Rust & Servo - An Introduction

Zhen Zhang izgzhen@gmail.com

May 18, 2016

Rust - The Modern System Programming Language

A Taste of Rust

- The mini interpreter
- Playground

A Taste of Rust (Cont.)

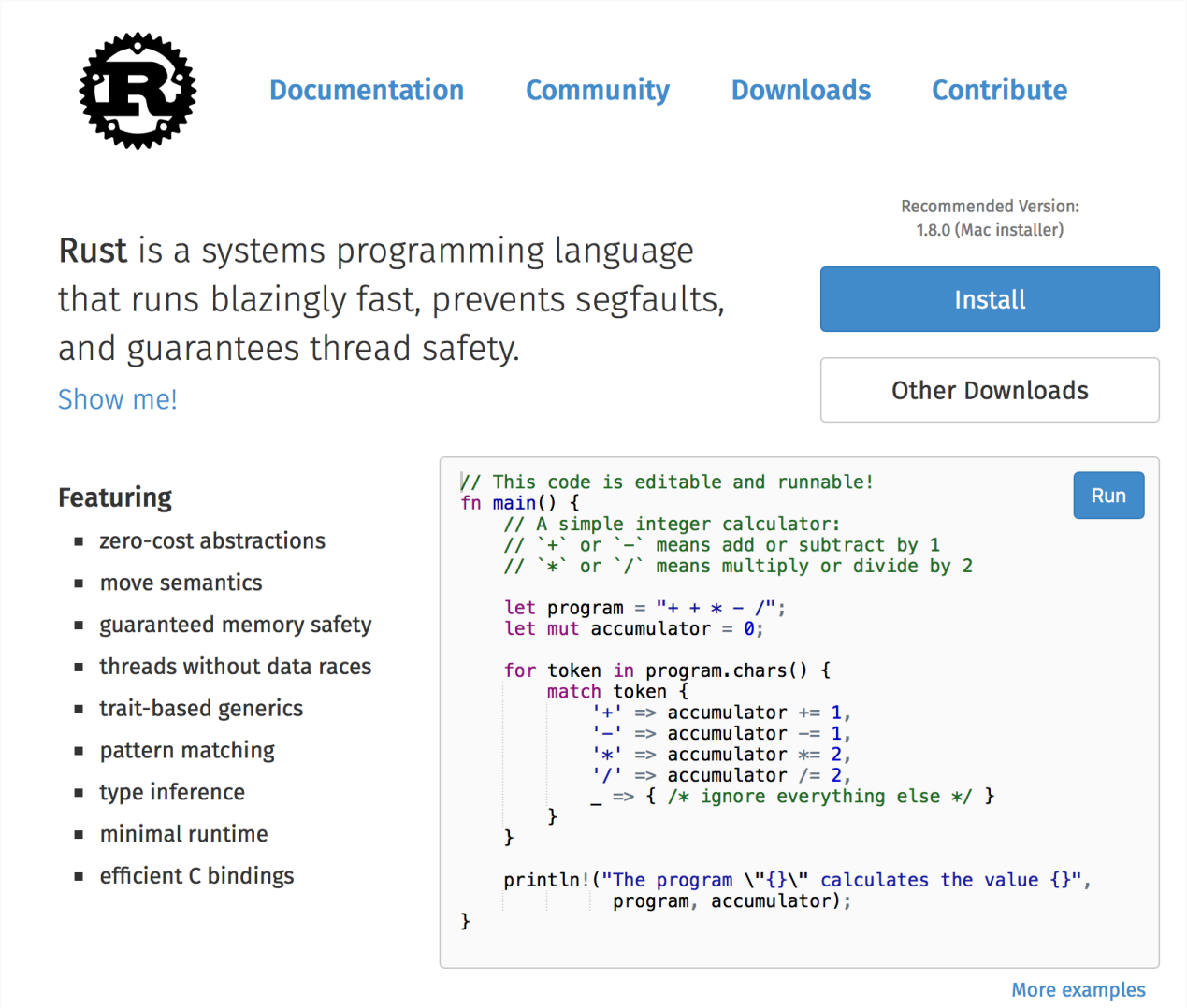


Figure 1: https://www.rust-lang.org

Why Rust is *Awesome*?

