Rust Trainings All in One

- High-level intro about Rust
- Ownership, borrow check, and lifetime
- Typesystem and Generic Programming
- Concurrency - primitives
- Concurrency - async/await
- Networking and security
- FFI with C/Elixir/Swift/Java
- WASM/WASI
- Rust for real-world problems

High-level Intro About Rust

Why Rust?

Let's talk about values and tradeoffs first

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- Performance
- Portability
- Productivity
- Resilliency
- Rigor
- Safety
- Security
- Simplicity
- Stability
- Thoroughness
- Transparent
- Velocity

C

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- **Performance**
- Portability
- Productivity
- Resilliency
- Rigor
- Safety
- Security
- **Simplicity**
- Stability
- Thoroughness
- **Transparent**
- Velocity

Erlang/Elixir

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- Performance
- Portability
- **Productivity**
- **Resilliency**
- Rigor
- **Safety**
- Security
- **Simplicity**
- Stability
- Thoroughness
- Transparent
- Velocity

Python

- **Approachability**
- Availability
- Compatibility
- Composability
- Debuggability
- **Expressiveness**
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- Performance
- Portability
- **Productivity**
- Resilliency
- Rigor
- Safety
- Security
- **Simplicity**
- Stability
- Thoroughness
- Transparent
- Velocity

Java (in early days)

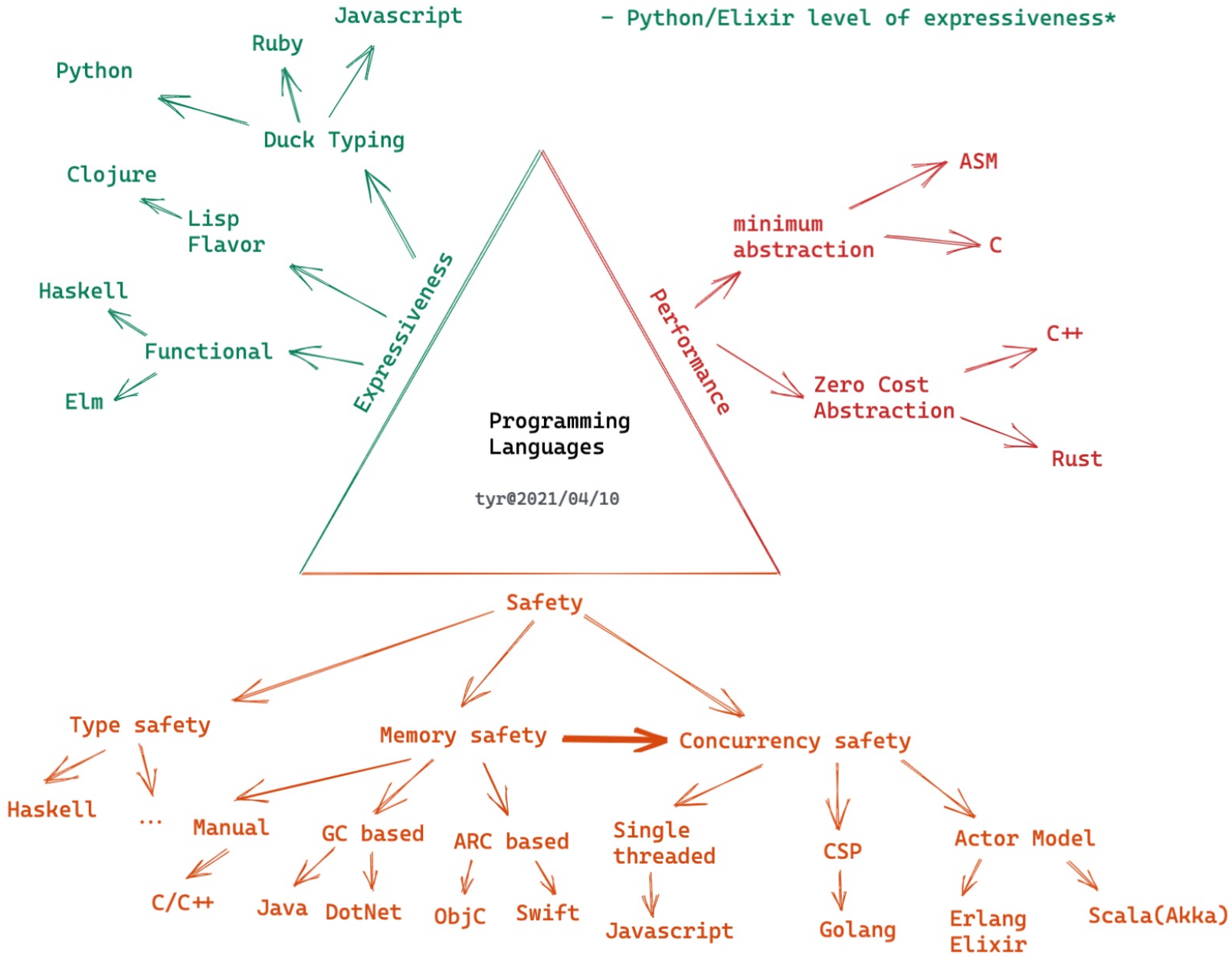
- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- **Performance**
- **Portability**
- Productivity
- Resilliency
- Rigor
- **Safety (memory)**
- **Security**
- Simplicity
- Stability
- Thoroughness
- Transparent
- Velocity

Rust

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- **Expressiveness**
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- **Performance**
- Portability
- **Productivity**
- Resilliency
- Rigor
- **Safety!!!**
- Security
- Simplicity
- Stability
- Thoroughness
- Transparent
- Velocity

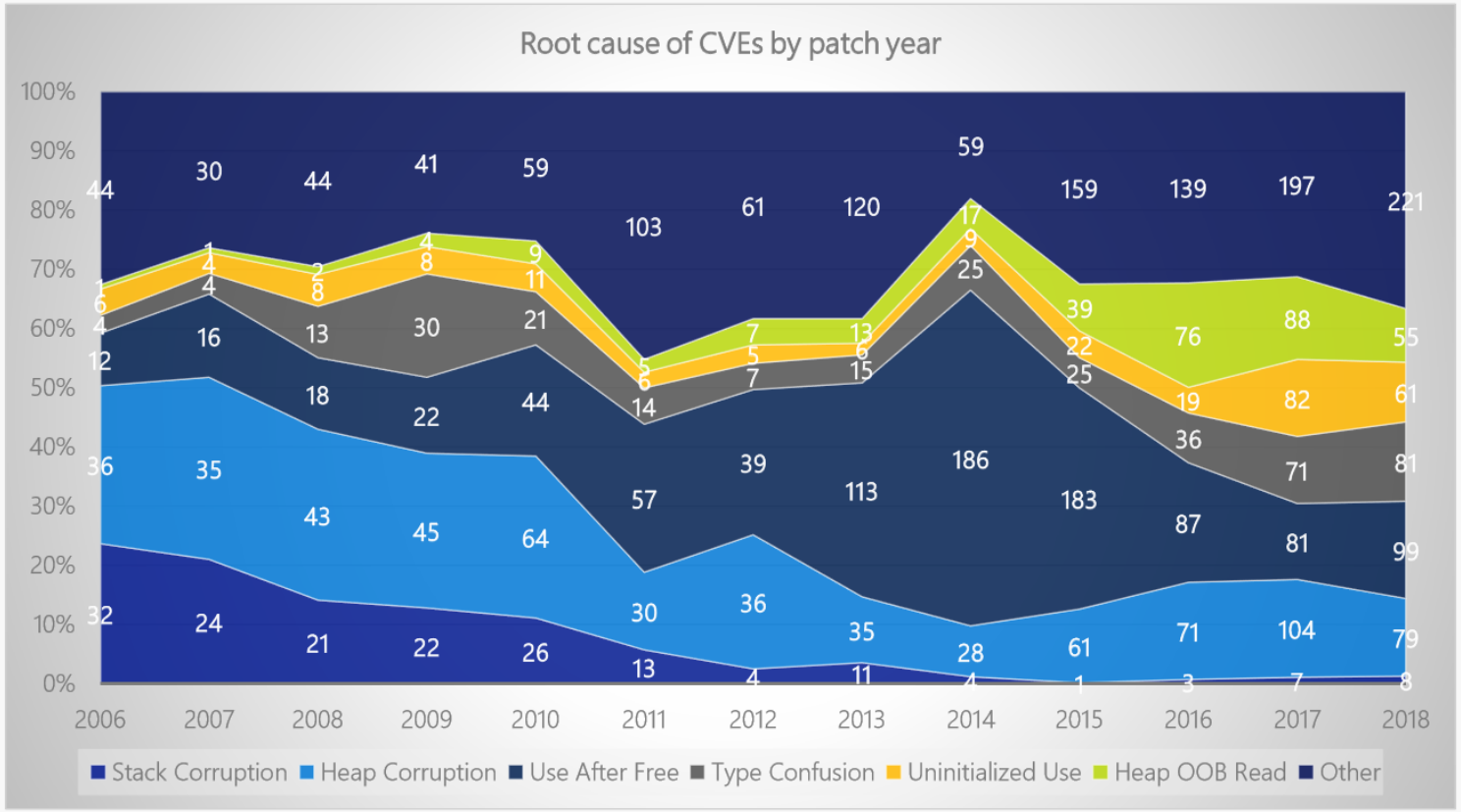
Rust:

- Type, Memory and Concurrency safety
- C/C++ level performance
- Python/Elixir level of expressiveness*



Why safety is important?

Drilling down into root causes



Stack corruptions are essentially dead

Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

Top root causes since 2016:

- #1: heap out-of-bounds
- #2: use after free
- #3: type confusion
- #4: uninitialized use

Note: CVEs may have multiple root causes, so they can be counted in multiple categories

(Source: Memory Safety Issues Are Still the Leading Source of Security Vulnerabilities)

Safety is hard!

- memory safety is not easy (you need to understand the corner cases)
- conccurency safety is really hard (without certain tradeoffs)
- Often you have to bear the extra layer of abstractions
  - normally it means performance hit

Memory safety

- Manually - C/C++: painful and error-prone
- Smart Pointers - C++/ObjC/Swift: be aware of cyclical references
- GC - Java/DotNet/Erlang: mubch bigger memory consumption, and STW
- Ownership - Rust: learning curve

Concurrency safety

- single-threaded - Javascript: cannot leverage multicore
- GIL - Python/Ruby: multithreading is notorious inefficient
- Actor model - Erlang/Akka: at the cost of memory copy and heap allocation
- CSP - Golang: at the cost of memory copy and heap allocation
- Ownership + Type System - Rust: super **elegant** and **no extra cost**!

How Rust achieves

memory and conccurency safety

without extra cost?

tyr@04/10/2021

value in a scope

⑤

i32, usize, [T;4]*
copy on move

Box, Arc, MutexGuard, RwLockGuard

Allow a type to do dereference or mutable deferences

Captured value is passed by reference unless explicitly `move`

Deref/DerefMut

Closure (Fn/FnMut)

Copy Trait

Exception: Rc

Type could be moved between threads safely

Send Trait

Type Safety

Sync Trait

Exception: Rc, Cell, RefCell

Type could be shared between threads safely

①

When passing a value to other scope, move occurs.

Can have only one owner in a scope. When owner left scope, value dropped.

```
{
    let arr = vec![1];
    let arr1 = vec![2];
    sum(arr);
    println!("arr: {:?}", arr);
}
```

I'm the owner

new owner: sum

arr1 dropped

②

can have multiple immutable references

```
{
  let arr = vec![1];
  let total = sum(&arr);
  let v = max(&arr);
}
```

I'm the owner

1st borrower: sum

2nd borrower: max

③

can have and only have one mutable reference exclusively

```
{
  let arr = vec![1];
  let ir = &arr;
  push(&mut arr);
  println!("ir: {:?}", ir);
}
```

immutable ref

I'm the owner

mutable ref

④

reference cannot outlive owner

```
{
  let arr = vec![1];

  thread::spawn(|| {
    println!("{:?}", arr);
  });
}
```

closure may outlive

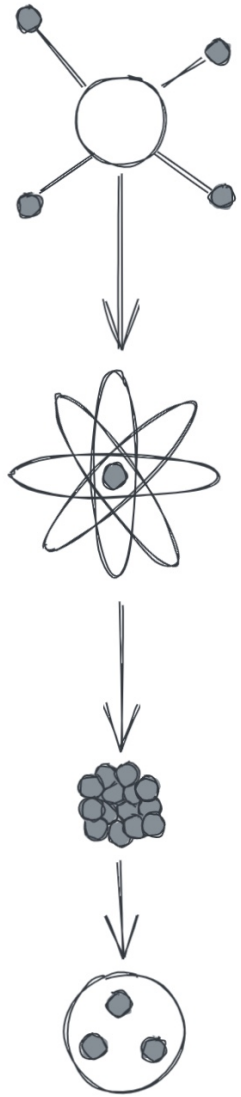
reference here

Show me the code!

```rust
fn main() {
    let mut arr: Vec<i32> = vec![1, 2, 3];    move occurs because `arr` has type `Vec<i32>`, which does not implement the `Copy` trait
    arr.push(4);

    let _result: Result<(), Error> = process(arr);    value moved here
    let _v: Option<i32> = arr.pop(); // failed since arr is moved    borrow of moved value: `arr`

    // you can have multiple immutable references
    let mut arr1: Vec<i32> = vec![1, 2, 3];
    let ir1: &Vec<i32> = &arr1;
    let ir2: &Vec<i32> = &arr1;    immutable borrow occurs here

    println!("ir1: {:?} ir2: {:?}", ir1, ir2);

    // but you can't have both mutable and immutable references
    let mr1: &mut Vec<i32> = &mut arr1;    cannot borrow `arr1` as mutable because it is also borrowed as immutable
    // let mr2 = &mut arr1;

    println!("mr1: {:?} mr2: {:?}", mr1, ir2);    immutable borrow later used here

    // by default, closure borrows the data
    let mut arr2: Vec<i32> = vec![1, 2, 3];
    thread::spawn(|| {    closure may outlive the current function, but it borrows `arr2`, which is owned by the current function
        arr2.push(4);    `arr2` is borrowed here
    });

    // we shall move the data explicitly
    let mut arr3: Vec<i32> = vec![1, 2, 3];
    thread::spawn(move || arr3.push(4));
}

fn thread_safety() {
    // but certain types cannot be moved to other thread safely
    let mut rc1: Rc<Vec<i32>> = Rc::new(vec![1, 2, 3]);
    thread::spawn(move || {    `Rc<Vec<i32>>` cannot be sent between threads safely
        rc1.push(4);
    });
}
```

```rust
fn thread_safety_reasoning() {
    let mut map: HashMap<&str, &str> = HashMap::new();    move occurs because `map` has type `HashMap<&str, &str>`, which does not implem
    map.insert(k: "hello", v: "world");

    // Arc is an atomic reference counter which can be moved safely across threads
    let mut ir: Arc<HashMap<&str, &str>> = Arc::new(data: map);    variable does not need to be mutable
    map.insert(k: "hello1", v: "world1"); // you can't do this since map is moved    borrow of moved value: `map`
    let ir1: Arc<HashMap<&str, &str>> = ir.clone(); // this is cheap, just reference counter clone
    thread::spawn(move || assert_eq!(ir1.get("hello"), Some(&"world")));
    // but arc is immutable, so this would fail
    thread::spawn(move || ir.insert(k: "hello2", v: "world2"));    cannot borrow data in an `Arc` as mutable

    // the compiler guide you to use types that provides mutable reference for threads

    // use Mutex - you can't clone a Mutex, thus you can't make it availabe for multiple threads
    let mut map1: HashMap<&str, &str> = HashMap::new();
    map1.insert(k: "hello", v: "world");
    let mr: Mutex<HashMap<&str, &str>> = Mutex::new(map1);
    let mr1 = mr.clone();    no method named `clone` found for struct `Mutex<HashMap<&str, &str>>` in the current scope
    thread::spawn(move || mr.lock().unwrap().insert(k: "hello1", v: "world1"));
    mr1.lock().unwrap().insert("hello2", "world2");

    // use Mutex with Arc - now you have mutable access and multi-thread cloning
    let mut map2: HashMap<&str, &str> = HashMap::new();
    map2.insert(k: "hello", v: "world");
    let mr: Arc<Mutex<HashMap<&str, &str>>> = Arc::new(data: Mutex::new(map2));
    let mr1: Arc<Mutex<HashMap<&str, &str>>> = mr.clone();

    thread::spawn(move || mr.lock().unwrap().insert(k: "hello1", v: "world1"));
    thread::spawn(move || mr1.lock().unwrap().insert(k: "hello2", v: "world2"));

    // can I use Box (smart pointer for heap allocation)?
    let mut map1: HashMap<&str, &str> = HashMap::new();
    map1.insert(k: "hello", v: "world");
    let mr: Arc<Box<HashMap<&str, &str>>> = Arc::new(data: Box::new(map1));
    let mr1: Arc<Box<HashMap<&str, &str>>> = mr.clone();
    thread::spawn(move || (**mr).insert(k: "hello1", v: "world1"));    cannot borrow data in an `Arc` as mutable
    mr1.insert(k: "hello2", v: "world2");    cannot borrow data in an `Arc` as mutable
}
```

First Principles Thinking



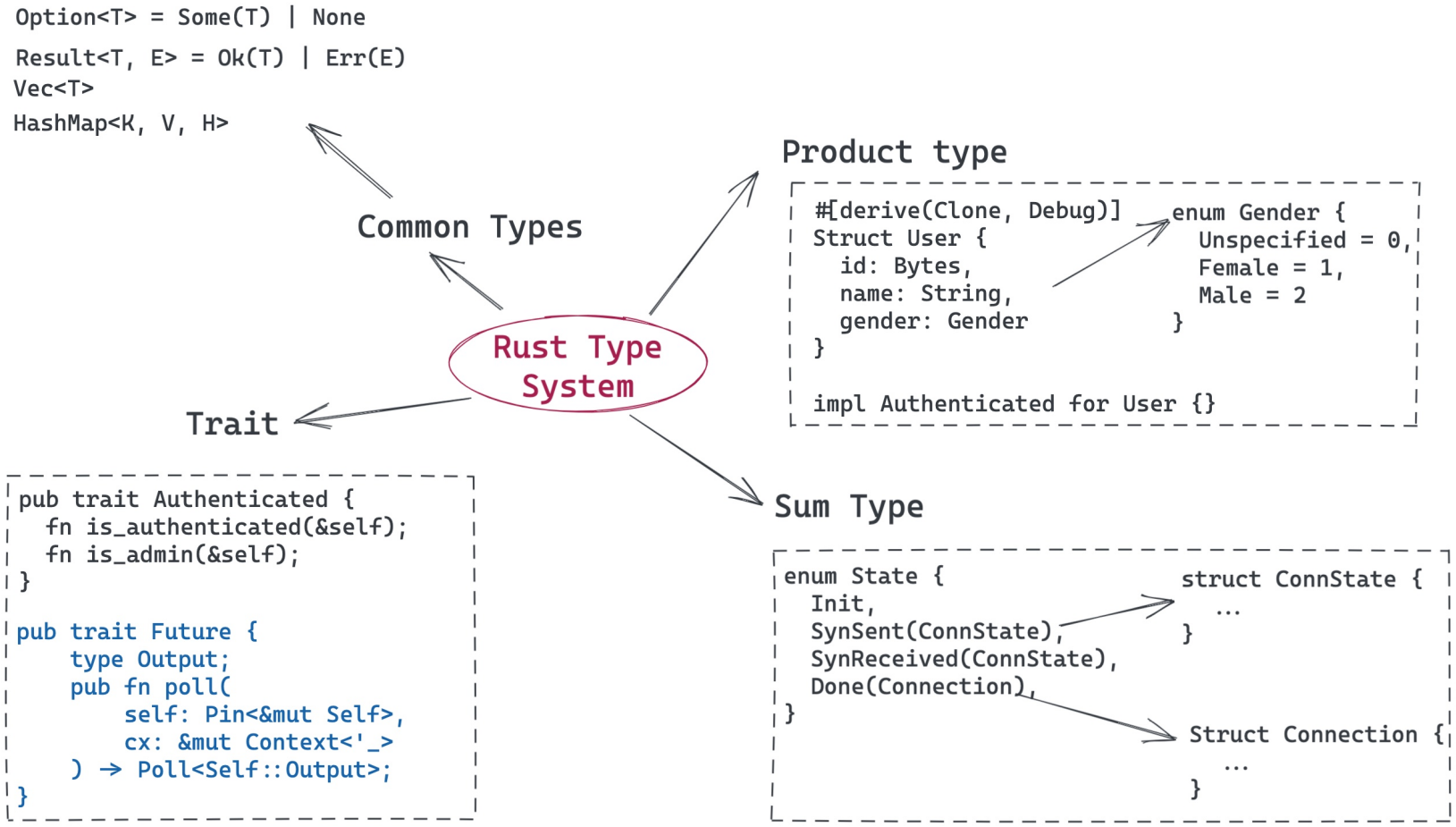
Boiling problems down to their most fundamental truth.

Recap

- One and only one owner
- Multiple immutable references
- mutable reference is mutual exclsive
- Reference cannot outlive owner
- **use type safety for thread safety**

With these simple rules, Rust achieved safety with zero cost abstraction

A glance at Rust Type System



How's Productivity of Rust?

