBytes];
    System.arraycopy(incoming.getData(), 0, response, 0, numBytes);
    return response;
  } catch (IOException ex) {
    return null;
  }
}
```

- create socket and datagram packet
- send it to host
- return bytes read from host

UDPPoke

```
  public static void main(String[] args) {
    InetAddress host;
    int port = 0;
    try {
      host = InetAddress.getByName(args[0]);
      port = Integer.parseInt(args[1]);
    } catch (RuntimeException | UnknownHostException ex) {
      System.out.println("Usage: java UDPPoke host port");
      return;
    }

    try {
      UDPPoke poker = new UDPPoke(host, port);
      byte[] response = poker.poke();
      if (response == null) {
        System.out.println("No response within allotted time");
        return;
      }
      String result = new String(response, "US-ASCII");
      System.out.println(result);
    } catch (UnsupportedEncodingException ex) {
      // Really shouldn't happen
      ex.printStackTrace();
    }
  }
}
```

this class makes it easy to write a UDP client for daytime, time, chargen,..

UDPTimeClient

- send an empty UDP packet to a specified host and port
- read the current time from the same host

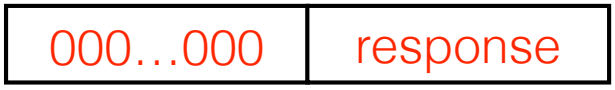
```
import java.net.*;
import java.util.*;

public class UDPTimeClient {

  public final static int PORT = 37;
  public final static String DEFAULT_HOST = "time.nist.gov";

  public static void main(String[] args) {

    InetAddress host;
    try {
      if (args.length > 0) {
        host = InetAddress.getByName(args[0]);
      } else {
        host = InetAddress.getByName(DEFAULT_HOST);
      }
    } catch (RuntimeException | UnknownHostException ex) {
      System.out.println("Usage: java UDPTimeClient [host]");
      return;
    }
```

UDPTimeClient

```
UDPPoke poker = new UDPPoke(host, PORT);
byte[] response = poker.poke();
if (response == null) {
  System.out.println("No response within allotted time");
  return;
} else if (response.length != 4) {
  System.out.println("Unrecognized response format");
  return;
}

// The time protocol sets the epoch at 1900,
// the Java Date class at 1970. This number
// converts between them.

long differenceBetweenEpochs = 2208988800L;

long secondsSince1900 = 0;
for (int i = 0; i < 4; i++) {
  secondsSince1900
      = (secondsSince1900 << 8) | (response[i] & 0x000000FF);
}

long secondsSince1970
      = secondsSince1900 - differenceBetweenEpochs;
  long msSince1970 = secondsSince1970 * 1000;
  Date time = new Date(msSince1970);

  System.out.println(time);
 }
}
```

4bytes (int in seconds) returned

secondsSince1900 (8bytes)



calculate time since 1970

CoIN LAB

UDPServer

base class to provide specific servers

```
import java.io.*;
import java.net.*;
import java.util.logging.*;

public abstract class UDPServer implements Runnable {
  private final int bufferSize; // in bytes
  private final int port;
  private final Logger logger = Logger.getLogger(UDPServer.class.getCanonicalName());
  private volatile boolean isShutDown = false;

  public UDPServer(int port, int bufferSize) {
    this.bufferSize = bufferSize;
    this.port = port;
  }

  public UDPServer(int port) {
    this(port, 8192);
  }
```

UDPServer

```
@Override
public void run() {
  byte[] buffer = new byte[bufferSize];
  try (DatagramSocket socket = new DatagramSocket(port)) {
    socket.setSoTimeout(10000); // check every 10 seconds for shutdown
    while (true) {
      if (isShutDown) return;
      DatagramPacket incoming = new DatagramPacket(buffer, buffer.length);
      try {
        socket.receive(incoming);
        this.respond(socket, incoming);
      } catch (SocketTimeoutException ex) {
        if (isShutDown) return;
      } catch (IOException ex) {
        logger.log(Level.WARNING, ex.getMessage(), ex);
      }
    } // end while
  } catch (SocketException ex) {
    logger.log(Level.SEVERE, "Could not bind to port: " + port, ex);
  }
}

public abstract void respond(DatagramSocket socket, DatagramPacket request)
    throws IOException;

public void shutDown() {
  this.isShutDown = true;
}

}
```

- create socket and listen on it
- read incoming datagram
- respond (as implemented in the subclass)

wait 10 seconds => no incoming datagram => shut down

shutDown should be called somewhere under some condition

subclass must override it

FastUDPDiscardServer

