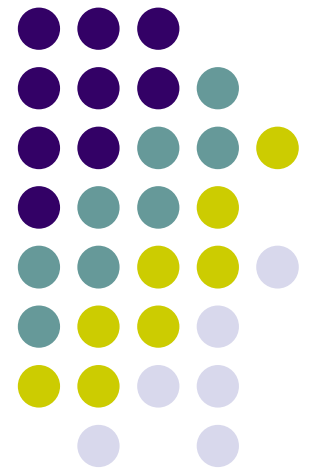
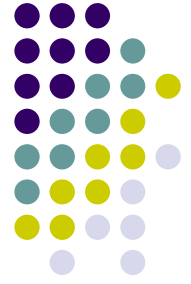


CS 352-Socket Programming & Threads

Dept. of Computer Science
Rutgers University





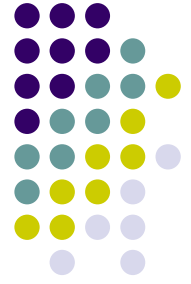
Major Ideas

- Stream and Datagram sockets
 - Byte streams, datagrams
 - Ports
 - Fully qualified communication
- Client and server programming
 - Java classes implementing sockets
 - Java Stream I/O
- Threads
 - Concurrent execution
 - Creating a new thread
 - Synchronization