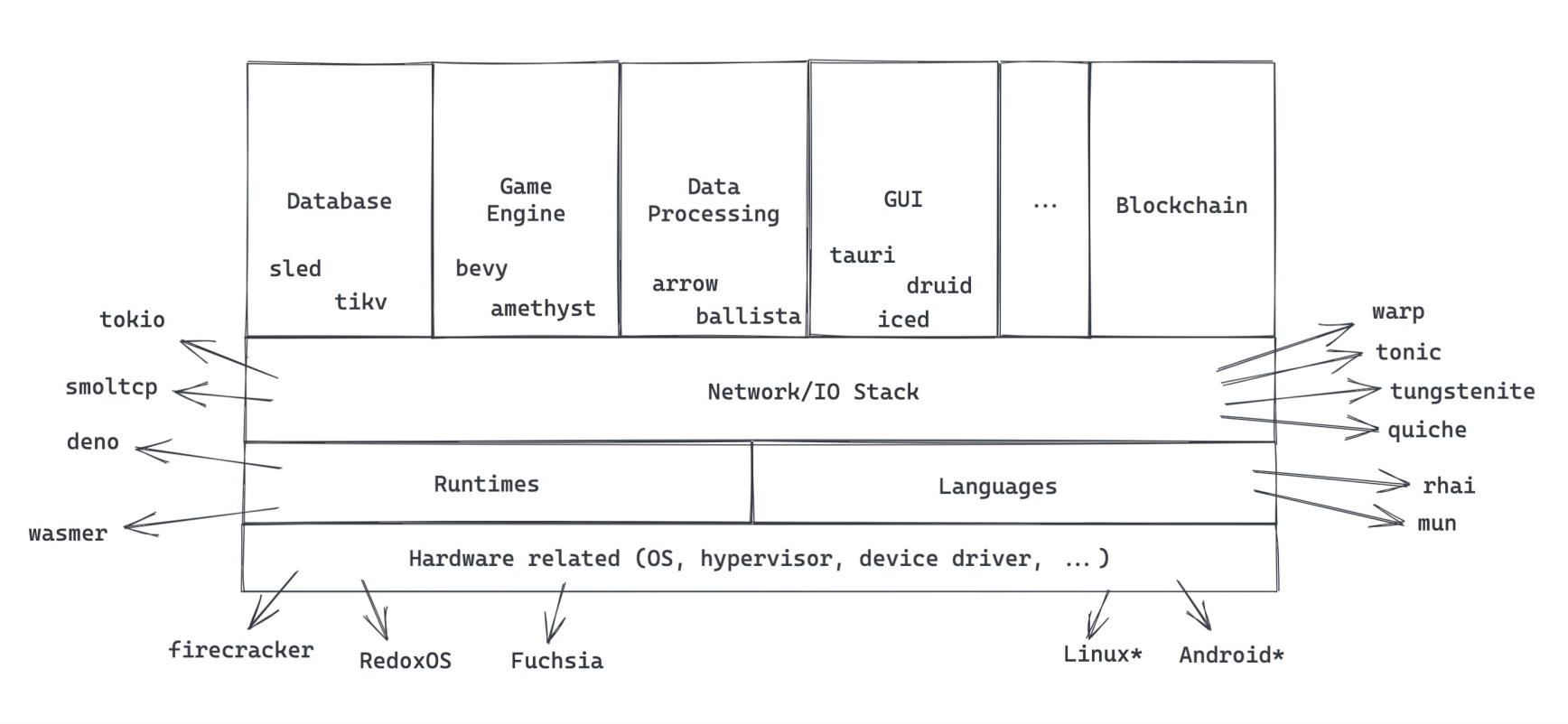
```
# client configuration

domain = "localhost"

[cert]
pem = """-----BEGIN CERTIFICATE-----
MIIBeTCCASugAwIBAgIBKjAFBgMrZXAwNzELMAkGA1UEBgwCVVMxFDASBgNVBAoM
C0RvbWFpbiBJbmMuMRIwEAYDVQQDDAlEb21haW4gQ0EwHhcNMjEwMzE0MTg0NTU2
WhcNMzEwMzEyMTg0NTU2WjA3MQswCQYDVQQGDAJVUzEUMBIGA1UECgwLRG9tYWlu
IEluYy4xEjAQBgNVBAMMCURvbWFpbiBDQTAqMAUGAytlcAMhAAzhorM9IPsXjBTx
ZxykGl5xZrsj3X2XqKjaAVutnf7po1wwWjAUBgNVHREEDTALgglsb2NhbGhvc3Qw
HQYDVR0OBBYEFD+NqChBZDOs5FMgefHJSIWiRTHXMBIGA1UdEwEB/wQIMAYBAf8C
ARAwDwYDVR0PAQH/BAUDAwcGADAFBgMrZXADQQA9sligQcYGaBqTxR1+JadSelMK
Wp35+yhVVuu4PTL18kWdU819w3cVlRe/GHt+jjlbk1i22TvfO5AaNmdxySkO
-----END CERTIFICATE-----"""

# server configuration

[identity]
key = """-----BEGIN PRIVATE KEY-----
MFMCAQEwBQYDK2VwBCIEII0kozd0PJsbNfNUS/oqI/Q/enDiLwmdw+JUnTLpR9xs
oSMDIQAtkhJiFdF9SYBIMcLikWPRIgca/Rz9ngIgd6HuG6HI3g==
-----END PRIVATE KEY-----"""

[identity.cert]
pem = """-----BEGIN CERTIFICATE-----
MIIBazCCAR2gAwIBAgIBKjAFBgMrZXAwNzELMAkGA1UEBgwCVVMxFDASBgNVBAoM
C0RvbWFpbiBJbmMuMRIwEAYDVQQDDAlEb21haW4gQ0EwHhcNMjEwMzE0MTg0NTU2
WhcNMjIwMzE0MTg0NTU2WjA5MQswCQYDVQQGDAJVUzEUMBIGA1UECgwLRG9tYWlu
IEluYy4xFDASBgNVBAMMC0dSUEMgU2VydmVyMCowBQYDK2VwAyEALZISYhXRfUmA
SDHC4pFj0SIHGv0c/Z4CIHeh7huhyN6jTDBKMBQGA1UdEQQNMAuCCWxvY2FsaG9z
dDATBgNVHSUEDDAKBggrBgEFBQcDATAMBgNVHRMEBTADAQEAMA8GA1UdDwEB/wQF
AwMH4AAwBQYDK2VwA0EAy7EOIZp73XtcqaSopqDGWU7Umi4DVvIgjmY6qbJZP0sj
ExGdaVq/7MOlZl1I+vY7G0NSZWZIUilX0CoOkrn0DA==
-----END CERTIFICATE-----"""
```

```rust
// you could also build your config with cert and identity separately. See tests.
let config: ServerTlsConfig = toml::from_str(config_file).unwrap();
let acceptor = config.tls_acceptor().unwrap();
let listener = TcpListener::bind(addr).await.unwrap();
tokio::spawn(async move {
    loop {
        let (stream, peer_addr) = listener.accept().await.unwrap();
        let stream = acceptor.accept(stream).await.unwrap();
        info!("server: Accepted client conn with TLS");

        let fut = async move {
            let (mut reader, mut writer) = split(stream);
            let n = copy(&mut reader, &mut writer).await?;
            writer.flush().await?;
            debug!("Echo: {} - {}", peer_addr, n);
        }

        tokio::spawn(async move {
            if let Err(err) = fut.await {
                error!("{:?}", err);
            }
        });
    }
});
```

Client: You, a month ago • init the project

```rust
let msg = b"Hello world\n";
let mut buf = [0; 12];

// you could also build your config with cert and identity separately. See tests.
let config: ClientTlsConfig = toml::from_str(config_file).unwrap();
let connector = config.tls_connector(Uri::from_static("localhost")).unwrap();

let stream = TcpStream::connect(addr).await.unwrap();
let mut stream = connector.connect(stream).await.unwrap();
info!("client: TLS conn established");

stream.write_all(msg).await.unwrap();

info!("client: send data");

let (mut reader, _writer) = split(stream);

reader.read_exact(buf).await.unwrap();

info!("client: read echoed data");
```

Things built with Rust



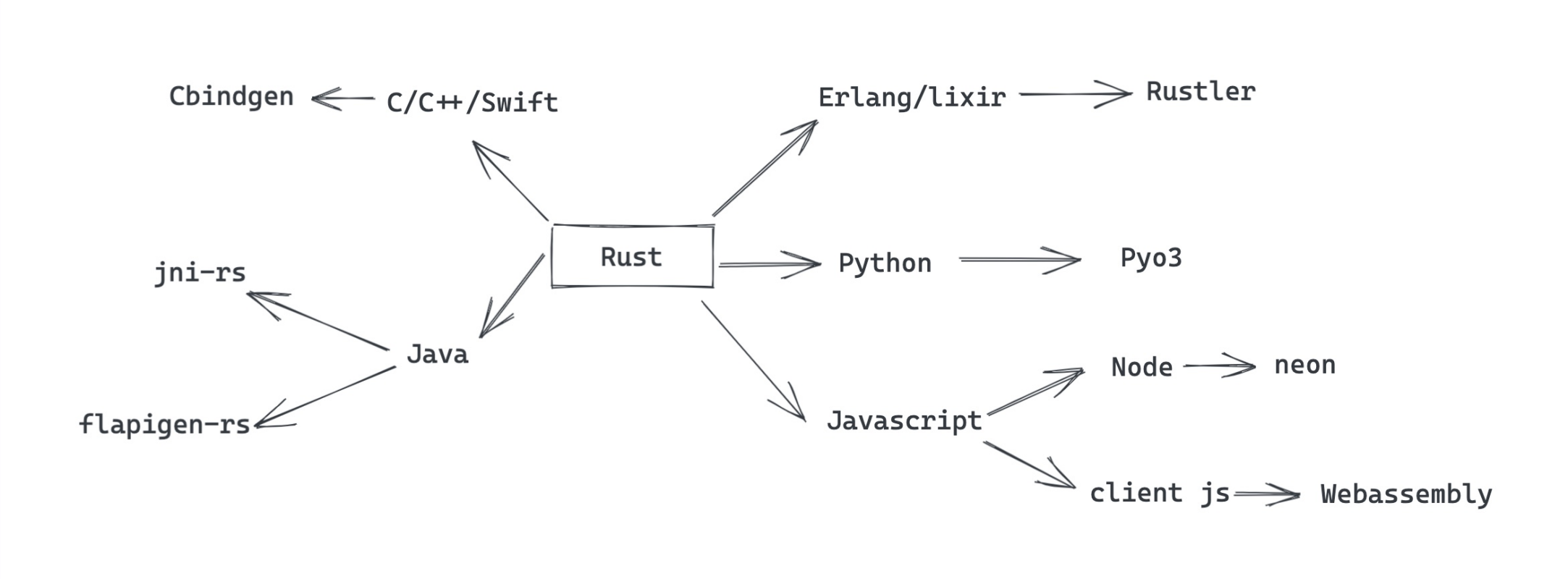
Should I use Rust?

- Rust is ideal when you need a system that reliable and performant
- Sometimes you don't, sometimes you do, sometimes you need that later
- it's all about tradeoffs

Rust for our use cases

- parts of the system that are bottlenecks
  - bottleneck on computation
  - bottleneck on memory consumption
  - bootleneck on I/O
- parser/decoder/encoder
- wants to leverage existingC/C++/Rust ecosystem (e.g. you need blake3 for hashing)

Rust FFI

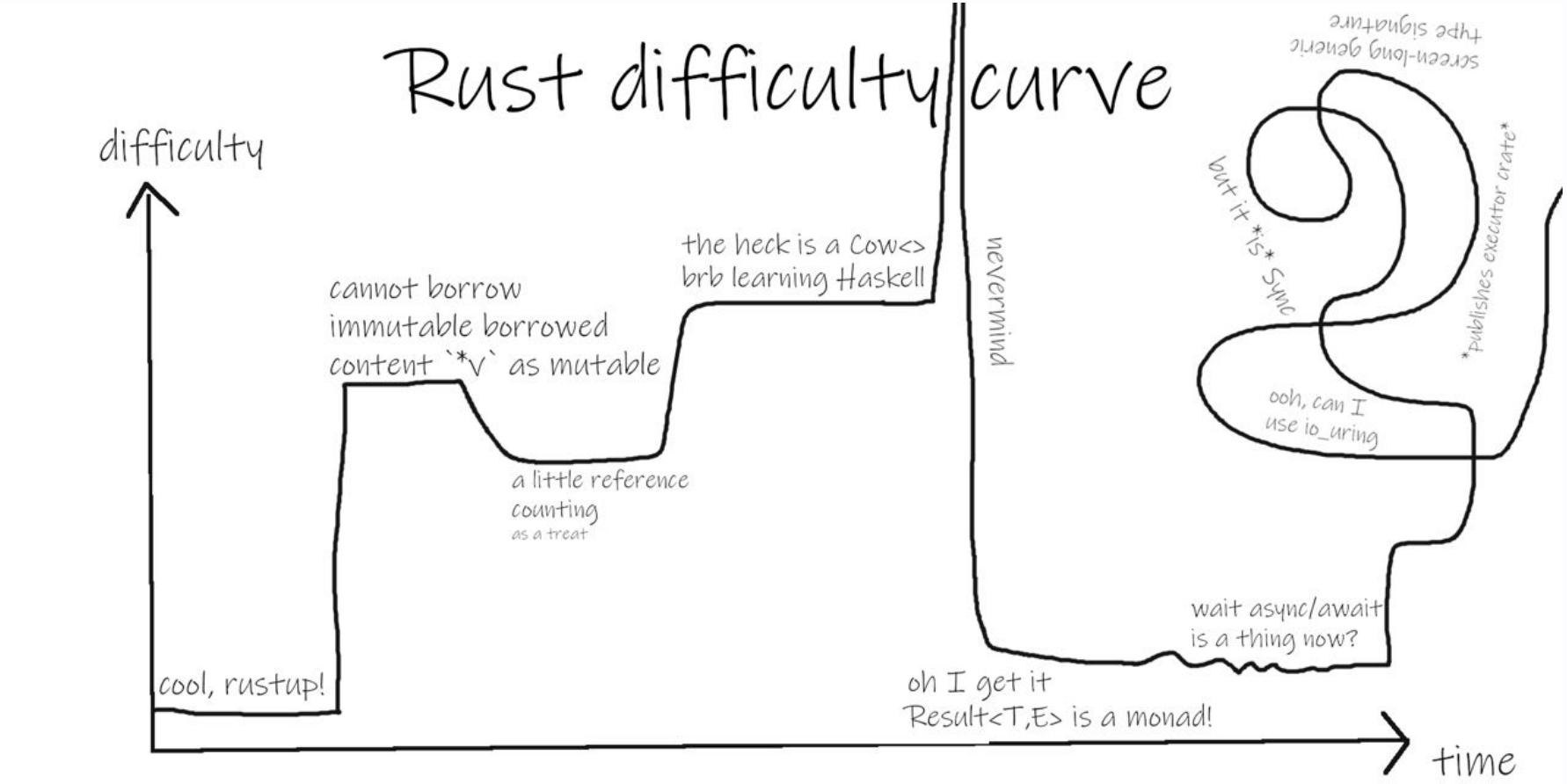

Cbindgen
C/C++/Swift
Erlang/lixir
Rustler
Rust
Python
Pyo3
jni-rs
Java
Node
neon
flapigen-rs
Javascript
client js
Webassembly

Learning rust as a(n)...

- Elixir eng: ownership model, type system, oh no mutation
- Scala eng: ownership model, oh no mutation
- Typescript eng: ownership model, multi-threaded programming
- Swift/Java eng: ownership model
- Python eng: ownership model, type system

The common misunderstandings

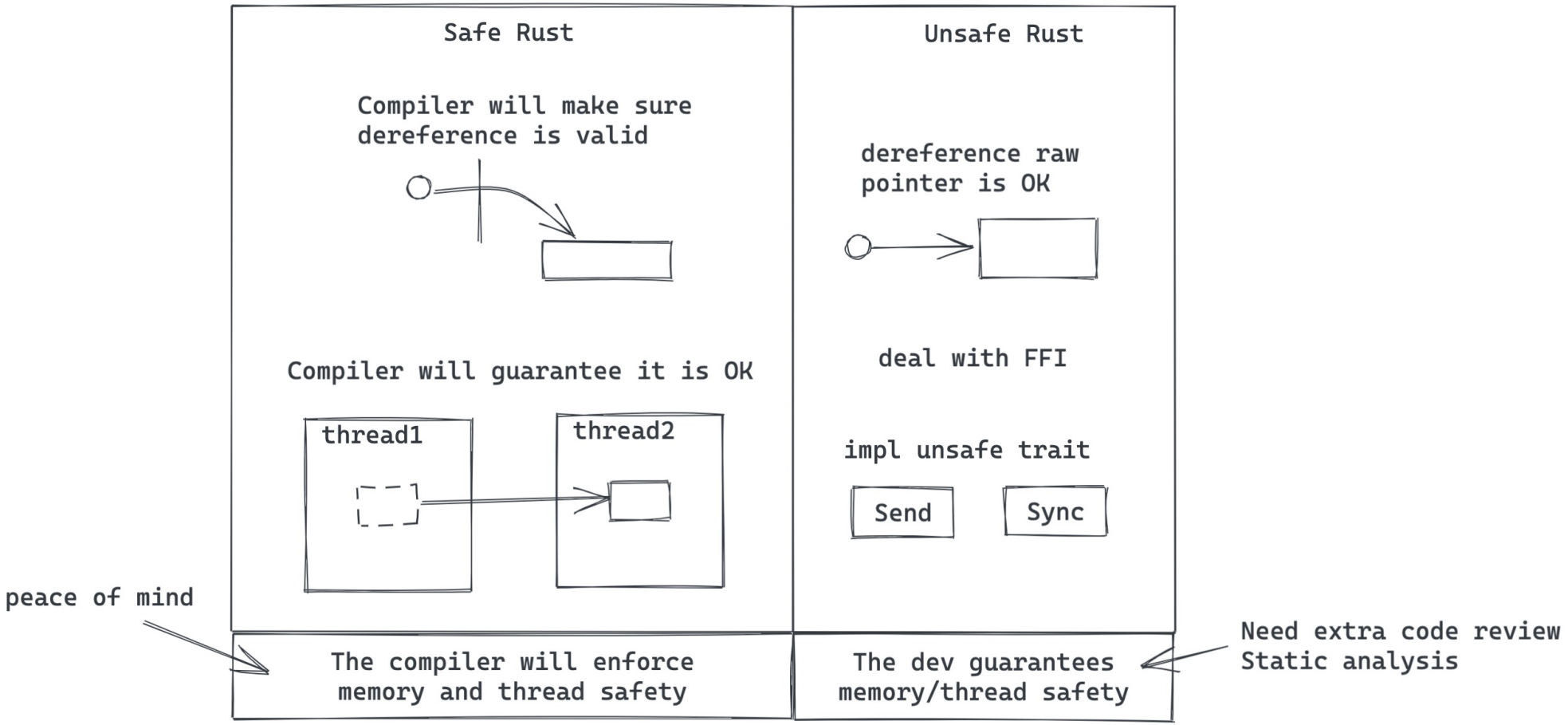
1. Rust is super hard to learn...



Rust is explicit

- Lots of knowledge about computer science is suddenly explicit to you
- If all your pain to learn a lang is 100%:
  - Rust:
    - Compiler help to reduce that to 90%
    - Then you suffer 70% the pains in first 3-6 months
    - Then the rest 20% in 3-5 years
  - Other:
    - You suffer 10-30% in first 3-6 months
    - Then 70%-90% in next 3-5 years

2. Unsafe Rust is evil...

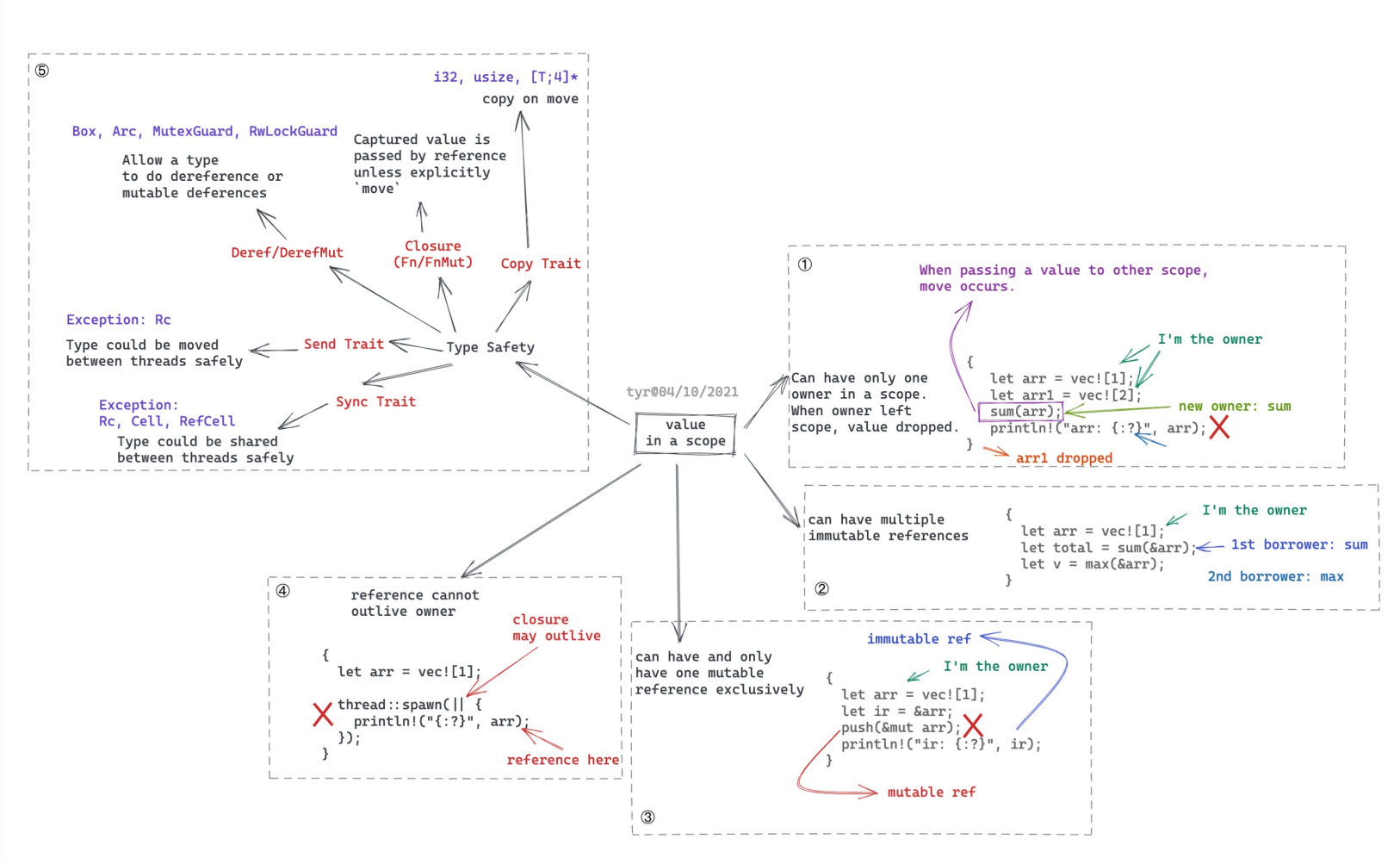


References

- The pain of real linear types in Rust
- Substructural type system
- Rust official book
- Rust official site
- Awesome Rust
- Are we web yet?
- Are we async yet?
- Are we gui yet?
- Are we learning yet?
- Are we game yet?
- Are we quantum yet?
- Are we IDE yet?
- Rust is for Professionals

