Network Communication and Remote Procedure Calls

COS 418: *Distributed Systems*
Lecture 2

Kyle Jamieson

Context and today's outline

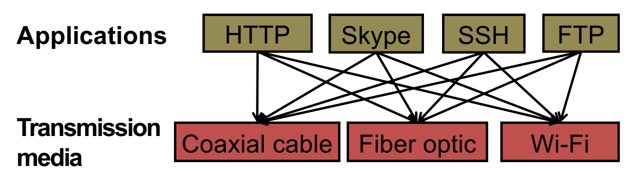
- A ***distributed system*** is many cooperating computers that appear to users as a single service
- **Today—** *How can processes on different cooperating computers* ***exchange information?***

1. **Network Sockets**
2. Remote Procedure Call
3. Threads

The problem of communication

- Process on **Host A** wants to talk to process on **Host B**
  - A and B must agree on the **meaning** of the bits being sent and received **at many different levels**, including:
    - *How many volts is a 0 bit, a 1 bit?*
    - *How does receiver know which is the last bit?*
    - *How many bits long is a number?*

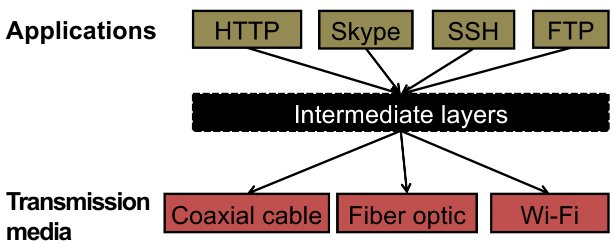
The problem of communication



- **Re-implement every application** for every new underlying transmission medium?
- **Change every application** on any change to an underlying transmission medium?

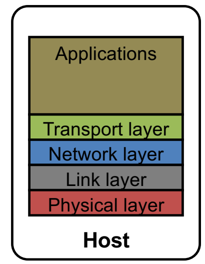
- **No!** But how does the Internet design avoid this?

Solution: Layering



- Intermediate ***layers*** provide a set of abstractions for applications and media
- New applications or media need only implement for intermediate layer's interface

Layering in the Internet

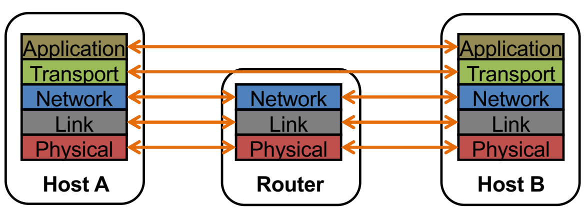


- **Transport:** Provide end-to-end communication between processes on different hosts
- **Network:** Deliver packets to destinations on other (heterogeneous) networks
- **Link:** Enables end hosts to exchange atomic messages with each other
- **Physical:** Moves bits between two hosts connected by a physical link

6

Logical communication between layers

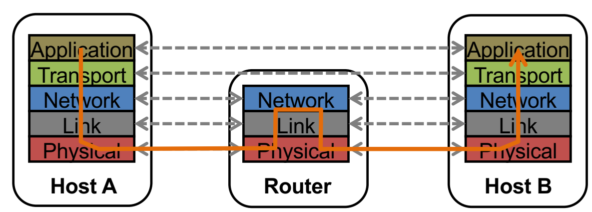
- *How to **forge agreement** on the **meaning** of the bits exchanged between two hosts?*
- ***Protocol:*** Rules that governs the format, contents, and meaning of messages
  – Each layer on a host interacts with its **peer** host's corresponding layer via the ***protocol interface***



7

Physical communication

- Communication goes down to the **physical network**
- Then from **network** peer to peer
- Then up to the **relevant application**

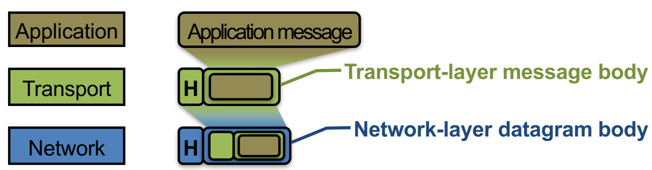


8

Communication between peers

- *How do peer protocols coordinate with each other?*

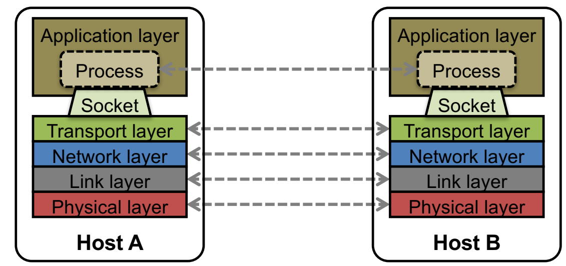
- Layer attaches its own ***header*** (**H**) to communicate with peer
  - Higher layers' headers, data ***encapsulated*** inside message
    - Lower layers don't generally inspect higher layers' headers