Featuring:

- **zero-cost abstractions**
- move semantics
- **guaranteed memory safety**
- threads without data races
- **trait-based generics**
- **pattern matching**
- **type inference**
- minimal runtime
- efficient C bindings

Awesomeness #1: Zero-cost abstraction

What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.
[Stroustrup, 1994]

Awesomeness #2: Trait-based generics

- The **ONLY** notion of interface
- Statically dispatched: C++ templates
- Dynamically dispatched: Java Interface
- Trait bound and inheritance

Awesomeness #2: Trait-based generics

Play

```
trait Hash {
    fn hash(&self) -> u64;
}

impl Hash for bool {
    fn hash(&self) -> u64 {
        if *self { 0 } else { 1 }
    }
}

impl Hash for i64 {
    fn hash(&self) -> u64 {
        *self as u64
    }
}

fn print_hash<T: Hash>(t: &T) {
    println!("The hash is {}", t.hash())
}

struct HashMap<Key: Hash + Eq, Value> {
    // Silly construct ... Just for demo purpose!
    k: Key,
    v: Value,
}

fn process(v: Vec<&Hash>) {
    // whatever
}
```

Figure 2: Example snippet - trait

Awesomeness #3: Memory safety

- A BIG topic in fact
- Static methods: Cyclone, MLKit ... Rust
- Dynamic methods: Garbage collection! Java, Python, C# ....
- Rust has many concepts...

A#3.1: Ownership and Borrowing

Classical problem: *Memory leak*

- If you own something, you may **borrow it out**, but you must make sure that you can get it back
- If you don't want something, you may **transfer the ownership**, but don't try to interfere with it after giving out
- If you own something, you must live *longer* than it

A#3.1: Ownership and Borrowing (Cont.)

Play

```rust
fn take(v: Vec<i32>) {
    // whatever
}

fn foo() {
    let v = vec![1, 2, 3];
    // now "foo" scope owns it
    // recall that Vec<T> points to an on-heap vector

    let v2 = v; // move ownership into v
    take(v); // another way of moving ownership
    take(v.clone()); // Clone another copy of vector

    println!("v[0] is: {}", v[0]); // error: use of moved value `v`
}
```

Figure 3: Example snippet – Ownership

A#3.2: References and Mutability

Classical problem: Concurrent data races

- Similar to C++
- Hardened with ownership control, lifetime and mutability
- Principles:
  - one or more references (`&T`) to a resource,
  - exactly one mutable reference (`&mut T`).

A#3.2: References and Mutability (Cont.)

Play

```
fn foo() {
    let mut x = 5;

    {
        let y = &mut x; // -+ &mut borrow starts here
        *y += 1;        //  |
    }                   // -+ ... and ends here

    println!("{}", x);  // <- try to borrow x here
}

fn bar() {
    let mut v = vec![1, 2, 3];

    for i in &v {
        println!("{}", i);
        v.push(34);
    }
}
```

Figure 4: Example snippet – References

A#3.3: Lifetimes

Classical problems: *Use after free*

- I acquire a handle to some kind of resource.
- I lend you a reference to the resource.
- I decide I'm done with the resource, and deallocate it, while you still have your reference.
- You decide to use the resource.

A#3.3: Lifetimes (Cont.)

