UNIVERSITATEA TEHNICĂ A MOLDOVEI

Network Programming

Alexandru Burlacu

Autumn 2020

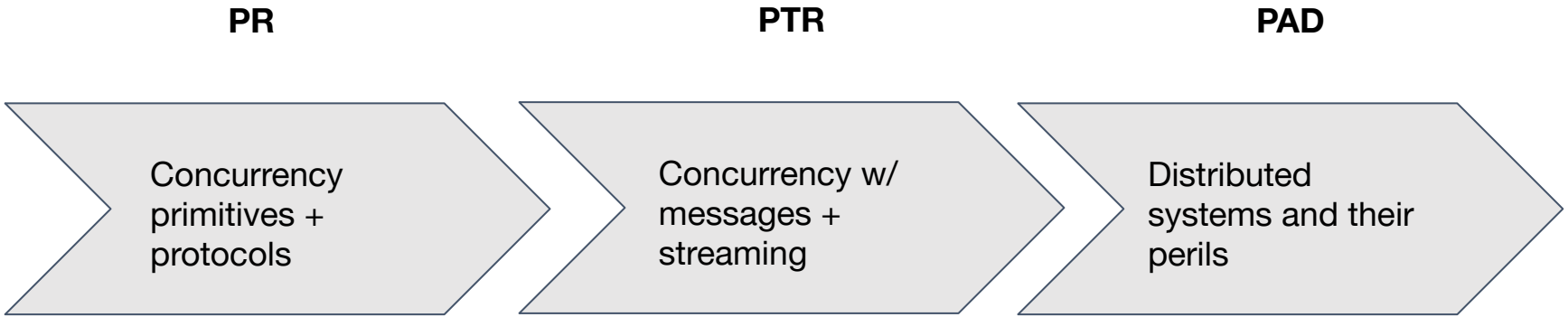
FACULTATEA
CALCULATOARE, INFORMATICĂ
ȘI MICROELECTRONICĂ

FCIM

Intro & Course Description

Alexandru Burlacu

Autumn 2020


PR
PTR
PAD
Concurrency primitives + protocols
Concurrency w/ messages + streaming
Distributed systems and their perils

- **Topics** - first Concurrency, and then Protocols, also a bit about networks and Git
- **Labs** - 2x, same as the main topics
- **Midterms** - two midterms, a lab (70%) + questions (3 Qs = 5 + 15 + 10)
- **Exam** - oral, 30 min preparation time, <16 min Q&A **Unless faculty decides otherwise**
- **Grading policy** - 10 is thresholded at 91 points, the rest are relative, following Gaussian dist.
- **Attendance** - Doesn’t matter. Just pass the exam and complete the labs on time

A. Git

B. Computer Networks

C. Communication

D. Concurrency

1. Definition of concurrency
2. Common problems
3. Synchronization mechanisms
4. Implementation variants
5. Concurrent Collections

E. Protocols

1. Definition of protocols
2. BSD Sockets API
3. TCP/UDP standards, and others
4. HTTP protocol
5. WebSockets and others
6. A bit about mail protocols (SMTP and POP3)
7. Error detecting and correcting codes

So, let's talk Git

Alexandru Burlacu

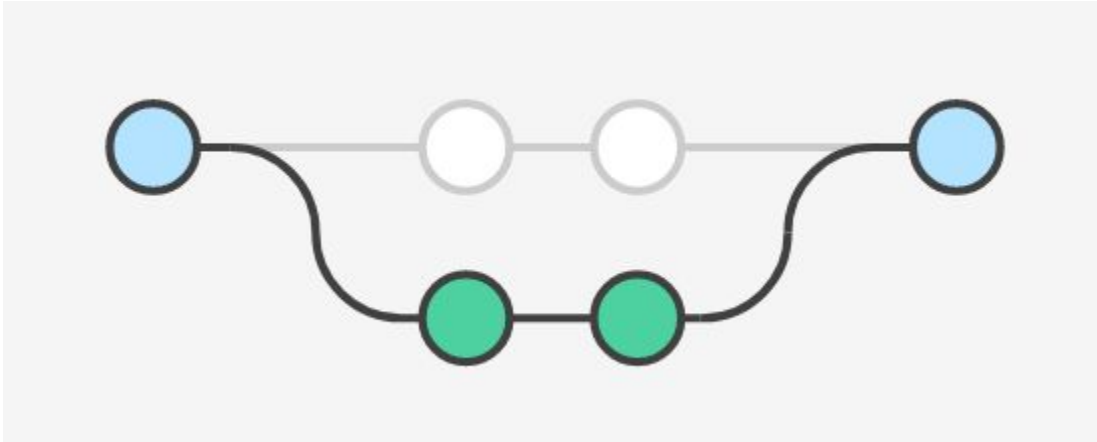
Autumn 2020

What is Git?

Alexandru Burlacu

Autumn 2020

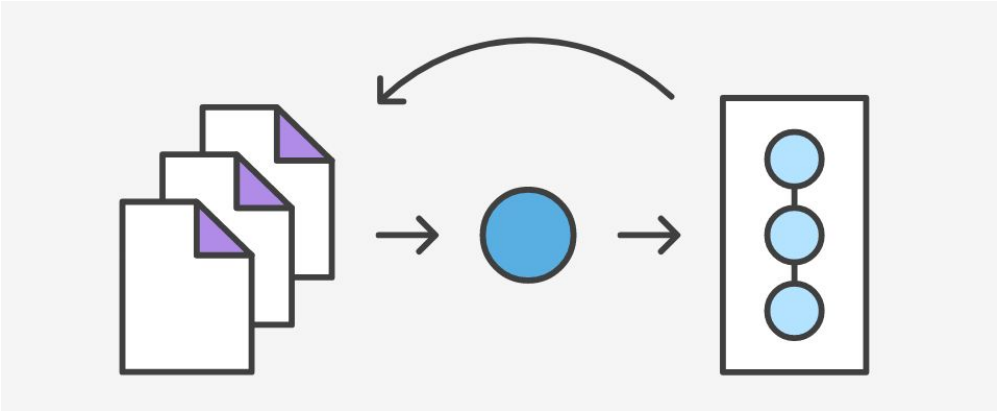
Quick Check 1



Merge vs Rebase?

*Img. Source: https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Quick Check 2



Go back to previous state?

*Img. Source: https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting

Quick Check 3, 4, 5

- How’s SVN different from Git?
- Git stash anyone?
- Git Flow? Forking workflow?

Must know on Git internals

Git is indeed an interesting VCS. It's author claims that he was heavily inspired by file systems rather than other VCS, that's why Git is so different.

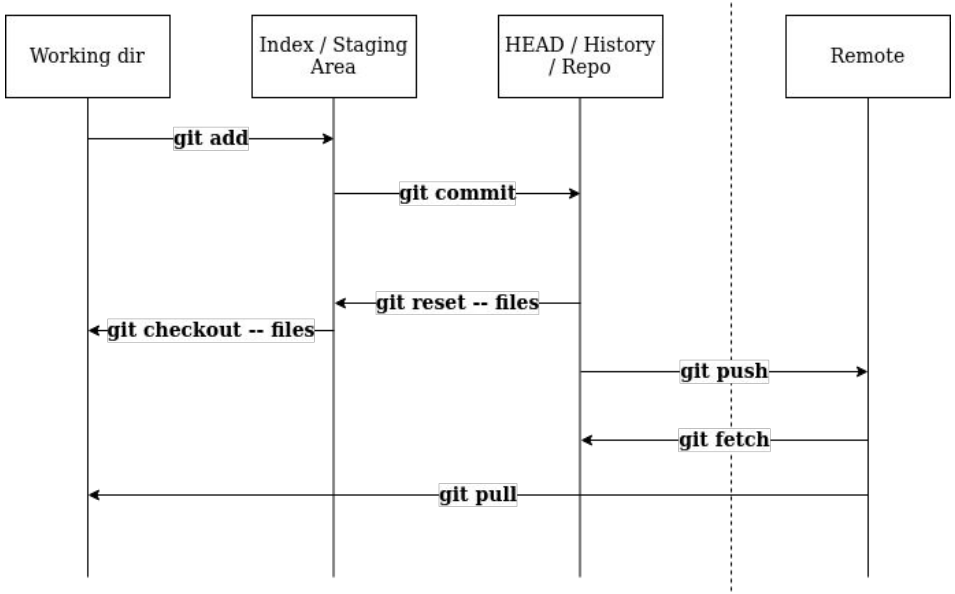
Git stores *snapshots* of the working directory. This allows fast checkout between commits and branches. But this consumes a lot of memory.

That's why sometimes when git does garbage collection, or is forced to (**`git gc`**) it will compress a number of snapshots into packfiles, that are basically binaries with deltas between objects.

Now, you really should check the Git book: https://git-scm.com/book/en/v2, especially 10th chapter.

Must know on Git internals #2

For more info with nice visualizations, check: marklodato.github.io/visual-git-guide/index-en.html



Some very cool things Git does

- **`(git cherry-pick)`** What if you solved a bug in a branch, and want to propagate the change to everyone else? Doing a full fledged merge could cause anything from a lot of downtime solving merge conflicts and messy commit history, to even losing credibility and respect at your workplace. Don't worry, **`git cherry-pick`** got you covered! Now you can pick a single commit and propagate your bugfix or whatever.

- **`(git bisect)`** So, you've done some commits and it turns out you accidentally introduced a bug. But where is it? And when did it happen? **`git bisect`** will help, doing binary search on a range of commits in a convenient way. Reduce your search space dramatically - today!

- **`(git daemon)`** You've heard about Git being decentralized, but never understood what's that? You're working with a buddy on a feature so pre-alpha that pushing it to central repo seems unnecessary? Try now **`git daemon`** - the static site of git servers!

Reading list

- http://tom.preston-werner.com/2009/05/19/the-git-parable.html
- https://nvie.com/posts/a-successful-git-branching-model/
- https://www.endoflineblog.com/gitflow-considered-harmful
- https://www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow
- https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow
- [For fanatics and no-lifers] Merkle Tree. https://www.codeproject.com/Articles/1176140/Understanding-Merkle-Trees-Why-use-them-who-uses-t

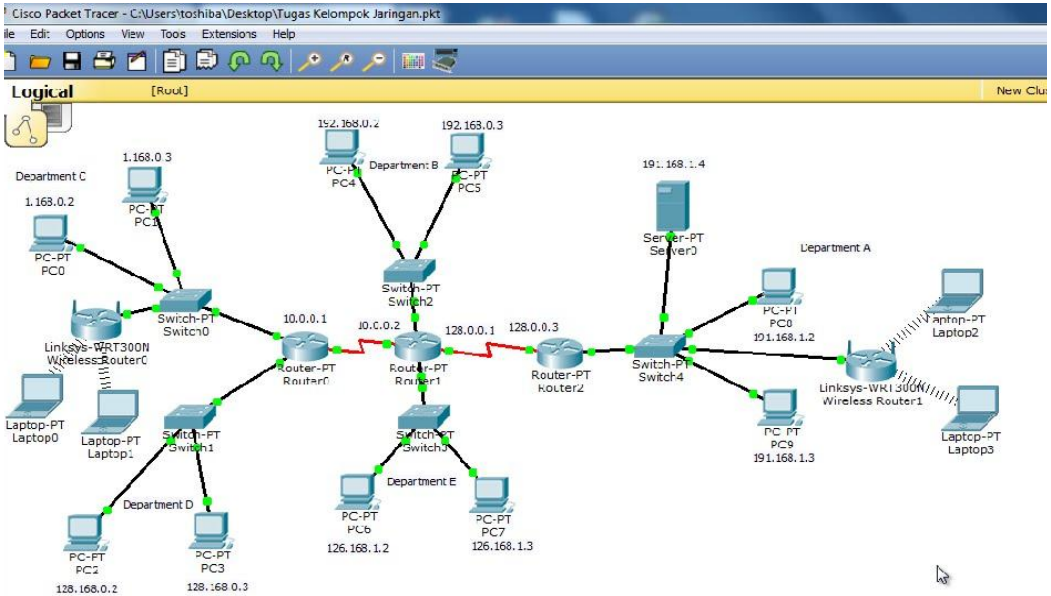
Computer networks

Alexandru Burlacu

Autumn 2020

How it all began...

"There was an idea
To bring together
A group of remarkable machines
So when we need them
They could compute the stuff
That we never could
… and also survive a couple of nukes" **- Nick Fury**



Ok, now seriously

Among the first computer networks where established by U.S. military in late 50s, notoriously SAGE aka Semi-automatic Ground Environment.
Another important milestone was SABRE, a commercial airline booking system, made in 1960, which was two mainframes connected.

The network that you probably all know about was 1669's ARPANET, the precursor of The Internet, which was connecting a number of research institutions and military facilities, financed by DoD, and having the main objective to survive a nuclear war.

What made it so innovative was the packet switching technology, the possibility to access the data and applications on any of the machines within the network from any other machine, also it had the first email.

Packet Switching. Why? How?

What makes modern computer networks so efficient is the packet switching technology.

It is used to ensure maximum chance of delivery, with latency and speed as good as possible.

Basically, that means that information is splitted into packets of predefined size and sent over network. In case there's some congestion on the way, the packets are switched through other routes, thankfully the networks have some redundancy.

Packets are sent through the fastest route, but that doesn't always mean it is the shortest route. Think GPS with information about traffic jams.

Packet Switching. Why? How?

A packet is sort of a wrapper around the chunk of data that the system is trying to send.
It has a header and a trailer. Most meta information is in the header, while the trailer is used to make clear how where the packet is ending.

For example, the header of a packet contains the sender and receiver addresses, the number of packets to be sent, the id of a specific packet, some control information.

For TCP, if a packet is lost or corrupted, it is resent, but more on that later.

A quick refresher: Topology

First there was the **the mainframe** with multiprogramming capabilities, and many client machines in a **star** topology.

Then, came **the bus** and later **the ring** topologies. But it was never enough.

And later, the router was invented and compute machines became affordable. At that point networks could be built just like they are now, more or less. So the **mesh** topology emerged.

Relatively recently peer-to-peer (P2P) networks emerged.

And there are also ad-hoc networks, like P2P but ad-hoc.

A quick refresher 2: Computer networks challenges

More computers, that are also interlinked, plus having devices to link them and provide different services, all this makes the system more fragile/prone to issues and vulnerabilities.

1. High maintenance cost
2. Increased budget for devices and software
3. Necessity for specialized staff to keep the network running normally
4. Security concerns (higher attack surface for hackers, spread of malware)
5. Authorization and access control
6. Basically your machines could burn, so you need backups
7. Et cetera

A quick refresher 3: Computer networks requirements

For a computer network to be useful, it should be analyzed, and perform well in certain dimensions.

1. Functionality - the more, the merrier
2. Efficiency (latency, response time, transfer speed, bandwidth)
3. Resilience and fault tolerance
4. Security
5. Availability
6. Quality of Service
7. Integrity and coherence
8. Monitoring and traffic control
9. Scalability
10. Adaptability

Computer networks standards

To be extensible and scalable… and easy and cheap to maintain, … and interoperable, the components of a network are desirable to be obey **Open Standards**. Both **Open** and **Standards** are important words here. Let's dissect.

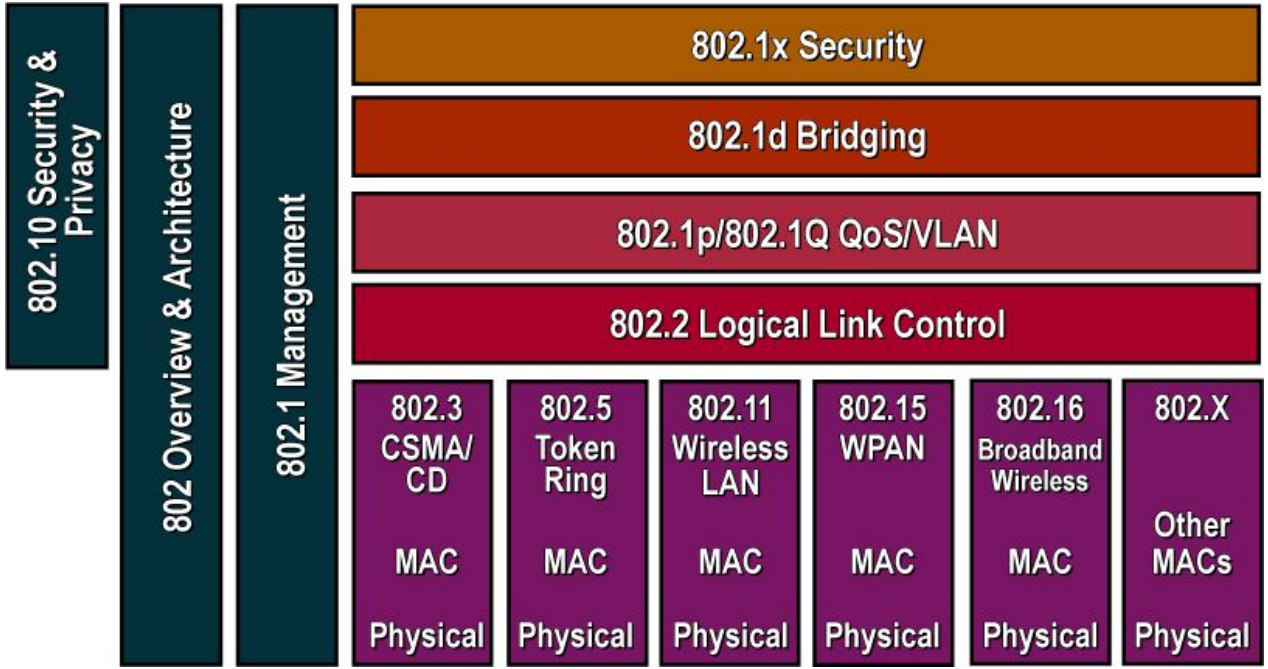
Open - open means that the specification is freely available for everyone, and doesn't have licencing costs. Specifications are not standards.