Network socket-based communication

- ***Socket:*** The interface the OS provides to the network
  - Provides inter-process **explicit message exchange**

- Can build distributed systems atop sockets: send(), recv()
  - *e.g.*: **`put(key,value)`** → message



Network sockets: Summary

- **Principle of transparency:** **Hide** that resource is physically distributed across multiple computers
  - Access resource same way as locally
  - Users can't tell where resource is physically located

> Network sockets provide apps with **point-to-point communication** between processes

- **`put(key,value)`** → message with sockets?


```
// Create a socket for the client
if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("Socket creation");
  exit(2);
}

// Set server address and port
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = inet_addr(argv[1]);
servaddr.sin_port = htons(SERV_PORT); // to big-endian

// Es
if (connect(sockfd, (struct sockaddr *) &servaddr,
            sizeof(servaddr)) < 0) {
  perror("Connect to server");
  exit(3);
}

// Transmit the data over the TCP connection
send(sockfd, buf, strlen(buf), 0);
```

Sockets don't provide transparency

Today's outline

1. Network Sockets
2. **Remote Procedure Call**
3. Threads

Why RPC?

- The typical programmer is trained to write single-threaded code that runs in **one place**
- **Goal:** Easy-to-program network communication that makes client-server communication **transparent**
  - Retains the "feel" of writing centralized code
    - Programmer needn't think about the network
- COS 418 programming assignments use RPC

What's the goal of RPC?

- Within a single program, running in a single process, recall the well-known notion of a **procedure call**:
  - ***Caller*** pushes arguments onto stack,
    - jumps to address of ***callee*** function
  - ***Callee*** reads arguments from stack,
    - executes, puts return value in register,
    - returns to next instruction in caller

RPC's Goal: To make communication appear like a **local** procedure call: **transparency** for procedure calls

RPC issues

1. **Heterogeneity**
   - Client needs to **rendezvous** with the server
   - Server must **dispatch** to the required function
     - What if server is **different** type of machine?
2. **Failure**
   - What if messages get **dropped?**
   - What if client, server, or network **fails?**
3. **Performance**
   - Procedure call takes ≈ 10 cycles ≈ 3 ns
   - RPC in a data center takes ≈ 10 μs ($10^3\times$ slower)
     - In the wide area, typically $10^6\times$ slower

Problem: Differences in data representation

- Not an issue for **local** procedure call

- For a remote procedure call, a **remote machine may:**
  - Represent data types using **different sizes**
  - Use a **different byte ordering** (*endianness*)
  - Represent floating point numbers **differently**
  - Have **different data alignment** requirements
    - *e.g.,* 4-byte type begins only on 4-byte memory boundary

Problem: Differences in programming support

- Language support **varies:**

  - Many programming languages have **no inbuilt concept** of remote procedure calls
    - *e.g.,* C, C++, earlier Java: won't generate stubs

  - Some languages have **support that enables RPC**
    - *e.g.,* Python, Haskell, **Go**

Solution: Interface Description Language

- Mechanism to pass procedure parameters and return values in a **machine-independent way**

- Programmer may write an ***interface description*** in the IDL
  - Defines API for procedure calls: names, parameter/return types

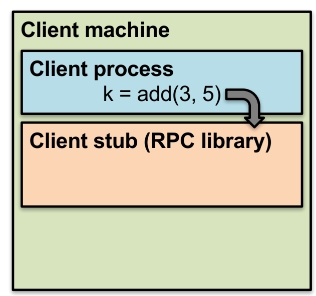
- Then runs an ***IDL compiler*** which generates:
  - Code to ***marshal*** (convert) native data types into machine-independent byte streams
    - And vice-versa, called ***unmarshaling***