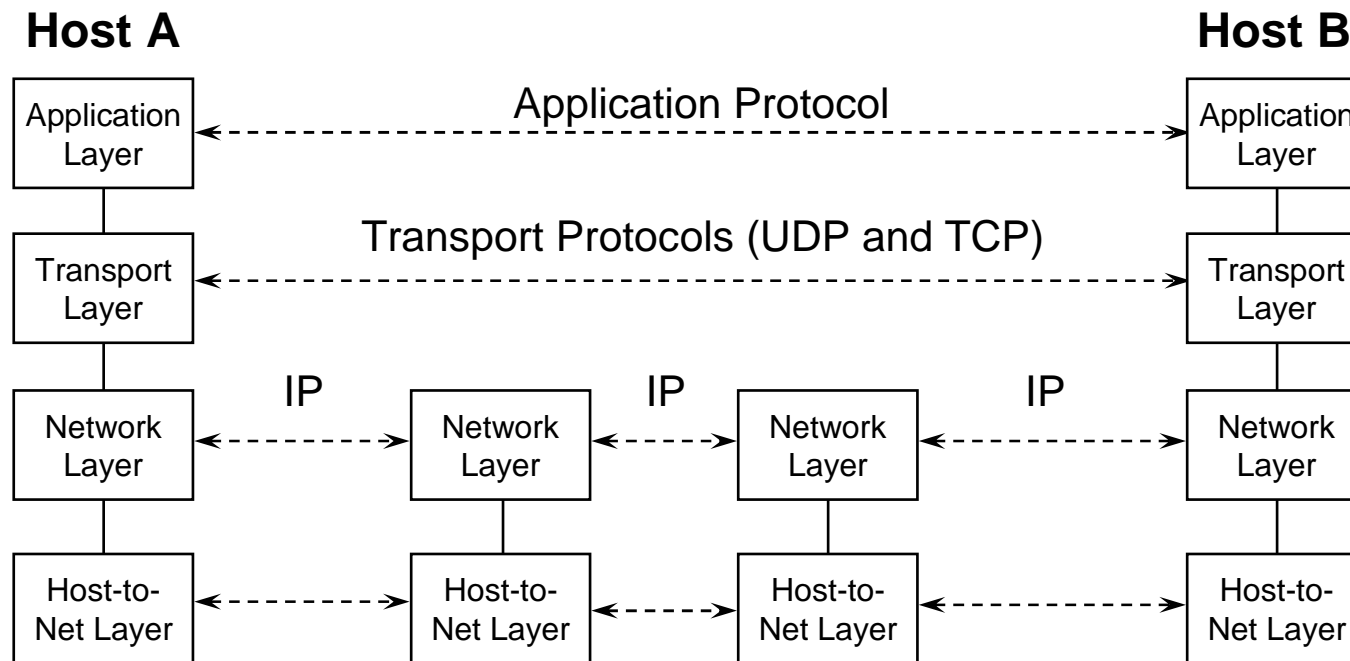


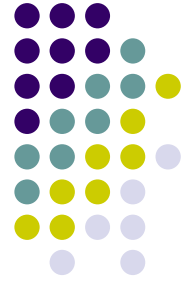
2 kinds of sockets

- Stream sockets
 - Abstract a byte-stream communications channel
 - Connection oriented
 - Follows a Circuit-switching model
- Datagram sockets
 - Abstract sending and receiving network packets
 - Unit of data are discrete byte arrays
 - Follows a Message switching model
- Typically, sockets run on Internet Protocols
 - But not necessarily! Sockets can be implemented on any protocol supporting stream or datagram abstractions