Play

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                      // -+ x goes into scope
                                //  |
    {                           //  |
        let y = &5;             // ---+ y goes into scope
        let f = Foo { x: y }; // ---+ f goes into scope
        x = &f.x;               //  | | error here
    }                           // ---+ f and y go out of scope
                                //  |
    println!("{}", x);          //  |
}                               // -+ x goes out of scope
```

Figure 5: Example snippet – Lifetimes

Awesomeness #4: ADT and Pattern Matching

- Algebraic data-types (functional flavor!)
- Destruct/match: More safe/expressive/elegant then `switch` or dynamic overload

Awesomeness #4: ADT and Pattern Matching (Cont.)

Play

```
enum Msg {
    GoodMorning,
    GiveYouSomeMoney(i32),
    GoodBye,
}

fn msg_dispatcher(msg: Msg) {
    match msg {
        Msg::GoodMorning => goodmorning_handler(),
        Msg::GiveYouSomeMoney(i) => money_handler(i),
        Msg::GoodBye => goodbye_handler(),
    }
}

fn main() {
    let (chan, recv) = mpsc::channel::<Msg>();
    // send chan to somewhere...
    loop {
        if let Ok(msg) = recv.recv() {
            msg_dispatcher(msg);
        }
    }
}
```

Figure 6: Example snippet – Pattern matching

Awesomeness #5: Type System

- Local type inference
- Generics (Parametric polymorphism)
- Trait-based polymorphism
- Associated types (type families)
- **NO** support for higher-kinded types yet

Awesomeness #5: Type System (Cont.)

Play

```rust
// local type inference
fn foo(x: i32) -> i32 { // Still needs signature here
    let _y = x + 1; // infer y: i32
    0
}

// Parametric Polymorphism
struct Pair<A, B> {
    a: A,
    b: B,
}
```

Figure 7: Example snippet – Type inference and Generics

Awesomeness #5: Type System (Cont.)

```rust
// Trait-based polymorphism
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

Figure 8: Example snippet – Trait based polymorphism

```rust
// Associated types
trait Graph {
    type N;
    type E;
    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}

struct Node;
struct Edge;
struct MyGraph;

impl Graph for MyGraph {
    type N = Node;
    type E = Edge;

    fn has_edge(&self, _n1: &Node, _n2: &Node) -> bool {
        true
    }

    fn edges(&self, _n: &Node) -> Vec<Edge> {
        Vec::new()
    }
}
```

Figure 9: Example snippet – Associated types

Rust's advantages over C++

1. `cargo` ecosystem
2. Safety
3. Simplicity of design
4. Modern language constructs

But Rust is not perfect

1. Value-returning flavor error handling
2. Lack of OOP
3. Too many macros
4. Steep learning curve (even for C/C++ hackers)
5. Still in fast evolving
6. Too many types of pointers, closures, strings ....

Rust in the wild

- Servo Browser Engine, Mozilla Foundation
- Rust Infrastructure, Mozilla Foundation
- Ruby app profiler, Skylight Inc.
- Infra, DropBox Inc.
- Xi Editor, Google Employee
- Redox OS

Misc

- Resource:
  - *The Book*
  - *Rust For Systems Programmers*
  - USTC mirror of latest binaries
  - Jump into the wheel-reinventing battle against C/C++/Go/D/... on GitHub!
- Discussions(in Chinese):
  - 如何看待 Dropbox 从 Go 转向 Rust ?
  - Rust 所宣称的 zero-cost abstractions 是怎么回事?

Servo - The Modern Parallel Browser Engine

Motivation

Why Mozilla want to invent a new language to write something to replace Gecko (in C++)?

- Takes advantage of **parallelism** at many levels
- Eliminating common sources of bugs and security vulnerabilities associated with incorrect **memory management and data races**

How does the effort pay off?

Performance and Benchmark

- *Mozilla's Servo Lets Rust Shine*
- *Mozilla's Servo Engine Is Crazy Fast Compared To Gecko*
- Benchmarking

Performance and Benchmark (Cont.)