```
import java.net.*;

public class FastUDPDiscardServer extends UDPServer {

  public final static int DEFAULT_PORT = 9;

  public FastUDPDiscardServer() {
    super(DEFAULT_PORT);
  }

  public static void main(String[] args) {
    UDPServer server = new FastUDPDiscardServer();
    Thread t = new Thread(server);
    t.start();
  }

  @Override
  public void respond(DatagramSocket socket, DatagramPacket request) {
  }
}
```

UDPEchoServer

```
import java.io.*;
import java.net.*;

public class UDPEchoServer extends UDPServer {

  public final static int DEFAULT_PORT = 7;

  public UDPEchoServer() {
    super(DEFAULT_PORT);
  }

  @Override
  public void respond(DatagramSocket socket, DatagramPacket packet)
      throws IOException {
    DatagramPacket outgoing = new DatagramPacket(packet.getData(),
        packet.getLength(), packet.getAddress(), packet.getPort());
    socket.send(outgoing);
  }

  public static void main(String[] args) {
    UDPServer server = new UDPEchoServer();
    Thread t = new Thread(server);
    t.start();
  }
}
```

UDP server doesn't usually need to be multi-threaded

UDPEchoClient

```java
import java.net.*;

public class UDPEchoClient {

  public final static int PORT = 7;

  public static void main(String[] args) {

    String hostname = "localhost";
    if (args.length > 0) {
      hostname = args[0];
    }

    try {
      InetAddress ia = InetAddress.getByName(hostname);
      DatagramSocket socket = new DatagramSocket();
      SenderThread sender = new SenderThread(socket, ia, PORT);
      sender.start();
      Thread receiver = new ReceiverThread(socket);
      receiver.start();
    } catch (UnknownHostException ex) {
      System.err.println(ex);
    } catch (SocketException ex) {
      System.err.println(ex);
    }
  }
}
```

SenderThread

```
import java.io.*;
import java.net.*;

class SenderThread extends Thread {

  private InetAddress server;
  private DatagramSocket socket;
  private int port;
  private volatile boolean stopped = false;

  SenderThread(DatagramSocket socket, InetAddress address, int port) {
    this.server = address;
    this.port = port;
    this.socket = socket;
    this.socket.connect(server, port);
  }
```

SenderThread

```
  public void halt() {
    this.stopped = true;
  }

  @Override
  public void run() {
    try {
      BufferedReader userInput
          = new BufferedReader(new InputStreamReader(System.in));
      while (true) {
        if (stopped) return;
        String theLine = userInput.readLine();
        if (theLine.equals(".")) break;
        byte[] data = theLine.getBytes("UTF-8");
        DatagramPacket output
            = new DatagramPacket(data, data.length, server, port);
        socket.send(output);
        Thread.yield();
      }
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

ReceiverThread

```
import java.io.*;
import java.net.*;

class ReceiverThread extends Thread {

  private DatagramSocket socket;
  private volatile boolean stopped = false;

  ReceiverThread(DatagramSocket socket) {
    this.socket = socket;
  }

  public void halt() {
    this.stopped = true;
  }
```

ReceiverThread

```
  @Override
  public void run() {
    byte[] buffer = new byte[65507];
    while (true) {
      if (stopped) return;
      DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
      try {
        socket.receive(dp);
        String s = new String(dp.getData(), 0, dp.getLength(), "UTF-8");
        System.out.println(s);
        Thread.yield();
      } catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
}
```

DatagramChannel

Nonblocking UDP

- DatagramChannel class
  - used for nonblocking UDP applications
    - used in the same way as SocketChannel and ServerSocketChannel
  - can be registered with a Selector

- Notes on UDP
  - single datagram socket can process requests from multiple clients for both input and output
  - => effect of nonblocking mode can be marginal

Using DatagramChannel

- Just read and write ByteBuffer, as in SocketChannel
- Opening a socket
  - create channel

```
DatagramChannel channel = DatagramChannel.open();
```