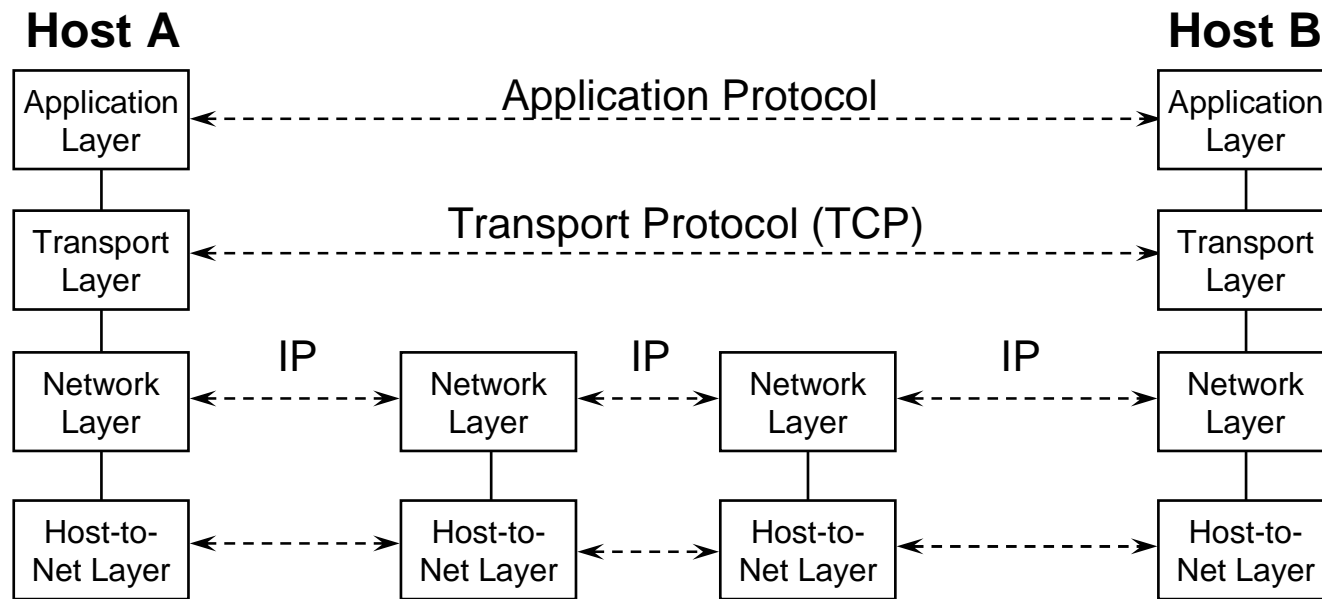


IP Layering Architecture





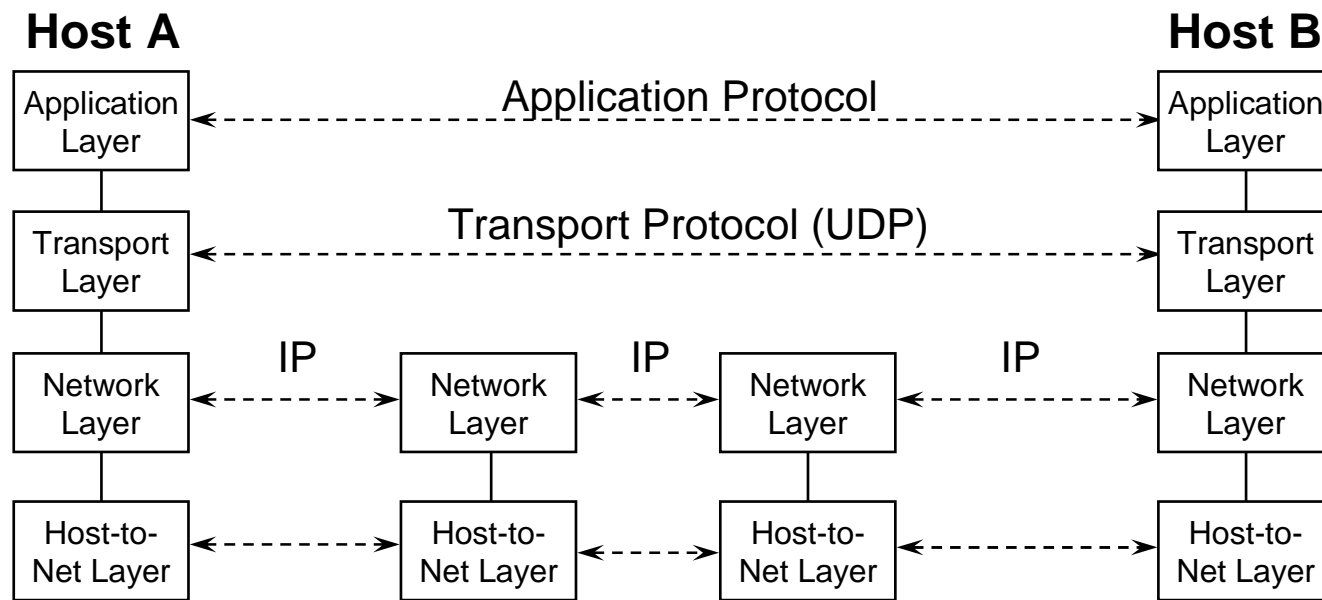
Stream Service



On the Internet, the Transmission Control Protocol (TCP) implements a byte stream network service



Datagram Service



On the Internet, the User Datagram Protocol (UDP) implements a datagram (packet) service

Abstract Stream Socket Service



- Asymmetric set-up, circuit abstraction
 - Server is passive, waits for connections
 - Client initiates the connections
- Bi-directional, continuous byte stream
 - TCP is free to break up and reorder the data however it likes as long as the user sees an ordered byte stream
 - But often doesn't
- Tries really hard when packets are lost
 - In reality can't recover from all errors
 - but timeouts on the order of 15 minutes

Abstract Datagram Socket Service



- No circuit abstraction, just send when ready
 - Server is passive, waits for datagrams
 - Client initiates the send
- Discrete packets (up to 64Kb long for UDP)
 - UDP/IP maintains packet integrity
 - All data in packet arrives or doesn't (e.g. no half-packet)
 - Even if lower layers fragment
 - No data corruption (e.g. bit errors)
- Best effort
 - Does not retransmit lost packets
 - 1 lost fragment -> whole packet is lost



Internet Addressing

- Layer 3 addressing
- Each IP entity (E.g. host) has a 4-byte address
 - As decimal: A.B.C.D
 - Can also be written as hexadecimal and binary!
- Recall the Domain Name System (DNS) translates symbolic names to IP addresses
 - E.g. remus.rutgers.edu -> 128.6.13.3



Ports

- Layer 4 addressing
- 2 byte port number differentiates destinations within an IP address
 - E.g. the mail server (25) vs. the web server (80)
- Think of ports as routing within a given computer
 - Only 1 program can have a port “open” at a time
 - Get “address already in use” error/exception

Fully Qualified communication



- *Fully qualified* IP communication at layer-4 requires 5 tuples:
 - A protocol identifier (UDP, TCP)
 - Source IP address, source port number
 - Destination IP address, destination port number
- Only with a fully qualified connection can we uniquely identify a connection
 - And thus get the data to the correct program!