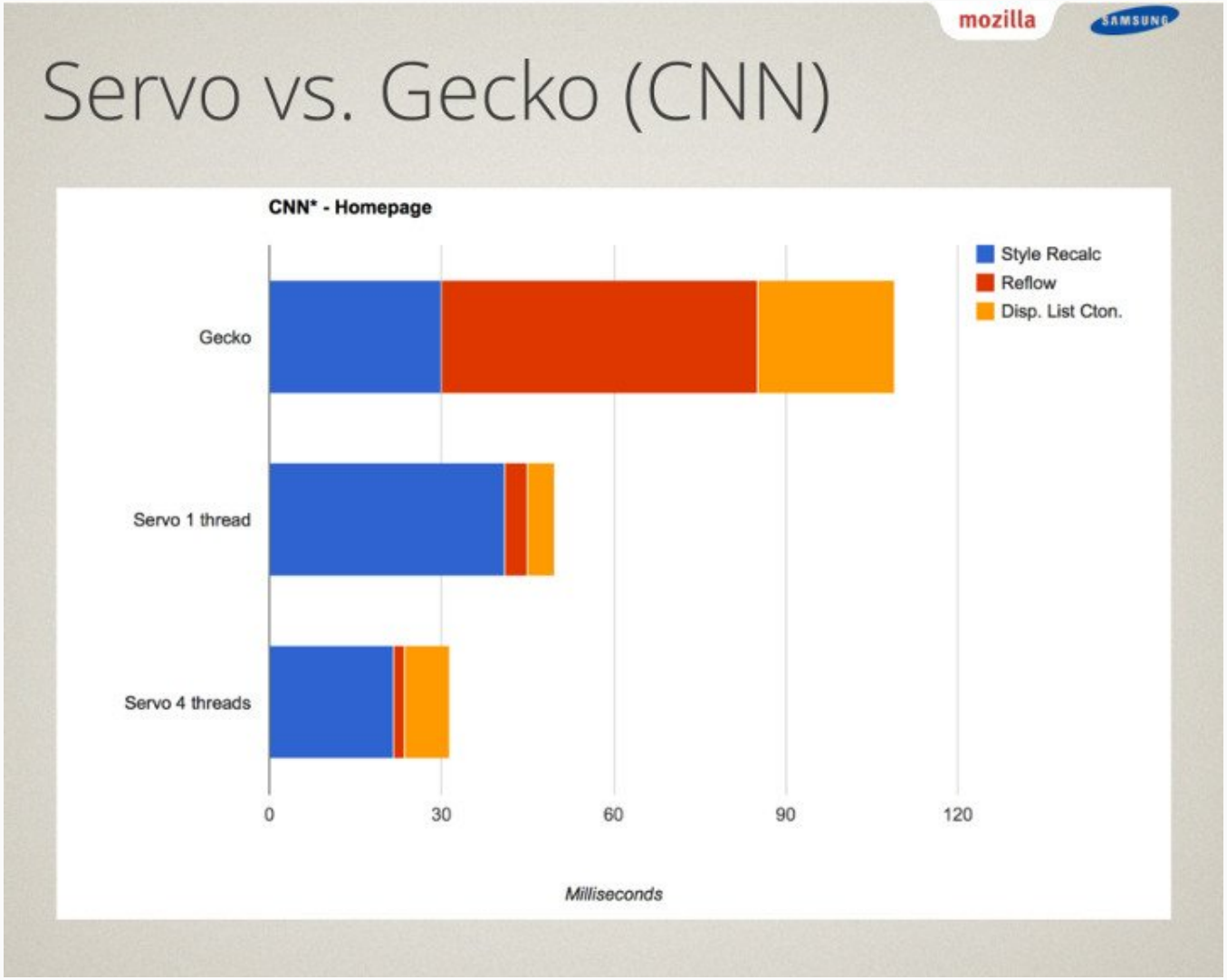


Figure 10: Servo v.s. Gecko

Browser Internals 101

- Layout engine
- JS runtime
- Network/Resource management
- UI

Browser Internals 101 (Cont.)