Ownership, borrow check, and lifetime

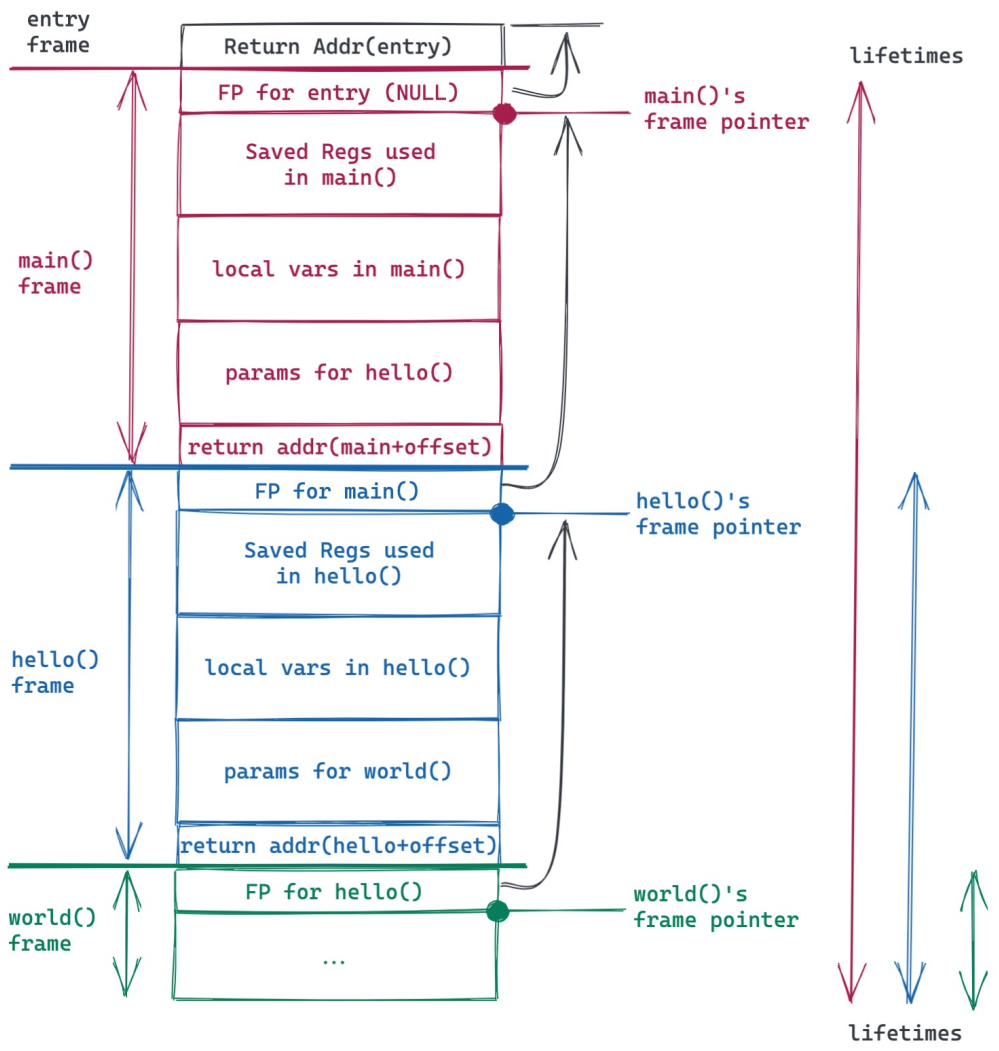
Ownership/Borrow Rules Review



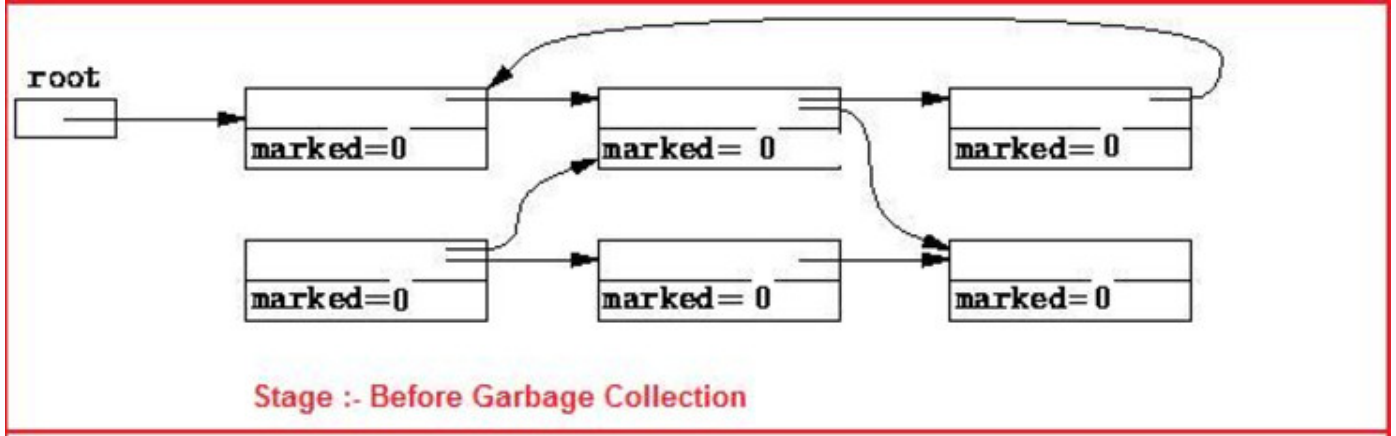
Lifetime, not a new idea

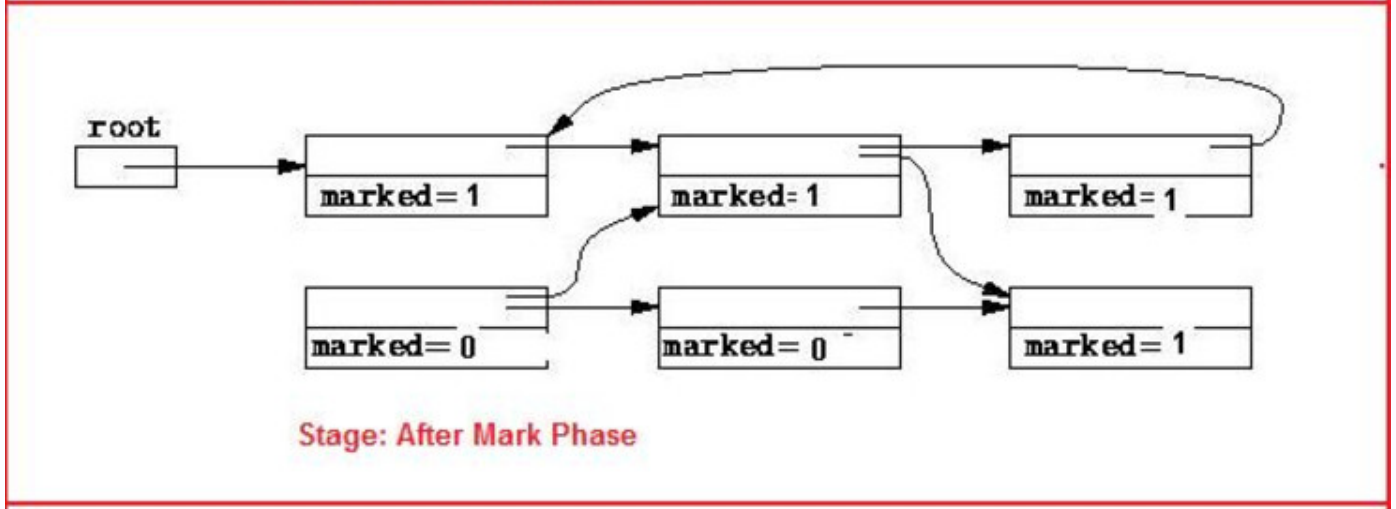
Lifetime: Stack memory

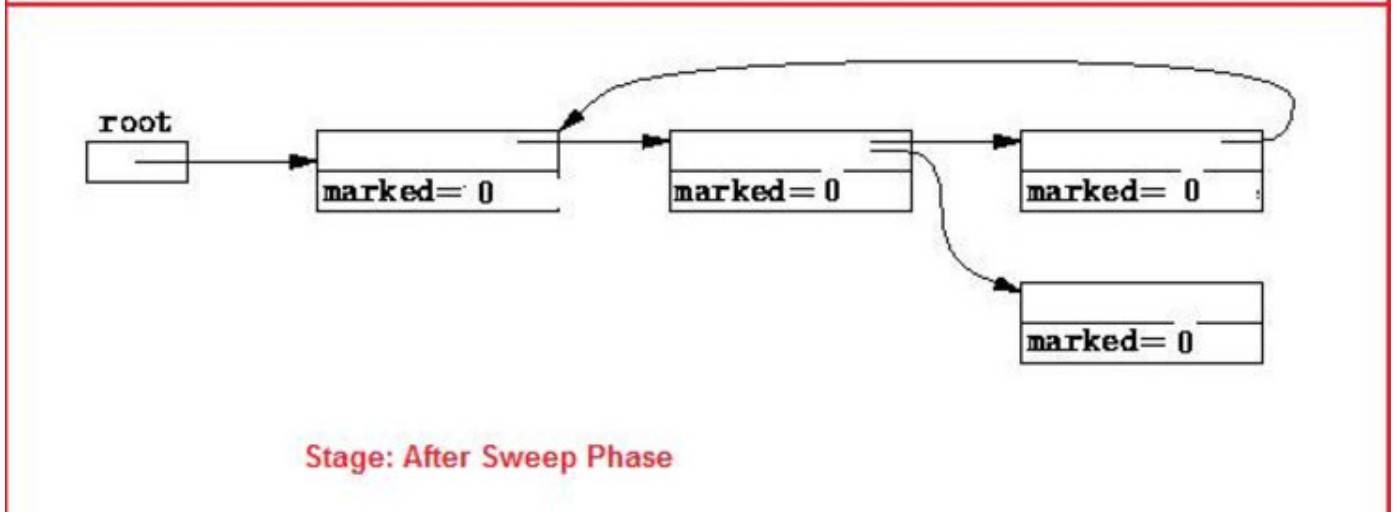
```c
#include <stdio.h>
static int VALUE = 42;

void world(char *st, int num) {
    printf("%s(%d)\n", st, num);
}


void hello(int num) {
    char *st = "hello world";
    int v = VALUE+num;
    world(st, v);
}

int main() {
    hello(2);
}
```



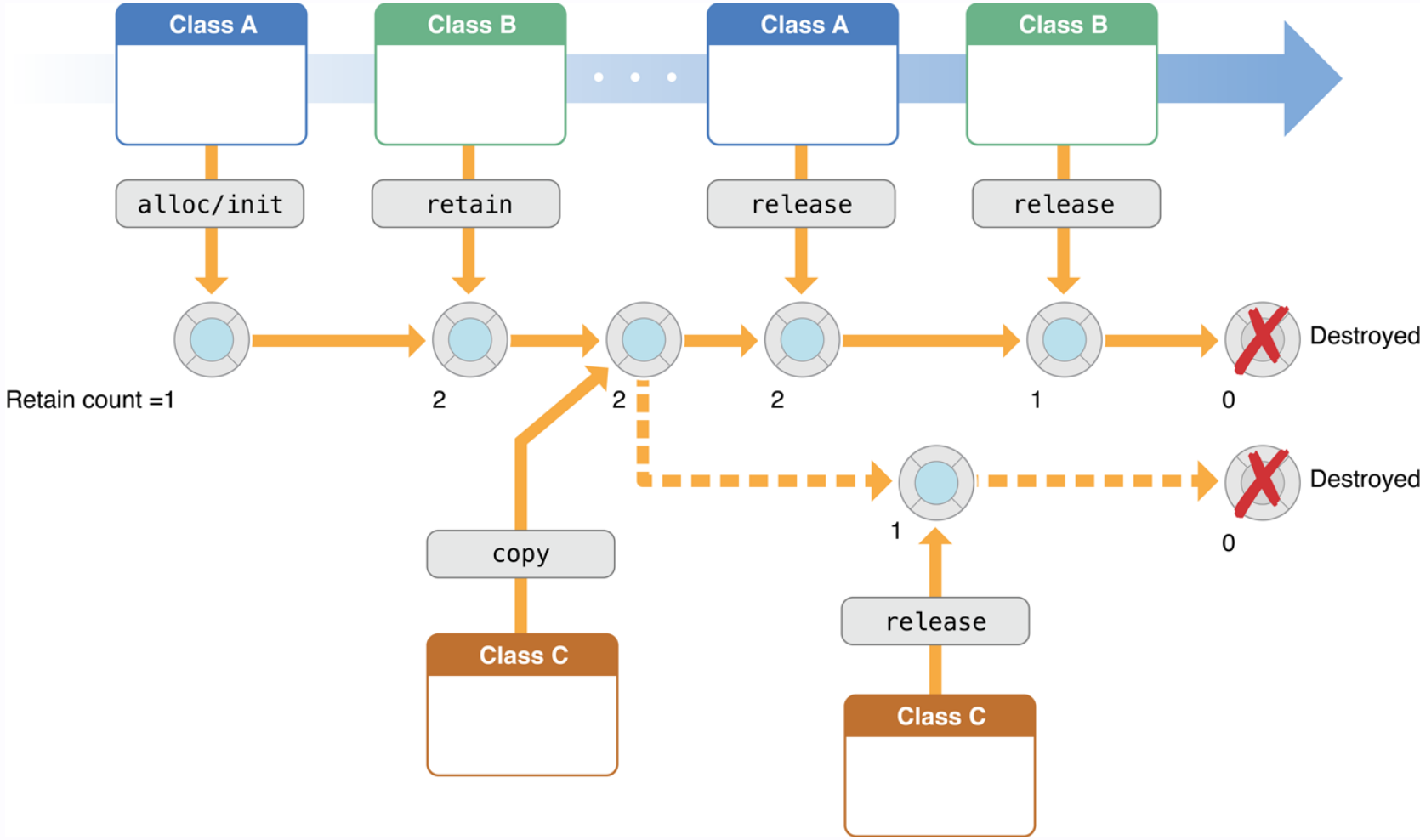
Lifetime: Heap memory (tracing GC)





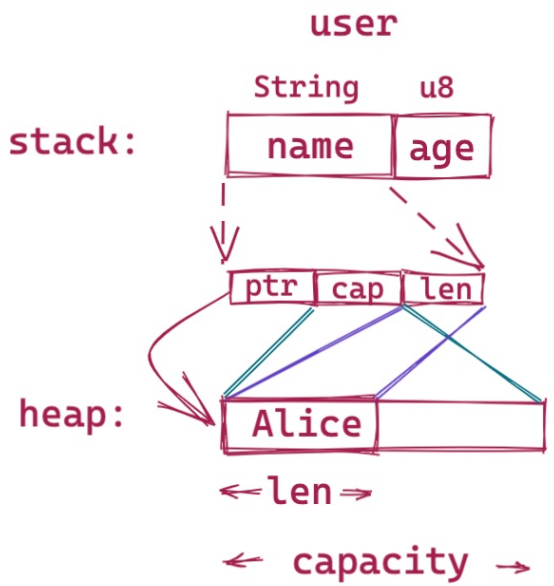


Lifetime: Heap memory (ARC)

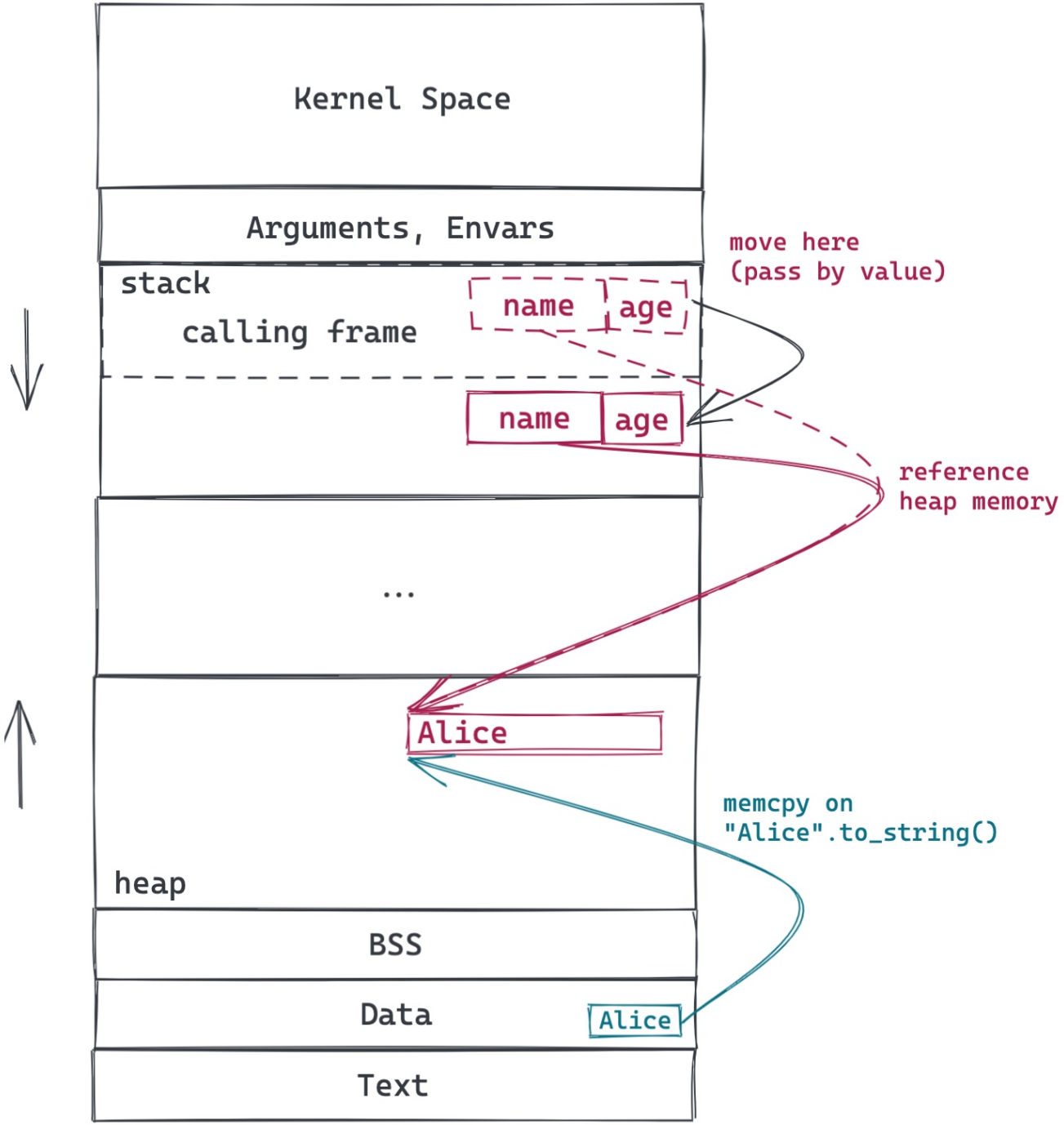


How Rust handles lifetime?

Move semantics

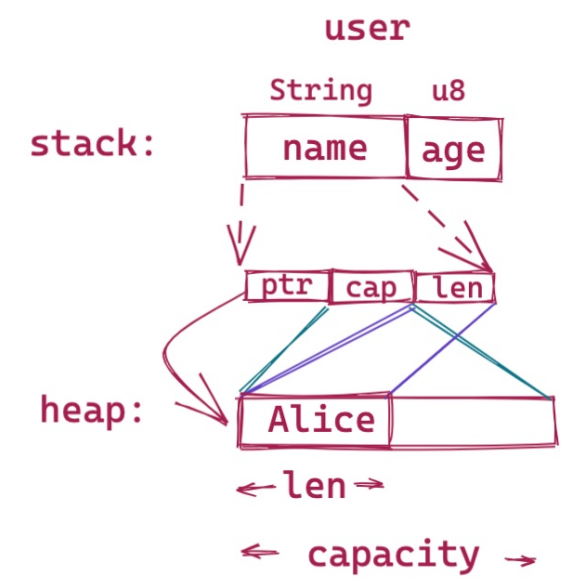



```
{
  let user = User {
    name: "Alice".to_string(),
    age: 20,
  };
  let result = insert(user);
   ...
}
```

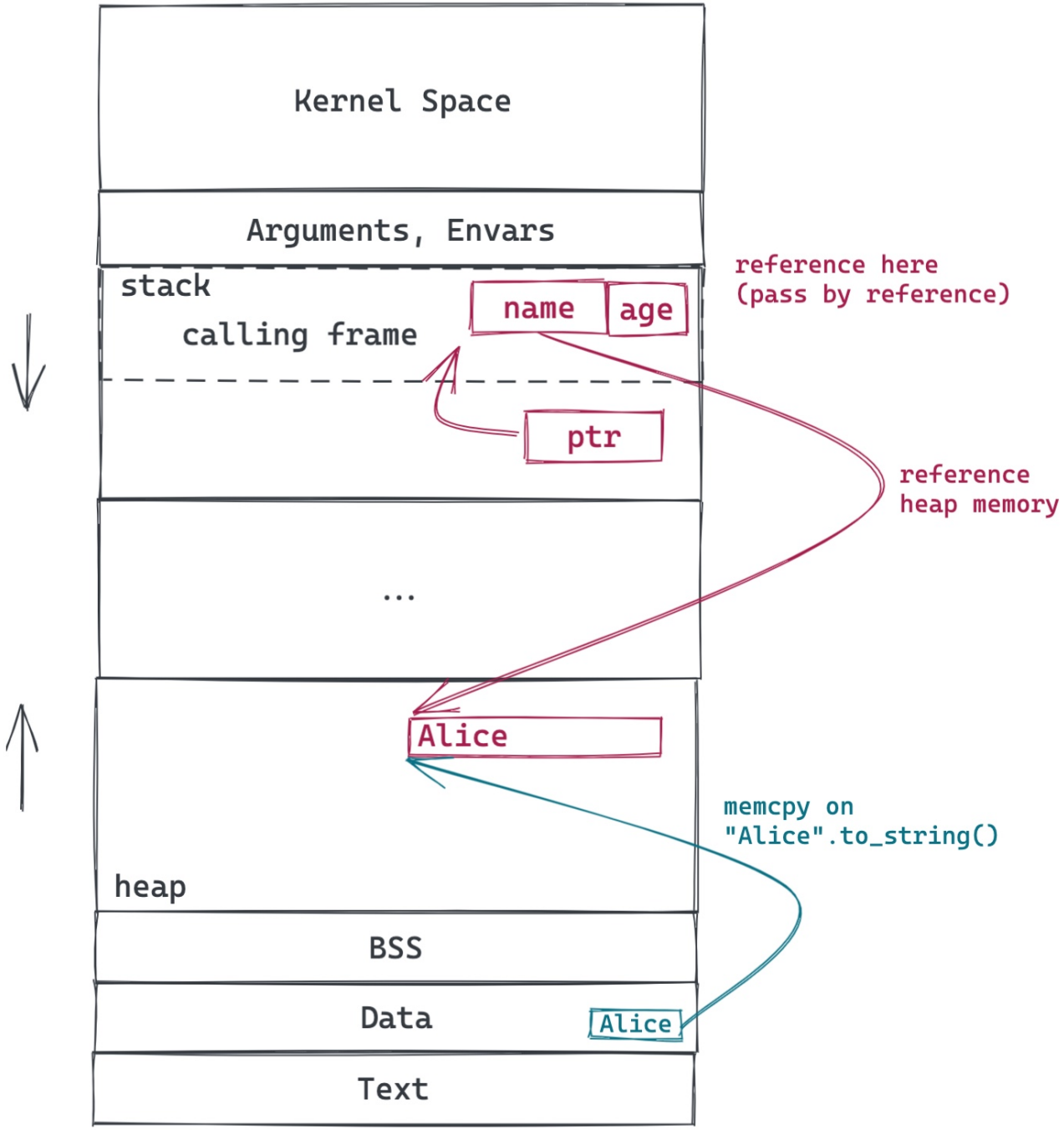


What a bout Borrow?

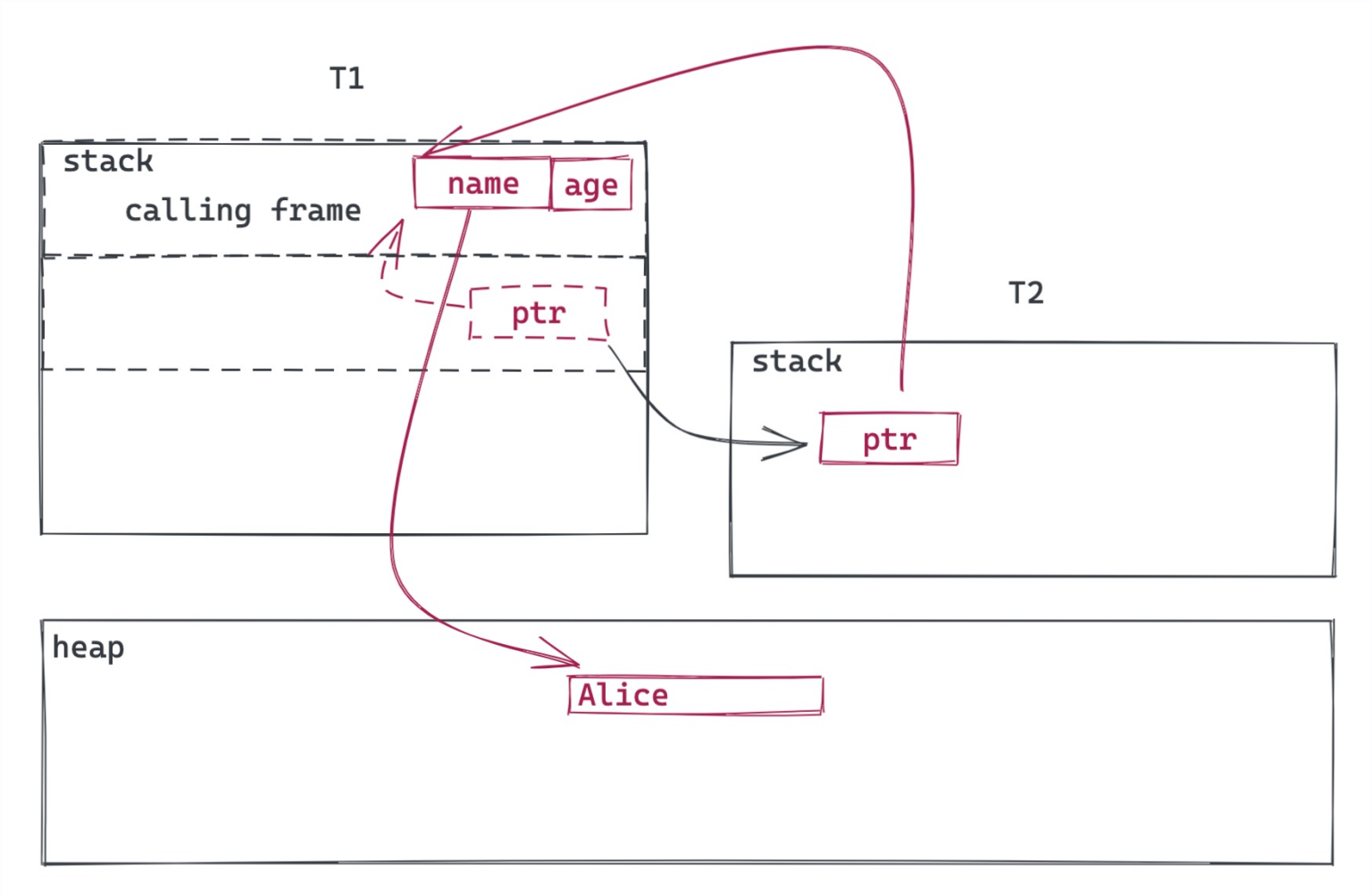
Think about: why does this work in Rust, but not C/C++?




```
{
  let user = User {
    name: "Alice".to_string(),
    age: 20,
  };
  let result = insert(&user);
   ...
}
```



Rust lifetime checker prevents this...