Standards - standards are complete, formal specifications that are crystalized and through a consensus decided to be canonical. Standards are maintained and developed by standardization institutions, like ISO, ANSI, IEEE, NIST, ECMA, or others.

The IEEE 802 project

Since February 1980, IEEE started working on 802.x suite of standards that cover all network components.

Just wikipedia it.



*Source: www.hill2dot0.com/wiki/index.php?title=IEEE_Project_802

Internet development organizations

ISOC (INTERNET Society) - Internet evolution and development. **IAB (Internet Architecture Board)** - technical control. **IETF (Internet Engineering Task Force) and IRTF (Internet Research Task Force)** - solve actual problems and do long term research, correspondingly.

Communication

Alexandru Burlacu

Autumn 2020

Communication objectives

We have 3 primary reasons to use communication:

- Data transfer, as in HPC (Accelerate computing), or microservices (Modularization)
- Data sharing, as in Google Drive, or Amazon S3
- Synchronization, or control, as in telling the other party what to do

*Source: Communication Over Networking - PR 2017, UTM, conf.univ. PhD Ciorbă Dumitru

Communication paradigms

Shared memory
- Efficient
- Error prone
- Really good when processes are local to a machine

Message passing
- Shines when running on multiple machines
- Abstracts away the location of the process
- Less error prone

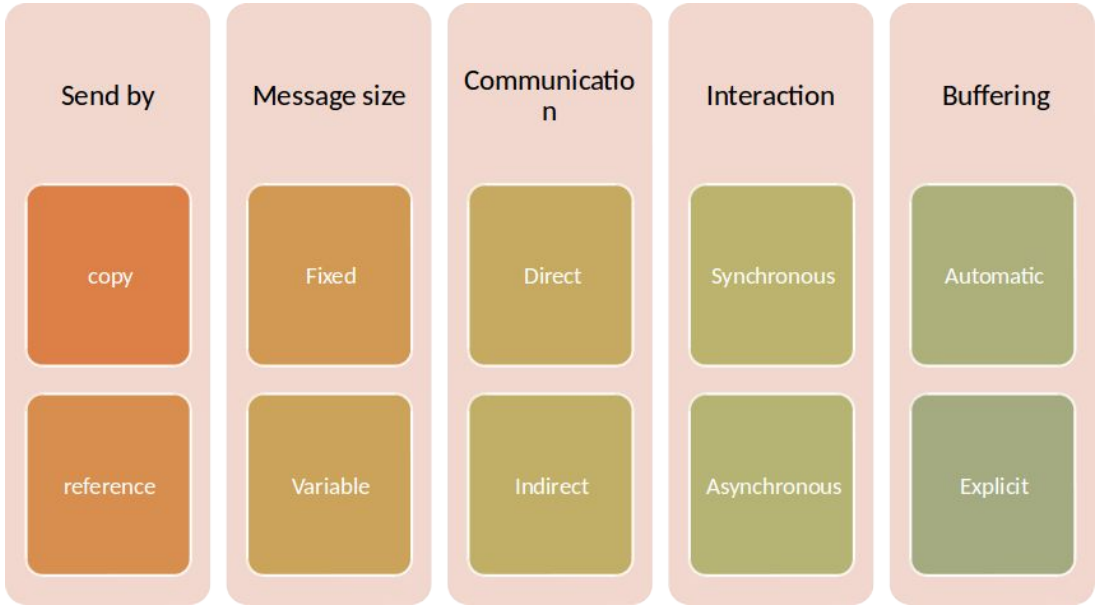
Remote invocation
- Just as Message Passing, but now you abstract away even the fact that you have another process. It feels like calling a function

Communication paradigms: Direct and Indirect

Direct Communication - processes communicate directly, obviously, it is efficient but not scalable. And not that flexible too. RPC and Queues and concurrent objects communicate like this.

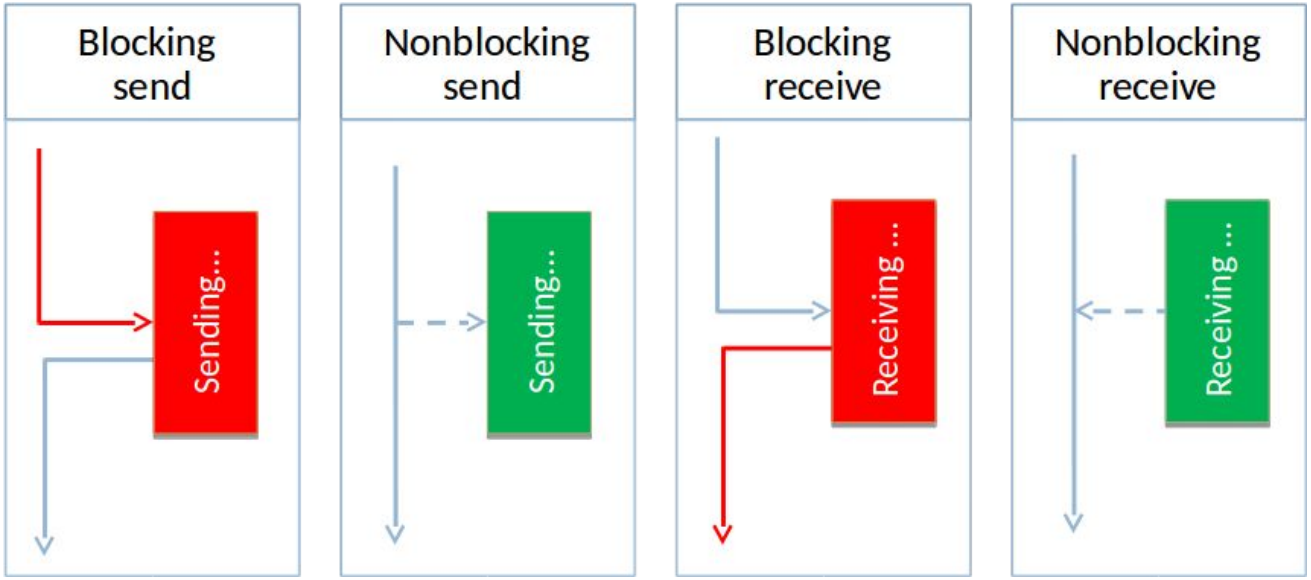
Indirect Communication - processes communicate through an intermediary, like a message queue or a mailbox. It is less efficient but lets you build systems which can handle variable number of agents communicating, efficiently.

Data Transfer. Dimensions.



*Source: Communication Over Networking - PR 2017, UTM, conf.univ. PhD Ciorbă Dumitru

Data Transfer. Interactions.



*Source: Communication Over Networking - PR 2017, UTM, conf.univ. PhD Ciorbă Dumitru

Data Transfer. Buffering.

Capacity	Buffering procedure

• Zero
  • ~~Asynchronous~~
  • Synchronous
• Bounded
• Unbounded
	• Automatic
	• Explicit

*Source: Communication Over Networking - PR 2017, UTM, conf.univ. PhD Ciorbă Dumitru

Communication granularity

Services - REST or SOAP or RPCs (Java RMI, .NET Remoting, CORBA, gRPC, Thrift) and all that network protocols of high level

Messages - Message Queues, WebSockets, Events and pipes/streams

Bytes - Files and pipes, Sockets (BSD ones)

Pipes

Pipes are an inter-process communication method (IPC) which shares some concepts with both shared memory and message passing paradigms.

On implementation level pipes are special temporary files. Pipes obey the so-called **stream processing** paradigm, which says that there's only one concrete producer and consumer and that the communication is unidirectional.

Pipes are usually OS primitives, and sometimes one can be bidirectional and named.

Pipes

The simplest example of a pipe, in UNIX systems, would be something like this expression **`ps -aux | grep -E "<some regexp pattern>"`** Do you know what it does?

The `|` operator creates a temporary store in memory in which the output content from the first command is written into, and simultaneously used as input for the second command. This is an **anonymous pipe**. Pipes are good as IPC within a system. They are usually faster than sockets.

There are also named pipes. Here's how these are created in Linux **`mkfifo <pipe_name>`**. Having a named pipe is useful because it can be used as a channel between processes and it won't go away when the processes are done. By alternating read and write sessions, these can ensure bidirectional communication, so are kind of Half-duplex (HDX).

See also: https://stackoverflow.com/questions/18568089/whats-the-difference-between-pipes-and-sockets
and http://hassansin.github.io/fun-with-unix-named-pipes
and https://stackoverflow.com/questions/1235958/ipc-performance-named-pipe-vs-socket#1238819

Stating the obvious

Network programming in most cases is about a **client** application/service/whatever talking to a **server**. This is even an architectural pattern, called client-server architecture.

Client
- Wants something from the server
- Establishes the connection

Server
- Fulfils clients desires
- Is oblivious of his clients up until the connection is established
- Can’t normally ask anything from the client

And because usually there are more clients than servers...

Concurrency

Alexandru Burlacu

Autumn 2020

Concurrency is NOT Parallelism!

Alexandru Burlacu

Autumn 2020

Concurrency is NOT Parallelism! Definition

“[...] concurrency refers to the decomposability property of a program, algorithm, or problem into order-independent or partially-ordered components or units.” - Leslie Lamport*
“A concurrent program whose processes „are executed in an abstract parallelism, that is, not necessarily on distinct processors” - Horia Georgescu**

Concurrency is about **ability** to perform certain program fragments independently, but not necessarily these running at the same time. I.o.w. one can run some code concurrently, yet have just one thread of execution running at the same time. Think JS, or Python/Ruby threads with GIL.

Parallelism is about actually running fragments of programs **simultaneously**.
Distribution is… pain… and unreliable components… and we’ll talk about it during PAD.

*"Time, Clocks, and the Ordering of Events in a Distributed System" - Leslie Lamport.
**”Programarea concurenta. Teorie si aplicatii” - Georgescu, Horia, 1996

What is a Process? Definition

Long story short, process is an instance of a computer program, **the execution of the computer program**.

Processes are said to *own* an image of the executable machine code, some memory, a call stack and a heap.

Being managed by the OS (unless otherwise specified), processes have also OS-specific attributes, like permissions, file descriptors, et cetera.

Their state is commonly called *context* - registers content and physical memory addressing.

What is a Thread? Definition

A thread is *“[...] the smallest sequence of programmed instructions that can be managed independently [...]”* - Leslie Lamport* (again)

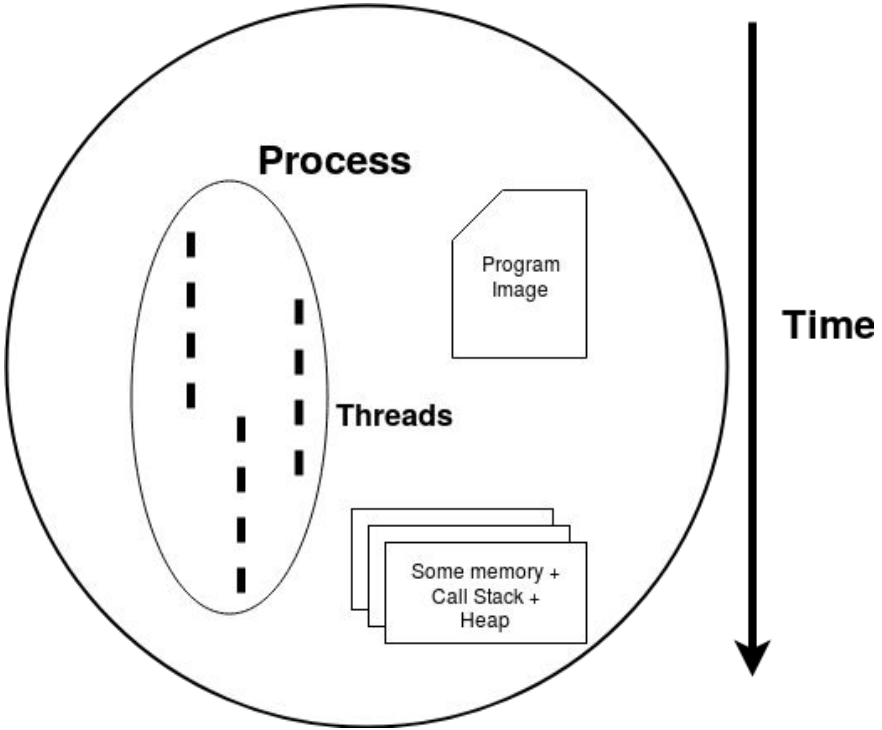
Remember! Threads are owned by processes. Multiple threads can exist within one process, executing concurrently and sharing resources (memory, variables, address space), while different processes do not share these resources.

Also remember - even classic threads are considerably cheaper to create and operate than processes.

*"How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs" - Leslie Lamport.

Threads and Processes

In Linux the computer can potentially can hold up to 32k+ processes, or sometimes up to ~4.2mln, depending on the OS (`cat /proc/sys/kernel/pid_max`) and
the number of threads per process is (`cat /proc/sys/kernel/threads-max`) around 125k, but can be overwritten.



Threads and Processes. Costs.

Context switch, that is moving from one process/thread to another process/thread is a costly operation, just as spawning a new one is costly too. Processes need a lot of resources, but threads do need some memory too, for the execution stack. On a modern processor, spawning an OS thread could be as costly as at least 8MB of memory*, although that value can be overwritten.

An OS thread needs between 1 to 5 microseconds, on average. A process on the other hand, depending on the number of file handles it owns, and memory, and OS, can take between 3-5 to sometimes as much as 35 microseconds.

Note that user-space threads, like goroutines, Ruby fibers or Python gevent threads, switch almost an order of magnitude faster, sometimes as fast as a couple of hundreds of nanoseconds.

*Source: https://unix.stackexchange.com/questions/127602/default-stack-size-for-pthreads

Threads: Preemptive vs Cooperative

Depending on the nature of the threads, the scheduling can be either preemptive or cooperative. Both have drawbacks. Usually, system threads are preemptively scheduled, via context switches while more lightweight ones (think green threads in Python, fibers in Ruby, or even Node's event loop) are managed in a cooperative way.

Cooperative - the thread should **yield** control of execution
    Drawback - high risk of starvation, deadlock possible

Preemptive - there's a scheduler that might interrupt one thread and let another one running
    Drawback - high risk of race conditions, convoy locks are possible

Threads: Standalone or Pooled?

Say you have to call a bunch of URLs, concurrently, and as you call them, more are coming (just like your lab)?
Recall that number of threads a system can handle is bounded by how many resources you have.
Besides there's always a cost (time) for launching a new thread.

What could you do?

Threads: Standalone or Pooled?

Recycle!! Recycle!! Recycle!!

Recycle!!

Recycle!! Recycle!! Recycle!!

Threads: Standalone or Pooled?

Recall the object pool pattern. Where you have a pool or resources/objects that represent some resources, and are shared. Once you are done using them, just put them back from where you got them.

In case of a shared pool of threads, there are 2 important variations - fixed and expandable pools. Depending on a use case, you might opt for one or another.

Think about trade-offs, experiment, see what's working best for you.

Threads: Standalone or Pooled?

When not to use a pool of threads?

1. You have a long running process that you are certain will be running
2. You want to isolate some critical code from the main thread, for security or whatever
3. The cost of switching between threads is too expensive for you
4. Yes

Talking about pooled: Work-stealing

When we're talking about a thread pool, we usually imply having a task queue, where we submit what needs to be done, and then as the workers do their work it is accessed for more tasks.

The issue - it is very possible to have the workers compete for access to the queue. A possible solution would be for each worker to have it's task queue, and the incoming tasks distributed uniformly among them.