  - create socket and bind it

```
SocketAddress address = new InetSocketAddress(3141);
DatagramSocket socket = channel.socket();
socket.bind(address);
```

  - or do it directly from DatagramChannel

```
SocketAddress address = new InetSocketAddress(3141);
channel.bind(address);
```

Receiving

- Method

```
public SocketAddress receive(ByteBuffer dst) throws IOException
```

- Caution
  - if datagram packet has more data than buffer can hold, extra data is thrown away without notification of the problem
    - no BufferOverflowException

UDPDiscardServerWithChannels

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;

public class UDPDiscardServerWithChannels {

  public final static int PORT = 9;
  public final static int MAX_PACKET_SIZE = 65507;

  public static void main(String[] args) {

    try {
      DatagramChannel channel = DatagramChannel.open();
      DatagramSocket socket = channel.socket();
      SocketAddress address = new InetSocketAddress(PORT);
      socket.bind(address);
      ByteBuffer buffer = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);
      while (true) {
        SocketAddress client = channel.receive(buffer);
        buffer.flip();
        System.out.print(client + " says ");
        while (buffer.hasRemaining()) System.out.write(buffer.get());
        System.out.println();
        buffer.clear();
      }
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

Sending

- Method

```
public int send(ByteBuffer src, SocketAddress target) throws IOException
```

  - return the number of bytes written

UDPEchoServerWithChannels

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;

public class UDPEchoServerWithChannels {

  public final static int PORT = 7;
  public final static int MAX_PACKET_SIZE = 65507;

  public static void main(String[] args) {

    try {
      DatagramChannel channel = DatagramChannel.open();
      DatagramSocket socket = channel.socket();
      SocketAddress address = new InetSocketAddress(PORT);
      socket.bind(address);
      ByteBuffer buffer = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);
      while (true) {
        SocketAddress client = channel.receive(buffer);
        buffer.flip();
        channel.send(buffer, client);
        buffer.clear();
      }
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

- this runs in blocking mode
- server doesn't wait for client to be ready
- much less opportunity for one client to get held up behind slow client
- blocking mode not a big problem for UDP

Connecting

- Methods

```
SocketAddress remote = new InetSocketAddress("time.nist.gov", 37);
channel.connect(remote);
```

  - limit send/receive only to designated remote host/port

```
public boolean isConnected()
```

  - true if and only if DatagramSocket is connected

```
public DatagramChannel disconnect() throws IOException
```

  - revert connect()

Reading

- Three read() methods

```
public int read(ByteBuffer dst) throws IOException
public long read(ByteBuffer[] dsts) throws IOException
public long read(ByteBuffer[] dsts, int offset, int length)
    throws IOException
```

  - can only be used on connected channels (alternative for receive)
  - more suitable for clients who know who they are talking to
  - return 0 if
    - channel is nonblocking and no packet was ready OR
    - datagram packet contained no data OR
    - buffer is full,…

Writing

- Three write() methods

```
public int write(ByteBuffer src) throws IOException
public long write(ByteBuffer[] dsts) throws IOException
public long write(ByteBuffer[] dsts, int offset, int length)
    throws IOException
```

  - can be used only for connected channels (alternative for send)
  - no guarantee to write completely
  - iterative method

```
while (buffer.hasRemaining() && channel.write(buffer) != -1) ;
```

UDPEchoClientWithChannels

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.*;

public class UDPEchoClientWithChannels {

  public  final static int PORT = 7;
  private final static int LIMIT = 100;

  public static void main(String[] args) {

    SocketAddress remote;
    try {
      remote = new InetSocketAddress(args[0], PORT);
    } catch (RuntimeException ex) {
      System.err.println("Usage: java UDPEchoClientWithChannels host");
      return;
    }

    try (DatagramChannel channel = DatagramChannel.open()) {
      channel.configureBlocking(false);
      channel.connect(remote);

      Selector selector = Selector.open();
```

```
channel.register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);