Benefit of lifetime-constrained borrow

- can borrow anything (stack object, heap object)
- safety can be guaranteed at compile-time (no runtime lifetime bookkeeping)
- Rust borrow checker is mostly a lifetime checker

Lifetime Annotation

- similar as generics, but in lowercase starting with `'`
- only need to put annotation when there's conflicts

```
// need explicit lifetime
struct User<'a> {
    name: &'a str,
    ...
}
fn process<T, 'a, 'b>(item1: &'a T, item2: &'b T) {}

// &'a User could be written as &User since on confliction
fn lifetime_example(user: &User) { // --+ Lifetime 'a
    if user.is_authed() {          //   |--+ Lifetime 'b
        let role = user.roles();   //   |  |
                                   //   |  |--+ Lifetime 'c
        verify(&role);             //   |  |  |
                                   //   |  |--+
    }                              //   |--+
}                                  // --+

fn verify(x: &Role) { /*...*/ }
```

Static Lifetime

- 'static
- data included in bss / data / text section
  - constants / static variables
  - string literals
  - functions
- if used as trait bound:
  - the type does not contain any non-static references
  - owned data always passes a `'static` lifetime bound, but reference to the owned data does not

Thread spawn

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
{
    Builder::new().spawn(f).expect("failed to spawn thread")
}
```

The 'static constraint is a way of saying, roughly, that no borrowed data is permitted in the closure.

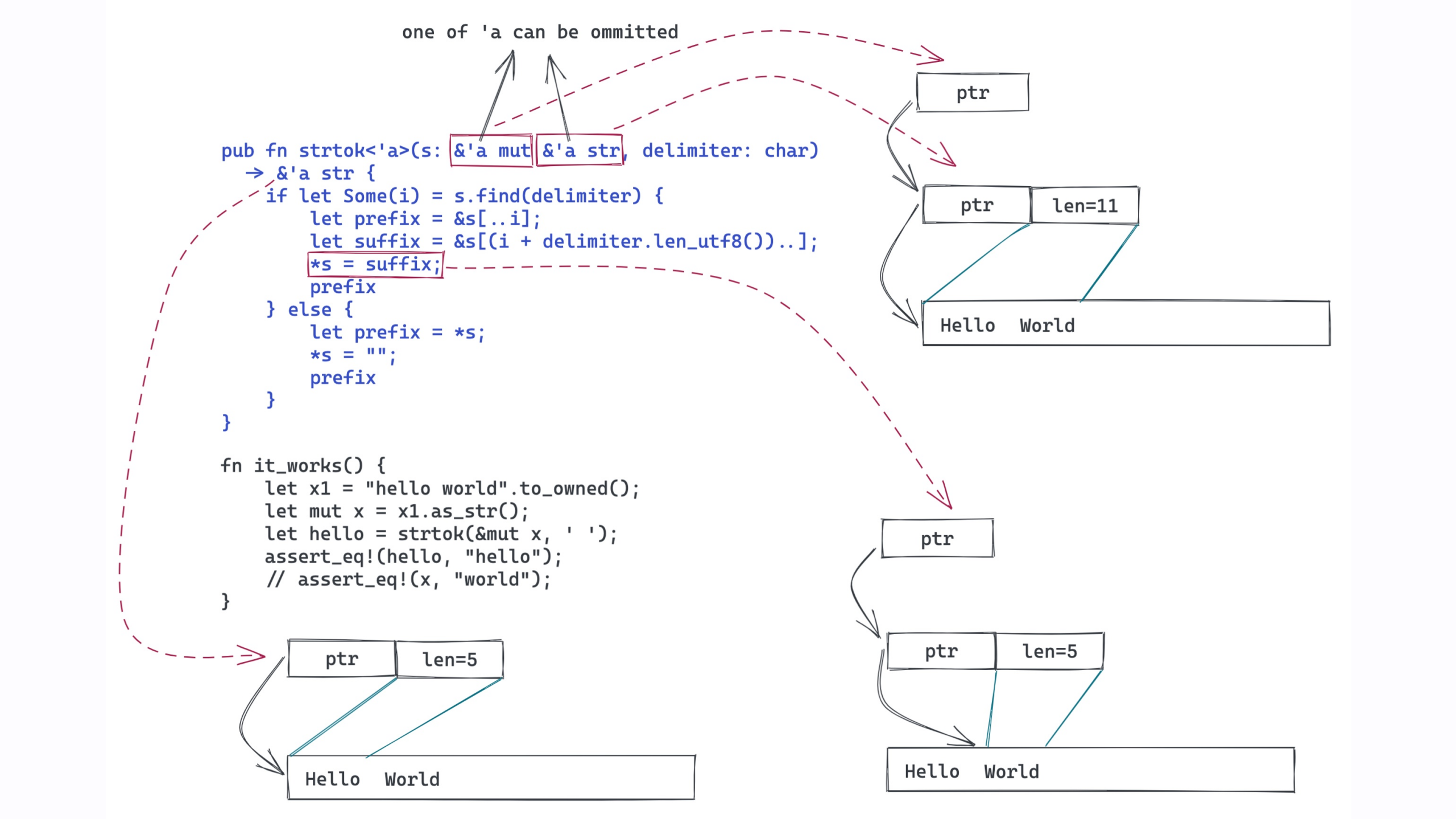
RAII (Resource Acquisition Is Initialization)

- initializing the object will also make sure resource is initialized
- releasing the object will also make sure resource is released

Drop Trait

- memory
- file
- socket
- lock
- any other OS resources

demo


one of 'a can be ommitted
pub fn strtok<'a>(s: &'a mut &'a str, delimiter: char)
-> &'a str {
if let Some(i) = s.find(delimiter) {
let prefix = &s[..i];
let suffix = &s[(i + delimiter.len_utf8())..];
*s = suffix;
prefix
} else {
let prefix = *s;
*s = "";
prefix
}
}
fn it_works() {
let x1 = "hello world".to_owned();
let mut x = x1.as_str();
let hello = strtok(&mut x, ' ');
assert_eq!(hello, "hello");
// assert_eq!(x, "world");
}
ptr
ptr
len=11
Hello World
ptr
ptr
len=5
Hello World
ptr
len=5
Hello World

Mental model

- write the code and defer the complexity about ensuring the code is safe/correct
- comfront the most of the safety/correctness problems upfront
- Mutate can only happen when you own the data, or you have a mutable reference
  - either way, the thread is guaranteed to be the only one with access at a time
- Fearless Refactoring
- reinforce properties well-behaved software exhibits
- sometimes too strict: rust isn't dogmatic about enforcing it

References

- Mark-And-Sweep (Garbage Collection Algorithm)
- Tracing garbage collection
- Swift: Avoiding Memory Leaks by Examples
- Reference counting
- Fearless concurrency with Rust
- Rust means never having to close a socket
- Programming Rust: ownership
- Crust of Rust: lifetime annotation (recommended)

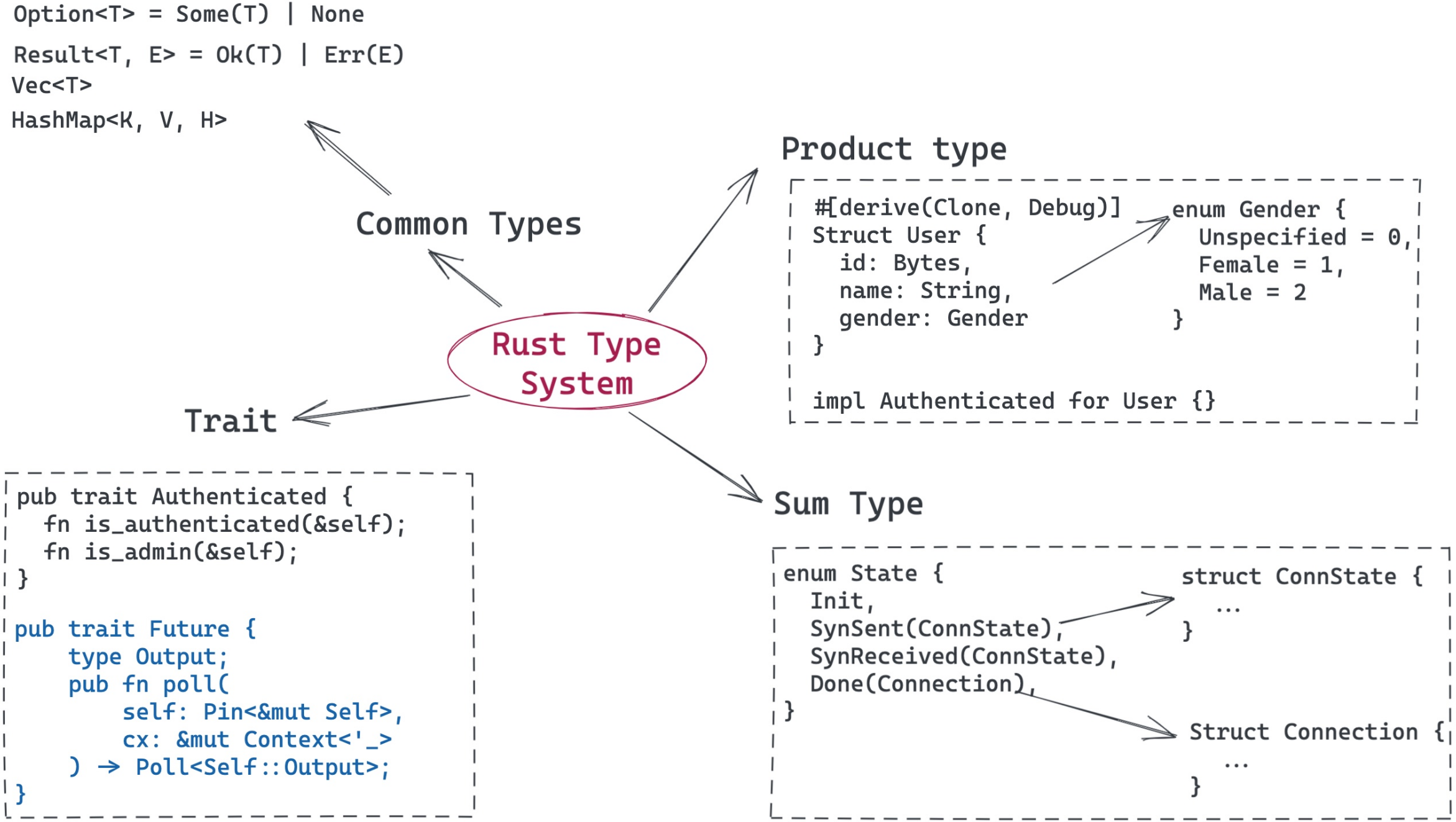
Cost of defects

- Don't introduce defect (this is impossible because humans are fallible).
- Detect and correct defect as soon as the bad key press occurs (within reason: you don't want the programmer to lose too much flow) (milliseconds later).
- At next build / test time (seconds or minutes later).
- When code is shared with others (maybe you push a branch and CI tells you something is wrong) (minutes to days later).
- During code review (minutes to days later).
- When code is integrated (e.g. merged) (minutes to days later).
- When code is deployed (minutes to days or even months later).
- When a bug is reported long after the code has been deployed (weeks to years later).

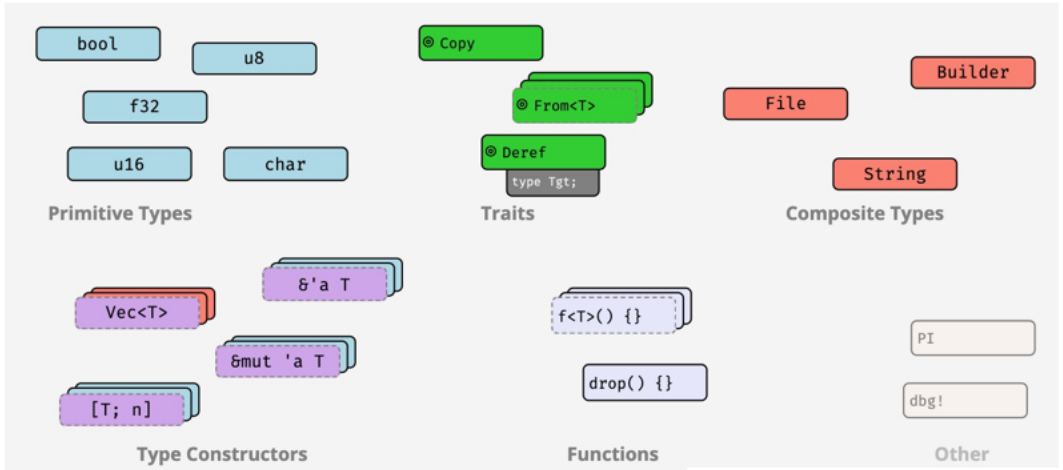
Ownership and Borrow rules

- Use after free: no more (reference can't point to dropped value)
- Buffer underruns, overflows, illegal memory access: no more (reference must be valid and point to owned value)
- memory level data races: no more (single writer or multiple readers)

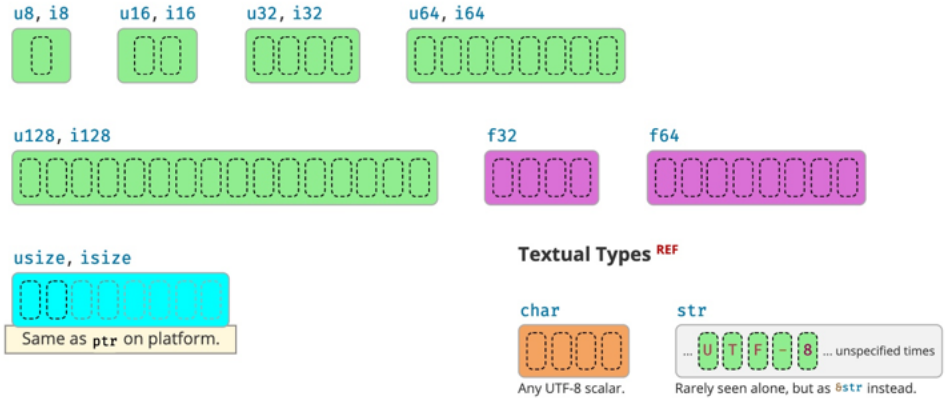
Typesystem and Generic Programming


Option<T> = Some(T) | None
Result<T, E> = Ok(T) | Err(E)
Vec<T>
HashMap<K, V, H>
Common Types
Rust Type System
Product type
#[derive(Clone, Debug)]
Struct User {
id: Bytes,
name: String,
gender: Gender
}
impl Authenticated for User {}
enum Gender {
Unspecified = 0,
Female = 1,
Male = 2
}
Trait
pub trait Authenticated {
fn is_authenticated(&self);
fn is_admin(&self);
}
pub trait Future {
type Output;
pub fn poll(
self: Pin<&mut Self>,
cx: &mut Context<'_>
) -> Poll<Self::Output>;
}
Sum Type
enum State {
Init,
SynSent(ConnState),
SynReceived(ConnState),
Done(Connection),
}
struct ConnState {
...
}
Struct Connection {
...
}

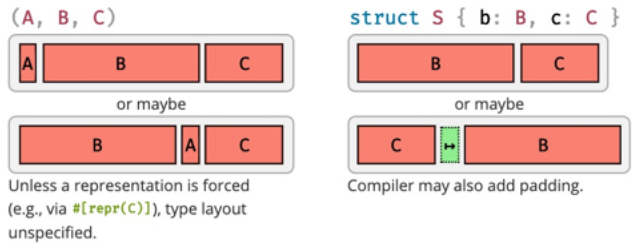
How types are layed out in memory?



Numeric Types REF

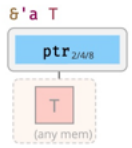


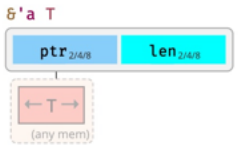
Textual Types REF

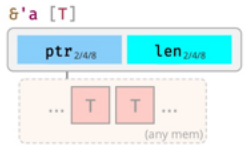


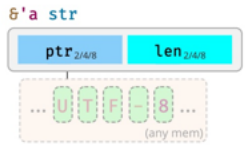
Pointer Meta

Many reference and pointer types can carry an extra field, **pointer metadata**. [STD] It can be the element- or byte-length of the target, or a pointer to a *vtable*. Pointers with meta are called **fat**, otherwise **thin**.

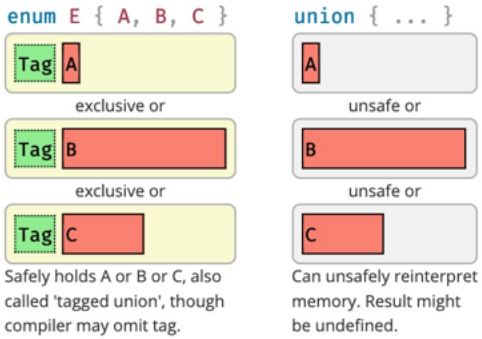


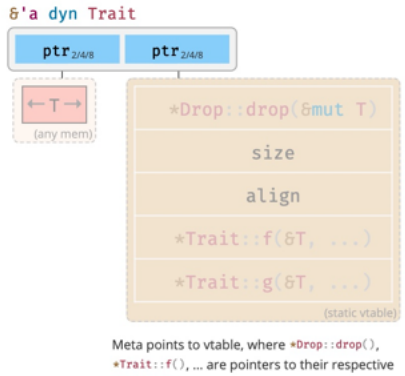






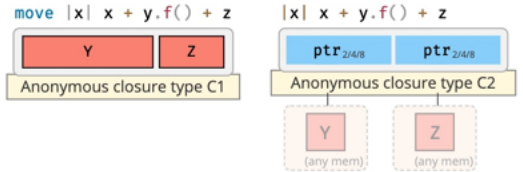
These **sum types** hold a value of one of their sub types:

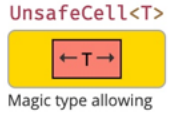


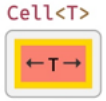


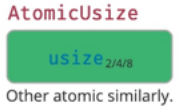
Closures

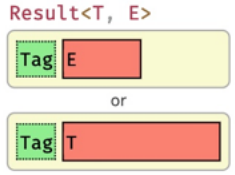
Ad-hoc functions with an automatically managed data block **capturing** [REF] environment where closure was defined. For example:

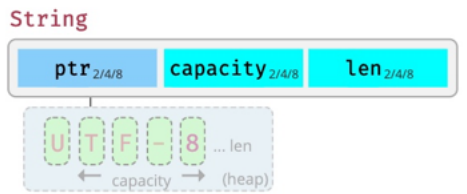


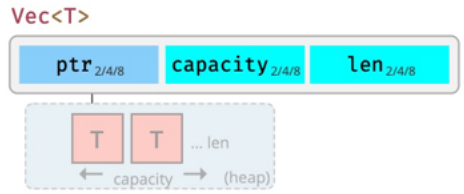












Trait (Typeclass)

refer to the actual data

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }

    ...
}
```

methods must be implemented

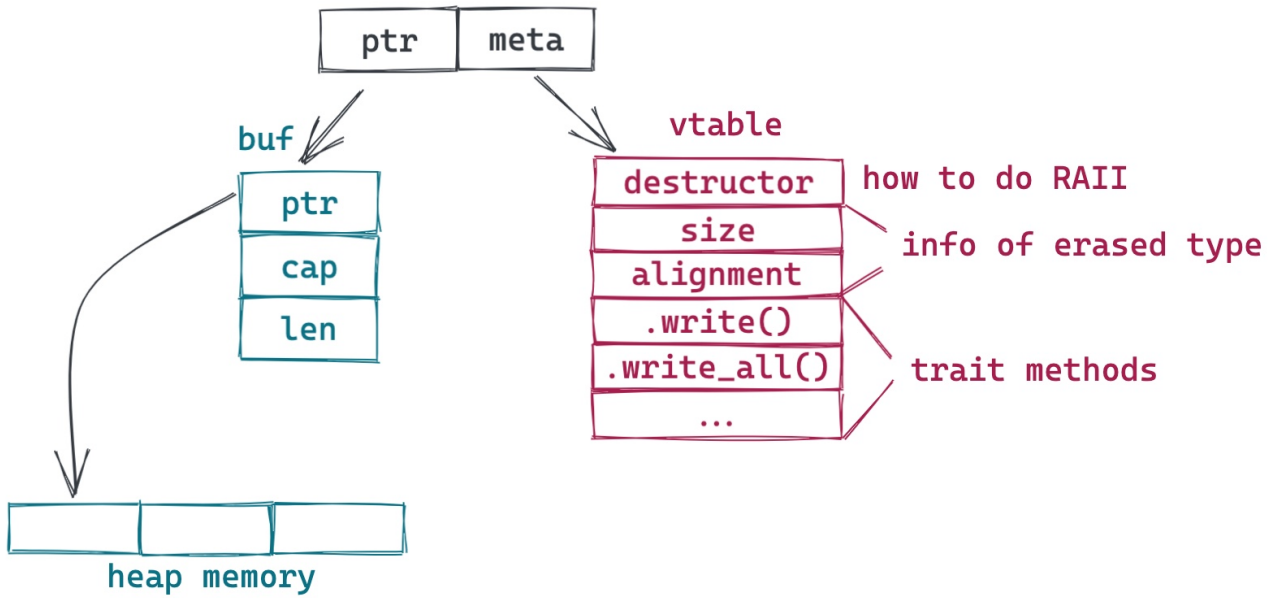
methods have a default impl

```
struct Sink {
    total: usize,
}

impl Write for Sink {
    fn write(&mut self, buf: &[u8]) -> Result<usize> {
        self.total += buf.len();
        Ok(buf.len())
    }

    fn flush(&mut self) -> Result<()> {
        Ok(())
    }
}
```

Coherence rule: either trait or type must be new in the current crate.

when implementing trait methods, you can access the actual data

```rust
use std::io::Write;

let mut buf: Vec<u8> = vec![];
let writer: Write = buf; // `Write` does not have a constant size

let writer: &mut Write = &mut buf; // ok. Trait object
```



Trait Object

- Unlike java, you can't assign a value to a trait (no implicit reference!!!)
- trait object is a fat pointer (automatically converted)
  - normal pointer reference to the value
  - vtable (vtable pointer)
    - unlike C++/Java, it is not a ptr in struct
- dynamic dispatch

```rust
pub trait Formatter {
    fn format(&self, input: &mut str) -> bool;
}

struct MarkdownFormatter;
impl Formatter for MarkdownFormatter {
    fn format(&self, input: &mut str) -> bool { todo!() }
}

struct RustFormatter;
impl Formatter for RustFormatter {
    fn format(&self, input: &mut str) -> bool { todo!() }
}

struct HtmlFormatter;
impl Formatter for HtmlFormatter {
    fn format(&self, input: &mut str) -> bool { todo!() }
}

pub fn format(input: &mut str, formatters: Vec<Box<dyn Formatter>>) {
    for formatter in formatters {
        formatter.format(input);
    }
}
```

```rust
pub trait Iterator {
    type Item;
    pub fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

associate type