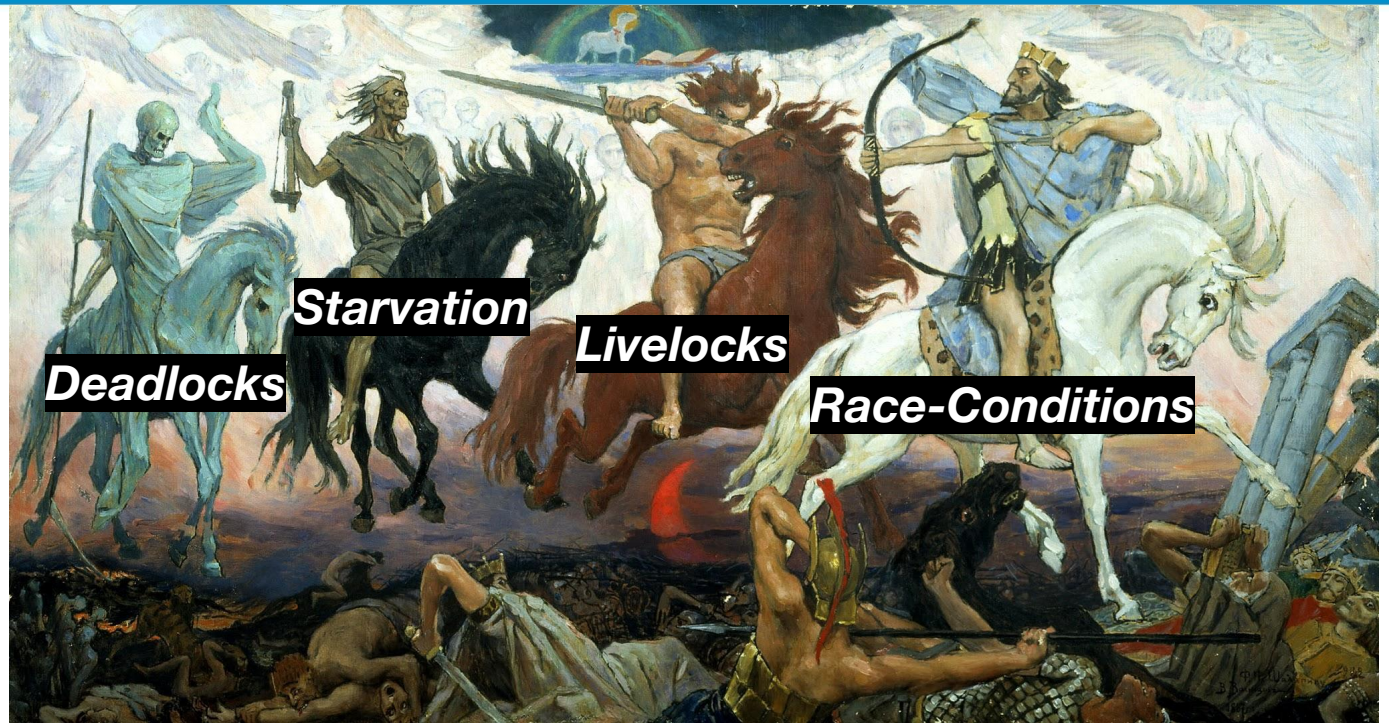
But what happens if some worker has longer running tasks, and others are already done?

Stealing time! The idle workers check their neighbours queue for work, and "steal" from them tasks. It seems like in practice, such an approach is faster than having just one queue.

Concurrency: Perils and Solutions

Alexandru Burlacu

Autumn 2020



*_Four Horsemen of the Apocalypse_ - Viktor Vasnetsov, 1887

Issue 1

Example 1: Say two threads access the code below, what will be `x`?

```
if (x == 42) { // The "Check"
   x = x * 2; // The "Act"
}
```

Example 2: Say N threads run the loop below, what will be `y`?

```
// given y = 0
for (int i = 0; i < 1000; ++i) {
   y = y + 1;
}
```

Issue 1: Race Conditions

A “race condition” can be defined as “Anomalous behavior due to unexpected critical dependence on the relative timing of events” [FOLDOC]

Given at least 2 threads that try to access some shared data and try to change it at the same time, a **race condition** occurs, due to the thread scheduling algorithm that can swap between threads at any time. Therefore, the threads are **racing** to access/change the data.

Possible solution: use locks, or atomic variables/constructs.

Special case: Data Race (when memory is corrupted)

Synchronization Primitives 1: Mutex

Mutex aka Mutual Exclusion is a solution to the race condition problem when multiple threads try to access a shared resource.

The requirement for mutual exclusion was first identified, and a solution was proposed, by Edsger Djikstra in a 1965 paper *"Solution of a problem in concurrent programming control"*.

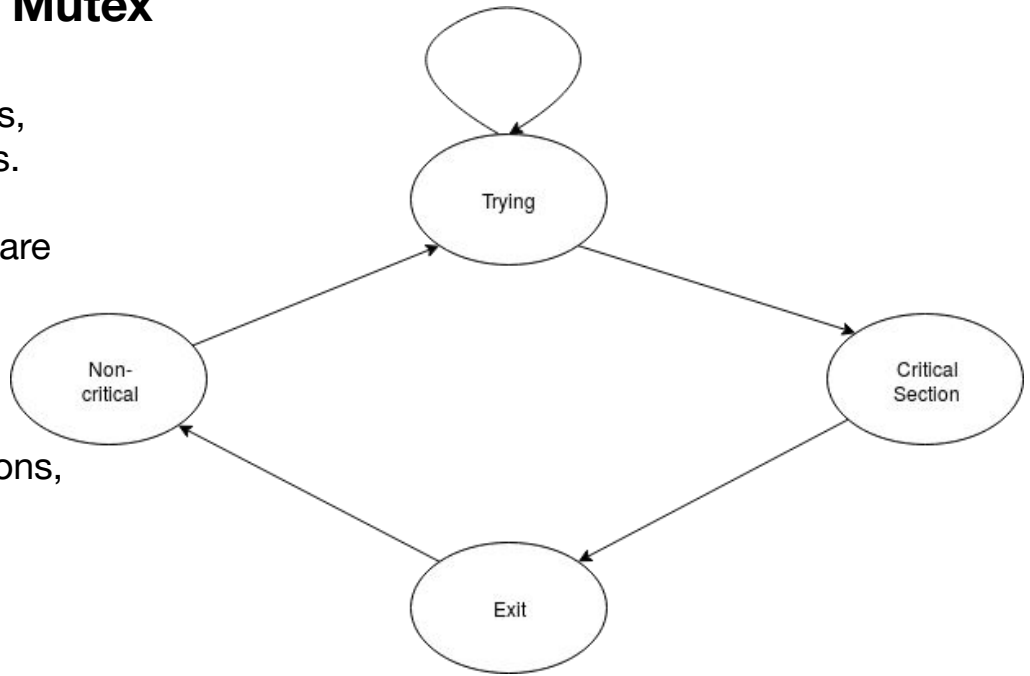
Mutex basically solves the problem by designating a so-called **critical section** in the program, a segment of the program where the manipulation of the shared resource happens, and enforcing that only one thread can access it at any given time, others have to wait.

Synchronization Primitives 1: Mutex

Thus, a program has 4 distinct segments, and therefore a thread can be in 4 states.

Mutexes can be enforced both on software and hardware level.

On hardware level common ways are preventing interrupts in critical sections, busy-waiting, or through atomic operations, most notably test-and-set and CAS (compare-and-swap).



Synchronization Primitives 1: Mutex

Can a mutex be locked more than once?

A mutex is a lock. Only one state (locked/unlocked) is associated with it. However, a *recursive mutex* can be locked more than once (POSIX compliant systems), in which a counter is associated with it, yet retains only one state (locked/unlocked). The programmer must unlock the mutex as many number times as it was locked.*

What happens if a non-recursive mutex is locked more than once?

Deadlock. If a thread which had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in deadlock.*

*Source: https://www.geeksforgeeks.org/mutex-vs-semaphore/

Synchronization Primitives 1: Mutex; Contention vs Overhead

Mutexes, also known as locks, have so-called granularity: how fine/coarse is the segment of the program inside the critical section.

Two important properties to keep in mind are **lock contention** and **lock overhead**.

Overhead happens due to the memory and computational requirements of a lock (it ain't for free). Contention happens when a thread has to wait until another thread releases a lock.

What one must remember is that there's always a trade-off: less locks mean smaller overhead but worse contention while more locks mean smaller contention but more overhead.

Readers-Writers problem

Supposing you have some threads that are trying to read from some resource, and some threads want to write into that shared resource. How to achieve mutual exclusion? What about reading only when values have changed?

Readers-Writers problem

In such a case, having multiple readers at the same time accessing the resource is fine.

While if there's at least one writer that is accessing the shared resource, mutual exclusion is required, otherwise all sorts of nasty things (race conditions, memory corruptions even) could happen.

It is possible to solve this problem using Mutexes, but that means blocking of reader threads, which is bad. Another mechanism is required.

Synchronization Primitives 2: Signaling

Q1: So, you can restrict access to a resource, but how can you make others know when it's available?

Synchronization Primitives 2: Signaling

Q2: How do you make possible to have dependent events. That is, order the execution of threads based on occurrences of particular events?

Synchronization Primitives 2: Signaling

Q1: So, you can restrict access to a resource, but how can you make others know when it's available?

Q2: How do you make possible to have dependent events. That is, order the execution of threads based on occurrences of particular events?

A: You need some sort of signaling mechanism. Let's first talk about the Semaphore.

Synchronization Primitives 2.1: Semaphore

Among the first synchronization mechanisms was the **semaphore**.

A semaphore is simply a variable, with two functions associated with it: `wait()` and `signal()`. Originally `P` and `V` from *passering* ("passing") and *vrijgave* ("release") respectively, from Dutch.

The concept of a semaphore was first proposed by Egdar Djikstra in early 1960s while working on the operating system for Electrologica X8 computer, later known as THE multiprogramming system.

Synchronization Primitives 2.1: Semaphore

The semaphore functions:

```
function V(semaphore S):
    S = S + 1

function P(semaphore S):
    while (S <= 0) {}
    S = S - 1
```

Synchronization Primitives 2.1: Semaphore

Semaphore tracks how many resources are available, not which ones are available. If only one resource is (binary semaphore) tracked, it is similar to a mutex/lock, though **you should NEVER assume they are equivalent!** If there are more than one resources, it is called counting semaphore.

The difference you should be aware about is that mutex/lock is a locking mechanism, while the semaphore is a signaling mechanism.

An important consideration about the semaphores: although it takes into account **how many** resources are available, it doesn't know about **which** resources are free.

Readers-Writers problem

Having semaphores, now it is feasible to solve the readers-writers problem.

The simplest, but not the general case, is the following algorithm:

1. Reader will run after Writer because of read semaphore.
2. Writer will stop writing when the write semaphore has reached 0.
3. Reader will stop reading when the read semaphore has reached 0.

3 Shades of Readers-Writers problem

In practice, there are at 3 most popular readers-writers problems which impose different constraints and might result in different unwanted behaviour.

Many languages even have a specialized reader-writer lock.

Shades of Readers-Writers problem: Basic

```
// Writers
resource.wait()
// critical section
resource.signal()
```

```
// Readers
mutex.wait()
     // critical section
     read_count++
     if read_count == 1
          resource.wait()
mutex.signal()

// perform reading

mutex.wait()
     // critical section
     read_count--
     if read_count == 0
          resource.signal()
mutex.signal()
```

Shades of Readers-Writers problem: No-starvation

```
// Writers
service_queue.wait()
      resource.wait()
      // critical section
service_queue.signal()

// perform writing
resource.signal()
```

```
// Readers
service_queue.wait()
      mutex.wait()
            // critical section
            if read_count == 0
                  resource.wait()
            read_count++
      mutex.signal()
service_queue.signal()

// perform reading

mutex.wait()
      // critical section
      read_count--
      if read_count == 0
            resource.signal()
mutex.signal()
```

Synchronization Primitives 2.2: Busy-waiting

Say you want to work on a resource given some condition/state. How do you do that?
You check it. Again and again? That's busy-waiting.

Generally it's an anti-pattern, but…

… given specific circumstances, it might be a good idea to not use numerous hardware interrupts but just check some value periodically.

Also, if a context switch would be more expensive, busy-waiting should be applied.

Synchronization Primitives 2.2: Busy-waiting

How to spot busy-waiting?

1. You have a multithreaded program
2. You have something like

```
while((local_val = some_global_val) == desired_state) {}
```

That's busy-waiting, enjoy.

Btw, as an extension to this idea, check Spinlocks, and these slides:
http://cs.iit.edu/~khale/class/intro-os/s19/handout/lec10.pdf

Synchronization Primitives 2.3: Condition variables

Generally, a condition variable is used to avoid busy-waiting. Although similar to the Semaphore, the difference is mostly on why use a specific primitive.

Semaphores are like condition variables that are always true/always execute. They are actually used to pass access to some resources.

Condition variables are like semaphores that are `signal`ed on some condition. They are used to notify sleeping threads of some event and awake them.

For more detailed info check: https://cs61.seas.harvard.edu/wiki/images/1/12/Lec19-Semaphores.pdf
https://stackoverflow.com/questions/3513045/conditional-variable-vs-semaphore#3514382
https://www.microsoft.com/en-us/research/wp-content/uploads/2004/12/ImplementingCVs.pdf

Synchronization Primitives 2.3: Condition variables

A condition variable could be thought as a wait-queue with a blocking wait and signal/wakeup operations. To use a condition variable one needs a condition and a mutex, to keep the condition-checking code thread-safe.

And so, the following steps happen:

1. The mutex is acquired
2. The condition is checked
3. Block and release mutex if condition is true, else release mutex

Synchronization Primitives 2.3: Condition variables

```
#include <iostream>
#include <thread>
#include <functional>
#include <mutex>
#include <condition_variable>
using namespace std::placeholders;
class Application
{
  std::mutex m_mutex;
  std::condition_variable m_condVar;
  bool m_bDataLoaded;
public:
  Application()
  {
    m_bDataLoaded = false;
  }
```

```
  void loadData()
  {
   // Make This Thread sleep for 1 Second
   std::this_thread::sleep_for(
          std::chrono::milliseconds(1000));
   std::cout<<"Loading Data from
XML"<<std::endl;
   std::lock_guard<std::mutex> guard(m_mutex);
   m_condVar.notify_one();
  }

  bool isDataLoaded()
  {
    return m_bDataLoaded;
  }
```

Synchronization Primitives 2.3: Condition variables

```
  void mainTask()
  {
    std::cout<<"Do Some Handshaking"<<std::endl;
    std::unique_lock<std::mutex> mlock(m_mutex);
    // Start waiting for the Condition Variable to get signaled
    // Wait() will internally release the lock and make the thread to block
    // As soon as condition variable get signaled, resume the thread and
    // again acquire the lock. Then check if condition is met or not
    // If condition is met then continue else again go in wait.
    m_condVar.wait(mlock, std::bind(&Application::isDataLoaded, this));
    std::cout<<"Do Processing On loaded Data"<<std::endl;
  }
};
```

*Code example from: https://thispointer.com/c11-multithreading-part-7-condition-variables-explained/

Synchronization Primitives 2.3: Condition variables

```
int main()
{
   Application app;
   std::thread thread_1(&Application::mainTask, &app);
   std::thread thread_2(&Application::loadData, &app);
   thread_2.join();
   thread_1.join();
   return 0;
}
```

*Code example from: https://thispointer.com/c11-multithreading-part-7-condition-variables-explained/

Issue 2: Livelocks

TL;DR: Having 2 threads, both are running/aren't blocked, yet the task isn't coming to completion. Because they switch states continuously.

Longer version: Livelocks happen usually when the lock hierarchy isn't respected, so even if the threads are running, they acquire or release the locks in such a way, that there's no possibility of both locks being released.

Issue 2: Livelocks

Example (C#-ish):

```
var lock1 = new LockObjectLikeMutexOrSemaphore();
var lock2 = new LockObjectLikeMutexOrSemaphore();
Thread.Start(()=> { // Thread 1
    while(true) {
        if(!lock1.Lock(1000)) {
            continue;
        }
        if (!lock2.Lock(1000)) {
            continue;
        }/// do some work});
// Next to the left
```

```
// Continue
Thread.Start(()=> { // Thread 2
    while(true) {
        if(!lock1.Lock(1000)) {
            continue;
        }
        if (!lock2.Lock(1000)) {
            continue;
        }/// do some work});
```

Issue 3: Deadlocks; Coffman conditions

TL;DR: When the system blocks due to the fact that necessary resources for one set of processes are acquired by another set of processes, these in turn requiring the resources held by the first set.

Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)
Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time)
Hold and Wait: A process is holding at least one resource and waiting for resources.
No Preemption: A resource cannot be taken from a process unless the process releases the resource.
Circular Wait: A set of processes are waiting for each other in circular form.
These conditions are known also as **Coffman conditions**.

