CS250: FOUNDATIONS OF COMPUTER SYSTEMS [SOFTWARE/NETWORKING]

Lugging a torrent of bits
From here to there
And through thin air

With fidelity ... for an error
begets a retransmission and then another

What's done to a bit, is done to the next
Be it a blockchain or a simple text

SHRIDEEP PALLICKARA
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Why is Python used so heavily if it's slow?
- Does stack have some role in the StackOverflowError?
- Is bytecode specific to Java?

Topics covered in this lecture

- Functions
- Execution of nested functions
- Networking

3

FUNCTIONS

Any problem in computer science can be solved with another level of indirection. Except for the problem of too many layers of indirection.
David Wheeler

4

Functions

- Every programming language is characterized by a fixed set of built-in operations
- In addition, high-level and some low-level languages offer the great freedom of **extending this fixed repertoire**
  - With an open-ended collection of **programmer-defined operations**
- Depending on the language, these canned operations are typically called subroutines, procedures, methods, or **functions**
  - We will collectively call these functions

Functions: the bread and butter of modular programming

- Functions are **standalone** programming units that are allowed to *call each other* for their effect
  - For example, `solve` can call `sqrt`
    - And `sqrt`, in turn, can call `power`
- This calling sequence can be as **deep** as we please, as well as **recursive**

Typically, the calling function (the caller) **passes arguments** to the called function (the callee)

- The **caller** *suspends* its execution until the callee completes its execution
- The **callee uses the passed arguments** to execute or compute something and then returns a value (which may be void) to the caller
- The caller then snaps back into action, **resuming** its execution

When one function (the caller) calls a function (the callee), someone must take care of the following:

- **Save the return address**, which is the address within the caller's code to which execution must return *after* the callee completes its execution
- **Save the memory resources** of the caller
- **Allocate** the memory resources required by the callee
- Make the **arguments** passed by the caller available to the callee's code
- **Start executing** the callee's code

When the callee terminates and returns a value, someone must take care of the following overhead:

- Make the callee's **return value** available to the caller's code
- **Recycle** the memory resources used by the callee
- **Reinstate** the previously saved memory resources of the caller
- **Retrieve** the previously saved return address
- **Resume executing** the caller's code, from the return address onward

Blissfully,

- High-level programmers don't have to ever think about all these nitty-gritty chores
- The assembly code generated by the **compiler** handles them
  - Stealthily and efficiently

In well-designed languages, built-in commands & programmer-defined functions have the same look and feel [1/2]

- For example, to compute $x + y$ using a stack machine, we push x, push y, and `add`
- In doing so, we expect the `add` implementation to pop the two top values off the stack, add them up, and push the result onto the stack

11

In well-designed languages, built-in commands & programmer-defined functions have the same look and feel [2/2]

- Suppose now that either we, or someone else, has written a power function designed to compute x^y
  - To use this function, we follow exactly the same routine: we push x, push y, and `call power`

12

This consistent calling protocol allows composing primitive commands and function calls seamlessly

- For example, expressions like $(x + y)^3$ can be evaluated using
  - push x, push y, add, push 3, `call power`
- The only difference between applying a primitive operation and invoking a function is the keyword `call` preceding the latter
- Everything else is exactly the same

Applying a primitive operation and invoking a function is exactly the same …

- Both operations require the caller to **set the stage** by pushing arguments onto the stack
- Both operations are expected to **consume their arguments**, and
- Both operations are expected to **push return values** onto the stack

Computing the hypotenuse

- $\sqrt{a^2 + b^2}$

```
Function main()
    Push 3
    Push 4
    call hypot
    return

Function hypot(x,y)
    Push x
    Push x
    Call mult
    Push y
    Push y
    Call mult
    add
    Call sqrt
    Return
```

```
Function mult(x,y)
    ....
    Return

Function sqrt(num)
    ...
    Return
```


During run-time, each function sees a **private world**, consisting of its own working stack and memory segments