Stream Sockets in Java

- **InetAddress** class
 - Object for containing and using IP addresses
 - methods for viewing and changing IP addresses and symbolic names
- **InPutStream, OutPutStream** classes
 - Send and receive bytes from a socket
- **Socket and ServerSocket**, classes
 - Both are TCP communication objects
 - Abstract asymmetry of client/server communication
 - Contain the stream objects once socket is connected



Stream Client Algorithm

- Create a socket object
 - Set destination address and port number
 - In constructor implies making a connection
- Get Input and Output Streams
- Call `read()`, `write()` and `flush()` methods
- `Close()` method when done
 - Be nice to the system, port use



Stream Client side

```
String machineName;  
int port;  
Socket sock = null;  
InputStream in;  
OutputStream out;  
  
sock = new Socket(machineName, port);  
in = sock.getInputStream();  
out = sock.getOutputStream();  
...  
bytesRead = in.read(byteBuffer);
```



Stream Server Algorithm

Create a serverSocket on a port

Loop:

- wait on accept() method for a new client
- accept() returns a new socket
- get input and output streams for this new socket
- close the socket when done



Stream Server Receive

```
ServerSocket ss = new ServerSocket(port);  
Socket nextClientSock;  
  
while ( ... ) {  
    nextClientSock = ss.accept(); // new socket  
    // the return socket is "bound" and can  
    // be used to send/receive data  
    in = nextClientSock.getInputStream();  
    out = nextClientSock.getOutputStream();  
  
}
```

Stream Server echo using exceptions

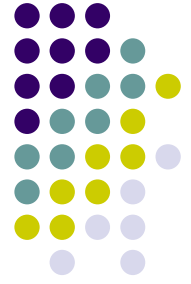


```
try {
while( (bytesRead = in.read(byteBuffer)) != -1)
{
    out.write(byteBuffer,0,bytesRead);
    out.flush();
    totalBytesMoved += (long) bytesRead;
}
nextClientSock.close();
} catch (IOException e) {
    System.out.println("Socket Error");
    nextClientSock.close();
}
```



Datagram Sockets in Java

- **InetAddress** class
 - Object for containing and using IP addresses
 - methods for viewing and changing IP addresses and symbolic names
- **DatagramPacket** class
 - Send and receive packets from a socket
 - Data bytes with associated (IP, port) information
- **DatagramSocket** class
 - Used at both client and server side
 - Input and output are datagram packets
 - User is responsible for extracting the data bytes information



Datagram Client Algorithm

- Create a socket object
- Construct DatagramPacket with server (IP,port) information
- Send packet through client socket
- Receive packet through client socket and extract corresponding data information
- Close() method when done
 - No contact with the server

Sample Datagram Echo Client



```
int port;
InetAddress address;
DatagramSocket socket = new DatagramSocket();
DatagramPacket packet;
byte[] sendBuf = new byte[256];
// this code sends the packet
byte[] buf = new byte[256];
InetAddress address = InetAddress.getByName(ServerName);
DatagramPacket packet = new DatagramPacket(buf,
    buf.length, address, port);
socket.send(packet);
// get the response from the server
packet = new DatagramPacket(buf, buf.length)
socket.receive(packet);
String received = new String(packet.getData());
System.out.println("Data as a String is:" + received);
```



Datagram Server Algorithm

Create a DatagramSocket on a designated port

Loop:

- wait on receive() method on socket for a new client
- Process the received DatagramPacket and serve the client request
- Construct DatagramPacket and send back to the client identified by the (IP,port) from the incoming packet



Sample Datagram Echo Server

```
Socket = new DatagramSocket(port);
int port;
byte[] buf = new byte[256];
DatagramPacket packet;
while (1) {
    // wait for the packet from the client
    DatagramPacket packet = new DatagramPacket(buf,
        buf.length);
    // server thread is blocked here
    socket.receive(packet);
    // echo packet back to the client
    InetAddress address = packet.getAddress(); //return address
    port = packet.getPort(); // return port
    packet = new DatagramPacket(buf, buf.length, address,
        port);
    socket.send(packet);
}
```



Other useful methods

- `inetAddress socket.getLocalAddress()`
 - get the machine's local address
- `socket.setSoTimeout(int milliseconds)`
 - block only for int milliseconds before returning
- `socket.toString`
 - get the IP address and port number in a string



Important Points

- Work with bytes, not strings, if possible
 - Conversions don't always work like you think
- Can use `BufferedReader` and `BufferedWriter` around base classes
 - But don't forget to flush!