Issue 3: Deadlocks

Example 1:

```
/* PROCESS 0 */
flag[0] = true;
while (flag[1])
    /* do nothing */;
/* critical section*/;
flag[0] = false;
// Continuation on the left
```

```
// Next
/* PROCESS 1 */
flag[1] = true;
while (flag[0])
    /* do nothing */;
/* critical section*/;
flag[1] = false;
```

*Example from: William Stallings, “Operating Systems: Internals and Design Principles”, Appendix A

Dining Philosophers

Initially formulated by Djikstra, and presented in its final form by Tony Hoare, this problem is meant to show synchronization problems in a concurrent system, and how to deal with them.
The original example is full of details but it is elemental to know this: there are 5 philosophers and a bowl of spaghetti, each philosopher has a named chair, and between philosophers there're forks, *"but such is the tangled nature of spaghetti a second fork is required to carry it to the mouth"**. So, there are 5 forks and 5 hungry mouths.

How to ensure that each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think?

*Source: "Communicating Sequential Processes", Tony Hoare, 1985

Dining Philosophers

Although formulated in order to illustrate the risk of a deadlock, depending on solution proposed, one can experience also livelock and resource starvation. Hell…

So, to get starting, imagine that all philosophers are competitive, and everyone picks his fork asap… and won't let go. At some point in time someone will have to call the ritual services.

A more elaborate plan would be to establish a rule by which every philosopher who's waiting more that T has to give up his fork and wait another T smthgs until trying to re-acquire that fork.

Will it work?

Dining Philosophers

It won't.

If by some unfortunate circumstances all 5 philosophers will grab their forks at the same time, forego of them, at the same time, wait T time, then try again ad infinitum, they will livelock.

What was that ritual services number again?

Dining Philosophers

So, there are 3 well-known solution to this problem.

First is to use a so-called **resource hierarchy**.
So, you number the forks (1-5 for example), and the rule is, if you're using two forks, you need to pick up the lower numbered fork first.
By this rule, the last philosopher won’t be able to grab the 5th fork (why?) and will have to wait until another fork is released.

No deadlock. No livelock even.

Still not good (why??).

Dining Philosophers

Because it's slow, as hell. And also hard to deal with if resources are not known in advance.

Another solution is have a policy that philosophers either grab 2 forks, or none and some enforcing mechanism, like a waiter.
In software this is implemented through a semaphore.

But there's a problem with this approach… and it's not just reduced parallelism by having another entity in the process.

Dining Philosophers

It's resource starvation. Because in principle it is possible for a philosopher to never eat under this policy (how?).

The last solution was proposed by K. Mani Chandy and J. Misra, and this one seems to solve all the problem of the previous ones.

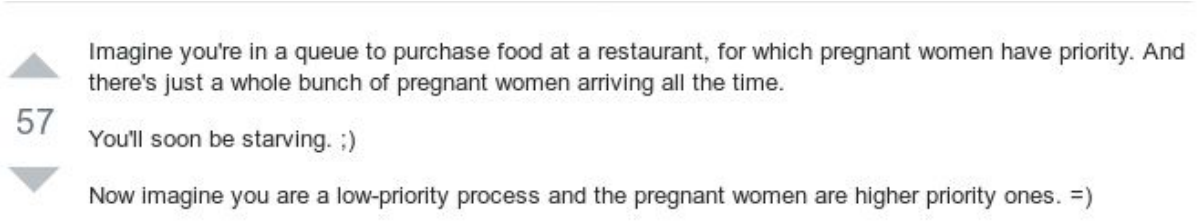
So, let's say all the forks are initially dirty. And the philosophers are ID-ed/numbered. For every pair of philosophers, give the fork to the one with the smaller ID. When a philosopher needs a fork, he asks his neighbor for a fork and either (1) he doesn't get a fork if it is clean, or (2) gets a fork if it dirty and the neighbor cleans it before sending over. When a philosopher has eaten, all his forks are dirty.
The guys who are starving get a higher priority than the guys who are eating! No more problems!

Dining Philosophers

You should really check this one out, especially if you saw “Parks and Recreation”:
adit.io/posts/2013-05-11-The-Dining-Philosophers-Problem-With-Ron-Swanson.html

Issue 4: Resource Starvation

Definition: When a process is perpetually denied access to a resource to complete its work, you get yourself **resource starvation**.



Causes: Issues with thread scheduling (too simple) or mutual exclusion algorithm (deadlocks), Resource leaks, Deliberate attacks (fork bombs).

*Source: https://stackoverflow.com/a/1162610 or https://stackoverflow.com/questions/1162587/what-is-starvation

Issue 4: Resource Starvation

Definition: When a process is perpetually denied access to a resource to complete its work, you get yourself **resource starvation**.

If for an algorithm it is impossible to have resource starvation, it is said to be starvation-free or **finite-bypass**, that is, a process will be bypassed a limited number of times before being allocated necessary resources.

Another way to solve it: **Aging technique** - gradually increasing the priority of processes that wait in the system for a long time.

Synchronization Primitives 3: Barrier

Think about it: You have 5 threads, all of them do some work, and once **all of them** are done, you want to proceed to the next stage in your program. How do you do it efficiently?

Synchronization Primitives 3: Barrier

Using control variables or other synchronization primitives could become cumbersome, that's why barriers where proposed.

A barrier is basically a shared mutex with a counter and a flag. When the `wait` method is called a predefined number of times the mutex is released for all threads simultaneously.

Barriers are mostly used in parallel programming, for example in MapReduce jobs or implicitly in OpenMPI Scatter/Gather/Reduce/Allreduce routines.

Synchronization Primitives 3: Barrier

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

*Code example from: https://docs.python.org/3/library/threading.html#barrier-objects

Producer-Consumer problem

Very similar to a readers writers problem, there's a bounded buffer, or queue, between the processes, rather than a shared resource. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

A possible solution is to make the producer either discard data or go to sleep if the buffer is full, and the consumer goes to sleep if the buffer is empty. Both need to communicate (be awaken) when there's either a free slot in the buffer or it is not empty anymore.

An inadequate solution could result in a deadlock where both the producer and the consumer are waiting to be awaken.

Synchronization Primitives 4: Queues and (Circular) Buffers

Let’s level up a bit.

Queues let us do message-passing/share-nothing (sort of) concurrency which is known to be safer and less error prone.

Also queues have a higher abstraction level than conditional variables, mutexes, semaphores, et cetera, which is good most of the time.

Complex concurrency models like CSP and Actor model (which we’ll discuss during the Real-Time programming course) use queues and message passing.

Synchronization Primitives 4: Queues and (Circular) Buffers

As written in Portland Pattern Repository - *“A circular buffer is a memory allocation scheme where memory is reused (reclaimed) when an index, incremented modulo the buffer size, writes over a previously used location. A circular buffer makes a bounded queue when separate indices are used for inserting and removing data. The queue can be safely shared between threads (or processors) without further synchronization so long as one processor enqueues data and the other dequeues it. (Also, modifications to the read/write pointers must be atomic, and this is a non-blocking queue--an error is returned when trying to write to a full queue or read from an empty queue). Note that a circular buffer with n elements is usually used to implement a queue with n-1 elements”*

In practice circular buffers are more efficient than simple queues in streaming scenarios where there are lots of `put`s and `take`s.

Synchronization Primitives 4: Queues and (Circular) Buffers

```
public class Squarer {

    private final BlockingQueue<Integer> in;
    private final BlockingQueue<SquareResult> out;
    // Rep invariant: in, out != null

    /** Make a new squarer.
     *  @param requests queue to receive requests from
     *  @param replies queue to send replies to */
    public Squarer(BlockingQueue<Integer> requests,
                   BlockingQueue<SquareResult> replies) {
        this.in = requests;
        this.out = replies;
    }
```

*Code example from: http://web.mit.edu/6.005/www/fa15/classes/22-queues/

Synchronization Primitives 4: Queues and (Circular) Buffers

```
/** Start handling squaring requests. */
 public void start() {
     new Thread(new Runnable() {
         public void run() {
             while (true) {
                 // TODO: we may want a way to stop the thread
                 try {
                     // block until a request arrives
                     int x = in.take();
                     // compute the answer and send it back
                     int y = x * x;
                     out.put(new SquareResult(x, y));
                 } catch (InterruptedException ie) {
                     ie.printStackTrace();
                 }
             }
         }}).start(); } }
```

*Code example from: http://web.mit.edu/6.005/www/fa15/classes/22-queues/

Priority Inversion problem

If by now you decided that preemptive multithreading/multitasking is probably the best solution, let me break it to you, it has its own demons. Like **Priority Inversion**.

Priority Inversion is a situation when a high priority task is indirectly preempted by a lower priority task effectively inverting the relative priorities of the two tasks. Why is it even dangerous? What if your log writer thread indirectly preempted the UI thread - then your UI is stuck, therefore bad user experience. Or what if instead of the UI thread, something more mission critical was preempted?

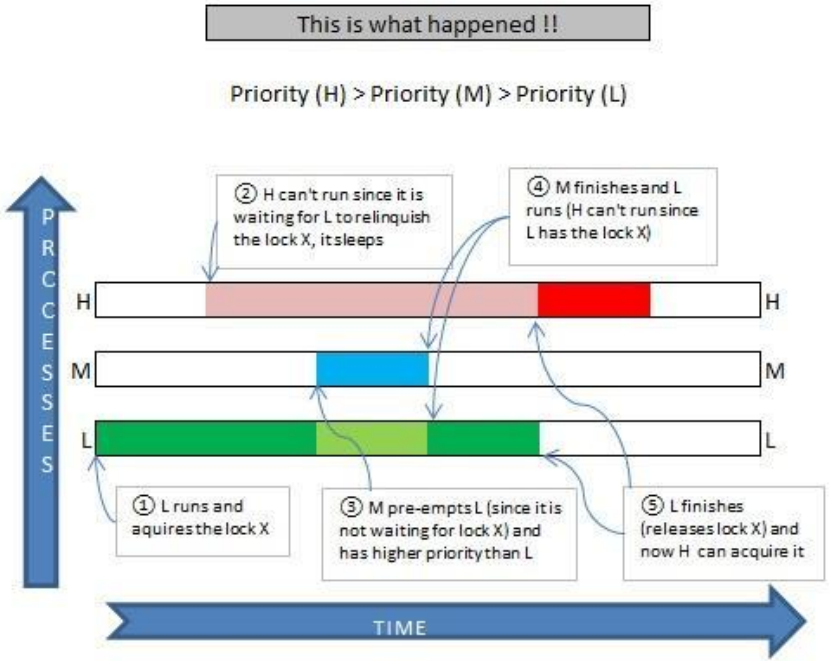
A good explanation here: https://stackoverflow.com/questions/4252158/what-is-priority-inversion

Priority Inversion problem

So how does this happen?

Say we have 3 threads, T1, T2, T3 with respective priorities.

If T1 acquires a lock, and is preempted by T2, then T3 which has highest priority, if requiring the same lock as T1, will be preempted too, effectively T2 preempted T3.



*Source: https://cmdlinelinux.blogspot.com/2013/12/priority-inversion-how-to-avoid-it.html

Priority Inversion problem

There are a number of solutions for this problem.

First, so called **priority ceiling**. It means that shared locks have very high priority, it sometimes being the highest priority of any task which may lock the resource, and the threads/processes holding them will be assigned same priority as these locks.

Another method, **priority inheritance**, temporarily assigns to the low priority task the priority of the highest waiting priority task for the duration of its use of the shared resource, thus keeping medium priority tasks from preempting the low priority task. Quite similar to priority ceiling.

Basically, priority ceiling doesn’t care if the lock is being waited by a higher priority task. Priority inheritance does.

Current Status

Many Flavors of Concurrency

Implementations

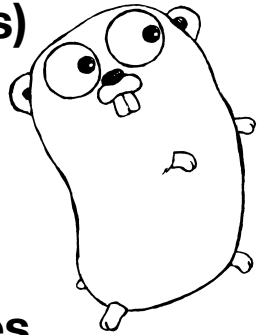
Alexandru Burlacu

Autumn 2020

Actors

Event Loop/ Coroutines

CSP (communicating sequential processes)



STM (software transactional memory)

Also:

- **Tasks/Futures**
- **Dataflow**
- **Just Threads**

Concurrency: Collections

Alexandru Burlacu

Autumn 2020

Concurrent Collections: Why?

For non-mutating operations like `map`/`filter`/`foreach` it is possible, even trivial, to run operations concurrently (or even in parallel). This sort of operations are called massively, or embarrassingly parallel.

It is even possible to run operations like `scan`/`reduce`/`aggregate` concurrently or in parallel, if the objects in the given collections obey certain laws (Monoid laws, but don't sweat it).

Now, given that it is possible to run some operations very efficient, how do you accomplish thread-safety for mutating operations or even getters? Just use synchronization mechanisms.*

*Beware, most concurrent collections have this mechanisms, so applying another mutex, for example, to work with them will likely cause a deadlock.

Synchronized vs Concurrent Collections

Some languages implement both *Synchronized* and *Concurrent* collection.

One might ask why?

Synchronized collections are a more coarse-grained way to ensure thread safety for collections.

Concurrent collections, on the other hand usually provide a more fine-grained synchronization mechanisms. For example usually one can read from a concurrent collection using multiple threads, while only one thread can write.

Synchronized Collections: Java

Found in `java.util.Collection` interface, it's static methods/wrappers provide ways to make any generic Java collection a thread-safe one, not in the most efficient way thought.
For example:

```
Collection<Integer> syncCollection =Collections.synchronizedCollection(new ArrayList<>());
    Runnable listOperations = () -> {
        syncCollection.addAll(Arrays.asList(1, 2, 3, 4, 5, 6));
    };

    Thread thread1 = new Thread(listOperations);
    Thread thread2 = new Thread(listOperations);
    thread1.start();
    thread2.start();
    thread1.join();
    thread2.join();

    assertThat(syncCollection.size()).isEqualTo(12);
}
```

Concurrent Collections: Java

Now, the *Concurrent* collections in Java can be found in `java.utils.concurrent` package, these achieve thread-safety by dividing their data into segments.

`BlockingQueue`, `ConcurrentMap` and `ConcurrentNavigableMap` interfaces can be found in this package, and their implementations too.

In a `ConcurrentSkipList`, the only implementation of `ConcurrentNavigableMap`, for example, different threads can acquire locks on each segment, so multiple threads can access the Skip List at the same time (a.k.a. concurrent access) making it extremely fast.

Synchronized Collections: C#

Just like Java (no surprise here) C# has both synchronized and concurrent collections.

To make a collections synchronized, wrap your collection in the `SynchronizedCollection<T>` class. Essentially, it will add locks to every method of your original collection, ensuring thread safety. It waits you in the `System.Collections.Generic` namespace since .NET 2.

Also there you'll find `SynchronizedKeyedCollection<K, V>` and `SynchronizedReadOnlyCollection<T>` classes there.

Nothing new. Moving on.

Concurrent Collections: C#

The System.Collections.Concurrent namespace is much newer (.NET 4+), and just like java.util.concurrent provides collections with much more fine grained access, therefore higher performance. They also use SpinLocks that are in a way, smart. If need to wait long, they use locks, otherwise, busy-waiting.

These classes no longer use locks to provide thread safety, which means they should scale better in a situation where multiple threads are accessing their data simultaneously.

Here you'll find BlockingCollection<T> that provides bounding and blocking functionality for any collection, also `ConcurrentDictionary<K,V>`, `ConcurrentQueue<T>`, `ConcurrentStack<T>`, `ConcurrentBag<T>` more specialized classes.

The only reason to opt for slower synchronized collections is if (1) you target older .NET 2 or earlier systems, or (2) need something that implements the `IList<T>` interface.

Concurrent Collections: Python

First things first, Python doesn't have true concurrent collections, as compared to Java and C#, but on the other hand a lot of existing collections are by default synchronized.

For example, all Queues in Python are thread safe. But only the Queue from the `multiprocessing` module can be used between processes.

In a way, all Python collections are thread safe. Why?

Another important point, when working with multiple threads or processes and shared variables and collections are desired, use Managers.

Concurrent Collections: Fork-Join approach

One could say that Fork-Join is a parallel design pattern. Basically one parallelizes (fork) certain sections of a program and then collects all the subsolutions (join).

Fork-Join can be applied recursively until a specific level of granularity is achieved.
A simple way to reason about it: Divide et Impera but the for each subtask spawn/fork a process.

```
mergesort(A, lo, hi):
    if lo < hi:                          // at least one element of input
        mid = ⌊lo + (hi - lo) / 2⌋
        fork mergesort(A, lo, mid)  // process (potentially) in parallel with main
task
        mergesort(A, mid, hi)           // main task handles second recursion
        join // join is NOT a barrier, only one thread continues
        merge(A, lo, mid, hi)
```

Reading list

- https://dwheeler.com/secure-programs/ By David A. Wheeler (there's a PDF on the site too)
- https://stackoverflow.com/questions/34510/what-is-a-race-condition
- https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/
- https://eli.thegreenplace.net/2018/launching-linux-threads-and-processes-with-clone/

Keywords (Good to know)

GIL, Atomicity, Lock convoy, Thundering herd problem
Optimistic concurrency control, Compare-and-swap, Read-Copy-Update
ReaderWriterLock, Petri nets, MVCC, Banker's Algorithm

Protocols

Alexandru Burlacu

Autumn 2020

Before we dive into Protocols, let's talk sockets

Alexandru Burlacu

Autumn 2020

BSD Sockets

Also known as Berkeley Sockets are an API for dealing with system sockets, and therefore communication over network. The API was first introduced in 1983, and still used today.