- These separate worlds are connected through two "wormholes"
  - When a function says `call mult?`
    - The arguments that it pushed onto its stack prior to the call are somehow passed to the argument segment of the callee
  - Likewise, when a function says return?
    - The last value that it pushed onto its stack just before returning is somehow copied onto the stack of the callee
    - Replacing the previously pushed arguments

A computer program consists of typically several and possibly many functions [1/2]

- Yet at any given point during run-time, only a few of these functions are actually doing something
- We use the term **calling chain** to refer, conceptually, to all the functions that are currently involved in the program's execution
- When a VM program starts running, the calling chain consists of one function only, say, `main`

A computer program consists of typically several and possibly many functions [2/2]

- At some point, `main` may call another function, say, `foo`, and that function may call yet another function, say, `bar`
  - At this point the **calling chain** is `main` → `foo` → `bar`
- *Each function* in the calling chain **waits** for the function that it called to return
- Thus, the only function **truly active** in the calling chain is the *last one*
  - Which we call the **current function**, meaning the currently executing function

Assisting functions in getting to do their work

- In order to carry out their work, functions normally use **local and argument variables**
  - These variables are **temporary**:
    - The memory segments that represent them must be allocated when the function starts executing and
    - Can be **recycled** when the function returns
- This memory management task is complicated by the requirement that function calling is allowed to be **arbitrarily nested** as well as **recursive**

COLORADO STATE UNIVERSITY | Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT | SOFTWARE & NETWORKING | L16.19

19

Each function lives and executes in its own private world

- During run-time, each function call must be **executed independently** of all the other calls
  - And maintain its own stack frame, local variables, and argument variables
- How can we implement this **unlimited nesting** mechanism and the *memory management* tasks associated with it?

COLORADO STATE UNIVERSITY | Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT | SOFTWARE & NETWORKING | L16.20

20

The property that makes this housekeeping task tractable is the **linear nature** of the call-and-return logic

- Although the function calling chain may be arbitrarily deep as well as recursive
  - At any given point in time, **only one function executes** at the **chain's end**
  - While all the other functions up the calling chain are waiting for it to return
- This **Last-In-First-Out** processing model lends itself perfectly to the stack data structure, which is also LIFO

Looking at the mechanics a little closer … [1/2]

- Assume that the current function is `foo`
- Suppose that `foo` has already pushed some values onto its working stack and has modified some entries in its memory segments
- Suppose that at some point `foo` wants to call another function, `bar`, for its effect
- At this point we have to put `foo`'s execution on hold **until** `bar` will terminate its execution

Looking at the mechanics a little closer … [2/2]

- Now, putting `foo`'s working stack on hold is not a problem:
  - Because the stack **grows only in one direction**
  - The working stack of `bar` **will never override** previously pushed values
- Therefore, saving the working stack of the caller is easy —
  - We get it "for free" thanks to the linear and unidirectional stack structure

But how can we save foo's memory segments?

- If we wish to put these segments on hold?
- We can push their pointers onto the stack and pop them later
  - When we'll want to bring `foo` back to life

Frames and multi-function settings

- We use the term **frame** to refer, collectively, to the **set of pointer values** needed for *saving and reinstating* the function's **state**
- We see that once we move from a single function setting to a multifunction setting?
  - The humble stack begins to attain a rather formidable role in our story

When handling the `call functionName` command

- The runtime **pushes the caller's frame** onto the stack
- At the end of this housekeeping, we are ready to jump to executing the callee's code

Returning from the callee to the caller when the former terminates is trickier

- Because the return command specifies no return address
- The caller's anonymity is inherent in the notion of a function call:
  - Functions like `mult` or `sqrt` are designed to serve **any caller**, implying that a return address *cannot be specified a priori*
  - Instead, a return command is interpreted as follows
    - **Redirect** the program's execution to the memory location holding the command *just following* the call command that invoked the current function

But where shall we save the return address?