  - **Client stub:** Forwards local procedure call as a request to server

  - **Server stub:** Dispatches RPC to its implementation

A day in the life of an RPC

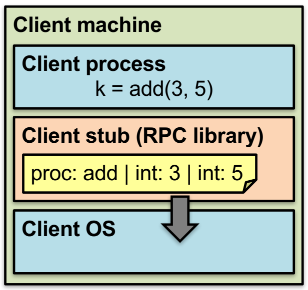
1. **Client calls stub function (pushes params onto stack)**



A day in the life of an RPC

1. Client calls stub function (pushes params onto stack)

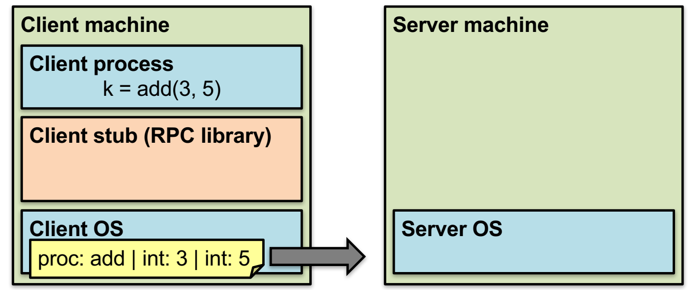
2. **Stub marshals parameters to a network message**



A day in the life of an RPC

2. Stub marshals parameters to a network message

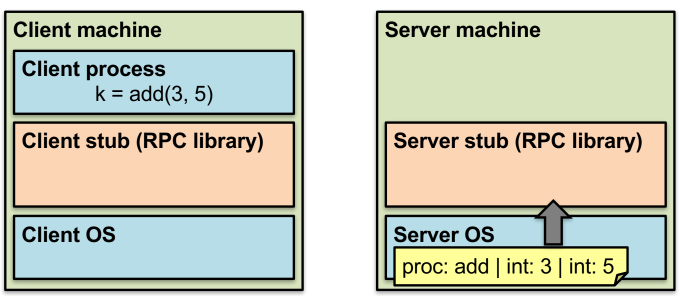
3. **OS sends a network message to the server**



A day in the life of an RPC

3. OS sends a network message to the server

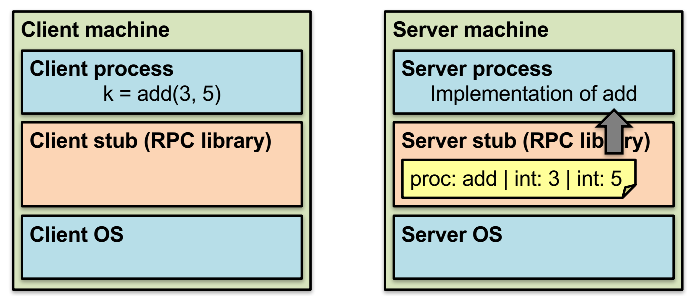
4. **Server OS receives message, sends it up to stub**



A day in the life of an RPC

4. Server OS receives message, sends it up to stub

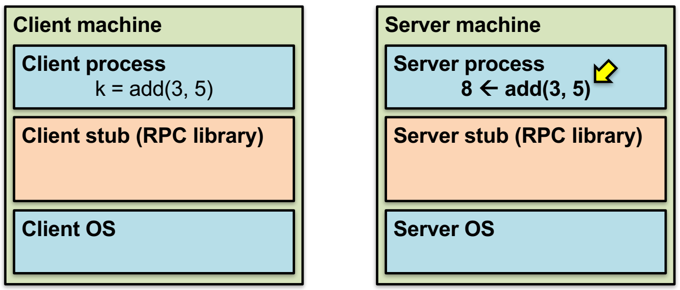
5. **Server stub unmarshals params, calls server function**



A day in the life of an RPC

5. Server stub unmarshals params, calls server function

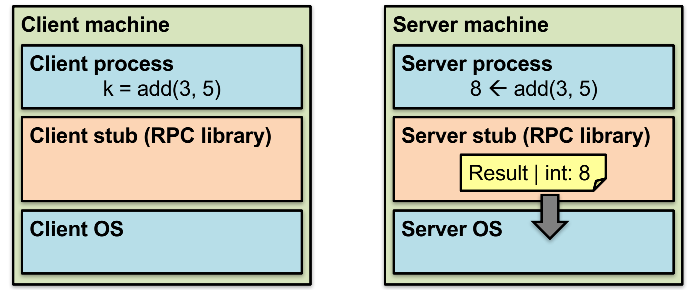
6. **Server function runs, returns a value**