A socket is an abstract representation (handle) for the local endpoint of a network communication path. The Berkeley sockets API represents it as a file descriptor (file handle) that provides a common interface for input and output to streams of data.

Usage of files is the result of adoption of Unix philosophy (better google it) which in this case states that “everything is a file”, that is streaming text interfaces are the lingua franca of Unix programs.

See also: https://cis.temple.edu/~giorgio/old/cis307s96/readings/docs/sockets.html

BSD Sockets

BSD Sockets are just a set of functions...

- *socket()* creates a new socket, of some type
- *bind()* associates a socket with an address
- *listen()* causes a bound TCP socket to enter listening state
- *connect()* assigns a free local port number to a socket
- *accept()* accepts a received incoming attempt to create a new connection, creates a new socket associated with the socket address pair of this connection
- *send()*, *recv()*, *sendto()*, and *recvfrom()* are used for sending and receiving data
- *close()* causes the system to release resources allocated to a socket
- et al.

BSD Sockets

BSD Sockets are just a set of functions, types…

socket(3) creates an endpoint for communication and returns a file descriptor for the socket.

- *domain* - [AF_INET, AF_INET6, AF_UNIX]
- *type* - [SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET, SOCK_RAW]
- *protocol* - specifying the actual transport protocol to use. The most common are IPPROTO_TCP, IPPROTO_SCTP, IPPROTO_UDP, IPPROTO_DCCP. Usually optional.

BSD Sockets

BSD Sockets are just a set of functions, type and a workflow (more or less).
Here's the TCP example:

Server
Create a `socket`
`bind` to an address
Set to `listen`
`accept` clients
`send`/`receive` data
`close` connection

Client
Create a `socket`
`connect` to some host/port
`send`/`receive` data
`close` connection

BSD Sockets: Ruby Example - simple

Creating a socket connection

```
require 'socket'
socket = TCPSocket.new("fcim.utm.md", 80)
socket.write "GET / HTTP/1.1"
socket.write "\r\n\r\n"
puts socket.recv(1024)
# guess what will it be?
```

... or a socket server

```
socket = TCPServer.new("0.0.0.0", 8080)
client = socket.accept
puts "New client! #{client}"
client.write("Hello from server")
client.close
```

And finally allowing the server to handle multiple clients (what could go wrong?)

```
socket = TCPServer.new('0.0.0.0', 8081)

def handle_connection(client)
  puts "New client! #{client}"
  client.write("Hello from server")
  client.close
end

puts "Listening on #{8081}"
loop do
  client = socket.accept
  Thread.new { handle_connection(client) }
end
```

BSD Sockets: Java Client Example - simple, and verbose

```
...
URL u = new URL(args[i]);
if (u.getPort() != -1) port = u.getPort();
if (!(u.getProtocol().equalsIgnoreCase("http"))) {
  System.err.println("I only understand http.");
}
if (!(u.getFile().equals(""))) file = u.getFile();
Socket s = new Socket(u.getHost(), port);
OutputStream theOutput = s.getOutputStream();
OutputStreamWriter out = new OutputStreamWriter(theOutput);
out.write("GET " + file + " HTTP/1.0\r\n");
out.write("Accept: text/plain, text/html, text/*\r\n");
out.write("\r\n");
out.flush();
InputStream in = s.getInputStream();
InputStreamReader isr = new InputStreamReader(in);
BufferedReader br = new BufferedReader(isr);
...  // check for IO exceptions and MalformedURLException
```

BSD Sockets: Java Server Example - simple, and verbose

```
try {
 ServerSocket ss = new ServerSocket(2345);
 Socket s = ss.accept();
 Writer out = new BufferedWriter(new OutputStreamWriter(s.getOutputStream()));
 out.write("Hello There!\r\n");
 out.write("Goodbye now.\r\n");
 out.flush();
 s.close();
}
catch (IOException e) {
 System.err.println(e);
}
```

*Code example from:
http://www.cafeaulait.org/slides/sd2000east/sockets/Java_Network_Programming__Part_2__Sockets__Server_Sockets__and_UDP.html

BSD Sockets: Python Example

First of all, see https://docs.python.org/3.6/library/socketserver.html
Python allows pretty simple implementation of socket servers by splitting server code from request handler and from concurrency. A more low-level example would be this:

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

BSD Sockets: Final words

BSD Sockets can be used for both network communication and IPC. Moreover, for IPC, on UNIX systems, there are UNIX-domain sockets, which are considerably faster than going through the TCP/IP stack.

Even using localhost is not nearly as fast as using UNIX sockets (OS feature), which are compatible with BSD Sockets (API).

Also, because it is a widespread and known API, you should opt for it rather than using lesser known, even if faster, approaches.

I/O Multiplexing: Selecting and Polling

The first step towards efficient I/O is to understand that most of the time the system is just waiting for read/write/connect operations, or going lower-level, filling some buffers.

Wouldn't it be great if we can start such a tasks and come back to it later, when it's done, in the meantime launch some other similar task, but for other resources?

Using file descriptors selection (`select` system call) or polling (`poll` or `epoll` system call) this becomes possible. Both approaches check for some events, usually READ and WRITE events, but it varies.

I/O Multiplexing: Selecting vs Polling

- They both handle file descriptors in a linear way. The more descriptors you ask them to check, the slower they get. More than a hundred file descriptors - depending on your hardware - you will start noticing that waiting for file descriptor activity and the following checking which file descriptor that it was, takes a significant time and becomes a bottleneck.
- The select() API with a "max fds" as first argument forces a scan over the bitmasks to find the exact file descriptors to check for, which the poll() API avoids. A small win for poll().
- select() only uses (at maximum) three bits of data per file descriptor, while poll() typically uses 64 bits per file descriptor. In each syscall invoke poll() thus needs to copy a lot more over to kernel space. A small win for select().

Polling and Selecting are basically the same speed-wise: slow. For more info, check this answer: https://stackoverflow.com/a/3951845/5428334

I/O Multiplexing: Async I/O

What you need to know about asynchronous I/O is that it is a single threaded solution which interleaves operation in a cooperative way (using `await`, or by calling a callback function) for example. An `await` basically means yielding the resources to the event loop so that it will decide who gets to run in foreground now.

An event loop is a loop that runs forever and knows about some functions and their state, so that it can schedule which one should be running. Python `asyncio`, JS, Ruby `EventMachine` and many other similar tools use event loops.

For Python, you should really check out this: https://realpython.com/async-io-python/.
Even if you use some other language, the principles mainly hold.

*Async the JS way: https://www.taniarascia.com/asynchronous-javascript-event-loop-callbacks-promises-async-await

I/O Multiplexing: Event loops

In order to achieve canonical async we need an **event loop**. An event loop is a programming pattern that in theory is very simple:

```
while True:
    events = event_queue.pop()
    for event in events:
        event.execute()
```

Within **`event`** it is possible to submit new tasks to the **`event_queue`**. It better be a priority queue or some kind of `select/poll/epoll`.

In practice it's much harder to do.

*How JS uses the event loop: https://www.youtube.com/watch?v=8aGhZQkoFbQ

I/O Multiplexing: Event loops

The main issues arise from the following:

- how to abstract away the event loop properly
- how to submit tasks (and how they should look like)
- making sure that everything that is passed in the event loop is non-blocking

In a way these are less of event loop's issues and more of how to efficiently embed them within a system.

*How JS uses the event loop: https://www.youtube.com/watch?v=8aGhZQkoFbQ

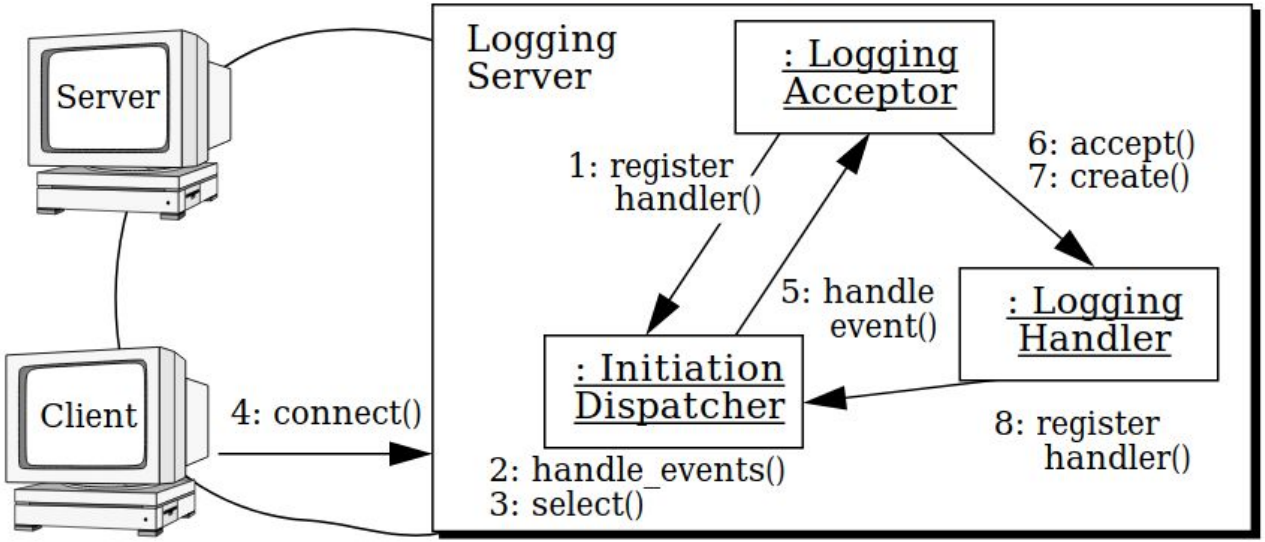
I/O Multiplexing: Reactors

Another beast related to event loops is the **reactor**. One of the most widely known reactor-based systems is the Twisted Python networking library. A reactor is a mechanism for **synchronous** execution of tasks.

Basically, with a reactor you will wait for a handler (read socket/file/timer/whatever) to be **ready** to be used. So a reactor is using system calls like `select` or `poll/epoll/kqueue` to check whenever a resource is ready to be run.

*Original reactor paper: http://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf
and a very nice, gradual implementation: https://hila.sh/2019/12/28/reactor.html

I/O Multiplexing: Reactors



I/O Multiplexing: ... and Proactors

An even more interesting approach is to wait not for handles to be **ready** to execute, but rather to be **done** executing. **Proactor** is similar to reactor pattern, with the difference that the execution on all stages is async. Usually for a proactor to work properly one needs OS support.

The Windows IOCP (IO completion ports) are similar to select and co, but adopt a more “proactor-like” behaviour.

For most languages, the proactor pattern is a bit of a hack, because to execute in the background they **might**, not necessarily do, spawn threads.

*Original proactor paper: https://www.dre.vanderbilt.edu/~schmidt/PDF/Proactor.pdf
also check: https://www.artima.com/articles/io_design_patterns.html

`select` Python Example

```
import selectors
sel = selectors.DefaultSelector()
# ...
lsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
lsock.bind((host, port))
lsock.listen()
print('listening on', (host, port))
lsock.setblocking(False)
sel.register(lsock, selectors.EVENT_READ, data=None)
# ...
while True:
    events = sel.select(timeout=None)
    for key, mask in events:
        if key.data is None:
            accept_wrapper(key.fileobj) # create a session
        else:
            service_connection(key, mask) # continue working
```

*Code example from: https://realpython.com/python-sockets/,
try to modify into Proactor: https://www.slideshare.net/Arbow/the-proactor-pattern

`select` Python Example

```
def accept_wrapper(sock):
    conn, addr = sock.accept()  # Should be ready to read
    print('accepted connection from', addr)
    conn.setblocking(False)
    data = types.SimpleNamespace(addr=addr, inb=b'', outb=b'')
    events = selectors.EVENT_READ | selectors.EVENT_WRITE
    sel.register(conn, events, data=data)
```

*Code example from: https://realpython.com/python-sockets/

`select` Python Example

```
def service_connection(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(1024)  # Should be ready to read
        if recv_data:
            data.outb += recv_data
        else:
            print('closing connection to', data.addr)
            sel.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if data.outb:
            print('echoing', repr(data.outb), 'to', data.addr)
            sent = sock.send(data.outb)  # Should be ready to write
            data.outb = data.outb[sent:]
```

*Code example from: https://realpython.com/python-sockets/

Transport protocols: TCP

Transmission Control Protocol (TCP, IETF RFC 793) is one of the most well-known network protocols, and dates back to ‘80s. It is a stream-oriented protocol that ensures congestion control, in-order data delivery and error-checking with re-transmission of corrupted or lost data chunks. In other words a very reliable protocol, made in the time where network was extremely unreliable.

Now, let’s break down the fancy words describing TCP:

- **stream-oriented** - data is sent as a stream, thus chunked into packets implicitly, not by the user
- **congestion control** - a mechanism that ensures that no more packets are sent than the connection allows. Think a way to quickly get rid of traffic jams.
- **in-order data delivery**, **error-checking**, and **re-transmission** should be self explanatory

Before we dive into transport layer...

We need to clear out some things about IP. It is a **best-effort** communication protocol that treats each packet separately, and therefore can lose data, transport corrupted data, and deliver packets out-of-order. In summary, it's unreliable.

It only checks whenever the header is error-free.

All the nice things we get from the Internet are because of higher level protocols that deal with it.

Also, the packets that IP uses can vary in size, because of the underlying network properties, that's why TCP packets usually use a significantly smaller size than the conventional IP maximum transmission unit (MTU) size. Mostly it's 1500 bytes.

Before we dive into transport layer...

One also needs to understand some even lower level details about the network in order to efficiently use it.

First of all, depending on the type of physical connection used, there are 2 ways signals can flow through, in **FDX (full-duplex)** and **HDX (half-duplex)**. It means that it is possibly to send signals both ways through the same channel, or it is necessary to alternate the destinations between transmissions, respectively. FDX is how modern Ethernet works, HDX is how named pipes work.

Also, take into account that FDX/HDX describe whenever or not the signals are flowing through the connection at the same time or alternating. HDX does not imply that communication is unidirectional. Just like with named pipes, it is possible to have bidirectional communication using unidirectional channels, or rather, half-duplex channels.

Congestion control

A few words on TCP congestion control (CCA). **There are multiple algorithms for it**.
Classic TCP uses a so-called Additive increase - multiplicative decrease (AIMD) mechanism to control congestion. The idea is simple - if congested, continuously decrease the number of packets sent by dividing the maximum number of packets within some time interval, and once congestion is gone, start increasing the traffic by adding packets sent per some time interval.

Another interesting mechanism is keeping the number of packets sent initially low and exponentially increasing that number once the connection was established. This is called slow-start.

There’s a lot more to TCP congestion control, so check out the references at the end of the slides.

Congestion control

How do we know we have a congestion in the network?

Most algorithms just use loss information as a proxy of how congested a network is.
Some use gray-box methods, like the mechanisms that employ fast retransmits based on estimation of round-trip time (RTT) and maximum bandwidth.

There are event so-called green-box methods, that is methods that receive information about network congestion from the network itself, via special protocols, flags in TCP headers, or other methods. All in all, these methods require special hardware and setup. But seem to be most optimal.

*For more info on CCA: https://stackoverflow.com/questions/8683722/how-can-i-do-congestion-control-for-a-udp-protocol

Congestion control

Among other techniques it's worth noting such mechanisms as choke packets, which could in principle emulate the behavior of green-box models.

Whatever method is used, some things remain constant, like the fact that TCP estimates all these things using congestion window (CWND) and receiver window (RWND). These are some counters.

Generally congestion control is subset of flow control.
An example of flow control in TCP is sending the RWND from receiver to the sender so it knows how much data to send to receiver and no more, so as not to overwhelm it.

*For more info on cwnd/rwnd check: https://blog.stackpath.com/glossary-cwnd-and-rwnd/

Congestion control

TCP is using sliding windows for both congestion and flow control.

A TCP *cwnd* window is the amount of unacknowledged data a sender can send on a particular connection before it gets an acknowledgment back from the receiver, that it has received some of the data.