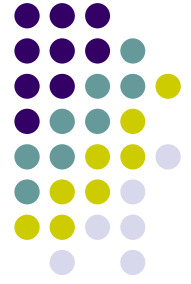


Using Strings

- Strings must be converted to bytes
- Use wrappers around basic byte stream
- Example:

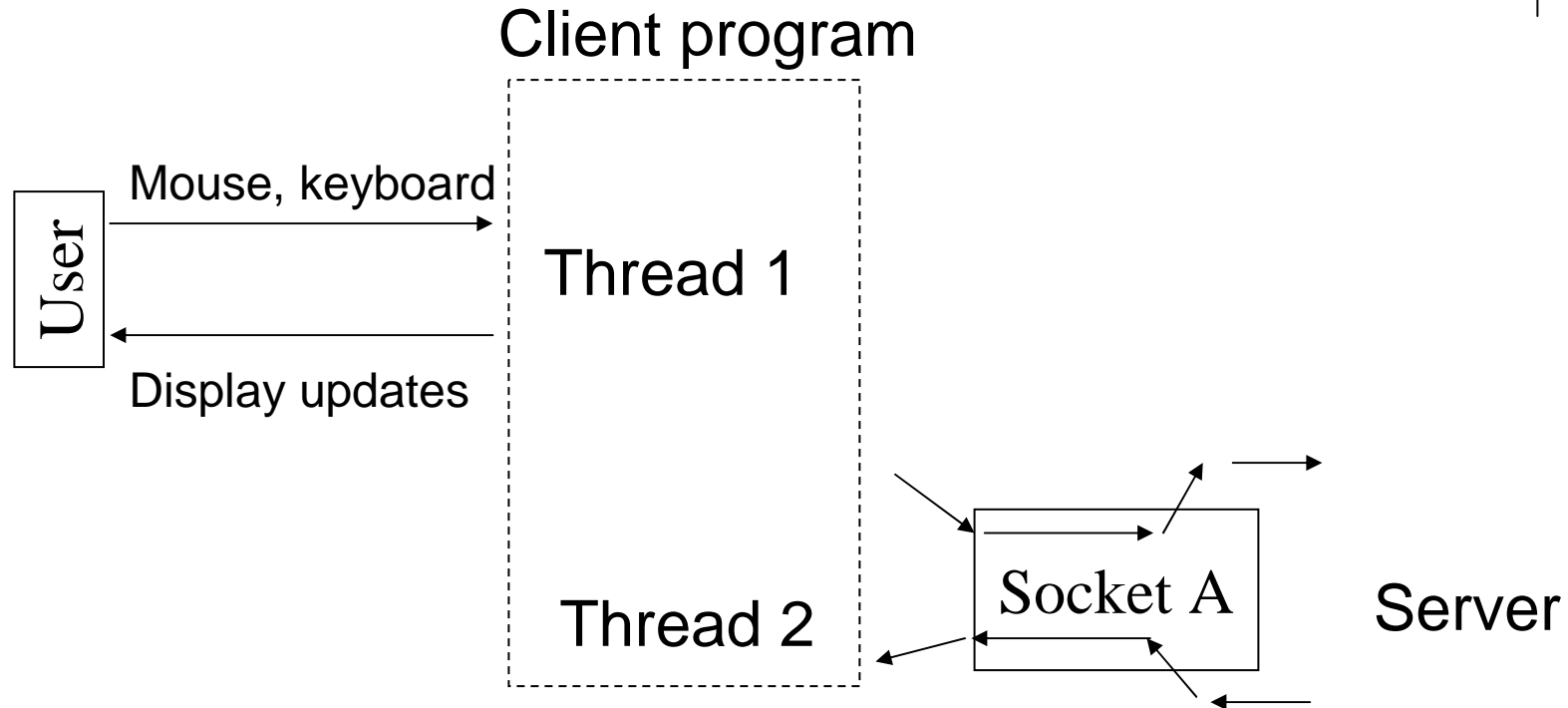
```
String InputLine;  
out = new  
    PrintWriter(sock.getOutputStream(), true);  
in = new BufferedReader( new InputStreamReader(  
    sock.getInputStream()));  
InputLine = in.readLine();  
out.println(InputLine);
```