A day in the life of an RPC

6. Server function runs, returns a value

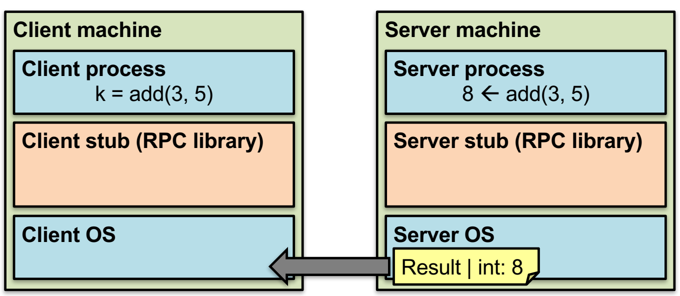
7. **Server stub marshals the return value, sends msg**



A day in the life of an RPC

7. Server stub marshals the return value, sends msg

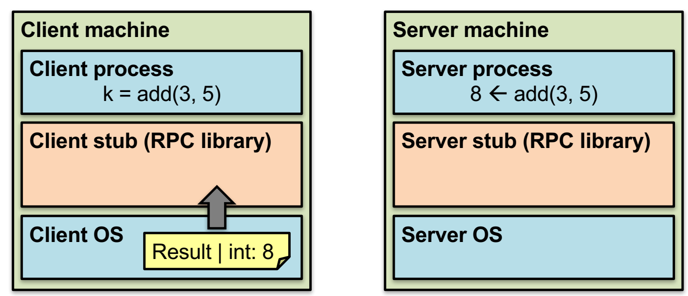
8. **Server OS sends the reply back across the network**



A day in the life of an RPC

8. Server OS sends the reply back across the network

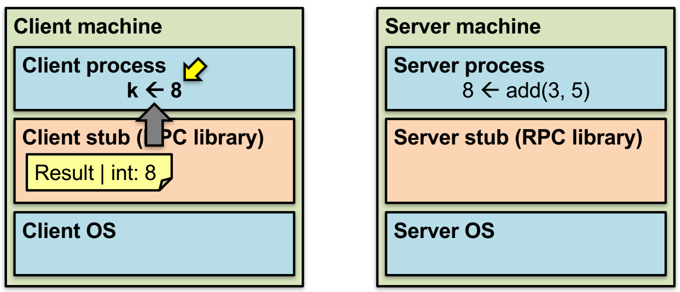
9. **Client OS receives the reply and passes up to stub**



A day in the life of an RPC

9. Client OS receives the reply and passes up to stub

10. Client stub unmarshals return value, returns to client



29

The server stub is really two parts

- ***Dispatcher***
  - Receives a client's RPC request
    - **Identifies** appropriate server-side method to invoke
- ***Skeleton***
  - **Unmarshals** parameters to server-native types
  - **Calls** the local server procedure
  - **Marshals** the response, sends it back to the dispatcher
- **All this is hidden from the programmer**
  - Dispatcher and skeleton may be integrated
    - Depends on implementation

30

Today's outline

1. Message-Oriented Communication
2. **Remote Procedure Call**
   - Rendezvous and coordination
   - **Failure**
   - Performance
3. Threads

31

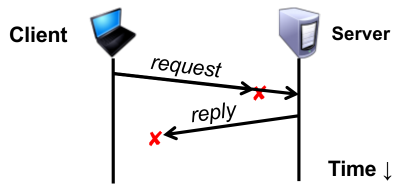
What could *possibly* go wrong?

1. Client may **crash and reboot**
2. Packets may be **dropped**
   - Some individual **packet loss** in the Internet
   - **Broken routing** results in many lost packets
3. Server may **crash and reboot**
4. Network or server might just be **very slow**

All these may look the same to the client…

32

Failures, from client's perspective



The cause of the failure is hidden from the client!