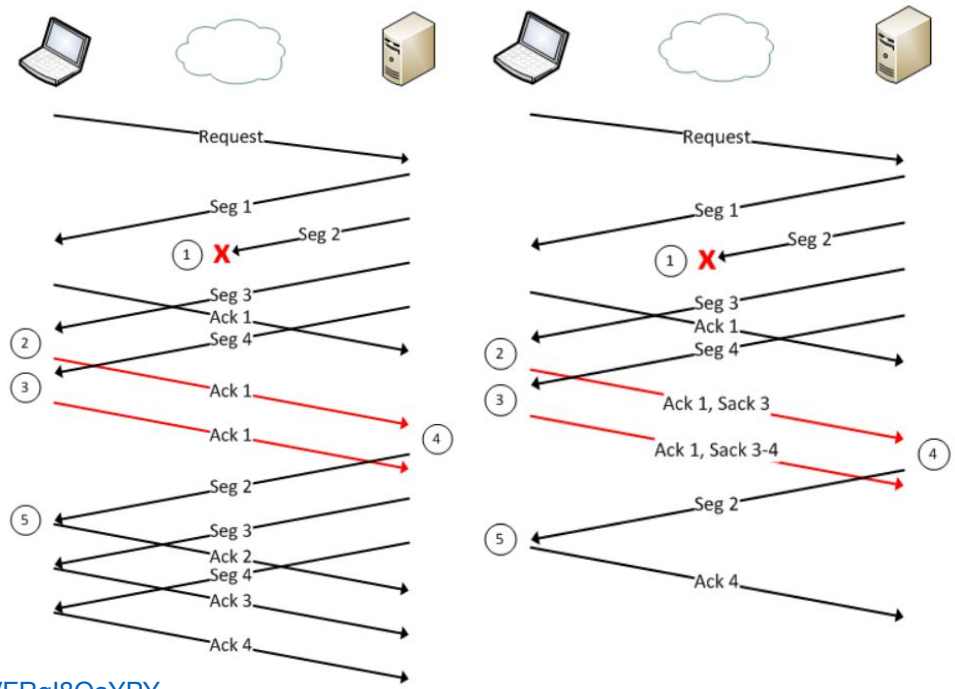
The receiving device should acknowledge each packet it received, indicating the sequence number of the last well-received packet. After receiving the ACK from the receiving device, the sending device slides the window to right side.

*For more info on cwnd/rwnd check: https://blog.stackpath.com/glossary-cwnd-and-rwnd/

Congestion control

TCP not-acknowledges packets by sending duplicated ACKs of for the last continuous series of packets. In such situations, all packets between the not acknowledged one and the ones already sent are retransmitted.

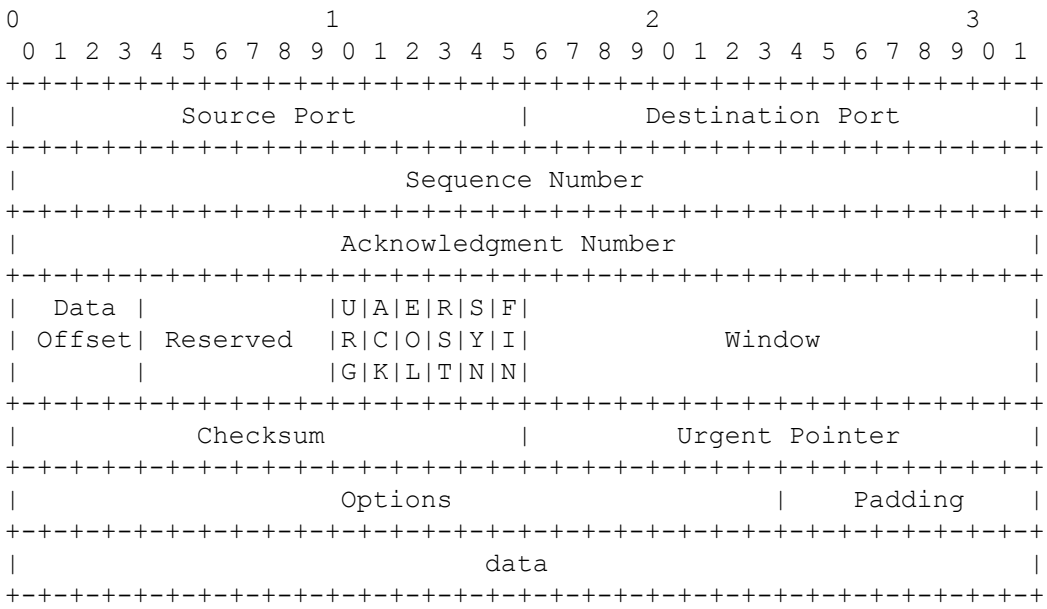
A solution for this is using so called selective acknowledgements (SACK) that acknowledge specific packets.



*For more info on sack check: https://www.youtube.com/watch?v=VERgI8QaYPY

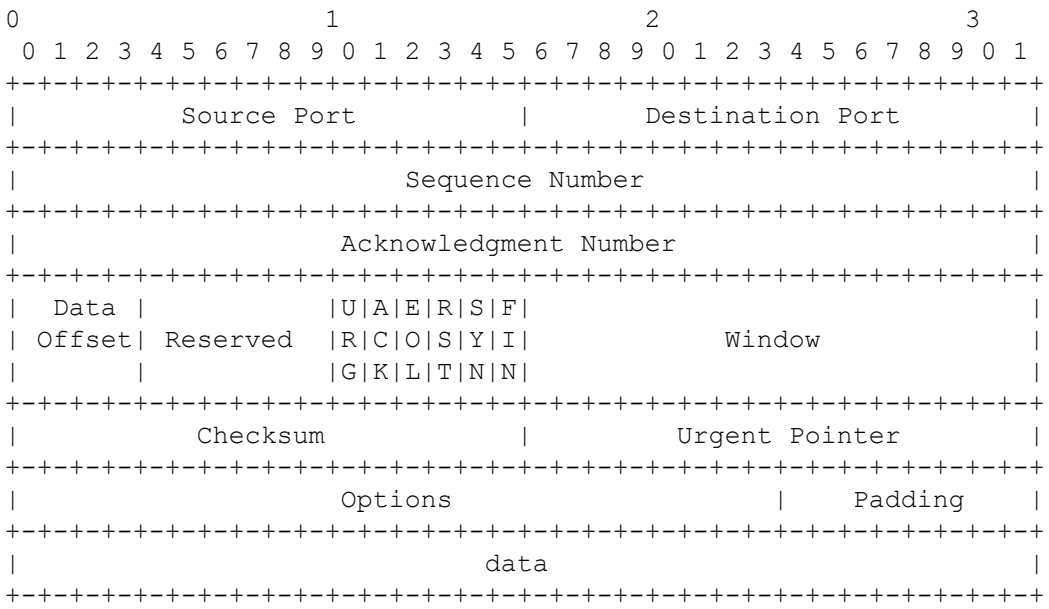
Transport protocols: TCP

Sequence Number - The sequence number of the first data octet in this segment (except when SYN is present).
Acknowledgement Number - If the ACK control bit is set this field contains the value of the next sequence number the sender of the segment is expecting to receive. Once a connection is established this is always sent.
Data Offset - Number of 32-bit words in the TCP header
Reserved - always 0



Transport protocols: TCP

URG - Urgent Pointer field significant
ACK - Acknowledgment field significant
EOL - End of Letter (0/1)
RST - Reset the connection (0/1)
SYN - Synchronize sequence numbers (0/1)
FIN - No more data (0/1)
Window - the number of data octets beginning with the one indicated in the acknowledgment field which the sender of this segment is willing to accept.



Transport protocols: TCP

TCP 3-Way Handshake for Connection Synchronization

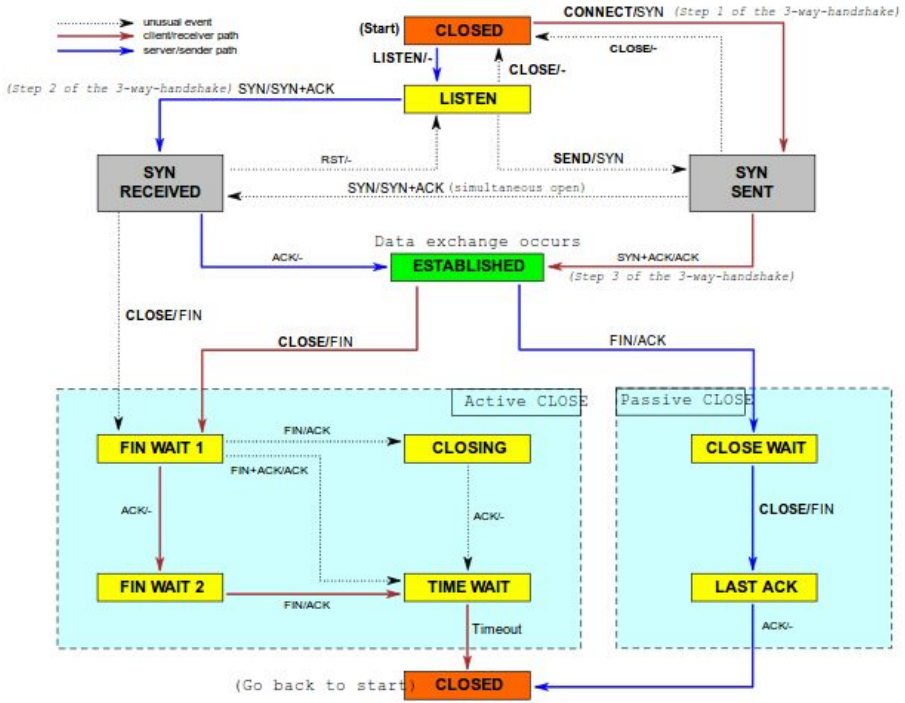
```
      TCP A                                                TCP B

  1.  CLOSED                                               LISTEN
  2.  SYN-SENT    --> <SEQ=100><CTL=SYN>               --> SYN-RECEIVED
  3.  ESTABLISHED <-- <SEQ=300><ACK=101><CTL=SYN,ACK>  <-- SYN-RECEIVED
  4.  ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK>       --> ESTABLISHED
  5.  ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK><DATA> --> ESTABLISHED
```

Basically, TCP has a special way to make sure that the connection is established.
First, (2) the sender sends a SYN message to the receiver, indicating that it will use sequence numbers starting with sequence number 100, then (3) the receiver responds with an ACK message, (4) and finally (before sending actual data) the sender sends and ACK as a response to receiver's ACK.
Note that the sequence number doesn't change between (4) and (5) - ACK ain't no data.

Transport protocols: TCP

There's a problem with the 3-way handshake mechanism.
It is possible to send a lot of SYN packets and to not send the client-ACK (after SYN-ACK) on purpose, thus having a lot of half open connections, and consuming server's ports, resulting in DoS (denial of server). This is called SYN flood.

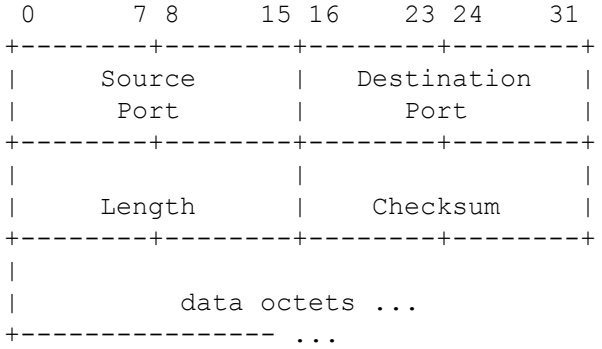
There are thankfully some mitigation strategies.



*Source: https://en.wikipedia.org/wiki/Transmission_Control_Protocol

Transport protocols: UDP

User Datagram Protocol, described in RFC 768, is a very simple, unreliable, message-oriented protocol, which is widely used when ordered delivery and packet loss avoidance can be sacrificed for increased performance.



User Datagram Header Format (taken from RFC 768)

Transport protocols: UDP

UDP supports multicast and broadcast, thus being used in different service discovery protocols, like DHCP, DNS and others.

UDP is connectionless, that means that it doesn't have any connection protocol and can send data right away to the desired destination. Recall that TCP does have a special connection protocol (3-way handshake) and thus is referred as a connection-oriented protocol.

Brief about a lesser known protocol

Scalable TCP was designed to provide higher throughput and scalability. Standard TCP halves the congestion window for every packet loss, and then slowly (*+1/cwnd*) starts to kick in to ramp the speed back up, which is slow in high-throughput, over 100Mbps, connections.

Scalable TCP uses a similar to classic TCP algorithm, but with different additive and multiplicative coefficients. It reduces the *cwnd* by ⅛ for every packet loss/timeout, resulting in a significantly increased throughput, it also ramps up the number of packets faster (*cwnd += 0.01* over *cwnd += 1/cwnd*), in fact turning into an exponential function.
Check http://datatag.web.cern.ch/datatag/papers/pfldnet2003-ctk.pdf.

Email protocols: SMTP/POP3/IMAP

Things you should know:
SMTP - is a standard (RFC 821 and 5321), runs on port 25, user-level email clients use it for sending on ports 587 or 465. Normally, SMTP is used to push the message to a mail server.

POP3 - used to retrieve mails from the mail server. The primary use case for it was/is to download data from the mail server on the local machine and then delete the messages. It uses port 110, or 995 if the traffic is encrypted (TLS, SSL, or whatever).

IMAP - is more complex than POP3. Has the notion of directories, and defaults to not deleting the messages on the server. Runs on 143, or 993 if encrypted traffic. IMAP supports MIME data (images, binaries, audio, almost anything) while POP3 doesn’t. It also supports multiple users of the same mailbox. It supports flagging messages too.

FTP - File transfer protocol

This one will be brief too.
FTP is quite an old protocol, and it uses 2 separate connections, one for data and another one for control.
FTP has to connection modes, active and passive. In both cases the client establishes a control connection to servers' port 21.
Active means the client tells the FTP server he's listening for incoming data connection on some port M.
Passive mode is used via sending a PASV message to the server, by the client, which results in the server responding with a port number used by the client to establish a data connection with the server.

FTP can be used via SSH (SFTP) or using TLS/SSL (FPTS). For more info, may Google be with you.

Brief-ish about another lesser known protocol

SCTP - Stream Control Transmission Protocol, described in the RFC 4960, provides features similar to both TCP (ensures reliable, in-sequence transport of messages with congestion control) and UDP (message oriented). It differs from those protocols by providing multi-homing aka redundant paths to increase resilience and reliability.

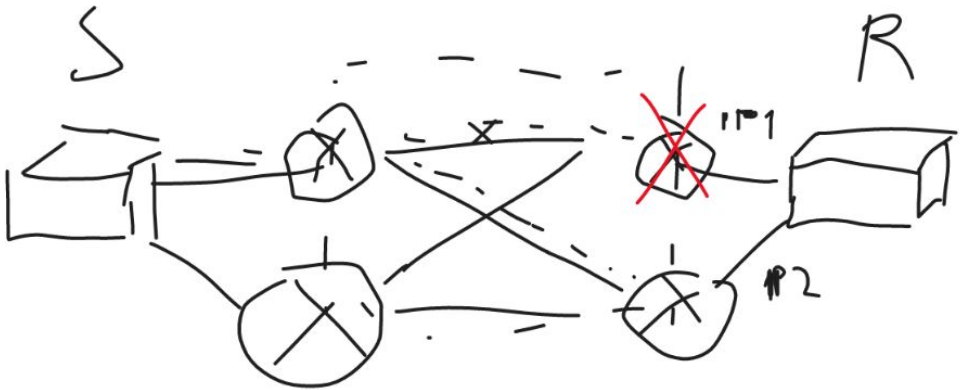
The so called **multi-homing** is multiple IP addresses per host to increase redundancy and make the protocol more resilient.

SCTP also supports stream multiplexing, that is, multiple datagrams/chunks, are sent within one SCTP packet. Stream multiplexing is possible with other protocols too, but they can lead to Head-of-Line blocking, as in TCP. SCTP only preserves the order of the messages within a single stream to mitigate HOL.

SCTP multi-homing

Multi-homing means you can connect your server to multiple routers, thus having multiple addresses for it.

Now, what makes things interesting is that SCTP knows it can do multi-homing, and while being connected to a client, can notify it about other valid addresses, such that in case one router fails, the server is still reachable by the client via the other address(es).



SCTP 4 way handshake

SCTP 4-way handshake

1. Sender **=INIT chunk=>** Receiver
   Receiver generates a secure hash.

2. Receiver **=INIT_ACK chunk=>** Sender
   Sender receives a chunk with a cookie with a secure hash and a MAC (Message Auth Code).

3. Sender **=COOKIE_ECHO=>** Receiver
   Sender sends back a copy of the cookie with the hash.

4. Receiver **=COOCKIE_ACK=>** Sender
   Receiver validates the information in the senders cookie. Connection is established.

HTTP

HTTP (HyperText Transmission Protocol) was initially proposed by TIm Berners-Lee at CERN in 1989. It is highly advised you read that proposal*.

HTTP1 and 1.1 are text based application level protocols that enforce a client server architecture. HTTP1/1.1/2 are built atop TCP, to ensure reliable transmission. The protocol was designed in order to facilitate easy, distributed and collaborative information management.

Hypertext part is probably one of the most important ones, because the idea is to have a web-like information structure, rather than a hierarchical one. Having links and so called “non-linear documents” facilitates quick discovery of necessary information.

Because of the necessity to work with text and perform many transitions, FTP was discarded.