Network Programming & Threads



- Network code involves logical simultaneous movement of data
- Multiple levels of movement at once
 - E.g. From the client to server and server to client
 - E.g. Between multiple clients and servers.
- Clients and servers must wait for events
 - While a client or server is waiting for network data, nothing else happens in your program
 - E.g. updating the screen, responding to the mouse and keyboard

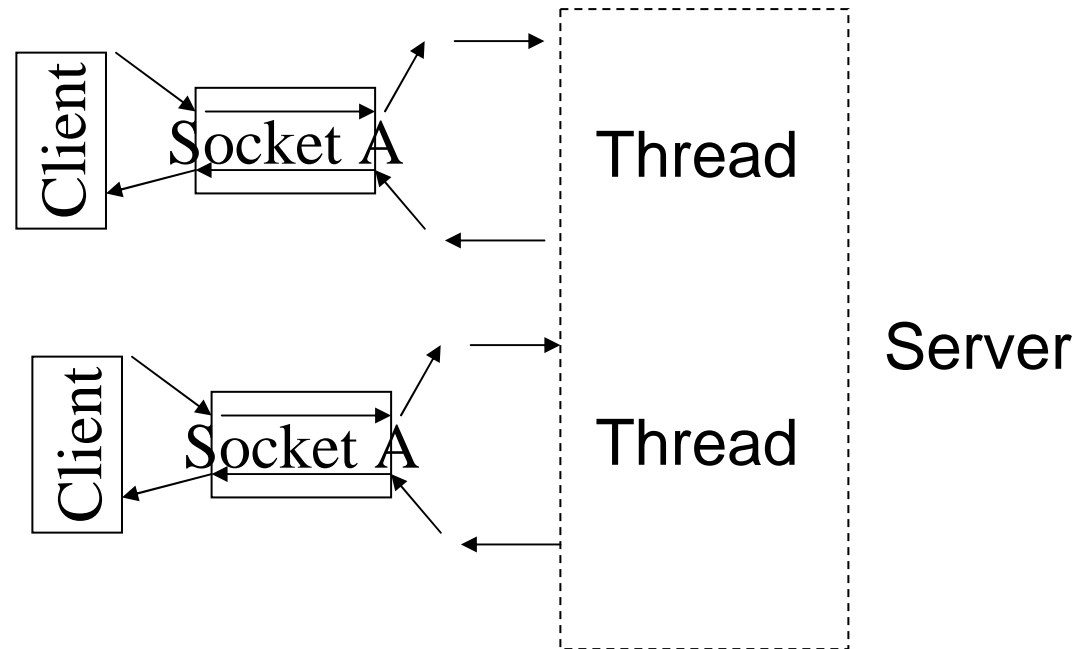
Multiple Logical Tasks for the client



- Need to support both the user and the server



Multiple Clients



- Need to support many channels at once
- Allows each client independent service



Concurrency in Java: Threads

- Threads solve these problems by abstracting multiple *simultaneous* execution paths in the program
- Usage
 - Create a class that extends thread
 - Must override the `run` method
 - Instantiate an object of that class
 - Invoking `run` method starts a new execution path
 - After the caller returns, the `run` method (and any methods it called) is still going!
 - Calling `join` method waits for the `run` method to terminate