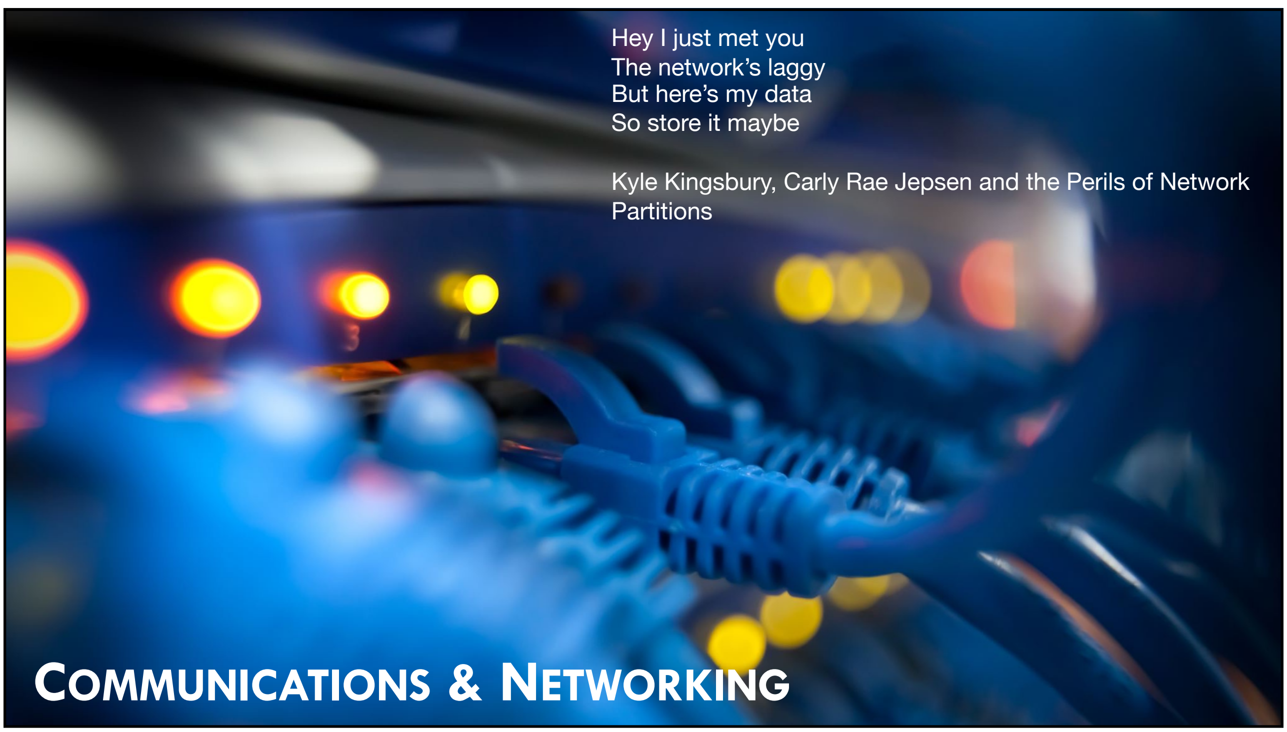
- Once again, the resourceful stack comes to the rescue
- The VM translator advances from one VM command to the next, generating assembly code as it goes along
  - When we encounter a `call foo` command in the VM code, we know exactly which command should be executed when foo terminates

The backstage on which this drama plays out is the stack

- Each call operation is implemented by saving the frame of the caller on the stack and jumping to execute the callee
- Each return operation is implemented by
  - Using the **most recently stored frame** for getting the *return address* within the caller's code and reinstating its memory segments
  - Copying the topmost stack value (the return value) onto the stack location associated with argument 0, and
  - Jumping to execute the caller's code from the return address onward
- All these operations must be realized by generated assembly code

COLORADO STATE UNIVERSITY | Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT | SOFTWARE & NETWORKING | L16.29

29



30

Example: Setting up connections to a server

- Programs open a **socket** to a server that's *listening* for connections
- To create a `Socket` you need to know the Internet host you want to connect to
- Servers don't know *who* will contact them
  - If it did, difficult to synchronize *when* this would happen

An analogy

- Server is like a person sitting by the phone
  - Doesn't know *who* will call and *when*
  - When the phone rings?
    - Talk to whoever is on the other line