```rust
pub trait From<T> {
    pub fn from(T) -> Self;
}
```

generic type

```rust
trait Person {
    fn name(&self) -> String;
}

trait Student: Person {
    fn university(&self) -> String;
}

trait Programmer {
    fn fav_language(&self) -> String;
}

trait CompSciStudent: Programmer + Student {
    fn git_username(&self) -> String;
}
```

super trait

trait composition

More about trait

- associated type
- generics
- supertrait
- trait composition

Generics

History of Generic Programming

The first step was a generalized machine architecture, exemplified by the IBM 360, based on a uniform view of the machine memory as a sequence of bytes, referenced by uniform addresses (pointers) independent of the type of data being referenced. The next step was the C programming language [KeRi78], which was effectively a generalized machine language for such architectures, providing composite data types to model objects in memory, and pointers as identifiers for such memory objects with operations (dereferencing and increment/decrement) that were uniform across types. The C++ programming language [Stroustrup97] was the next step. It allows us to generalize the use of C syntax, applying the built-in operators to user types as well, using class definitions, operator overloading, and templates. The final step in this progression is generic programming

(from: Fundamentals of Generic Programming)

Classification of Abstractions

- **Data Abstractions**: data types and sets of operations defined on them
  - e.g. `Vec<T>`, `HashMap<K, V>`
- **Algorithmic abstrations**: families of data abstractions that have a set of efficient algorithms in common (generic algorithms)
  - e.g. `quicksort`, `binary_search`
- **Structural abstractions**: a data abstraction A belongs to a structural abstraction S if and only if S is an intersection of some of the algorithmic abstractions to which A belongs.
  - e.g. singly-linked-lists
- **Representational abstractions**: mappings from one structural abstration to another, creating a new type and implementing a set of operations on that type.
  - e.g. `VecDeque<T>`
- Comes from: Generic Programming: http://stepanovpapers.com/genprog.pdf

Generics to Types

just like

Types to Values

```
auto add(auto a, auto b) { return a + b; }
```

```
T add(T)(T a, T b) { return a + b; }
```

```
fn add<T>(a: T, b: T) -> T { a + b }
```

```
func add<T>(_ a: T, _ b: T) -> T { a + b }
```

```
add :: t -> t -> t
add a b = a + b
```

```
auto add(auto a, auto b) { return a + b; }
```

```
T add(T)(T a, T b) { return a + b; }
```

```
fn add<T: std::ops::Add<Output = T>>(a: T, b: T) -> T { a + b }
```

```
pub trait Add<Rhs = Self> {
    type Output;
    pub fn add(self, rhs: Rhs) -> Self::Output;
}
```

SystemTime + Duration => SystemTime

```
func add<T: Numeric>(_ a: T, _ b: T) -> T { a + b }
```

```
add :: Num t => t -> t -> t
add a b = a + b
```

Example	Explanation
`S<T>`	A **generic** BK EX type with a type parameter (`T` is placeholder name here).
`S<T: R>`	Type short hand **trait bound** BK EX specification (`R` *must* be actual trait).
`T: R, P: S`	**Independent trait bounds** (here one for `T` and one for `P`).
`T: R, S`	Compile error, you probably want compound bound `R + S` below.
`T: R + S`	**Compound trait bound** BK EX, `T` must fulfill `R` and `S`.
`T: R + 'a`	Same, but w. lifetime. `T` must fulfill `R`, if `T` has lifetimes, must outlive `'a`.
`T: ?Sized`	Opt out of a pre-defined trait bound, here `Sized`. ?
`T: 'a`	Type **lifetime bound** EX; if T has references, they must outlive `'a`.
`T: 'static`	Same; does esp. *not* mean value `t` *will* live `'static`, only that it could.
`'b: 'a`	Lifetime `'b` must live at least as long as (i.e., *outlive*) `'a` bound.
`S<const N: usize>`	**Generic const bound**; ? user of type `S` can provide constant value `N`.
`S<10>`	Where used, const bounds can be provided as primitive values.
`S<{5+5}>`	Expressions must be put in curly brackets.
`S<T> where T: R`	Almost same as `S<T: R>` but more pleasant to read for longer bounds.
`S<T> where u8: R<T>`	Also allows you to make conditional statements involving *other* types.
`S<T = R>`	**Default type parameter** BK for associated type.
`S<'_>`	Inferred **anonymous lifetime**; asks compiler to *'figure it out'* if obvious.
`S<_>`	Inferred **anonymous type**, e.g., as `let x: Vec<_> = iter.collect()`
`S::<T>`	**Turbofish** STD call site type disambiguation, e.g. `f::<u32>()`.
`trait T<X> {}`	A trait generic over `X`. Can have multiple `impl T for S` (one per `X`).
`trait T { type X; }`	Defines **associated type** BK REF `X`. Only one `impl T for S` possible.
`type X = R;`	Set associated type within `impl T for S { type X = R; }`.
`impl<T> S<T> {}`	Implement functionality for any `T` in `S<T>`, here `T` type parameter.
`impl S<T> {}`	Implement functionality for exactly `S<T>`, here `T` specific type (e.g., `S<u32>`).
`fn f() -> impl T`	**Existential types**, BK returns an unknown-to-caller `S` that `impl T`.
`fn f(x: &impl T)`	Trait bound,"**impl traits**", BK somewhat similar to `fn f<S:T>(x: &S)`.
`fn f(x: &dyn T)`	Marker for **dynamic dispatch**, BK REF `f` will not be monomorphized.
`fn f() where Self: R;`	In `trait T {}`, make `f` accessible only on types known to also `impl R`.
`fn f() where Self: R {}`	Esp. useful w. default methods (non dflt. would need be impl'ed anyway).
`for<'a>`	**Higher-ranked trait bounds.** NOM REF
`trait T: for<'a> R<'a> {}`	Any `S` that `impl T` would also have to fulfill `R` for any lifetime.

Generic Programming Example

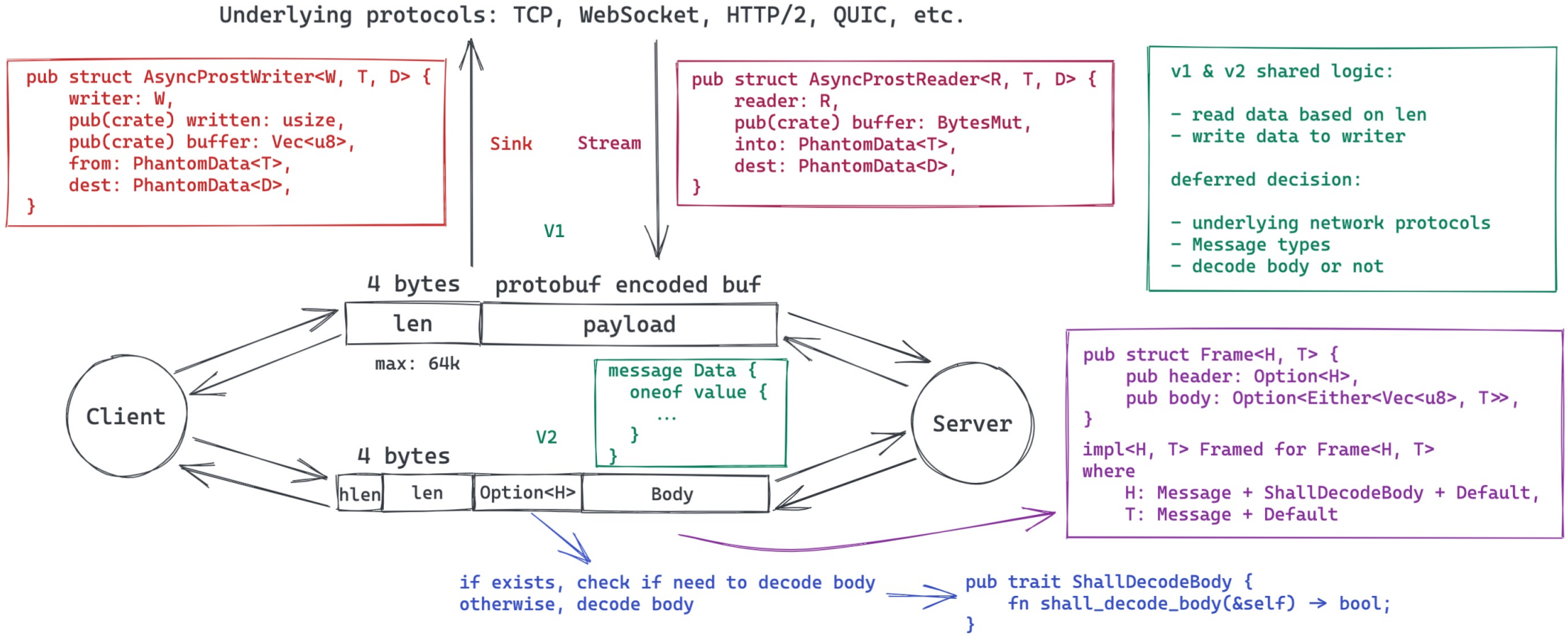
```
int binary_search(int x[], int n, int v) {
  int l = 0;
  int u = n;

  while (true) {
    if (l > u) return -1;

    int m = (l + u) / 2;

    if (x[m] < v) l = m + 1;
    else if (x[m] == v) return m;
    else u = m - 1;
  }
}
```

```
template <class I, class T>
I lower_bound(I f, I l, const T& v) {
  while (f ≠ l) {
    auto m = next(f, distance(f, l) / 2);

    if (*m < v) f = next(m);
    else l = m;
  }
  return f;
}
```

first — ForwardIterator — value_type(I), comparable — last

demo: rust implmentation

Realworld GP example

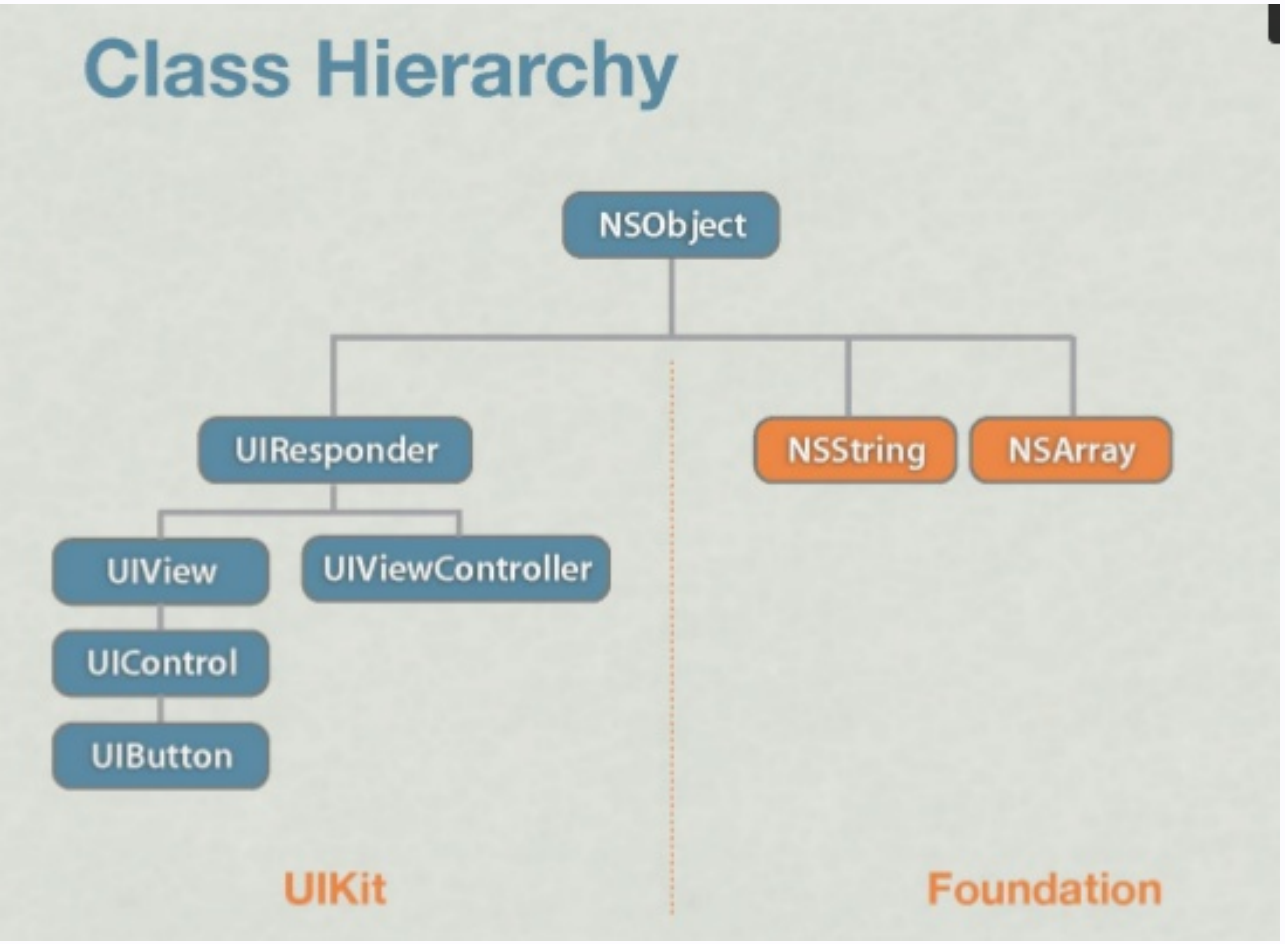

Underlying protocols: TCP, WebSocket, HTTP/2, QUIC, etc.
pub struct AsyncProstWriter<W, T, D> {
writer: W,
pub(crate) written: usize,
pub(crate) buffer: Vec<u8>,
from: PhantomData<T>,
dest: PhantomData<D>,
}
Sink
Stream
pub struct AsyncProstReader<R, T, D> {
reader: R,
pub(crate) buffer: BytesMut,
into: PhantomData<T>,
dest: PhantomData<D>,
}
v1 & v2 shared logic:
- read data based on len
- write data to writer
deferred decision:
- underlying network protocols
- Message types
- decode body or not
V1
4 bytes
protobuf encoded buf
len
payload
max: 64k
message Data {
oneof value {
...
}
}
V2
Client
Server
4 bytes
hlen
len
Option<H>
Body
pub struct Frame<H, T> {
pub header: Option<H>,
pub body: Option<Either<Vec<u8>, T>>,
}
impl<H, T> Framed for Frame<H, T>
where
H: Message + ShallDecodeBody + Default,
T: Message + Default
if exists, check if need to decode body
otherwise, decode body
pub trait ShallDecodeBody {
fn shall_decode_body(&self) -> bool;
}
impl<W, T, D> Sink<T> for AsyncProstWriter<W, T, D>
where
W: AsyncWrite + Unpin,
Self: ProstWriterFor<T>,
pub trait ProstWriterFor<T> {
fn append(&mut self, item: T) -> Result<(), io::Error>;
}
impl<W, F: Framed> ProstWriterFor<F>
for AsyncProstWriter<W, F, AsyncFrameDestination> { ... }
impl<W, T: Message> ProstWriterFor<T>
for AsyncProstWriter<W, T, AsyncDestination> { ... }
impl<R, T> Stream for AsyncProstReader<R, T, AsyncDestination>
where
T: Message + Default,
R: AsyncRead + Unpin { ... }
impl<R, T> Stream for AsyncProstReader<R, T, AsyncFrameDestination>
where
R: AsyncRead + Unpin,
T: Framed + Default { ... }

> Functions delay binding: data structures induce binding.
> Moral: Structure data late in the programming process.
>
> — Epigrams on programming

References

- All about trait objects
- Protocol-oriented programming in swift
- Generic Data Types
- Generics (Rust by Example)

Why not object oriented?



Class is awesome

- Encapsulation
- Access control
- Abstraction
- Namespace
- Extensibility

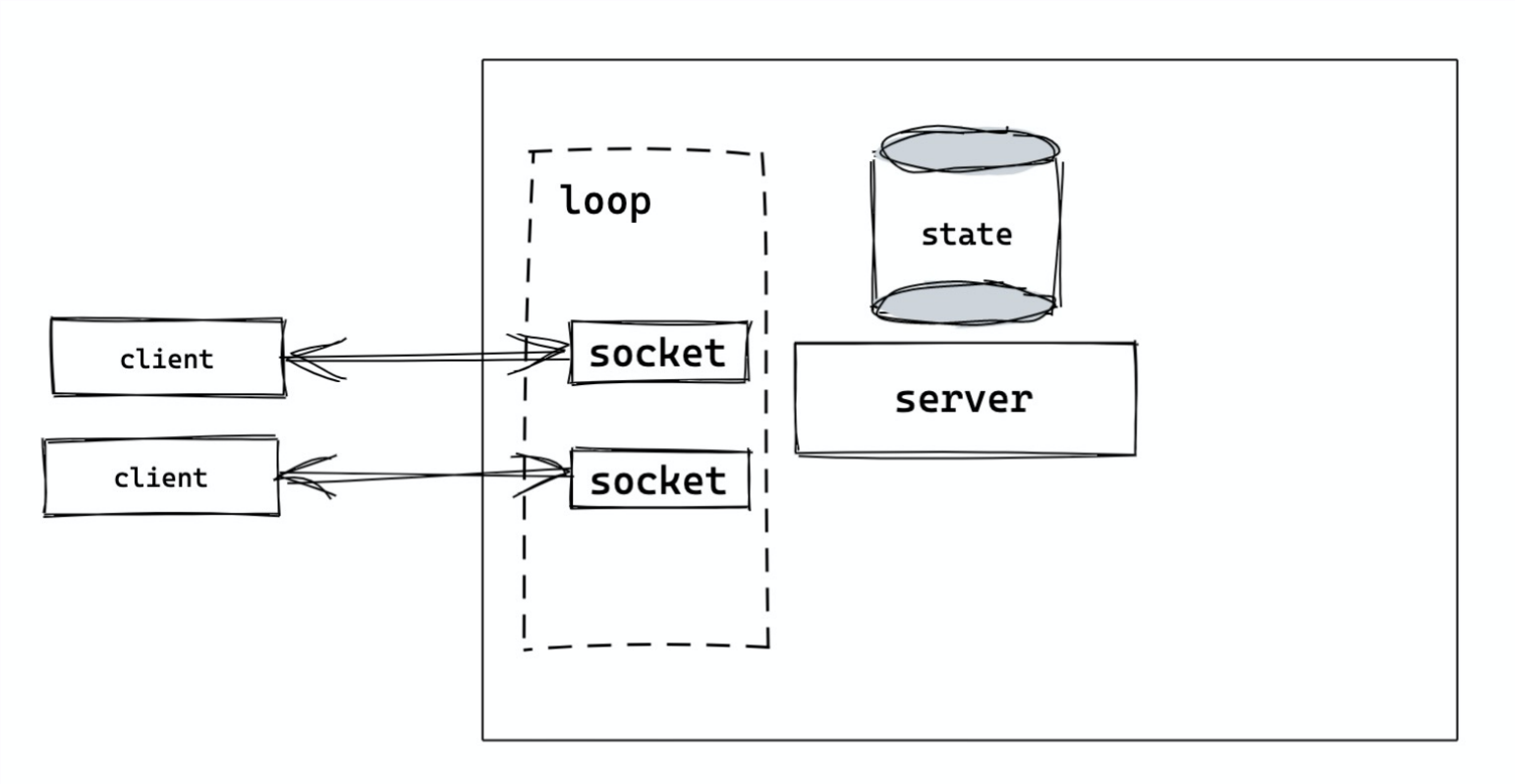
Class has problems

- inheritance is pretty limited - choose superclass well!
- know what/how to override (and when not to)
- superclass may have properties
  - you have to accept it
  - initialization burden
  - don't break assumptions of superclass
- hard to reuse outside the hierachy (composition over inheritance)

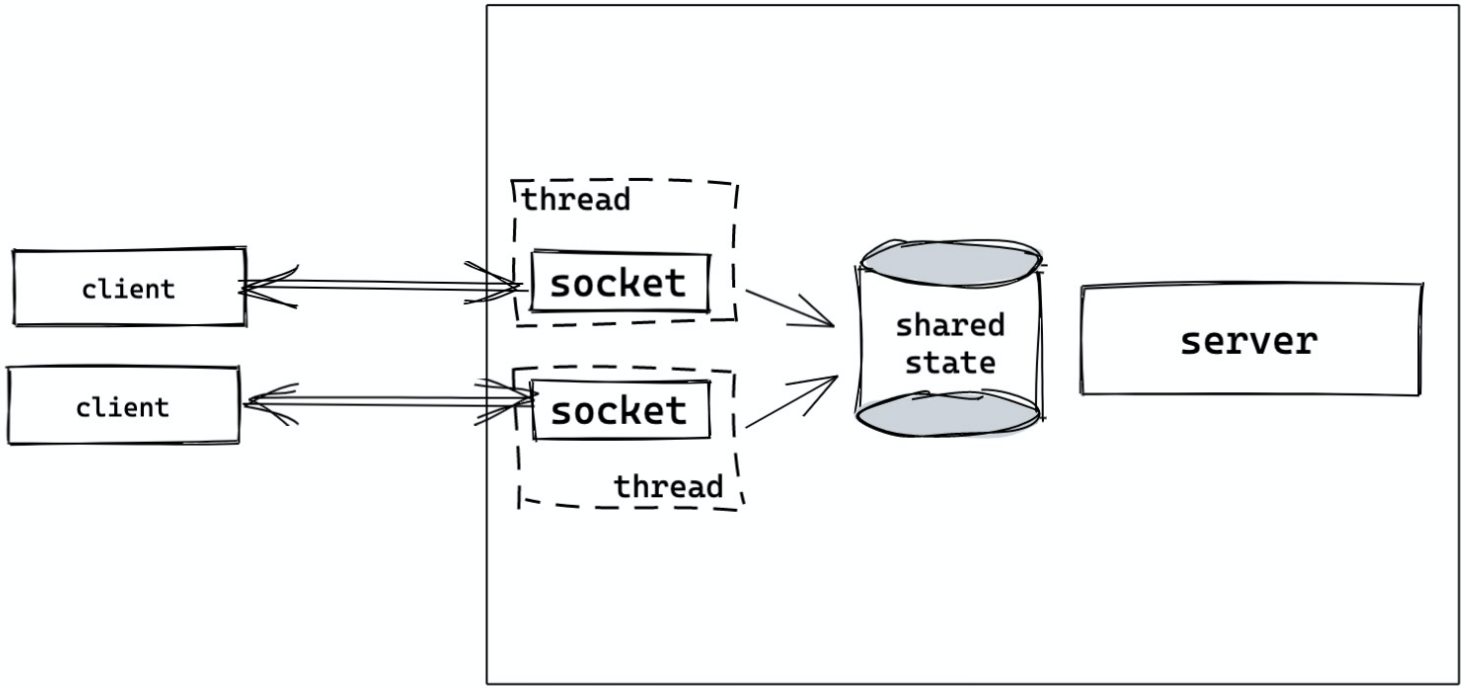
Concurrency - primitives

Let's solve a real-world problem

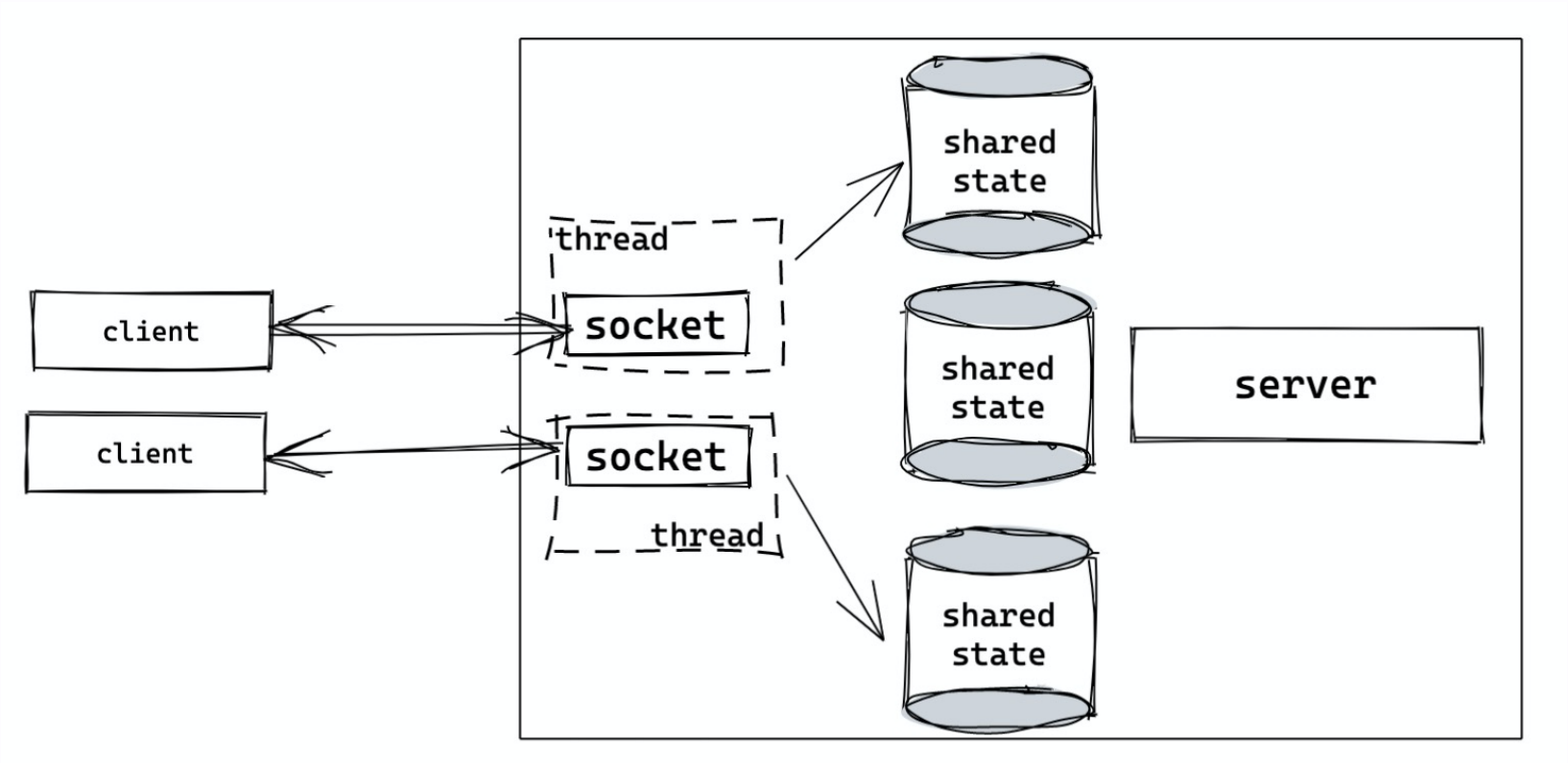
v1: Simple loop



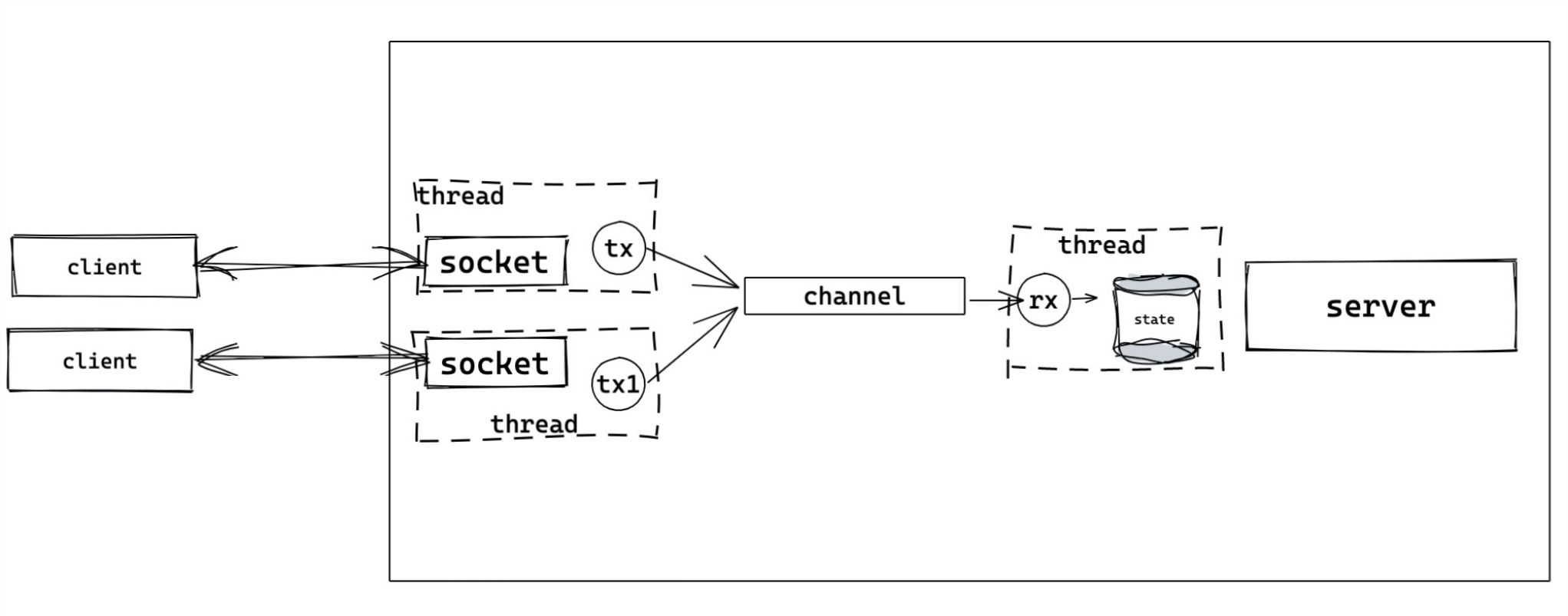
v2: Multithread with shared state



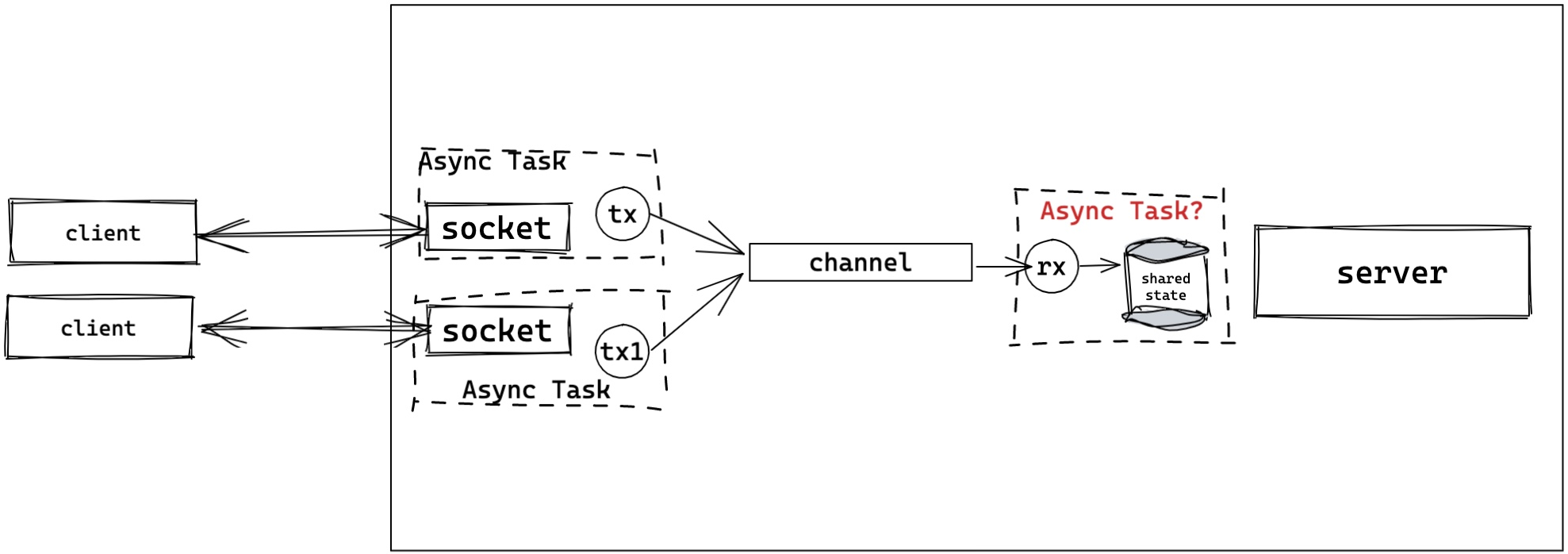
v3: Optimize the lock



v4: Share memory by communicating



v5: Async Task



What have we used so far?

- Mutex Lock
- Channel/Actor
- Async Task (coroutine, promise, etc.)

How is Mutex implemented?

An naive implementation

