CS455 - Introduction To Distributed Systems
[Lab Session 4]

Jason Stock
Computer Science
Colorado State University

Colorado State University

Topics Covered in Today's Lab

- Quiz 2 Review
- HW1 questions and discussion
- Introduction to HW2
- Introduction to Java NIO

NOTE: Feel free to bring laptops, code, and questions!

Quiz 2 Review [1/4]

1. The OSI architecture does not imply a strict layering of the protocol stack
   - **False**
2. The Internet architecture does not imply a strict layering of the protocol stack
   - **True**
3. The process-per-protocol model results in minimal overheads and highest possible efficiency when processing messages.
   - **False**

Quiz 2 Review [2/4]

4. In the Internet architecture, every node (i.e. hosts, routers, switches, etc.) does not need to include support for the IP protocol.
   - **False**
5. IP attempts to recover from reassembly failure at the receiver side by sending retransmission request to the source
   - **False**
6. Packet formats are designed to align on 32-bit boundaries to simplify the task of processing them in software.
   - **True**

Quiz 2 Review [3/4]

7. The demultiplexing key is useful for identifying the higher-level protocol that a message should be sent to.
    - **True**
8. The presence or absence of Options in the IPv4 header cannot be determined solely from the length of the header (HLen).
    - **False**

Quiz 2 Review [4/4]

9. Consider an IPv4 packet whos checksum field indicates that the header has been corrupted in transit. The IPv4 checksum also doubles as an error-correcting code and can be used to recover from data corruptions.
    - **False**
10. Packets transmitted using IPv4 or IPv6 can never be duplicated - i.e., a packet can never be received more than once.
    - **False**

HW1 Questions and Discussion…

- Programming Component
  - **Due Date**: Wednesday, February 19, 2020 @ 5:00 PM (MST)
- Written Component
  - **Released**: Wednesday, February 19, 2020
  - **Due Date**: Friday, February 21, 2020 @ 5:00 PM (MST)

Introduction to HW2

Due Date: Wednesday March 11, 2020 @ 5:00 PM (MST)

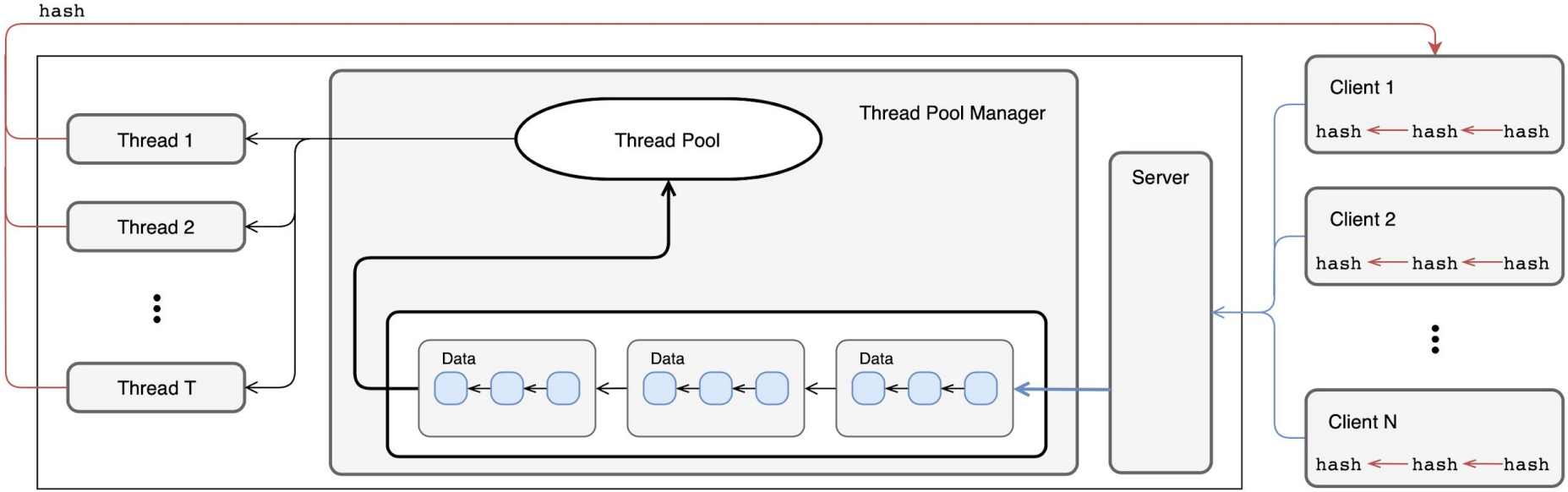
High Level Overview [1/3]

- Developing a server to handle network traffic with the design of a **custom thread pool**
- Will be able to handle 100+ connections from clients on other machines
- Worker threads will perform tasks related to network communication
  - **Accepting** incoming network connections
  - **Receiving** data over these network connections
  - **Sending** data back over any connection

High Level Overview [2/3]

- The Clients
  - Generate random messages as a `byte[]` to send to the Server
  - Maintain a local list of hashed messages
- Server Node
  - Manages a thread pool that will perform a variety of tasks
  - Acknowledges requests by recomputing hashes on the data

High Level Overview [3/3]



Thread Pool

- The Thread Pool remains fixed -- all threads are created during server initialization
- Comprised of default number of threads passed as a command line argument
- Individual threads will process a variety of different tasks throughout their lifetime