Java provides a `ServerSocket` to enable writing servers

- `ServerSocket` runs on the server
  - **Listens** for *incoming* network connections on a particular **port** on the host that it runs on
- When a client socket on a remote host attempts to connect to that server port
  1. Server **wakes** up
  2. *Negotiates* a **connection** between the client and server
  3. **Opens** a regular `Socket` between the two hosts

Some more about the two types of sockets

- `ServerSockets` **wait** for connections
- Client `Sockets` **initiate** connections
- Once the `ServerSocket` has set up the connection?
  - **Data <u>always</u> travels over the regular `Socket`**

Using the `ServerSocket`

- Created on a particular **port** using the `ServerSocket(port)` constructor
- Listen for communications on that port using `accept()`
  - **Blocks** *until* a client attempts to make connection
  - Returns a `Socket` object that **connects** the client to the server
- Use the `Socket`'s `getInputStream()` and `getOutputStream()` to communicate

Creating the `ServerSocket`

- ```
  ServerSocket serverSocket =
          new ServerSocket(5000);
  ```
  - Tries to create a server socket on port 5000
- ```
  ServerSocket serverSocket =
         new ServerSocket(5000, 100);
  ```
  - Can hold up to 100 incoming connections
- ```
  ServerSocket serverSocket =
     new ServerSocket(5000, 100,
       InetAddress.getHostByName
             ("address2.cs.colostate.edu"));
  ```
  - On a **multi-homed** host, specify the network-address over which connections should be accepted
  ```

Accepting network connections

```
ServerSocket serverSocket =
     new ServerSocket(portNum);
while(true) {
   Socket socket = serverSocket.accept();
   ...
}
```


Closing the client and server sockets

- Closing a `ServerSocket` **frees** a port on the host that it runs on
- Closing a `Socket` **breaks** the connection between the local and remote hosts

We exchange byte streams over the socket

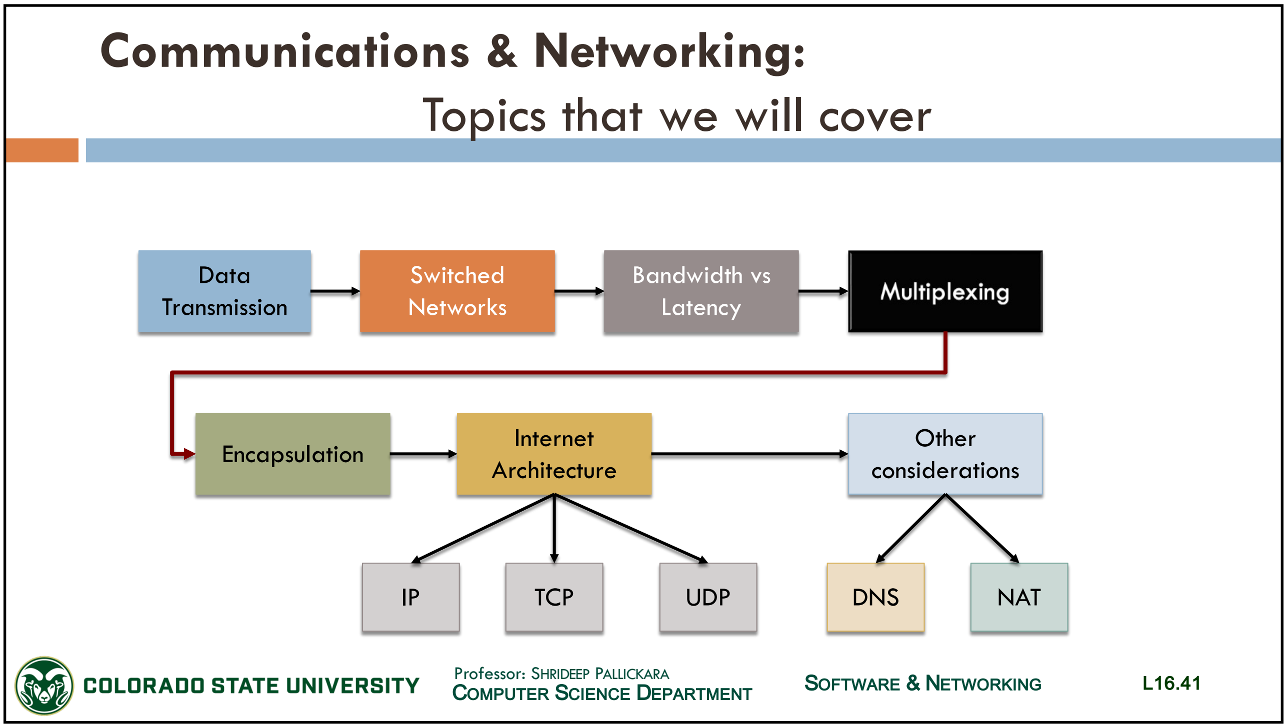
- The `java.io` package contains the `DataInputStream` and `DataOutputStream` that lets you do this elegantly