```
struct Lock<T> {
  locked: bool,
  data: T,
}

impl<T> Lock<T> {
  pub fn new(data: T) -> Lock<T> { ... }

  pub fn lock<R>(&mut self, op: impl FnOnce(&mut T) -> R) -> R {
    // spin if we can't get lock
    while self.locked != false {} // **1
    // ha, we can lock and do our job
    self.locked = true; // **2
    // execute the op as we got lock
    let ret = op(self.data); // **3
    // unlock
    self.locked = false; // **4
    ret
  }
}

// You may call it like this:
let l = Lock::new(0);
l.lock(|v| v += 1);
```

Issues

- Atomicity
  - In multicore environment, race condition between 1 and 2 - other thread may kick in
  - Even in single core environment, OS may do preempted multitasking, causing other thread kick in
- OOO execution
  - Compiler might generate optimized instructions that put 3 before 1
  - CPU may do OOO execution to best utilize pipeline, so putting 3 before 1 might also happen

How to solve this?

- We need CPU instruction to guarantee Atomicity and non-OOO
- Algorithm: **CAS (Compare-And-Swap)**
- data structure: `AtomicXXX`

Atomics

Updated lock

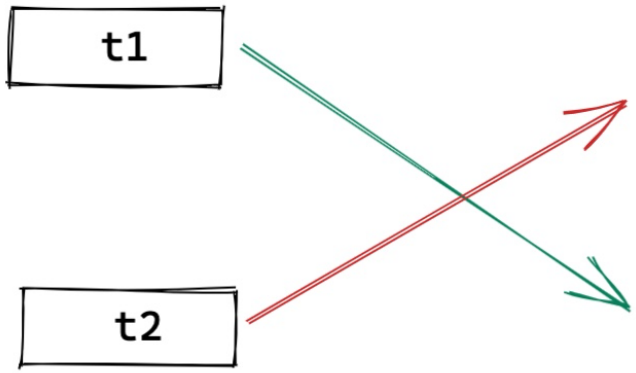
```
struct Lock<T> {
  locked: AtomicBool, // ***
  data: UnsafeCell<T>, // ***
}
unsafe impl<T> Sync for Lock<T> where T: Send {} // need to explicitly impl `Send`

impl<T> Lock<T> {
  pub fn new(data: T) -> Self { ... }
  pub fn lock<R>(&mut self, op: impl FnOnce(&mut T) -> R) -> R {
    // spin if we can't get lock
    while self
      .locked
      .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
      .is_error() {}
    // execute the op as we got lock
    let ret = op(unsafe { &mut *self.v.get() }); // **3
    // unlock
    self.locked.store(false, Ordering::Release); // **4
    ret
  }
}

// You may call it like this:
let l = Lock::new(0);
l.lock(|v| v += 1);
```

What does ordering mean?

- Relaxed: No restriction to compiler/CPU, OOO is allowed
- Release:
  - for current thread, any read/write inst cannot be OOO after this inst (`store`);
  - for other thread, if they use `Acquire` to read, they would see the changed value
- Acquire
  - for current thread, any read/write inst cannot be OOO before this inst (`compare_exchange`)
  - for other thread, if they use `Release` to update data, the modification would be see for current thread
- AcqRel: combination of Acquire and Release
- SeqCst: besides `AcqRel`, all threads would see same operation order.

Optimization

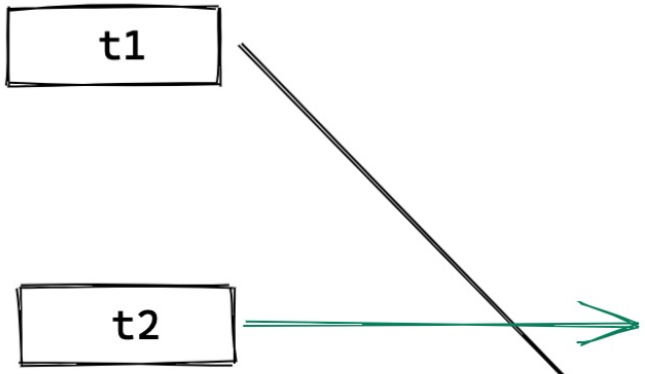
```
pub struct Lock<T> {
    locked: AtomicBool,
    data: UnsafeCell<T>,
}
unsafe impl<T> Sync for Lock<T> where T: Send {}

impl<T> Lock<T> {
  pub fn new(data: T) -> Self {...}
  pub fn lock<R>(&self, op: impl FnOnce(&mut T) -> R) -> R {
      while self
          .locked
          .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
          .is_err()
      {
          while self.locked.load(Ordering::Relaxed) == true {
              std::thread::yield_now(); // we may yield thread now
          }
      }
      let ret = op(unsafe { &mut *self.data.get() });
      self.locked.store(false, Ordering::Release);
      ret
  }
}
```




```
pub fn with_lock<R>(&self, f: impl FnOnce(&mut T) -> R) -> R {
    while self
            .locked
            .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
            .is_err()
    {
        while self.locked.load(Ordering::Relaxed) == true {}
    }
    let ret = op(unsafe { &mut *self.v.get() });
    self.locked.store(false, Ordering::Release);
    ret
}
```

t1 finished op, released lock

t2 finished spin, entered critical section



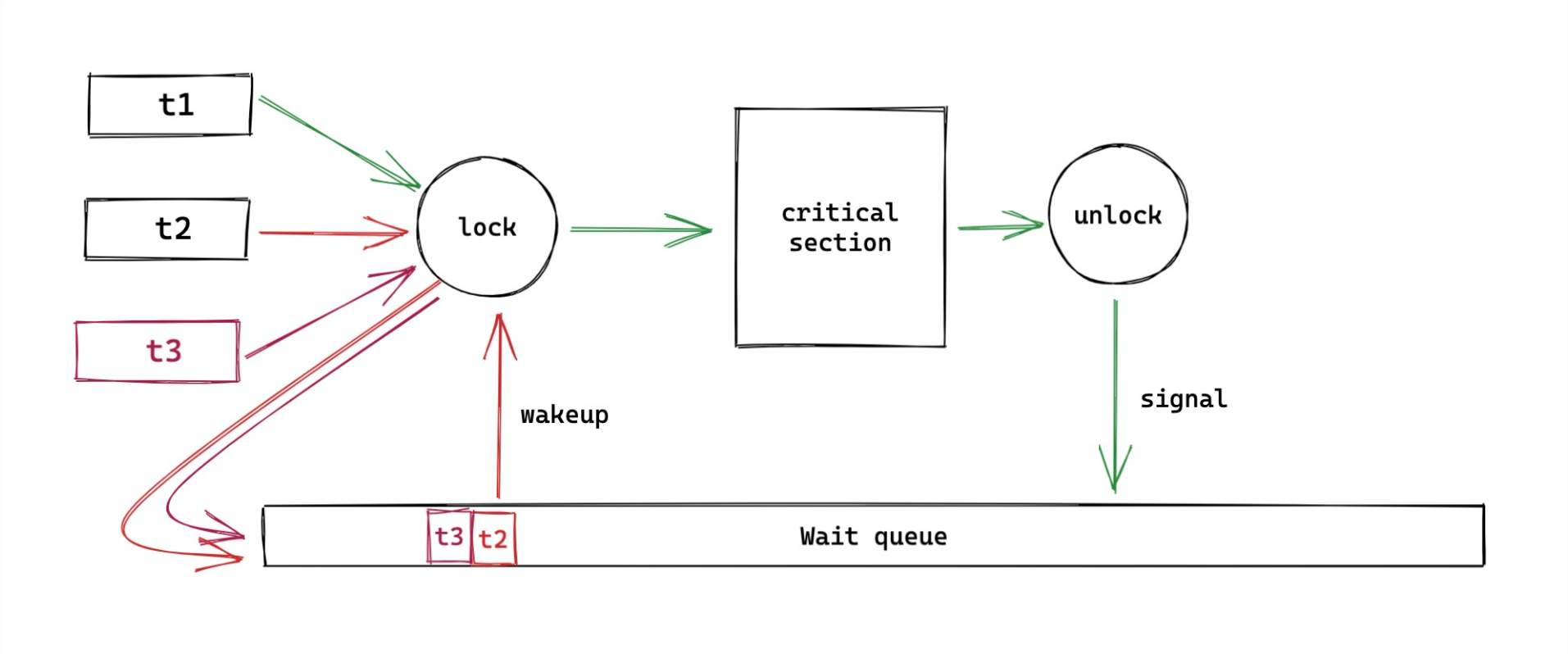

```
pub fn with_lock<R>(&self, f: impl FnOnce(&mut T) -> R) -> R {
    while self
            .locked
            .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
            .is_err()
    {
        while self.locked.load(Ordering::Relaxed) == true {}
    }
    let ret = op(unsafe { &mut *self.v.get() });
    self.locked.store(false, Ordering::Release);
    ret
}
```

This is basically how

Mutex

works

Real world Mutex

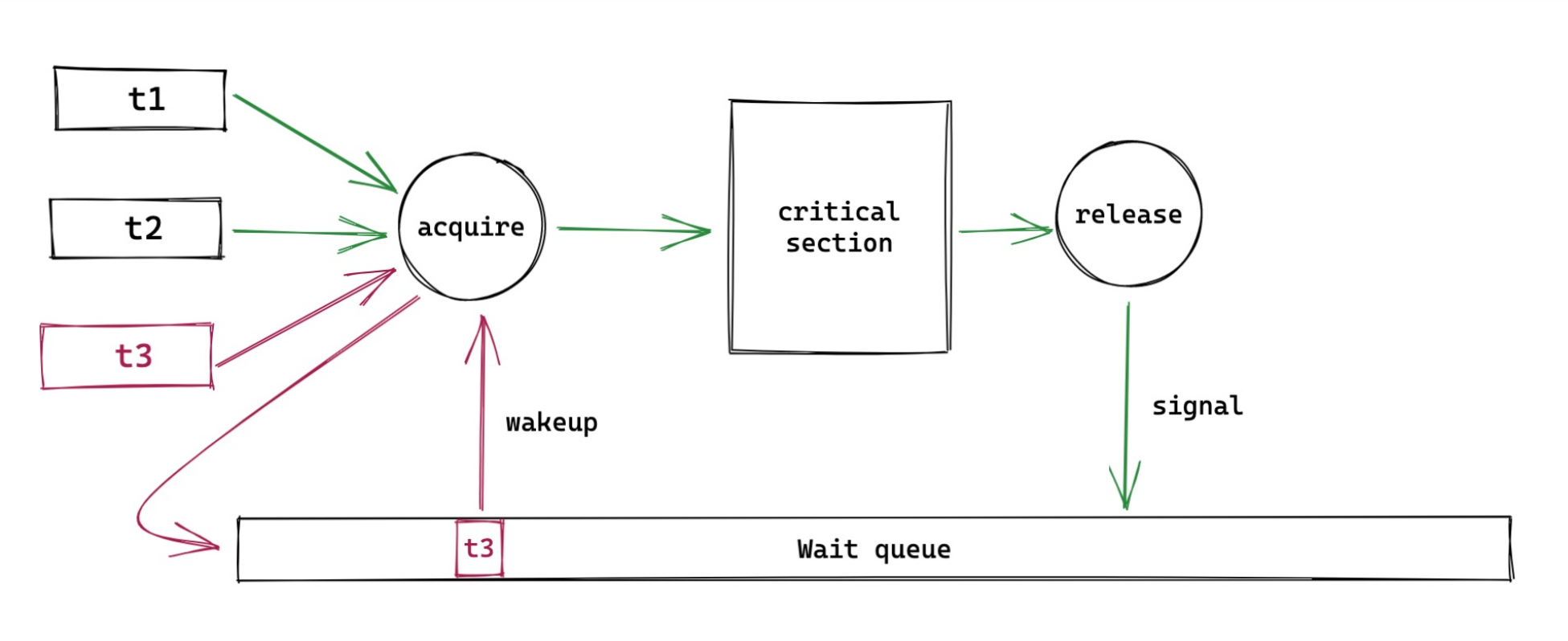


Semaphore

“In computer science, a **semaphore** is a variable or abstract data type used to control access to a common resource by multiple processes and avoid critical section problems in a concurrent system such as a multitasking operating system. A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions.

A useful way to think of a semaphore as used in a real-world system is as **a record of how many units of a particular resource are available**, coupled with operations to adjust that record safely (i.e., to avoid race conditions) as units are acquired or become free, and, if necessary, wait until a unit of the resource becomes available.”

Semaphore as a generalized Mutex

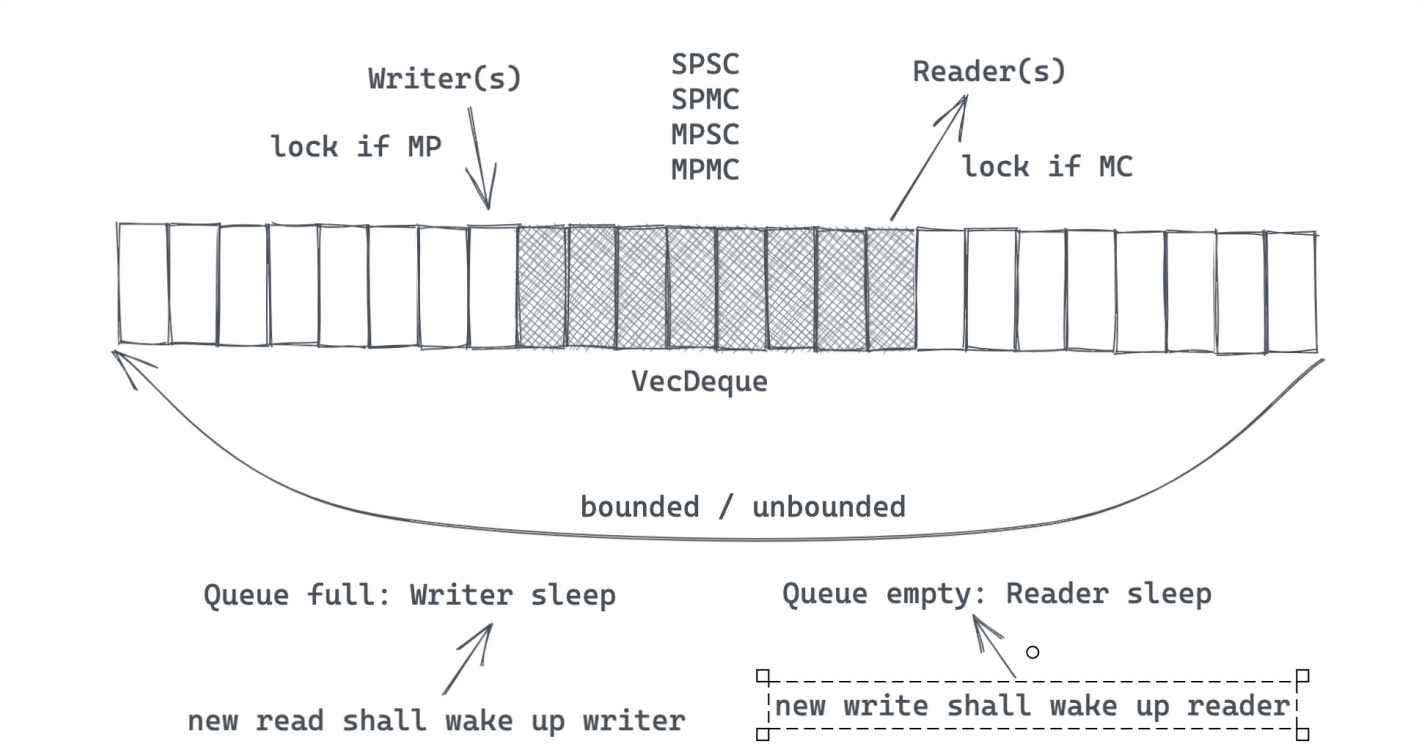


Demo code: restricted HTTP client

code/primitives/src/http_semaphore.rs

Channel

Channel basics



Flavours of Channels

- sync: sender can block, limited capacity
  - `Mutex` + `Condvar` + `VecDeque`
  - Atomic VecDeque (atomic queue) + `thread::park` + `thread::notify`
- async: sender cannot block, unbounded
  - `Mutex` + `Condvar` + `VecDeque`
  - `Mutex` + `Condvar` + `DoubleLinkedList`
- rendezvous: sync with capacity = 0. Used for thread sync.
  - `Mutex` + `Condvar`
- oneshot: only one call to `send()`. e.g. Ctrl+C to stop all threads
  - atomic swap
- async/await
  - basically same as sync but waker is different

Demo code: naive MPSC

code/primitives/src/channel.rs

Demo code: naive actor

- Questions:
  - Which type of channel shall we use? SPSC, SPMC, MPSC?
  - When creating an actor, what is its `pid`?
  - When sending a message to an actor, how the actor reply (`handle_call`)?
- Code: code/primitives/src/actor.rs

References

- CAS: https://en.wikipedia.org/wiki/Compare-and-swap
- Ordering: https://doc.rust-lang.org/std/sync/atomic/enum.Ordering.html
- std::memory_order: https://en.cppreference.com/w/cpp/atomic/memory_order
- Atomics and Memory Ordering: https://www.youtube.com/watch?v=rMGWeSjctlY
- spinlock: https://en.wikipedia.org/wiki/Spinlock
- spin-rs: https://github.com/mvdnes/spin-rs
- parking lot： https://github.com/Amanieu/parking_lot
- Flume: https://github.com/zesterer/flume
- Crossbeam channel： https://docs.rs/crossbeam-channel

Things to do with atomics

- lock free data structure
- in memory metrics
- id generation

Concurrency - async/await

Using threads in Rust