Thread Pool Manager

- Maintains a collection of tasks that needs to be performed by the threads
- Task are added to the tail of a FIFO `LinkedBlockingQueue` when
  - A client is **registering** with the server
  - **Reading incoming data** from a client
  - An intermediate batch (data queue) has reached a length of **batch-size**
  - A **batch-time** has expired since the previous unit was processed
- An available worker thread is assigned to the task when it becomes available

Software Test and Verification

- Design units of work as an `interface`
  - An abstract type specifying a behavior that generic classes can implement
- Test your Thread Pool independently and prior to servicing network traffic
- Queue a set of test tasks to ensure the Thread Pool is functional

Introduction to Java NIO

Comparison with IO

Java IO	Java NIO
Blocking IO	Non-Blocking IO
Stream Oriented	Buffer Oriented
Large Messages	Smaller Messages
More Threads	Selector
Intuitive	Efficient
Restaurant Waiter	*Restaurant Bartender*

Selectors

- Created with: `Selector selector = Selector.open();`
- Tool for managing channels - similar to a `Set`
- Allows selecting channels that are to be read from / written to
- Channels must be registered with a `Selector` before they can be used

Channels

- Instead of `ServerSocket` (java.io) a `ServerSocketChannel` (java.nio) is used
  - E.g., `ServerSocketChannel serverSocket = ServerSocketChannel.open();`
- Instead of `Socket` (java.io) a `SocketChannel` (java.nio) is used
  - E.g., `SocketChannel socketChannel = serverSocket.accept();`
- Does `serverSocket.accept()` still block? **No**
- You only call `accept()` on `ServerSocketChannel`s that have incoming connections
- How do you know that there is an incoming connection without calling `accept()`?

Registering Channels

- A `Selector` will examine and determine which NIO channels are ready
- All channels need to be registered with the `Selector` using a `SelectionKey`
  - `SelectionKey.OP_ACCEPT`
  - `SelectionKey.OP_CONNECT`
  - `SelectionKey.OP_READ`
  - `SelectionKey.OP_WRITE`

Registering `ServerSocketChannels`

- `ServerSocketChannel`s only support the accepting of new connections, which is denoted by `SelectionKey.OP_ACCEPT`
- Bind the `ServerSocketChannel` to a host and port to accept new connections
  - E.g.,

```
Selector selector = Selector.open(); // created once
ServerSocketChannel serverSocket = ServerSocketChannel.open();
serverSocket.socket().bind( new InetSocketAddress( host, port ) );
serverSocket.configureBlocking( false );
serverSocket.register( selector, SelectionKey.OP_ACCEPT );
```

Registering `SocketChannels` [1/2]

- `SocketChannel`s only support connecting, reading and writing which is denoted by `SelectionKey.OP_CONNECT | SelectionKey.OP_READ | SelectionKey.OP_WRITE`
- Register the `SocketChannel` to read incoming data
  - E.g.,

```
SocketChannel socketChannel = serverSocket.accept();
socketChannel.configureBlocking( false );
socketChannel.register( selector, SelectionKey.OP_READ );
```

Registering `SocketChannels` [2/2]

- Create a `SocketChannel` that connects to a remote `ServerSocketChannel`
  - E.g.,

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.configureBlocking( false );
socketChannel.connect( new InetSocketAddress( host, port ) );
```

Selecting Channels [1/2]

- Iterate through the `Selector` to check if a `key` is acceptable, writable, or readable
- `SelectionKey`s are objects that maintain references to channels (sockets)
- `key.isAcceptable()` will return true if that channel associated with that key has an incoming connection that it can accept
- This is how you can call `serverSocket.accept()` without ever blocking

Selecting Channels [2/2]

```
while ( true ) {
    if ( selector.selectNow() == 0 ) continue;
    Iterator<SelectionKey> iter = selector.selectedKeys().iterator();
    while ( iter.hasNext() ) {
        SelectionKey key = iter.next();
        if ( key.isAcceptable() ) {
            // a connection was accepted by a ServerSocketChannel
        }
        else if ( key.isReadable() ) {
            // a channel is ready for reading
        }
        iter.remove();
    }
}
```

SelectionKeys

- `SelectionKey`s are objects that maintain references to channels (sockets)
  - `SocketChannel socketChannel = ( SocketChannel ) key.channel();`
- An object can be attached to a `SelectionKey`
  - Useful for keeping buffers with their associated key, or
  - Indicating that the key is currently in use by a thread
  - E.g.,

```
key.attach( new Object() );
...
Object o = ( Object ) key.attachment();
...
key.attach( null );
```

Reading Data from `SocketChannels`

- Reading from a channel never blocks - causes complications because messages might not arrive as entire chunk
- **Solution**: allocate a buffer of appropriate length, and read until buffer is full
  - E.g.,

```
ByteBuffer buffer = ByteBuffer.allocate( Constants.BUFFER_SIZE );
int bytesRead = 0;
while ( buffer.hasRemaining() && bytesRead != -1 ) {
    bytesRead = socketChannel.read( buffer );
}
```

- Do not forget to `rewind()` the buffer before getting the data from it

Writing Data to SocketChannels

- Similarly, channels will not wait to write all of data
- **Solution**: write from a buffer, and write until no data remains
  - E.g.,

```
ByteBuffer buffer = ByteBuffer.wrap( byteArray );
while ( buffer.hasRemaining() )
{
    socketChannel.write( buffer );
}
```

Questions and Discussion…