- ```
  DataInputStream din =
    new DataInputStream(socket.getInputStream());
  ```

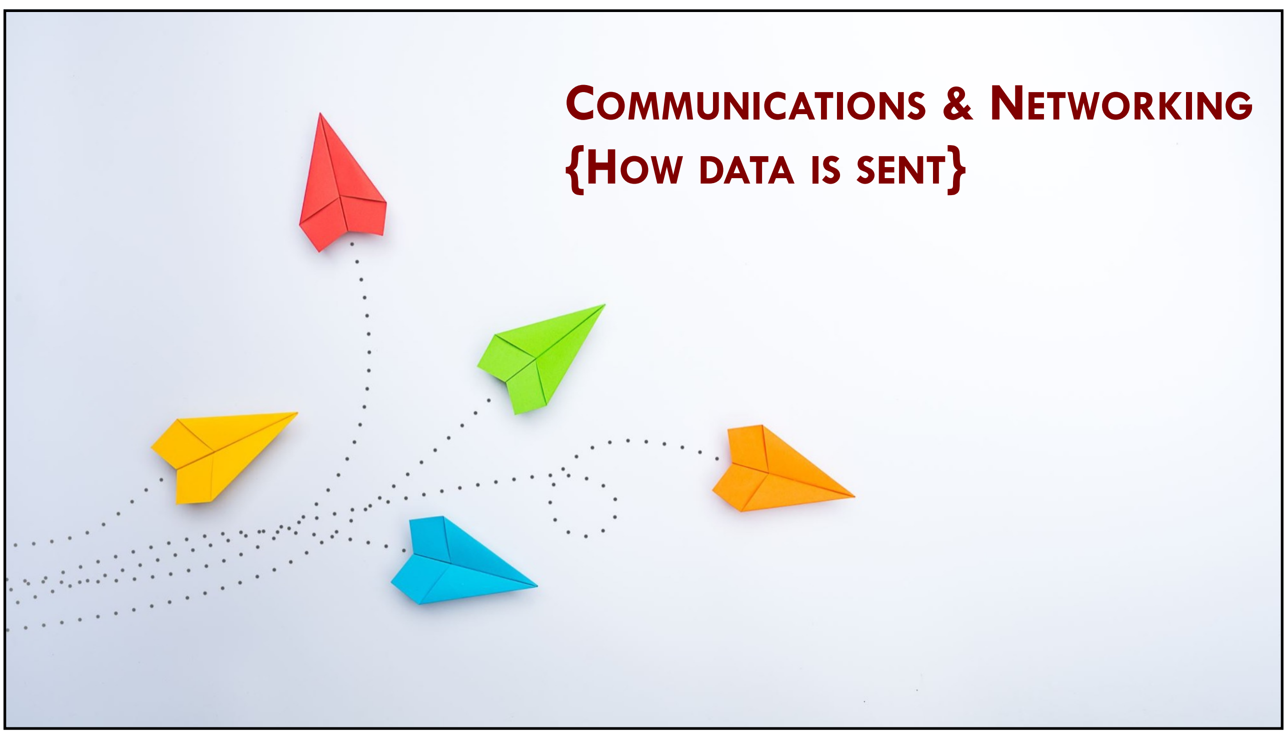
- ```
  DataOutputStream dout =
  new DataOutputStream(socket.getOutputStream());
  ```


39

40


Communications & Networking:
Topics that we will cover
Data Transmission
Switched Networks
Bandwidth vs Latency
Multiplexing
Encapsulation
Internet Architecture
Other considerations
IP
TCP
UDP
DNS
NAT
COLORADO STATE UNIVERSITY
Professor: SHRIDEEP PALLICKARA
COMPUTER SCIENCE DEPARTMENT
SOFTWARE & NETWORKING
L16.41

41


COMMUNICATIONS & NETWORKING
{HOW DATA IS SENT}

42

How is the data sent?

- Are we sending **1**'s and **0**'s?
- Whatever the physical medium, we use **signals**
  - Electromagnetic waves traveling at the speed of light
    - Speed of light is different in different mediums

Components of encoding binary data in a signal

- Modulation
- Duplexity

Encoding binary data: Modulation

- Objective is to send a ***pair*** of **distinguishable** signals

- Vary frequency, amplitude, or phase of the signal to transmit information
  - E.g. vary the power (amplitude) of signal
  - $x(t) = A \sin(2\pi ft + \theta)$

Encoding binary data: Duplexity

- *How many* bit streams can be encoded on a link at a time?
  - If it is one: nodes must share access to link
- Can data flow in both **directions** at the same time?
  - Yes → full-duplex
  - No → half-duplex

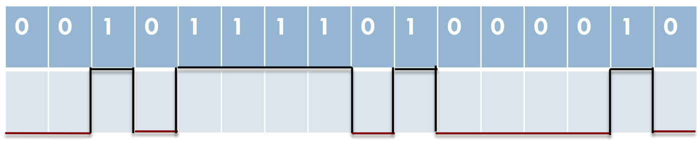
For our purposes, let's ignore details of modulation