*The proposal can be found at: w3.org/History/1989/proposal.html

HTTP

HTTP protocol has a number of components: header fields, body, status codes, URLs and request methods.

Body - the hypertext document that was requested. It may contain URL addresses of other resources.

URL - universal resource locator, that's the technical name of a web address. It uses a URI encoding with the scheme **http** or **https** (**s** for secure).
An URI looks like this:

```
https://jane.doe:passwd@www.somesite.org:8081/posts/?tag=atag&ord=asc#top
^^^^    ^^^^^^^^^^^^^^    ^^^^^^^^^    ^^  ^^^^^  ^^^^^^^^^^^^    ^^
  |       Username +          Host        Port  Path         Query        Fragment
Scheme    Password                            (or Route)
```

HTTP

Header fields - contain information about either the request or response, like, status codes, the type of response the user wants, the type of response the user receives, cache information, access tokens, and so on.

Status codes - once the request was made, the response follows with a special code, that describes how did it go. Generally 1xx codes are informative, 2xx mean success, 3xx - redirection, 4xx - client errors, 5xx - server errors. There are a number of error codes that are standard, and there are others that are not (nginx 499), or are a joke (418, check HTCPCP).

Most common status codes are: 200 - OK, 201 - Created, 301 - Moved permanently, 400 - Bad request, 403 - Forbidden, 404 - Not Found, 500 - Internal Server error.

HTTP

Request Methods - sometimes referred as verbs, define how a specific resource should be acted upon. Commonly, methods are divided into safe and unsafe, what makes the distinction is the purpose of the method. If the method is intended for retrieving information, it is safe.
So, GET, OPTIONS, HEAD and TRACE are safe, while POST, PUT, DELETE and some others are not.

Another important consideration of HTTP methods is **idempotency**, which is the property of an action to have the same effect whenever it was applied once or multiple times in a raw.

A primer, if you DELETE a resource twice (or any number of times), without anything else accessing it in between actions, the result will be the same as if you delete it once.
Safe methods are idempotent, but not the other way around (PUT is idempotent).

HTTP

HTTP, since version 1.1 allows connection reuse, which greatly improves performance (due to TCPs 3 way handshake and slow start mechanisms). This feature is commonly known as **keep-alive**.

There's also support for **pipelining**, that is sending multiple requests before receiving a response. It has some issues with HOL blocking, that's why it's not used by default.

Another feature of HTTP is the possibility to do **byte serving**, commonly referred to as range requests, that allows to fetch only some data from the resource. For this to work the server has to contain *Accept-Ranges* header in the response, and then the client must specify the range via *Range: bytes=start-end*.

SSE - Server Sent Events

What if it would be possible to keep a connection between server and client and allow server to push data to the client without explicit requests?

Enter SSE, or Server-sent events.

SSE is built atop HTTP, and EventSource API is actually part of HTML5.

Depending on the language and framework used the implementations will vary, but generally you need to return in one way or another an iterator from your route controller and also set the *Content-Type* header to **text/event-stream**.

SSE events must have at least the *data* key, but can also have *id*, and *event*.

SSE - Server Sent Events

```
# Flask setup omitted for brevity
@route("/stream")
def stream():
    def eventStream():
        while True:
            # Poll data from the database
            # and see if there's a new message
            if len(messages) > len(previous_messages):
                yield "data:
                    {}\n\n".format(messages[len(messages)-1])"

    return Response(eventStream(), mimetype="text/event-stream")
```

WebSockets

WebSockets are much more powerful compared to SSE. They are fully bi-directional, that is a client and a server can communicate via a single connection, sending each other messages.

WebSockets although being different, is designed to work over HTTP. For example, WebSockets uses HTTP to establish a connection.

Request

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

Response

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept:
HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

Briefly on HTTP/2

Compared to HTTP/1.1, HTTP/2 is now binary, and primarily focused on increased performance and lower latency. It is also commonly used in secure variant, using TLS (transport layer security) encryption.

HTTP/2 uses HPACK compression of the headers, provides server push mechanism (like SSE), multiplexing multiple requests over a single TCP connection and stream prioritization.

It must be noted that since wide adoption of HTTP/2 in 2015, the WebSockets technology use is declining, although there are still some domains where it performs fairly well.

Briefly on HTTP/2: Binary, multiplexing and stream prioritization

The data in HTTP/2 is converted from text to binary. The binary representation is embedded into frames, and then into a stream, finally the streams can be prioritized. There are many benefits to it.

Binary representation is both easier to parse and has smaller footprint.
Having frames gives the possibility to interleave multiple request and response frames within a connection, this way alleviating the HOL problem.

By giving priority to different streams it is possible to make sure the most important parts of a web page arrive and render first, for example HTML and CSS files, then some core JS dependencies and finally the additional JS code and maybe some images. Not only this, HTTP/2 allows do define parent-child relation between streams.

Briefly on HTTP/2: Header compression

HTTP/2 header compression uses a special designed HPACK algorithm, which uses a combination of static and dynamic dictionaries, and Huffman encoding.

First, the most common keys, or key value pairs are stored in a static dictionary and referenced. Then, the key-values that do not change during the requests are stored in a dynamic dictionary and referenced in the actual header. Finally, what's left is Huffman encoded.

For more info, check http2.github.io/http2-spec/compression.html.

Protocol development. Representation. State machines.

An important part in designing protocols is to specify in a very detailed, non-ambiguous way, how it works. Natural language is ambiguous, and in order to alleviate this problem, special domain language with very strict definitions, and tons of documentation are usually needed.

An alternative, would be to use something a priori formal, an automaton, or state machine.
For our use case, designing and documenting a protocol, there are 2 main types of state machines:

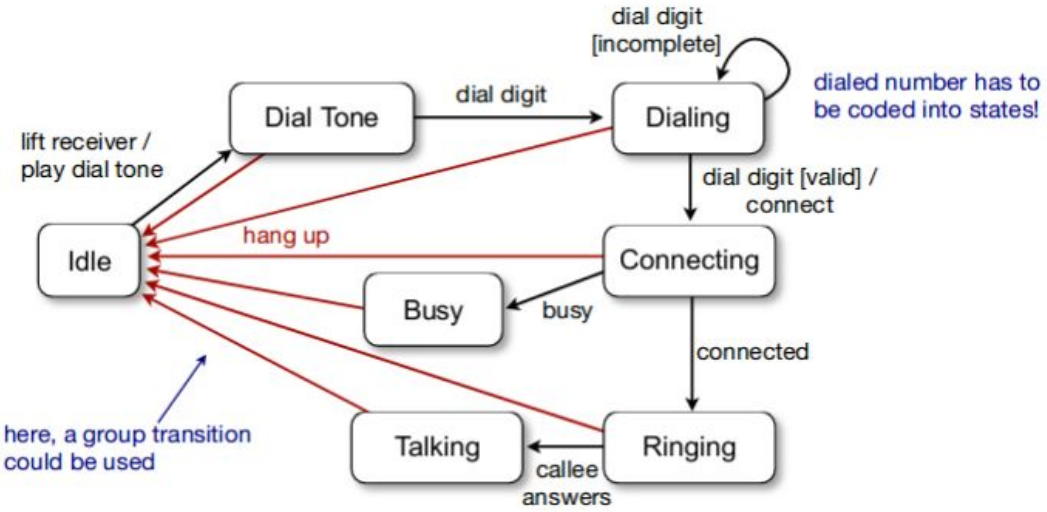
- Mealy (output determined by current state and input)
- Moore (output determined by current state only)
- both are equivalent.

Based on: http://archive.cone.informatik.uni-freiburg.de/teaching/vorlesung/protocol-design-s09/slides/04-Protocol_Specification_1.pdf

Protocol development. Representation. State machines.

There's a problem with finite state machines, actually a number of them

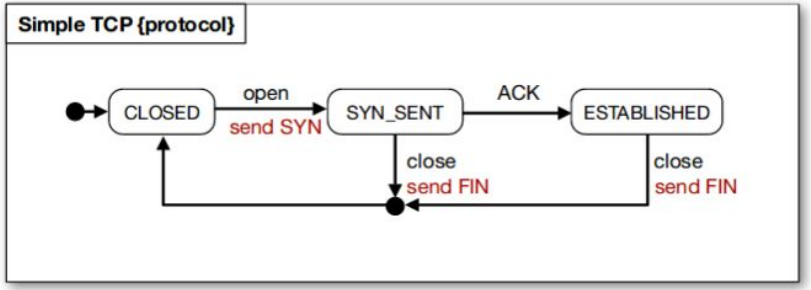
- They don't have abstraction and composition
- They don't allow for data variables, thus requiring every change to be in the state space
- They break when having concurrency
- Problems with memory, incapable of properly formalizing variable length messages



Based on: http://archive.cone.informatik.uni-freiburg.de/teaching/vorlesung/protocol-design-s09/slides/04-Protocol_Specification_1.pdf

Protocol development. Representation. State machines.

Due to all this problems, various solution where proposed. Note that they either lack the superior formality of FSMs, or are too complex to use for formal verification. Still, are widely used for documentation and design. For this course, State and Sequence diagrams will be used.



Send signal actions are not modeled here.

Based on: http://archive.cone.informatik.uni-freiburg.de/teaching/vorlesung/protocol-design-s09/slides/04-Protocol_Specification_1.pdf

Protocol development. Representation. State machines.

Before we start, a couple of words on Sequence diagrams, their strengths and weaknesses.

Modeling a protocol in terms of a “story” where the client communicates with the server, by outlining the sequence of messages sent to one another is indeed intuitive and easy to do, but there’s a problem.

Sequence diagrams normally will show you only one, “happy” flow of messages between clients and servers without becoming cluttered. This sometimes results in a number of sequence diagrams to formally describe the protocol.

Protocol development. Representation. State machines.

A protocol specification consists of five distinct parts. To be complete, each specification should include explicitly:

1. The service to be provided by the protocol
2. The assumptions about the environment in which the protocol is executed
3. The vocabulary of messages used to implement the protocol
4. The encoding (format) of each message in the vocabulary
5. The procedure rules guarding the consistency of message exchanges

The 5th part is the hardest to validate.

From: *"Design and Validation of Computer Protocols"* - Gerard J. Holzmann

Protocol development. Representation. State machines.

As an example, let's see the so called Lynch's protocol, which he describes in 1968 as *"... a reasonable looking but inadequate scheme published by one of the major computer manufacturers in a system information manual."*

First, **service specification**:
The purpose of the protocol is to transfer text files as sequences of characters across a telephone line while protecting against transmission errors, assuming that all transmission errors can in fact be detected.
The protocol is defined for full-duplex (allows for transfers in two directions simultaneously) file transfer. Positive and negative acknowledgments for traffic from A to B are sent on the channel from B to A, and vice versa. Every message contains two parts: a message part, and a control part.

From: *"Design and Validation of Computer Protocols"* - Gerard J. Holzmann

Protocol development. Representation. State machines.

Assumptions about the Environment:
The ‘‘environment’’ in which the protocol is to be executed consists minimally of two users of the file transfer service and a transmission channel.

The users can be assumed to simply submit a request for file transfer and await its completion.

The transmission channel is assumed to cause arbitrary message distortions, but not to lose, duplicate, insert, or reorder messages.

From: *“Design and Validation of Computer Protocols”* - Gerard J. Holzmann

Protocol development. Representation. State machines.

Protocol Vocabulary:
The protocol vocabulary defines three distinct types of messages:

- **ack** for a message combined with a positive acknowledgment
- **nak** for a message combined with a negative acknowledgment
- **err** for a message with a transmission error.

The vocabulary can be succinctly expressed as a set: *V = { ack, err, nak }*.

Each message type can further be refined into a class of lower-level messages, consisting for instance of one sub-type for each character code to be transmitted.

From: *"Design and Validation of Computer Protocols"* - Gerard J. Holzmann

Protocol development. Representation. State machines.

Message Format:
Each message consists of a control field identifying the message type and a data field with the character code. For the example we assume that the data and control fields are of a fixed size. The general form of each message can now be represented symbolically as a simple structure of two fields: *{ control tag, data }* which in a C-like specification may be specified in more detail as follows:

```
enum control { ack, nak, err };
struct message { enum control tag; unsigned char data; };
```

The line starting with the keyword enum declares an enumeration type named control with three possible values: one for each message type used. The message structure itself contains two fields: a tag of type control, and a data field declared as an unsigned character (one byte).

From: *"Design and Validation of Computer Protocols"* - Gerard J. Holzmann

Protocol development. Representation. State machines.

Procedure Rules:

The procedure rules for the protocol were informally described as follows:

1. *If the previous reception was error-free, the next message on the reverse channel will carry a positive acknowledgment; if the reception was in error it will carry a negative acknowledgment.*
2. *If the previous reception carried a negative acknowledgment, or the previous reception was in error, retransmit the old message; otherwise fetch a new message for transmission.*

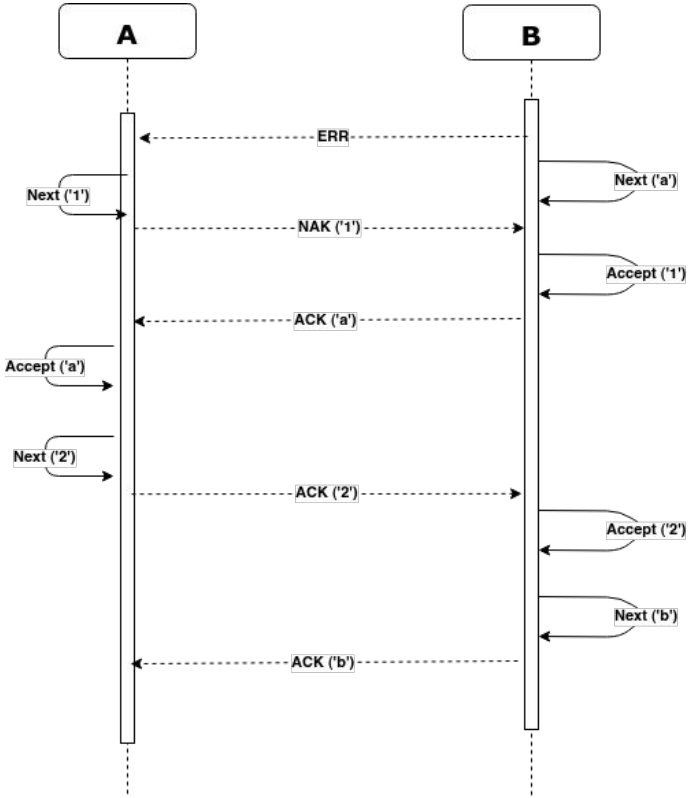
To formalize the rules, a state and a sequence diagram are presented.

From: *"Design and Validation of Computer Protocols"* - Gerard J. Holzmann

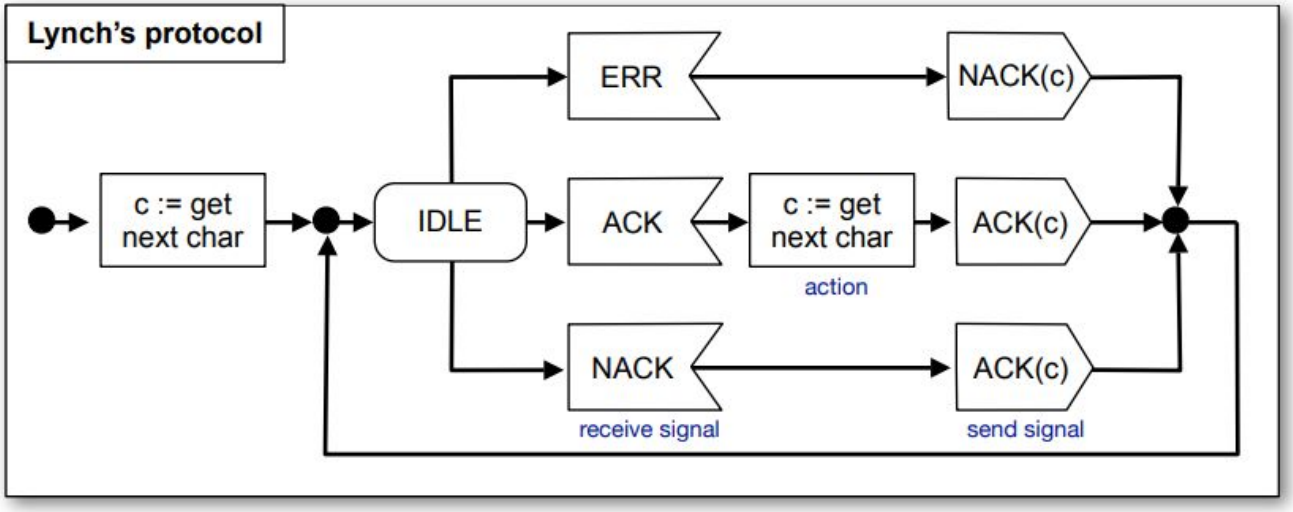
Protocol development.

On the left the protocol is represented as a sequence of actions/calls. It is already visible that the protocol has issues.