```
use std::thread;

fn main() {
    println!("So we start the program here!");
    let t1 = thread::spawn(move || {
        thread::sleep(std::time::Duration::from_millis(200));
        println!("We create tasks which gets run when they're finished!");
    });

    let t2 = thread::spawn(move || {
        thread::sleep(std::time::Duration::from_millis(100));
        println!("We can even chain callbacks...");
        let t3 = thread::spawn(move || {
            thread::sleep(std::time::Duration::from_millis(50));
            println!("...like this!");
        });
        t3.join().unwrap();
    });
    println!("While our tasks are executing we can do other stuff here.");

    t1.join().unwrap();
    t2.join().unwrap();
}
```

Drawbacks of threads

- large stack (not suitable for heavy loaded concurrent jobs - e.g. a web server)
- context switch is out of your control
- lots of syscall involved (costly when # of threads is high)

What are alternative solutions?

Green threads/processes

(stackful coroutine)

Golang/Erlang

Green Threads

- Run some non-blocking code.
- Make a blocking call to some external resource.
- CPU "jumps" to the "main" thread which schedules a different thread to run and "jumps" to that stack.
- Run some non-blocking code on the new thread until a new blocking call or the task is finished.
- CPU "jumps" back to the "main" thread, schedules a new thread which is ready to make progress, and "jumps" to that thread.

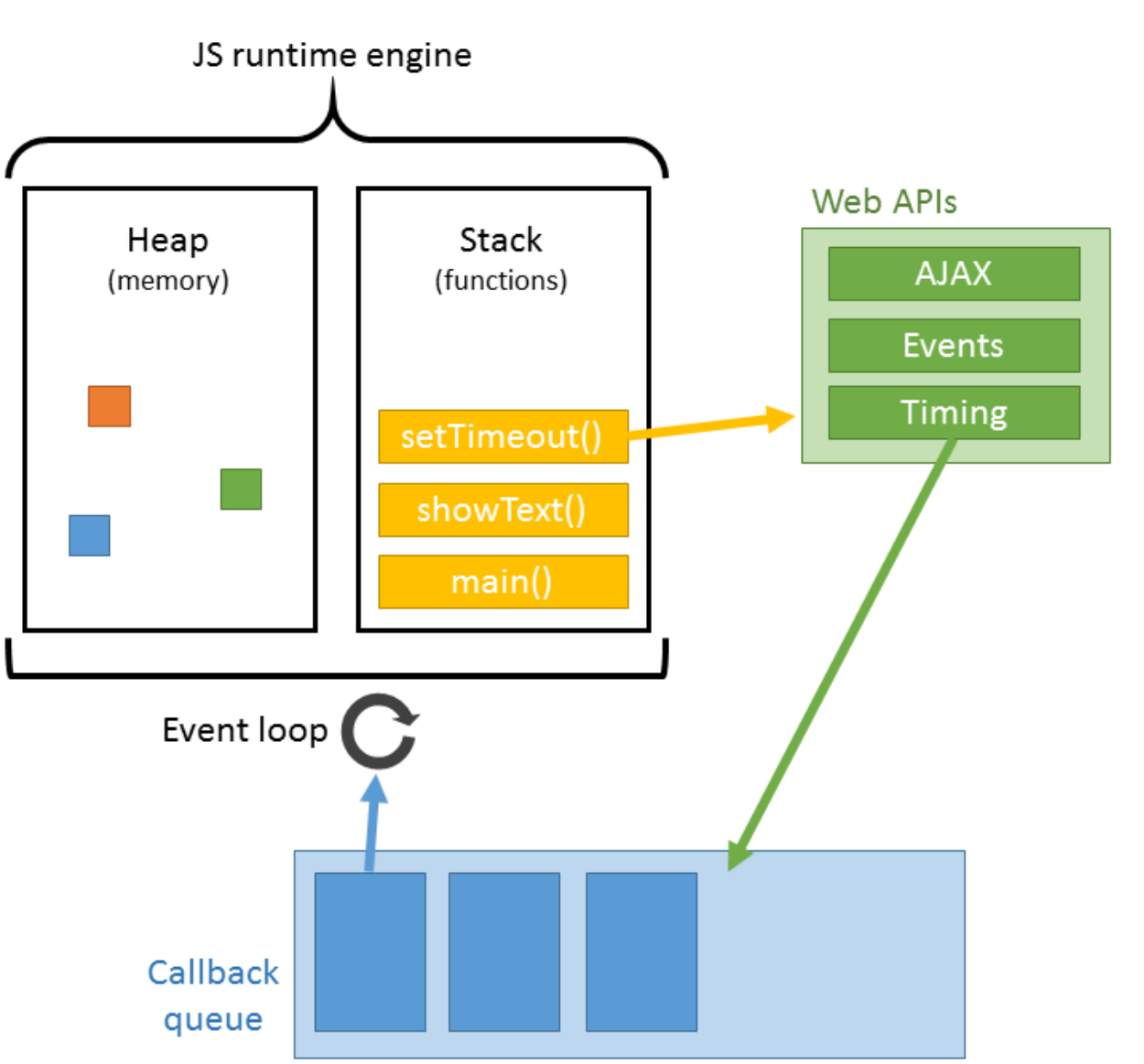
Green Threads - pros/cons

- Pros:
  - Simple to use. The code will look like it does when using OS threads.
  - A "context switch" is reasonably fast.
  - Each stack only gets a little memory to start with so you can have hundreds of thousands of green threads running.
  - It's easy to incorporate preemption which puts a lot of control in the hands of the runtime implementors.
- Cons:
  - The stacks might need to grow. Solving this is not easy and will have a cost.
  - You need to save the CPU state on every switch.
  - It's not a zero cost abstraction (Rust had green threads early on and this was one of the reasons they were removed).
  - Complicated to implement correctly if you want to support many different platforms.

Poll based event loops

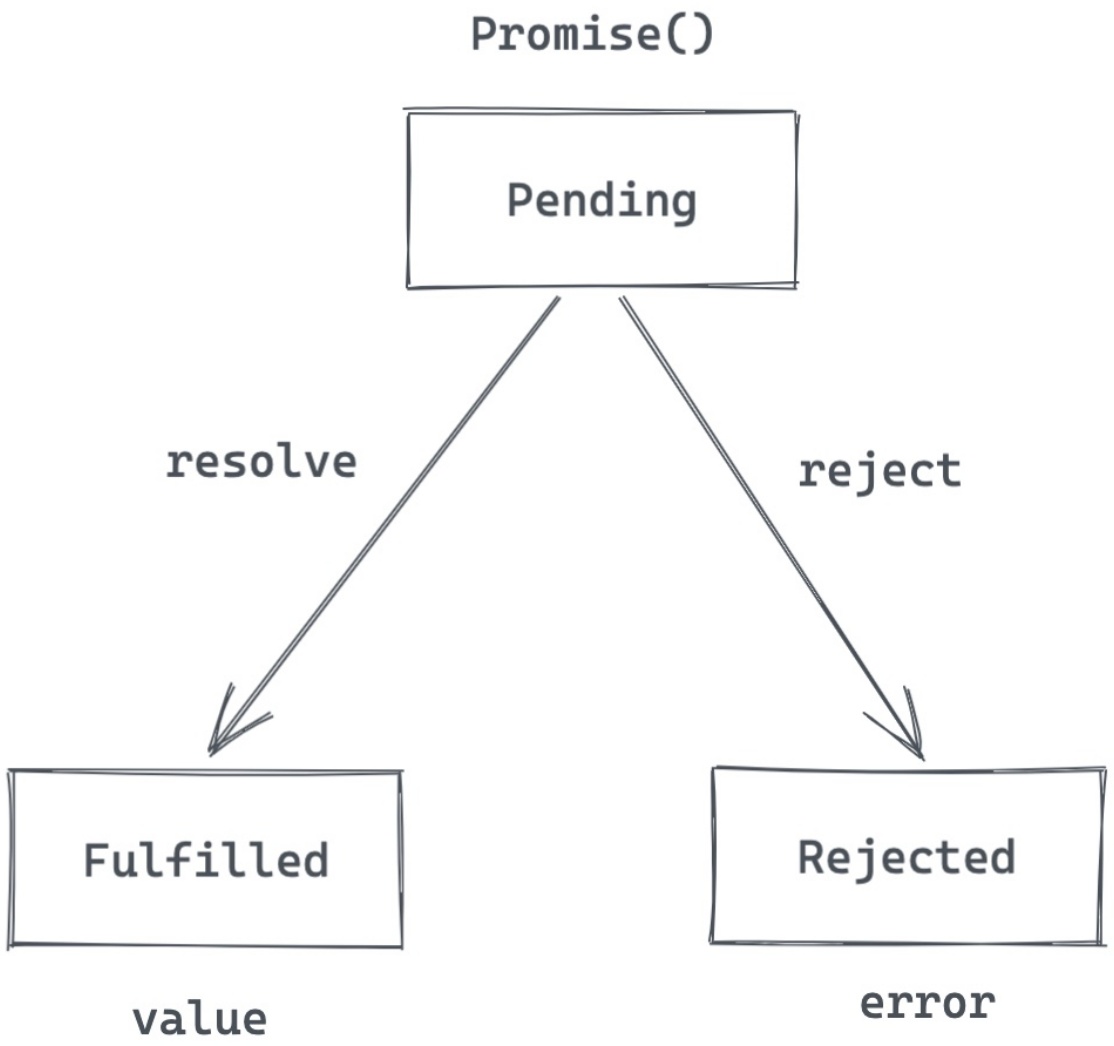
(stackless coroutine)

Javascript/Rust



Callback

```
setTimer(200, () => {
  setTimer(100, () => {
    setTimer(50, () => {
      console.log("I'm the last one");
    });
  });
});
```



Promise

```
function timer(ms) {
    return new Promise(
      (resolve) => setTimeout(resolve, ms)
    );
}

timer(200)
.then(() => timer(100))
.then(() => timer(50))
.then(() => console.log("I'm the last one"));
```

Async/Await

```
async function run() {
    await timer(200);
    await timer(100);
    await timer(50);
    console.log("I'm the last one");
}
```

The Rust approach

```
fn step1<'a>(xZ: &'a u8) -> impl Future<Output=u8> + 'a {}
```

syntax sugar to

```
async fn step1(x: &u8) -> u8 {}
async fn do_tasks() {
  step1().await;
  let step2 = async {
    ...
  };
  step2.await;
}
```

```
pub trait Future {
    type Output;
    pub fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Self::Output>;
}
```

```
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

```
pub struct Context<'a> {
    waker: &'a Waker,
    _marker: PhantomData<fn(&'a ()) -> &'a ()>,
}
```

```
pub struct Waker {
    waker: RawWaker,
}
pub struct RawWaker {
    data: *const (),
    vtable: &'static RawWakerVTable,
}
```

Internally generated code
(Zero Cost Abstraction)

illustrated

```
struct DoTasks {
  step1: FutStep1,
  step2: FutStep2,
  state: State,
}

enum State {
  AwaitingStep1,
  AwaitingStep2,
  Done,
}
```

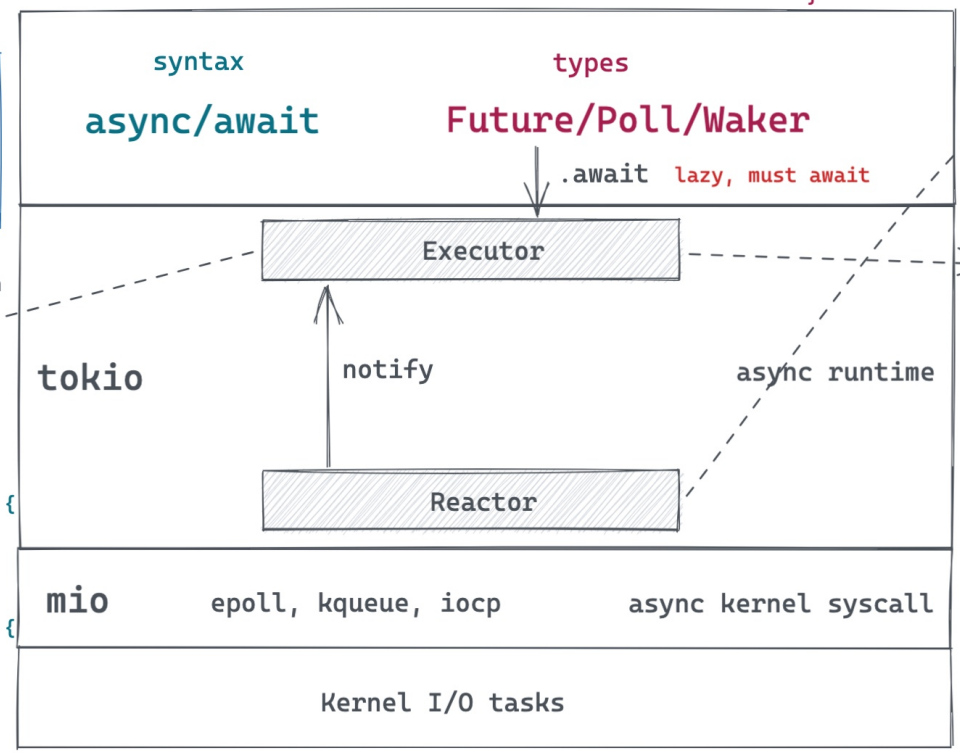
optimized

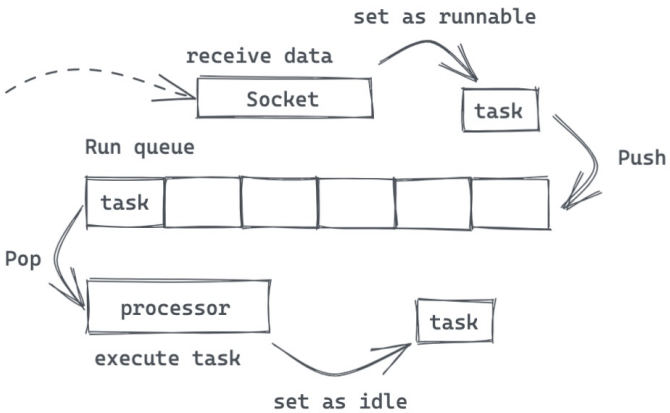
actual

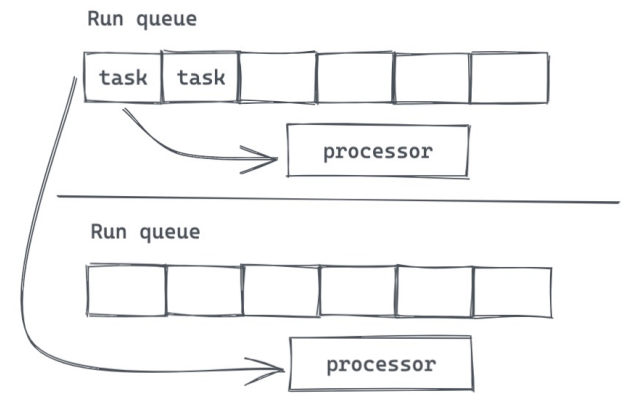
```
enum DoTasks {
  Init,
  AwaitingStep1(FutStep1),
  AwaitingStep2(FutStep2),
  Completed,
}
```

poll future until return

```
impl Future for DoTasks {
  type Output = ();

  fn poll(mut self: Pin<&mut Self>,
       cx: &mut Context<'_>) -> Poll<()> {
    loop {
      match self.state {
        State::AwaitingStep1 => match self.step1.poll(..) {
          Poll::Ready(()) =>
            self.state = State::AwaitingStep2,
          Poll::Pending => return Poll::Pending,
        },
        State::AwaitingStep2 => match self.step2.poll(..) {
        Poll::Ready(()) => self.state = State::Done,
          Poll::Pending => return Poll::Pending,
        },
        State::Done => return Poll::Ready(()),
      }
    }
  }
}
```







Work stealing scheduler

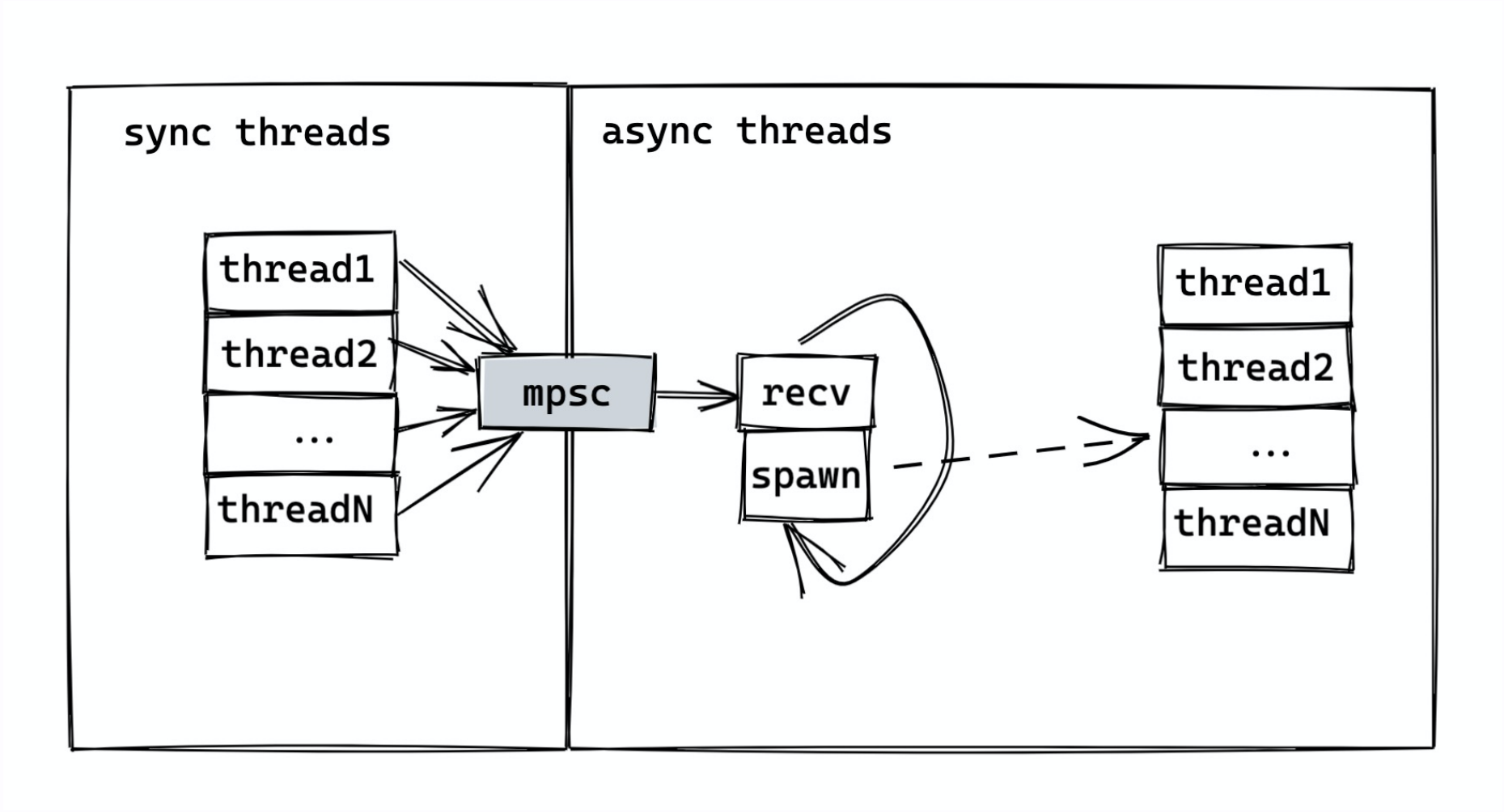
What is Pin<P>?

a Pin<P> ensures that the pointee of any pointer type P has a stable location in memory, meaning it cannot be moved elsewhere and its memory cannot be deallocated until it gets dropped. We say that the pointee is “pinned”.