- Assume we are working with two signals
  - High and low
- In practice:
  - Different voltages on a copper-based link
  - Different power-levels on an optical link

Let's do the obvious thing

- Map **1** to a high signal
- Map **0** to a low signal

Non-return to zero (NRZ)



Problems with NRZ because of consecutive 1's and 0's: **BASELINE WANDER**

- Receiver keeps **average** of the signal seen so far
- Average is used to *distinguish* between low and high
- Lots of consecutive **1**/**0**'s will make it difficult to detect a significant change

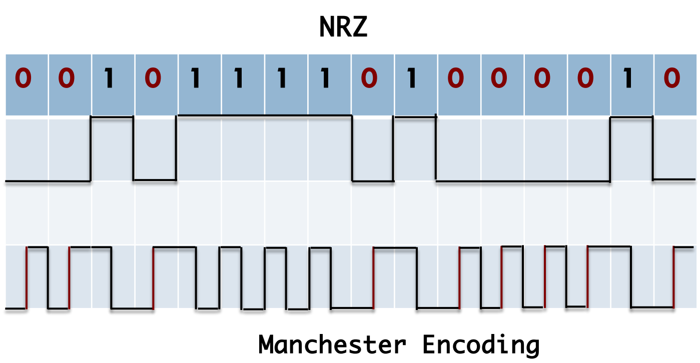
Problems with NRZ because of consecutive 1's and 0's: **CLOCK RECOVERY**

- Every clock cycle, sender transmits and the receiver receives
- Sender and receiver's clocks must be perfectly **synchronized**
  - Otherwise, it is not possible to decode the signal

Manchester encoding

- **0** is a low-to-high transition
- **1** is a high-to-low transition

Manchester encoding and NRZ



Some more about Manchester encoding

- Doubles the rate at which signal **transitions** are made on the link
  - Receiver has ½ the time to *detect* each pulse
- Rate of signal changes: baud rate
- Bit rate is ½ the baud rate
  - Encoding is considered 50% efficient

The contents of this slide-set are based on the following references

- Noam Nisan and Shimon Schocken. *The Elements of Computing Systems: Building a Modern Computer from First Principles.* 2nd Edition. ISBN-10/ ISBN-13: 0262539802 / 978-0262539807. MIT Press. [Part II, Chapter 7-8]