ByteBuffer buffer = ByteBuffer.allocate(4);
int n = 0;
int numbersRead = 0;
while (true) {
  if (numbersRead == LIMIT) break;
  // wait one minute for a connection
  selector.select(60000);
  Set<SelectionKey> readyKeys = selector.selectedKeys();
  if (readyKeys.isEmpty() && n == LIMIT) {
    // All packets have been written and it doesn't look like any
    // more are will arrive from the network
    break;
  }
  else {
    Iterator<SelectionKey> iterator = readyKeys.iterator();
    while (iterator.hasNext()) {
      SelectionKey key = (SelectionKey) iterator.next();
      iterator.remove();
      if (key.isReadable()) {
        buffer.clear();
        channel.read(buffer);
        buffer.flip();
        int echo = buffer.getInt();
        System.out.println("Read: " + echo);
        numbersRead++;
      }
```

```
            if (key.isWritable()) {
              buffer.clear();
              buffer.putInt(n);
              buffer.flip();
              channel.write(buffer);
              System.out.println("Wrote: " + n);
              n++;
              if (n == LIMIT) {
                // All packets have been written; switch to read-only mode
                key.interestOps(SelectionKey.OP_READ);
              }
            }
          }
        }
      }
      System.out.println("Echoed " + numbersRead + " out of " + LIMIT +
                                                              " sent");
      System.out.println("Success rate: " + 100.0 * numbersRead / LIMIT +
                                                              "%");
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

```
Wrote: 0
Read: 0
Wrote: 1
Wrote: 2
Read: 1
Wrote: 3

  ...

Wrote: 23
...
Wrote: 97
Read: 72
Wrote: 98
Read: 73
Wrote: 99
Read: 75
Read: 76
...
Read: 97
Read: 98
Read: 99
Echoed 92 out of 100 sent
Success rate: 92.0%
```

Notes

- you need to notice when data transfer is complete and shut down;
- in this case, wait one minute after all transmissions, and shut down
- in case of TCP, it would be like “just complete
- transmission and wait until all responses are received”

Closing

- Methods
  - `public void close() throws IOException`
    - free up port and any other resources the channel has been holding
  - `public boolean isOpen()`
    - true if open, false if closed

```
DatagramChannel channel = null;
try {
  channel = DatagramChannel.open();
  // Use the channel...
} catch (IOException ex) {
  // handle exceptions...
} finally {
  if (channel != null) {
    try {
      channel.close();
    } catch (IOException ex) {
      // ignore
    }
  }
}
```

```
try (DatagramChannel channel = DatagramChannel.open()) {
  // Use the channel...
} catch (IOException ex) {
  // handle exceptions...
}
```

Socket Options

Option	Type	Constant	Purpose
SO_SNDBUF	`StandardSocketOptions.`	`Integer`	Size of the buffer used for sending datagram packets
SO_RCVBUF	`StandardSocketOptions.SO_RCVBUF`	`Integer`	Size of the buffer used for receiving datagram packets
SO_REUSEADDR	`StandardSocketOptions.SO_REUSEADDR`	`Boolean`	Enable/disable address reuse
SO_BROADCAST	`StandardSocketOptions.SO_BROADCAST`	`Boolean`	Enable/disable broadcast messages
IP_TOS	`StandardSocketOptions.IP_TOS`	`Integer`	Traffic class
IP_MULTICAST_IF	`StandardSocketOptions.IP_MULTICAST_IF`	`NetworkInterface`	Local network interface to use for multicast
IP_MULTICAST_TTL	`StandardSocketOptions.IP_MULTICAST_TTL`	`Integer`	Time-to-live value for multicast datagrams
IP_MULTICAST_LOOP	`StandardSocketOptions.IP_MULTICAST_LOOP`	`Boolean`	Enable/disable loopback of multicast datagrams

Methods

- Three methods

```
public <T> DatagramChannel setOption(SocketOption<T> name, T value)
    throws IOException
public <T> T getOption(SocketOption<T> name) throws IOException
public Set<SocketOption<?>> supportedOptions()
```

  - example

```
try (DatagramChannel channel = DatagramChannel.open()) {
  channel.setOption(StandardSocketOptions.SO_BROADCAST, true);
  // Send the broadcast message...
} catch (IOException ex) {
  // handle exceptions...
}
```

DefaultSocketOptionValues

```
import java.io.IOException;
import java.net.SocketOption;
import java.nio.channels.DatagramChannel;
public class DefaultSocketOptionValues {

  public static void main(String[] args) {
    try (DatagramChannel channel = DatagramChannel.open()) {
      for (SocketOption<?> option : channel.supportedOptions()) {
        System.out.println(option.name() + ": " + channel.getOption(option));
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```