At-Least-Once scheme

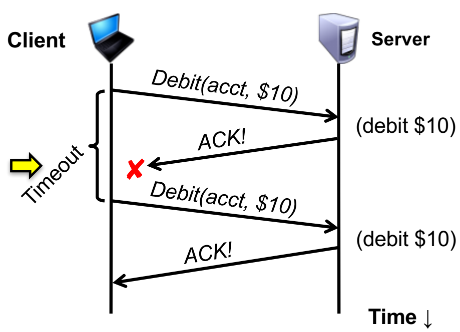
- **Simplest** scheme for handling failures

1. Client stub **waits for a response**, for a while
   – Response takes the form of an ***acknowledgement*** message from the server stub
2. If no response arrives after a fixed ***timeout*** time period, then client stub **re-sends the request**

- Repeat the above a few times
  – *Still no response?* Return an error to the application

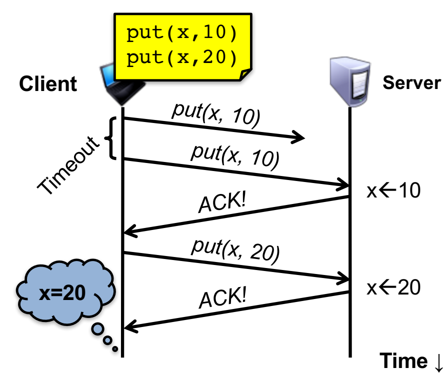
At-Least-Once and side effects

- Client sends a "debit $10 from bank account" RPC



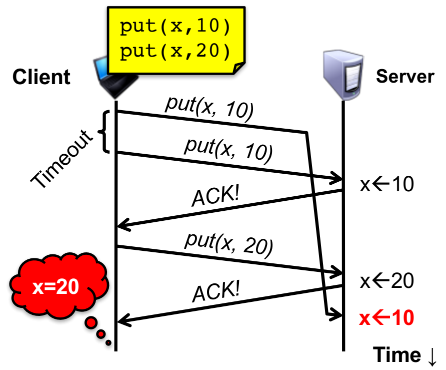
At-Least-Once and writes

- put(x, *value*), then get(x): expect answer to be *value*



At-Least-Once and writes

- Consider a client storing **key-value pairs** in a **database**
  - put(x, *value*), then get(x): expect answer to be *value*



So is At-Least-Once *ever* okay?

- **Yes:** If they are read-only operations with no side effects
  - *e.g.*, read a key's value in a database
- **Yes:** If the application has its own functionality to cope with duplication and reordering
  - You will need this in Assignments 3 onwards

At-Most-Once scheme

- **Idea:** server RPC code detects duplicate requests
  - Returns previous reply **instead of re-running handler**
- *How to detect a duplicate request?*
  - **Test:** Server sees same function, same arguments twice
    - **No!** Sometimes applications **legitimately** submit the same function with same augments, twice in a row

At-Most-Once scheme

- *How to detect a duplicate request?*
  - Client includes unique ***transaction ID*** (***xid***) with each one of its RPC requests
  - Client uses **same xid** for retransmitted requests

```
At-Most-Once Server
if seen[xid]:
        retval = old[xid]
else:
        retval = handler()
        old[xid] = retval
        seen[xid] = true
return retval
```

At Most Once: Providing unique XIDs

1. Combine a unique client ID (*e.g.*, IP address) with the current time of day
2. Combine unique client ID with a sequence number
   – Suppose the client crashes and restarts. *Can it reuse the same client ID?*
3. Big random number (probabilistic, not certain guarantee)

At-Most-Once: Discarding server state

- **Problem:** `seen` and `old` arrays will **grow without bound**
- **Observation:** By construction, when the client gets a response to a particular xid, it will **never re-send it**
- Client could **tell** server "I'm done with xid *x* – delete it"
  – Have to tell the server about **each and every** retired xid
    • Could **piggyback** on subsequent requests

Significant overhead if many RPCs are in flight, in parallel

At-Most-Once: Discarding server state

- **Problem:** `seen` and `old` arrays will **grow without bound**
- Suppose xid = ⟨unique client id, sequence no.⟩
  – *e.g.* ⟨42, 1000⟩, ⟨42, 1001⟩, ⟨42, 1002⟩
- Client includes "seen all replies ≤ *X*" with every RPC
  – Much like TCP sequence numbers, acks
- *How does the client **know** that the server received the information about retired RPCs?*
  – Each one of these is cumulative: later seen messages subsume earlier ones

At-Most-Once: Concurrent requests

- **Problem:** How to handle a duplicate request while the original is still executing?
  – Server doesn't know reply yet. Also, we don't want to run the procedure twice
- **Idea:** Add a `pending` flag per executing RPC
  – Server waits for the procedure to finish, or ignores