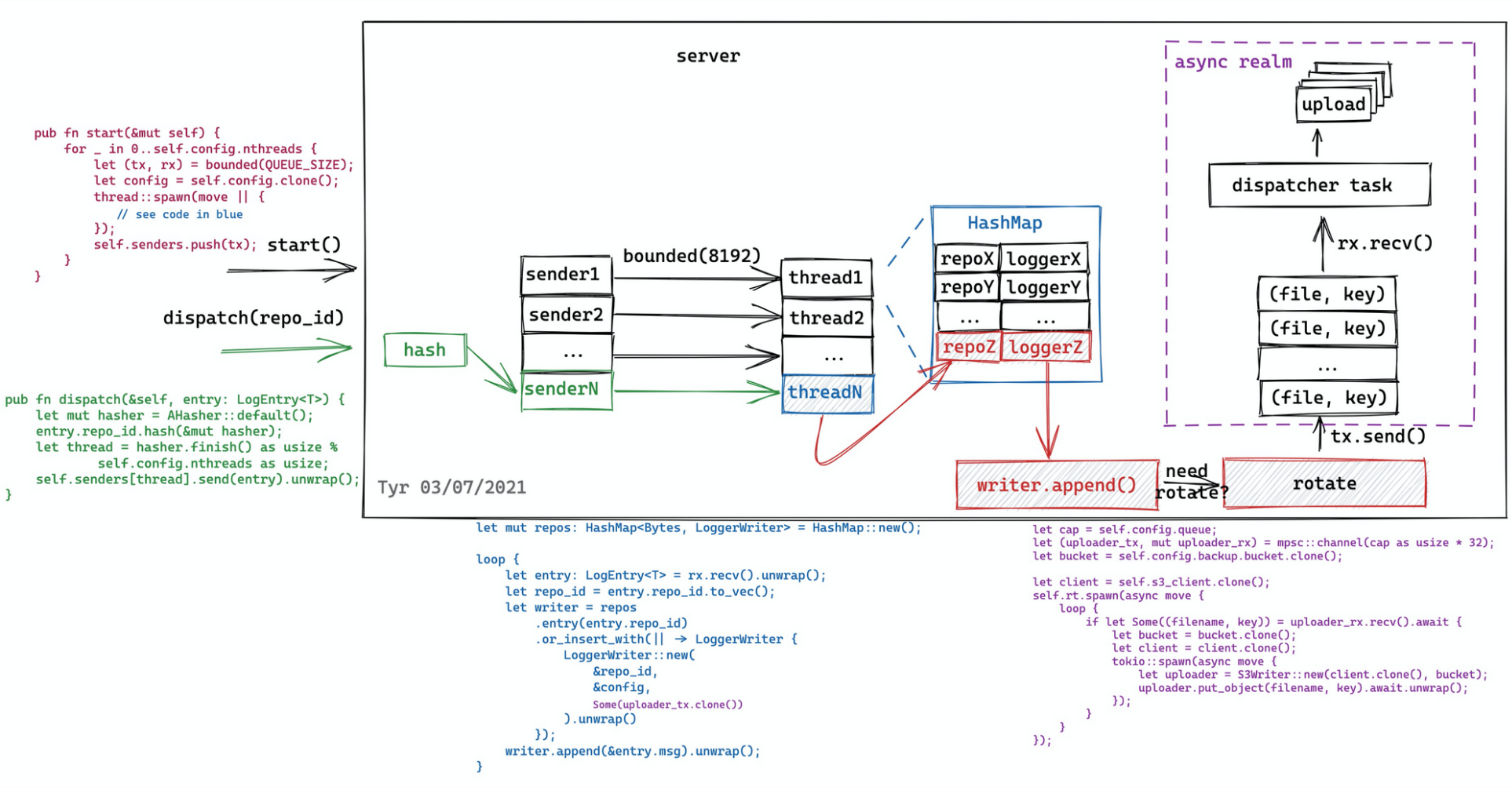
Demo: writing a server in rust

It uses async-prost, tokio and prost

One more thing...



An event store example




```
pub fn start(&mut self) {
    for _ in 0..self.config.nthreads {
        let (tx, rx) = bounded(QUEUE_SIZE);
        let config = self.config.clone();
        thread::spawn(move || {
            // see code in blue
        });
        self.senders.push(tx);
    }
}
```

```
pub fn dispatch(&self, entry: LogEntry<T>) {
    let mut hasher = AHasher::default();
    entry.repo_id.hash(&mut hasher);
    let thread = hasher.finish() as usize %
            self.config.nthreads as usize;
    self.senders[thread].send(entry).unwrap();
}
```

```
let mut repos: HashMap<Bytes, LoggerWriter> = HashMap::new();

loop {
    let entry: LogEntry<T> = rx.recv().unwrap();
    let repo_id = entry.repo_id.to_vec();
    let writer = repos
        .entry(entry.repo_id)
        .or_insert_with(|| -> LoggerWriter {
            LoggerWriter::new(
                &repo_id,
                &config,
                Some(uploader_tx.clone())
            ).unwrap()
        });
    writer.append(&entry.msg).unwrap();
}
```

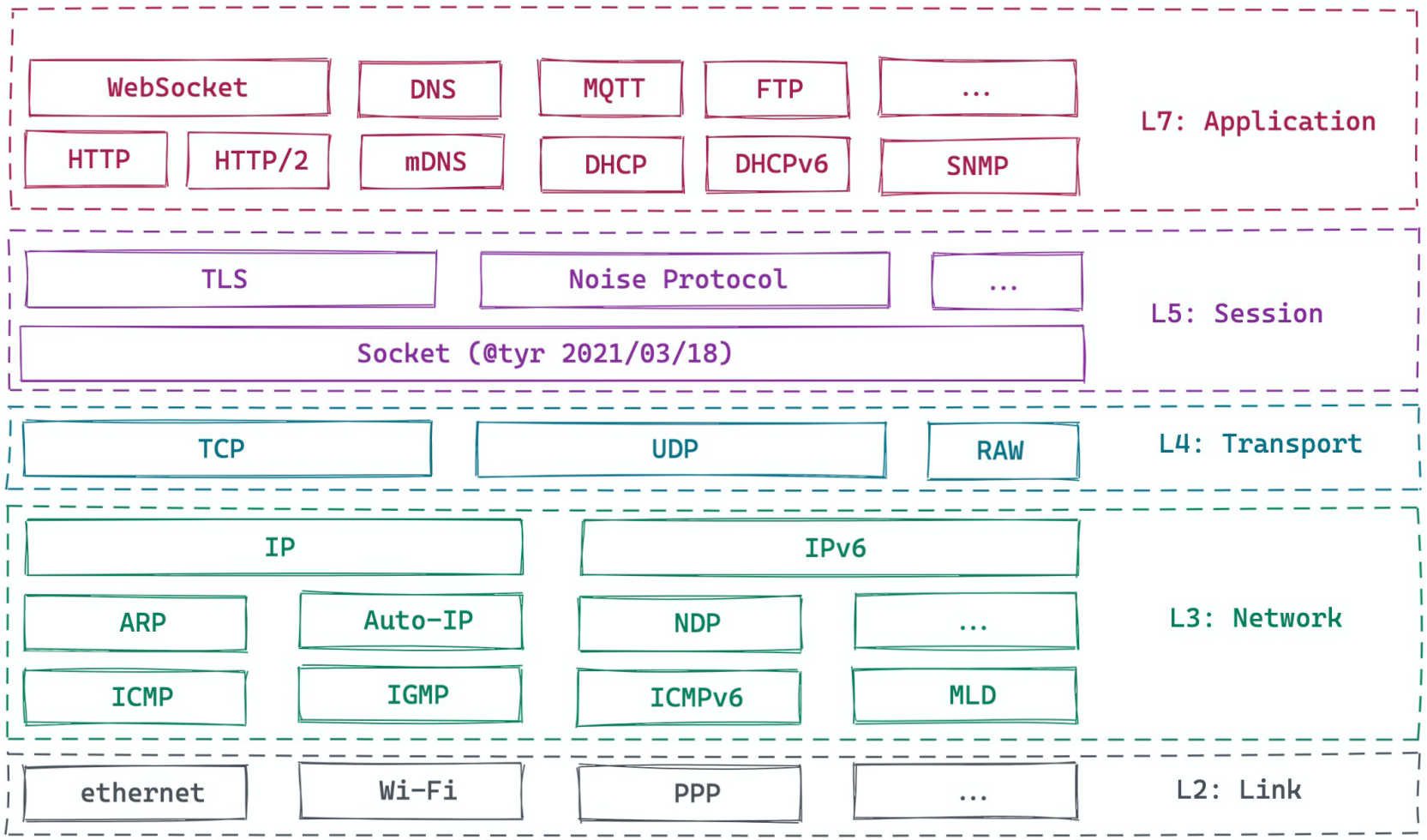
```
let cap = self.config.queue;
let (uploader_tx, mut uploader_rx) = mpsc::channel(cap as usize * 32);
let bucket = self.config.backup.bucket.clone();

let client = self.s3_client.clone();
self.rt.spawn(async move {
    loop {
        if let Some((filename, key)) = uploader_rx.recv().await {
            let bucket = bucket.clone();
            let client = client.clone();
            tokio::spawn(async move {
                let uploader = S3Writer::new(client.clone(), bucket);
                uploader.put_object(filename, key).await.unwrap();
            });
        }
    }
});
```

References

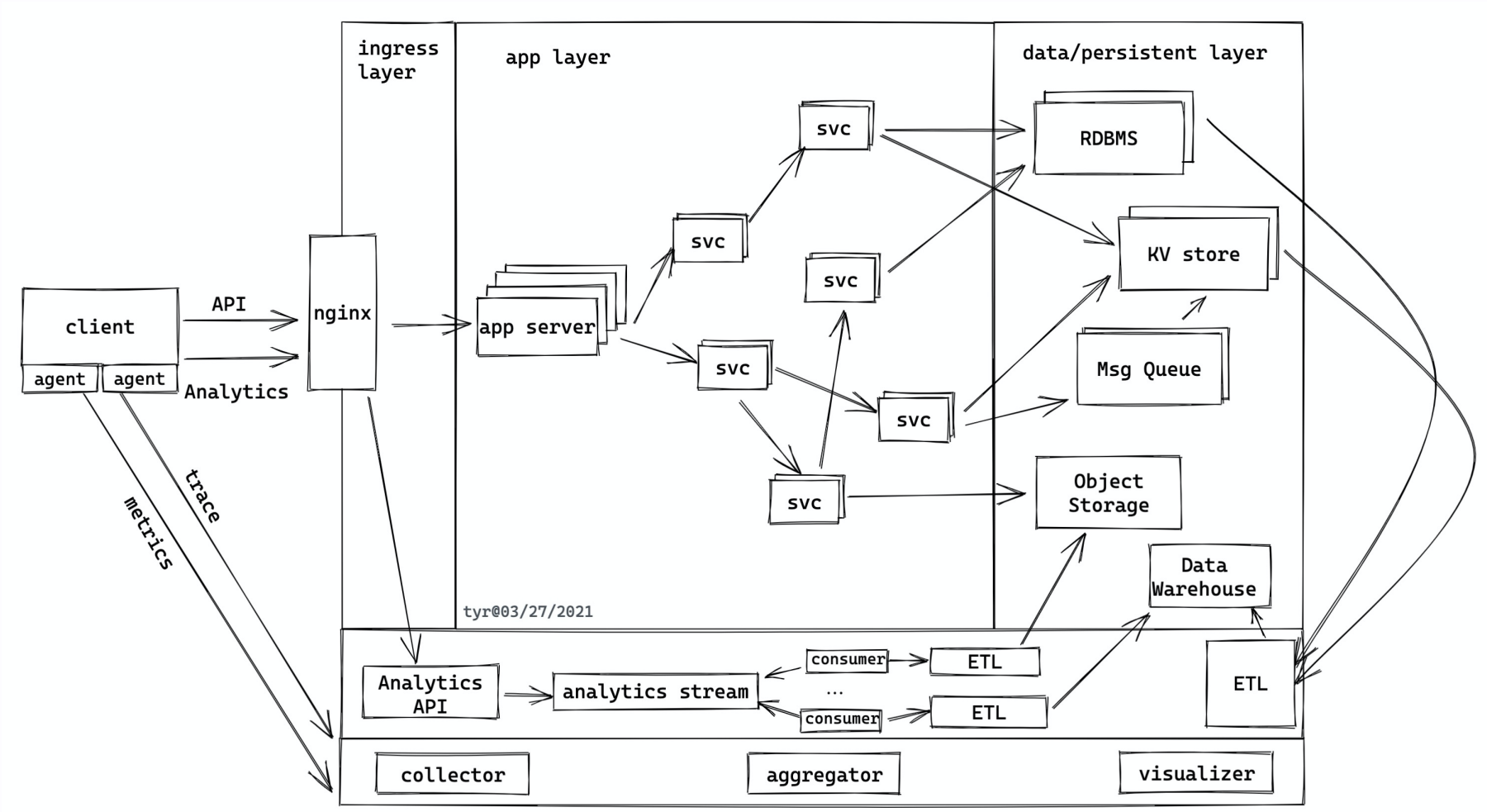
- Future explained
- Rust async book
- calloop: a callback based event loop

Networking and security

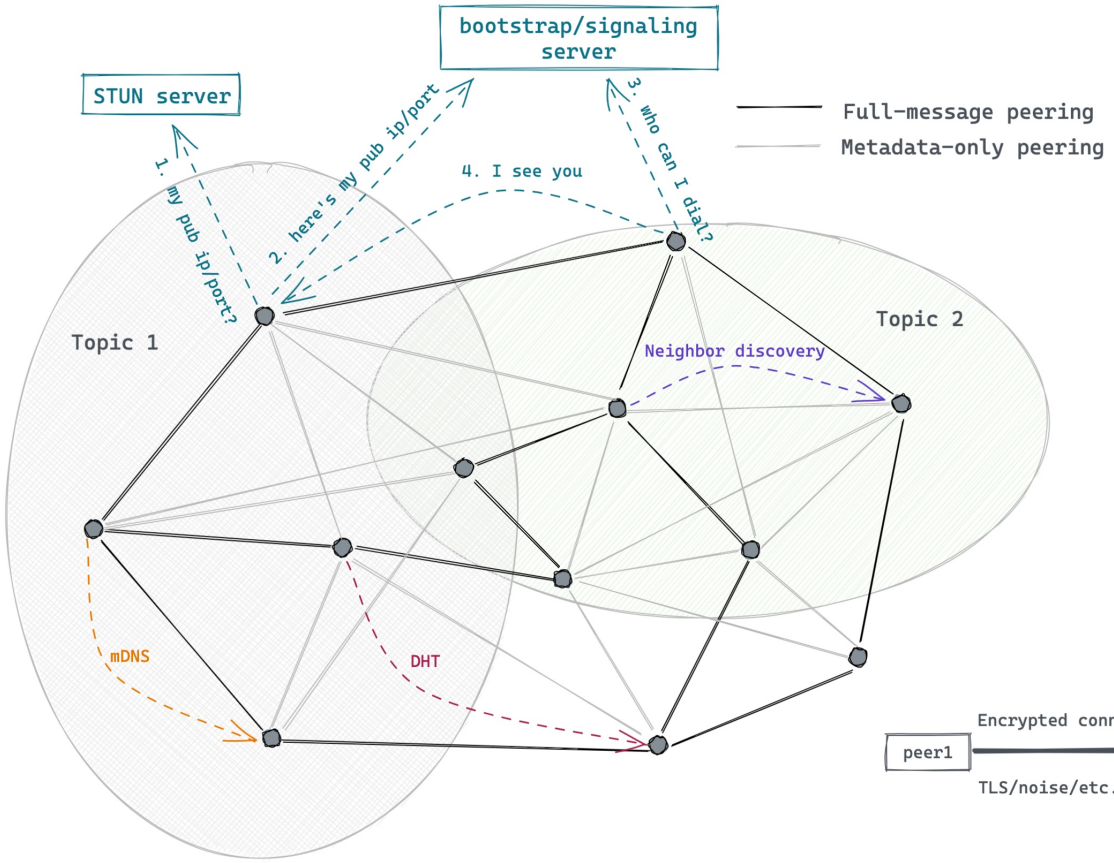
Network Stack



App for centralized network



App for p2p network



Network identity: normally a keypair

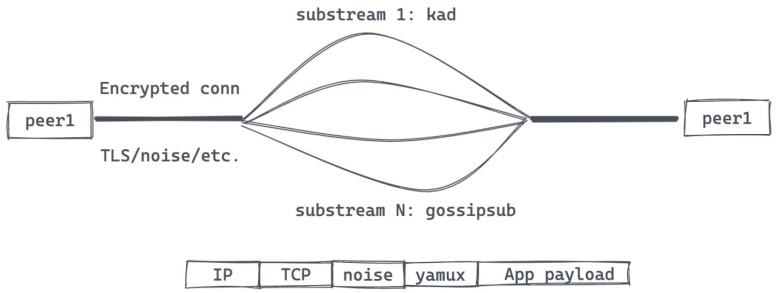
Transport: TCP/QUIC/WS/WebRTC ...

Security: TLS / noise

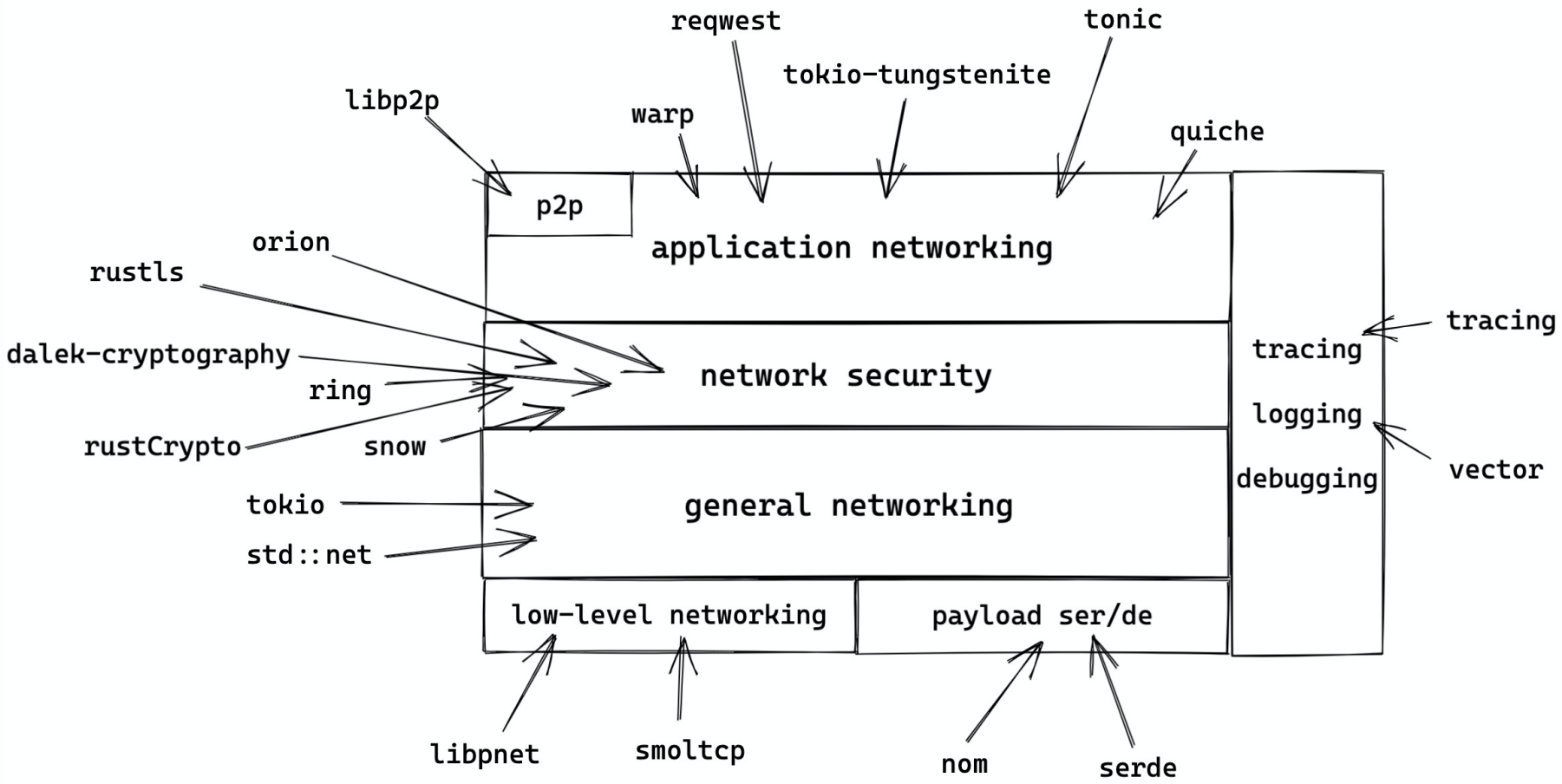
Multiplex: Yamux/Mplex/QUIC

Protocol:

- peer discovery: mDNS, bootstrap, KadDHT
- peer routing: Kad DHT
- content discovery: pubsub, Kad DHT
- NAT traversal: STUN/TURN/ICE
- App protocol: think gRPC



Rust Network Stacks



Demo: Build a TCP server (sync/async)

deps: stdlib / tokio

Demo: HTTP Client/Server

deps: reqwest / actix-web

Demo: gRPC

deps: prost / tonic

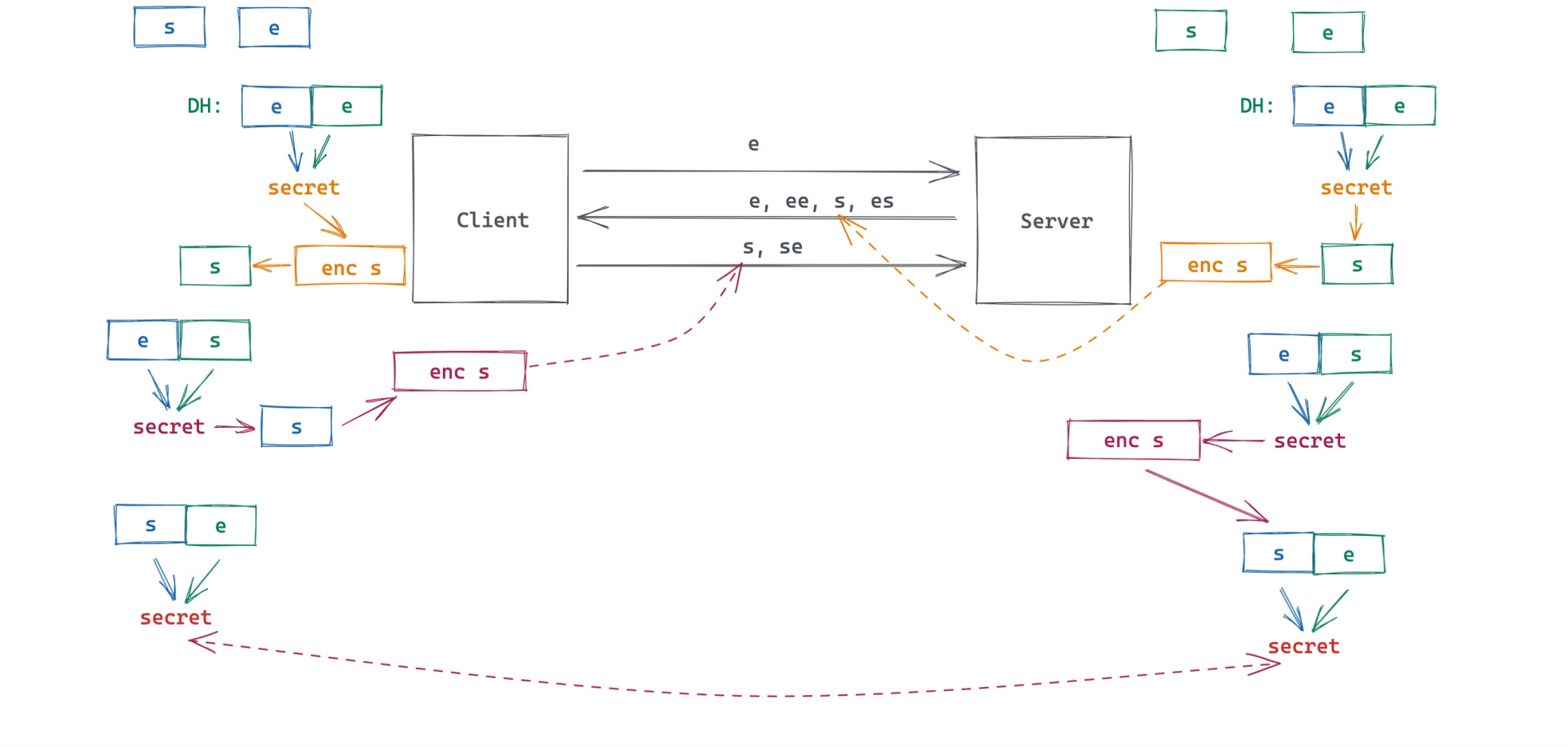
Steps to write a server

- data serialization: serde / protobuf / flatbuffer / capnp / etc.
- transport protocol: tcp / http / websocket / quic / etc.
- security layer: TLS / noise protocol / secio / etc.
- application layer: your own application logic

Network Security

TLS (skip)

Noise Protocol



Demo for noise protocol

References

- GRPC Protocol
- Are we web yet?
- Tonic: rust grpc framework

FFI with C/Elixir/Swift/Java

WASM/WASI

Rust for real-world problems

May the Rust be with you