First and foremost, it lacks a proper setup and teardown procedure. Secondly, it can only move characters if it receives other characters. There are also more subtle bugs in it.



Protocol development. Representation. State machines.



Based on: http://archive.cone.informatik.uni-freiburg.de/teaching/vorlesung/protocol-design-s09/slides/04-Protocol_Specification_1.pdf

Error detection and error correcting codes

Errors in transmission happen for whatever reasons. Most common ones are rearranged bits, bit flips, duplicate bits, deleted/lost bits, and sometimes even inserted bits. All it depends on the network characteristics.

For example, the probability of an error in computer system circuits is 1e-15, in case of optical fiber cables - 1e-9, coaxial ones - 1e-6, and in case of telephone links - between 1e-4 and 1e-5. The difference is huge indeed.

Just so you would understand better, *“At a rate of 9600 bits per second, it [error probability of 1e-15] would cause one single bit error every 3303 years of continuous operation. At the same data rate, a bit error rate of 1e–4 causes a bit error, on average, once a second.”**

*From: *“Design and Validation of Computer Protocols”* - Gerard J. Holzmann

Error detection and error correcting codes

*Errors are usually of 2 types:

- Linear distortion of the original data, for instance, as caused by bandwidth limitations of the raw data channel
- Non-linear distortion that is caused by echoes, cross-talk, white noise, and impulse noise

Also, depending on the characteristics of a given system, it might be more probable to get single bit errors, that do not “cascade” (like RAM memory, or CPU caches) or, more likely in practice, burst errors, that is, if an error occurs for a given bit/byte/sequence, the probability of the next sequence to be distorted is much higher than the baseline. This understanding is important when choosing how the system will cope with errors.

*From: *“Design and Validation of Computer Protocols”* - Gerard J. Holzmann

Error detection and error correcting codes

It should be clear that in order to cope with errors, anyhow, we need redundancy in the information that we are trying to send.

The simplest way to at least identify that the data being sent was distorted, is to have a **parity check**. The principle is truly simple. Append to the data a single bit that is either 0 or 1, in such a way that if we XOR all the bits in the packet, they will be 0.

Say, you want to send a bit string like 011011001 over the network and ensure that if an error occurs, you will find out about it. With a simple parity check, you add to the bit string 1, and now when the receiver XORs all bits sequentially he will know if the data was corrupted along the way.

Or will it?

Error detection and error correcting codes

Error correction, or forward error control, is used to not only check for the presence of distortions in the messages, but also, to some extent, correct them.

Say, you want to broadcast a simple weather forecast: Sunny, Rainy, Foggy, Cloudy. The minimum amount of bits needed are 2. How many bits are necessary for a reliable transmission of this information?

To help you with this, think of how many bits are necessary to send reliably (with error correction) that a message was either received or not (*ack* and *nak*). The answer is 3. Treat *ack* as 000 and *nak* as 111, if one bit is flipped in either case, you can correct it. That should work most of the time.

Error detection and error correcting codes

Say, you want to broadcast a simple weather forecast: Sunny, Rainy, Foggy, Cloudy. The minimum amount of bits needed are 2. How many bits are necessary for a reliable transmission of this information?

The answer is 5 bits. And you place the bits in a very interesting pattern, that is called **Hamming Code**.

	Orig	w/ Par	w/ Hamming
Sunny -	00	-> 000	-> 00000
Rainy -	01	-> 011	-> 10011
Foggy -	10	-> 101	-> 11100
Cloudy-	11	-> 110	-> 01111

Error detection and error correcting codes

Hamming code, with an additional, overall parity bit, is a so-called Single Error Correction, Double Error Detection (**SECDED**) scheme. It is primarily used in computer memory (ECC RAM).

The scheme works in the following way: the redundant bits are waived into the data bits at powers of 2 positions (1st bit, 2nd bit, 4th bit, 8th bit, you got the idea) and these are parity bits. 1st bit is a parity bit for the sequence 3, 5, 7… 2k+1, then 2nd bit checks the parity of 2, 3, 6, 7… 2k, 2k+1, where k is odd. For other bits, the pattern grows.

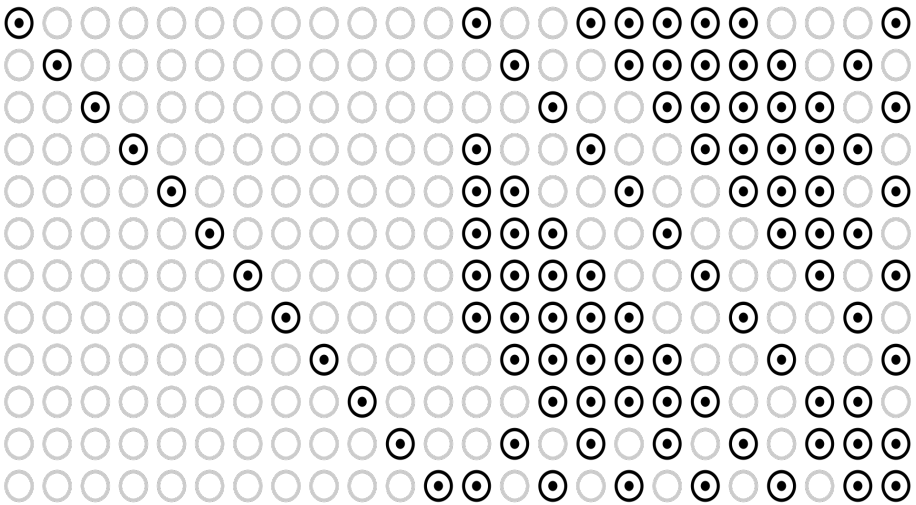
An interesting trick is to do interleaving of hamming codes, making it possible to detect and correct multiple bit flips per block.

See: https://www.youtube.com/watch?v=b3NxrZOu_CE

Error detection and error correcting codes

Hamming code works well only in settings where the probability of an error is small, and the errors are independent of each other. Enter Golay codes, namely the Extended Golay Code G24, or [24, 12, 8] code, which is capable of correcting up to 3 errors, and detecting up to 7.

G24 can encode a 12 bit message into a 24 bit one. It multiplies a generator matrix G with the original message V. G is made by horizontal concatenation of a 12 dimensional identity matrix with a special matrix.



Error detection and error correcting codes

Error detection, or feedback error control, aims to detect errors and ask for retransmission. For this purpose checksums are used. Recall simple parity check - it is a feedback error control mechanism.

There is of course more to it. CRCs, or cyclic redundancy checks are some of the most used methods, because of the speed and reliability of the method. CRC uses special polynomials, represented as bits, something like: **x2 - 1** will be equivalent with 101. CRCs also use modulo-2 arithmetic (lots of XORs).

The idea is to have a known, generator polynomial G(x) and the data D(x) to be sent also represent a polynomial. Then, to the data a sequence of x zeroes is appended, and divided to G(x). The receiver will check the remainder, and if it is not 0 (or sometimes 1) it will known an error in transmission happened.

Error detection and error correcting codes

```
// Pseudocode for CRC 32
def crc32_encode(data: Bytes) -> UInt32:
  crc32 = 0xFFFFFFFF
  for byte in data:
    nLookupIndex = (crc32 xor byte) and 0xFF
    crc32 = (crc32 shr 8) xor CRCTable[nLookupIndex]
    //CRCTable is an array of 256 32-bit constants, for faster compute

  crc32 = crc32 xor 0xFFFFFFFF
  return crc32
```

Error detection and error correcting codes

CRC is good at detecting all 1 and 2 bit errors, but also at detecting bursts of errors that are less than the length of the given polynomial, for example for the popular CRC-32, the maximum error burst that it can detect is 32 bits.

Because of its binary nature, it can be implemented efficiently in hardware, and it is, in fact.

Error detection and error correcting codes

Keep in mind, sometimes retransmission is a better option that trying to correct the errors. For example when errors occur in bursts, decoding/correcting the altered message is more time consuming then retransmission or when the error rate is low.

On the other hand, error correction is still of use, for example when broadcasting information, when it is stored on some device rather than propagated via some network or when the error rate is high.

Also, keep in mind that you should favor one solution over the other based on the medium via which messages are sent, the types of errors that happen and other criteria outlined above. Don't try to use Hamming code when you know you'll have burst errors, and don't try to fix bit flips when you mainly get reordered data.

Text vs Binary representations

By now it should be known that the messages sent through the network can be either binary or in text. But why do we have both?

Biggest pro for using text representation is that it is human readable, and therefore easier to debug. Besides, a message in text format should in principle be more portable.

On the other hand, binary is more compact, generally easier to parse. Therefore more efficient. But it is much harder to debug and might have some compatibility issues.

The wide adoption of text formatted messages despite being less efficient should be a hint on how important readability is in software engineering.

Serialization? Marshaling?

Some say these are different, some dare to disagree, still the issue is the following.

Sending some local data structure (list, dict, class, struct), with or without references, over the network. This means transforming it into some “transferable” format, like bytes, or JSON, or something else, text or binary.

Special care should be taking in regards to the references. Why? Because a reference on one computer won’t make sense on another. Same address, different city situation.

That usually means materializing (substituting) the reference with the actual value.

Serialization? Marshaling?

Serialization is a big topic in and of itself, with issues like backward compatibility, forward compatibility, dealing with references, even being able to make different runtimes communicate, if our format is cross-language/cross-platform.

Also, there are major security threats related to object serialization/deserialization.

We don’t have time for these topics, regretfully.

See: http://erights.org/data/serial/jhu-paper/intro.html
also check: M. Kleppman, Designing Data-Intensive Applications, “Formats for Encoding Data” at p. 112

Byte ordering. Endianness.

Say you have an int32 number represented in hexadecimal: **0x00C0FFEE** (that's 12648430).
How will you store it in memory?

Byte ordering. Endianness.

The most obvious way would be **[00, c0, ff, ee]**, that is with the least significant bit (LSB) on the right. In other words, the most significant byte (MSB) is stored first and the least significant last.

But there's another way!
[ee, ff, c0, 00] - LSB stored first and MSB last.

These two ways of ordering the bytes are called **big** and **little endian** respectively.

The reason little-endianness even exists is because it is sometimes easier to design hardware, compared to big-endian order.

Byte ordering. Endianness.

The reason little-endianness even exists is because it is sometimes easier to design hardware, compared to big-endian order.

Check this C code:

```
union { uint8_t u8; uint16_t u16; uint32_t u32; uint64_t u64; } u = { .u64 = 0x4A };
puts(u.u8 == u.u16 && u.u8 == u.u32 && u.u8 == u.u64 ? "true" : "false");
```

Normally, in a little-endian system the output should be `"true"`.
Such tricks are mainly used by programmers who write hardware drivers.

Byte ordering. Endianness.

Now, why should you care?

Today, however, big-endianness is the dominant ordering in networking protocols (IP, TCP, UDP). Conversely, little-endianness is the dominant ordering for processor architectures (x86, most ARM implementations) and their associated memory. There are some architecture that are bi-endian, like ARM Arch64 and Power ISA by IBM.

Internet uses big-endian order for historical reasons, and because the “backbone” of the Internet is almost impossible to update entirely, we have to live with this.

The translation between endianness is done when reading data or sending it: **hton** (host2net) and **ntoh** (net2host).

https://betterexplained.com/articles/understanding-big-and-little-endian-byte-order/

Reading list

- http://web.mit.edu/6.005/www/fa15/classes/21-sockets-networking/
- http://www.kegel.com/c10k.html
- https://daniel.haxx.se/docs/poll-vs-select.html
- https://thetechsolo.wordpress.com/2016/02/29/scalable-io-events-vs-multithreading-based/
- http://ecomputernotes.com/computernetworkingnotes/communication-networks/what-is-congestion-control-describe-the-congestion-control-algorithm-commonly-used
- https://www.queryhome.com/tech/34483/sctp-handshake-vs-tcp-handshake
- http://archive.cone.informatik.uni-freiburg.de/teaching/vorlesung/protocol-design-s09/slides/04-Protocol_Specification_1.pdf
- https://people.cs.aau.dk/~ask/Undervisning/MVP/html/mvp10a.pdf
- “Design and Validation of Computer Protocols” - Gerard J. Holzmann

Keywords (Good to know)

VPN, QUIC and HTTP/3, Kerberos, SSH
RDMA, Head-of-Line blocking, TLS handshakes
Reed-Solomon codes, Viterbi algorithm and convolution error correcting codes

Annex A: Cigarette smokers problem

Assume a cigarette requires 3 ingredients to make, tobacco, matches and paper. There are 3 people at a table, each with an infinite amount of a single resource. There's also a non-smoker who non-deterministically chooses 2 of the resources to put on the table. The 3rd smoker takes the provided resources, makes a cigarette and smokes for a while.

For every resource there's a semaphore, signaling when a resource is available. Smokers have a semaphore too, to signal to the non-smoker that they're done.

However, this can lead to deadlock; if the agent places paper and tobacco on the table, the smoker with tobacco may remove the paper, leaving the smoker with matches unable to make their cigarette. The problem is to define additional processes and semaphores that prevent deadlock, without modifying the agent.

Annex A: Cigarette smokers problem

When initially proposed, in a paper researching the limits of semaphores, the problem had 2 constraints.

1. One can not modify the smoker's code, only add
2. One can not use conditionals

The problem is impossible to solve using just semaphores, but using an array of them, and some controlling process makes it feasible.

David Parnas later argued that the second constraint doesn't make sense.* Someone complies while someone else changes the game. Actually, the argument made sense, because the problem is meant to model operating system use cases, and not using conditional statements when they are available is plain stupid.

*See: https://kilthub.cmu.edu/articles/On_a_solution_to_the_cigarette_smokers_problem/6607826

Annex B: Sleeping barber problem

Assume a barber, a chair and 2 rooms, one where the customers are waiting, and another one where they sit in that chair and ~~their throats are slit~~ the barber cuts their hair.

If there are no clients, barber sleeps, if there are no seats in the waiting room, the client leaves.

Clients check on the barber and if he sleeps, they wake him up and wait for the haircut.

Based on a naïve analysis, the above decisions should ensure that the shop functions correctly, with the barber cutting the hair of anyone who arrives until there are no more customers, and then sleeping until the next customer arrives. In practice, there are a number of problems that can occur that are illustrative of general scheduling problems.

Annex B: Sleeping barber problem

The problems are all related to the fact that the actions by both the barber and the customer (checking the waiting room, entering the shop, taking a waiting room chair, etc.) all take an unknown amount of time.

For example, a customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While they're on their way, the barber finishes their current haircut and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to their chair and sleeps. The barber is now waiting for a customer, but the customer is waiting for the barber.

In another example, two customers may arrive at the same time when there happens to be a single seat in the waiting room. They observe that the barber is cutting hair, go to the waiting room, and both attempt to occupy the single chair.

Annex B: Sleeping barber problem

The problem is atributed to Djikstra. So, you might have figured out semaphores are involved.

One of the solutions involves 3 semaphores, one for the client, one for the barber and one for the waiting room. The idea is to restrict simultaneous state transitions of entities.

```
def barber_proc():
  while true:
    custReady_sem.wait()
    accessWRSeats_sem.wait()
    numberOfFreeWRSeats += 1
    barberReady_sem.signal()
    accessWRSeats_sem.signal()
    # start cutting here
```

```
def customers_proc():
  while true:
    accessWRSeats.wait()
    if numberOfFreeWRSeats > 0:
      numberOfFreeWRSeats -= 1
      custReady.signal()
      accessWRSeats.signal()
      barberReady.wait()
      # Have hair cut here
    else:
      accessWRSeats.signal()
      # (Leave without a haircut.)
```

Annex C: Emitter-Receiver problem

Assume an emitter that has to send N messages to N receivers, and must not send another message before the previous one was received by all receivers.

So, a trivial solution would be to have some way of blocking and signaling for receivers.

```
# emiter
broadcast_channel.send()
receiver.signal(n_receiver)

# receiver
receiver.wait()
broadcast_channel.receive()
```

Annex C: Emitter-Receiver problem

The first solution is far from good. I has no way of making the emitter wait for all the receivers to receive the message before sending a new one. This solution solves that.

```
# emiter
broadcast_channel.send()
receiver.signal(n_receiver)
emitter.wait()
```

```
# receiver
receiver.wait()
broadcast_channel.receive()
mutex.wait()
cur_receivers += 1
if cur_receivers == n_receivers:
    emitter.signal()
mutex.signal()
```

Annex C: Emitter-Receiver problem

```
# emiter
emitter.wait()
broadcast_channel.send()
receiver.signal(n_receiver)
```

```
# receiver
all_receivers.wait()
mutex.wait()
cur_receivers -= 1
if cur_receivers == 0:
    emitter.signal()
else:
    all_receivers.signal()
mutex.signal()
receiver.wait()
broadcast_channel.receive()
mutex.wait()
cur_receivers += 1
if cur_receivers == n_receivers:
    all_receivers.signal()
mutex.signal()
```