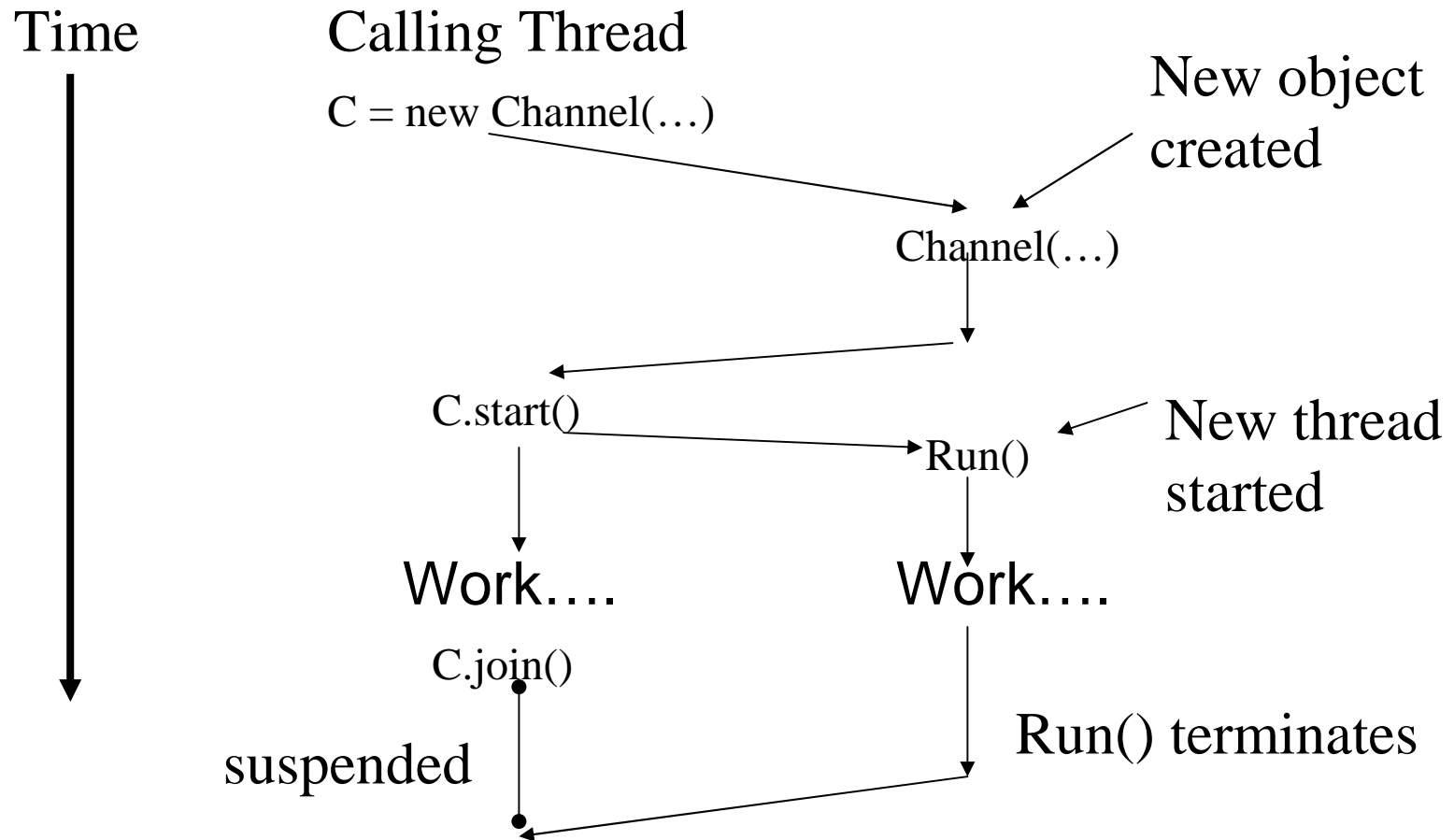


Threads in Java

```
Class Channel extends Thread {
    Channel(...) {    // constructor
    }
    public void run() {
        /* Do work here */
    }
}

/* other code to start thread */
Channel C = new Channel(); // constructor
C.start();    // start new thread in run method
C.join();    // wait for C's thread to finish.
```