At Most Once: Server crash and restart

- **Problem:** Server may crash and restart
- *Does server need to write its tables to disk?*
- Yes! On **server crash and restart:**
  - If `old[]`, `seen[]` tables are only in memory:
    - Server will forget, **accept duplicate requests**

Go's net/rpc is at-most-once

- Opens a TCP connection and writes the request
  - TCP may retransmit but server's TCP receiver **will filter out duplicates internally,** with sequence numbers
  - No retry in Go RPC code (*i.e.* will **not** create a second TCP connection)
- However: Go RPC **returns an error** if it doesn't get a reply
  - Perhaps after a TCP timeout
  - Perhaps server didn't see request
  - Perhaps server processed request but server/net failed before reply came back

RPC and Assignments 1 and 2

- Go's RPC **isn't enough** for Assignments 1 and 2
  - It only applies to a single RPC call
  - If worker doesn't respond, master **re-sends** to another
    - Go RPC **can't detect** this kind of duplicate
  - **Breaks at-most-once** semantics
    - No problem in Assignments 1 and 2 (handles at application level)
- In Assignment 3 **you** will explicitly detect duplicates using something like what we've talked about

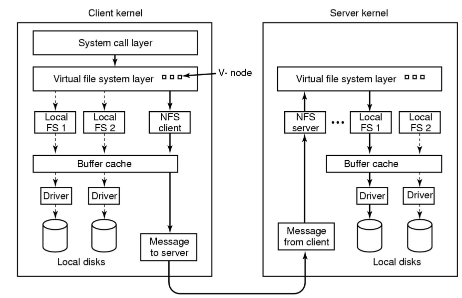
Exactly-once?

- Need retransmission of at least once scheme
- Plus the duplicate filtering of at most once scheme
  - To survive **client** crashes, client needs to record pending RPCs on disk
    - So it can replay them with the same unique identifier
- Plus story for making server reliable
  - Even if server fails, it needs to continue with full state
  - To survive **server** crashes, server should log to disk results of completed RPCs (to suppress duplicates)
- Similar to Two-Phase Commit (later)

Exactly-once for external actions?

- Imagine that the remote operation triggers an external physical thing
  – *e.g.,* dispense $100 from an ATM
- The ATM could crash immediately before or after dispensing and lose its state
  – Don't know which one happened
    - Can, however, make this window very small
- **So can't achieve exactly-once in general,** in the presence of external actions

Summary: RPC



- RPC is everywhere!
- **Necessary** issues surrounding machine heterogeneity
- **Subtle** issues around handling **failures**

Today's outline

1. Network Sockets
2. Remote Procedure Call
3. **Threads**

Threads

- One goal of this class is to give you experience and wisdom dealing with threads – they are tricky!
- **Go terminology:** threads = **goroutines**
- Thread = Program counter + set of registers: an execution context
  – Can be multiple threads in the same shared memory address space

Data races

- Challenge: Sharing data
  - Two threads write same memory location
  - One thread writes same memory location, other reads
- Called a ***race***
- x = 0 initially. **Thread 1:** x ← x+1; **Thread 2:** x ← x+1
  - Answer has to be 2, but if they run together can get 1
    - Both threads read x before either writes back
- To fix: wrap access to the same variable with a go ***mutex***

53

Waiting

- One thread wants to wait for the other thread to finish
- In Go, use **Channels** for communication between threads
- But beware ***deadlock:*** can be cycles in the waiting
  - Thread 1 waiting for thread 2 to do something
  - Thread 2 waiting for thread 1 to do something
  - Sounds silly but comes up if you are not careful!

54

Wednesday topic:
Network File Systems

Friday precept:
Concurrency In Go, MapReduce

55

APPENDIX

56

Review: Byte order

- x86-64 is a ***little endian*** architecture
  - **Least** significant byte of multi-byte entity at **lowest** memory address
    - "Little end goes first"

int 5 at address 0x1000:

0x1000:	**0000 0101**
0x1001:	0000 0000
0x1002:	0000 0000
0x1003:	0000 0000

- Some other systems use ***big endian***
  - **Most** significant byte of multi-byte entity at **lowest** memory address
    - "Big end goes first"

int 5 at address 0x1000:

0x1000:	0000 0000
0x1001:	0000 0000
0x1002:	0000 0000
0x1003:	**0000 0101**