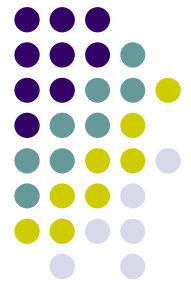
Threads in Java



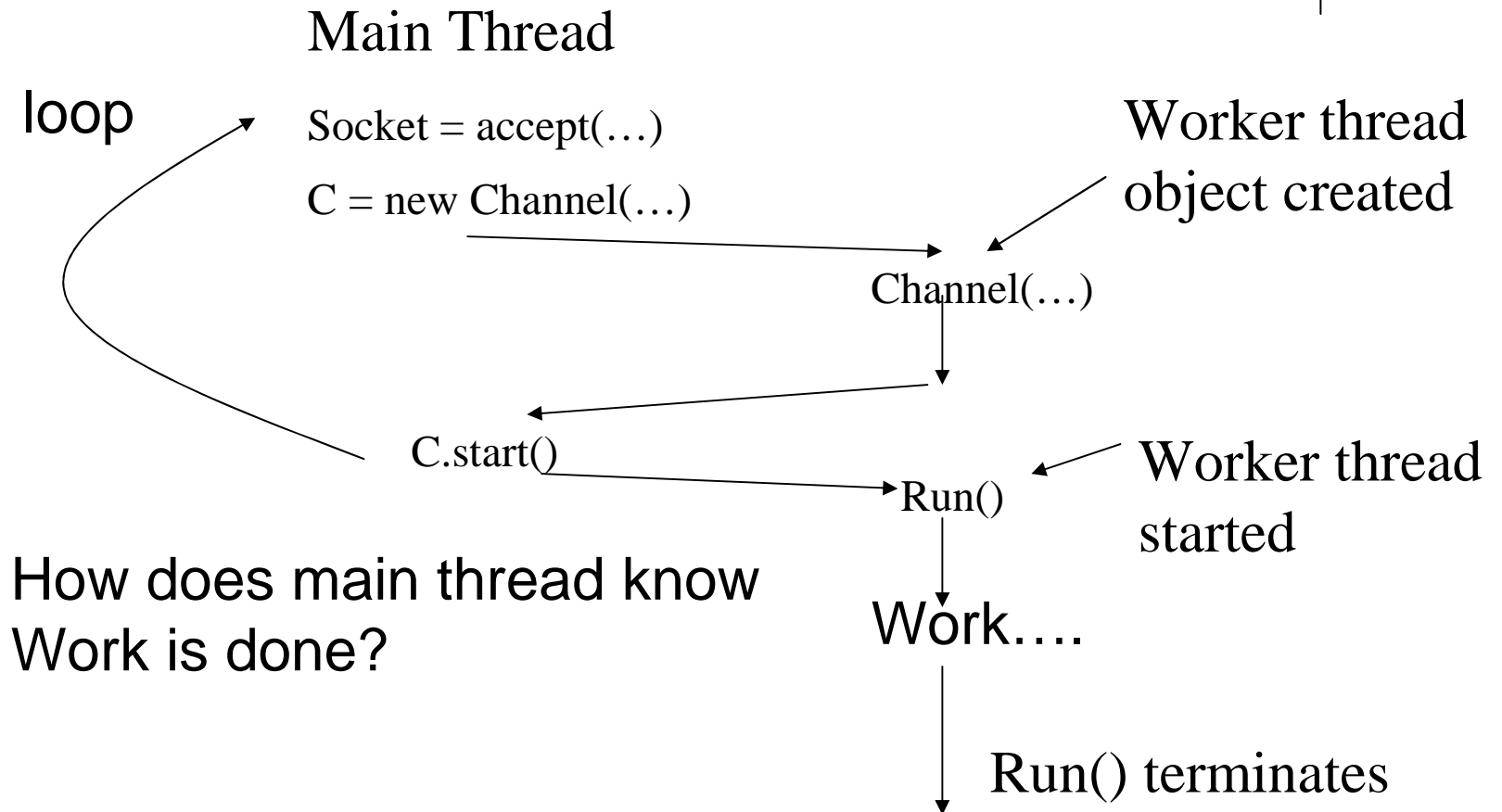


Example when to use threads

- Server main thread waits for client requests
- After `accept()` method, main thread creates a new worker thread to handle this specific socket connection
- Main thread returns to accept new connections
- Work thread handles this request
- Provides logical independent service for each client



Threads in Java





Synchronized Methods

- Can add keyword `synchronized` to a method
- Result is that at most 1 thread can execute the method at a time
- Use for altering data structures shared by the threads
 - Be careful! No operations should block in the synchronized method or program can get stuck.