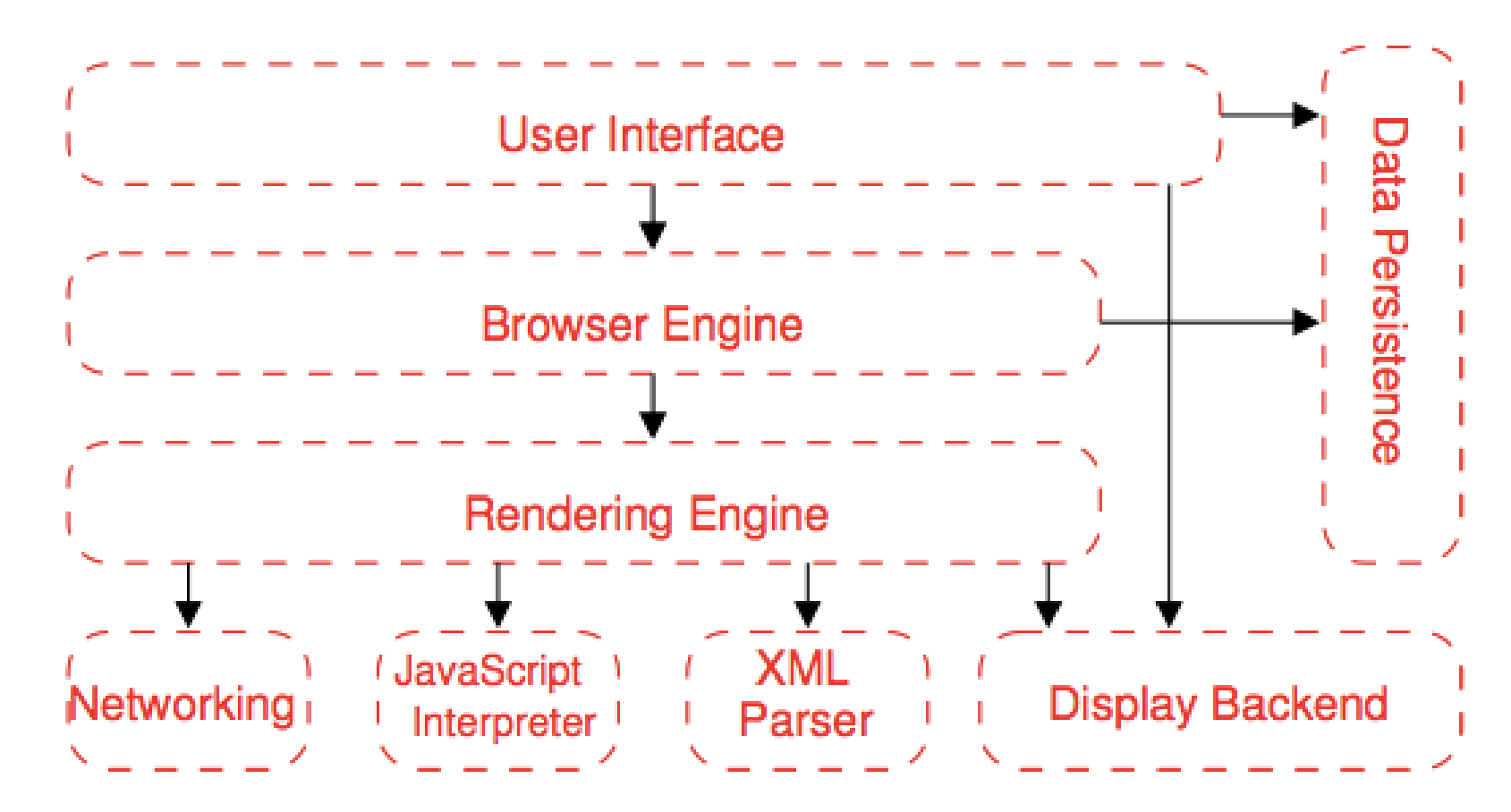


Figure 11: Browser Reference Architecture

Browser Internals 101 (Cont.)

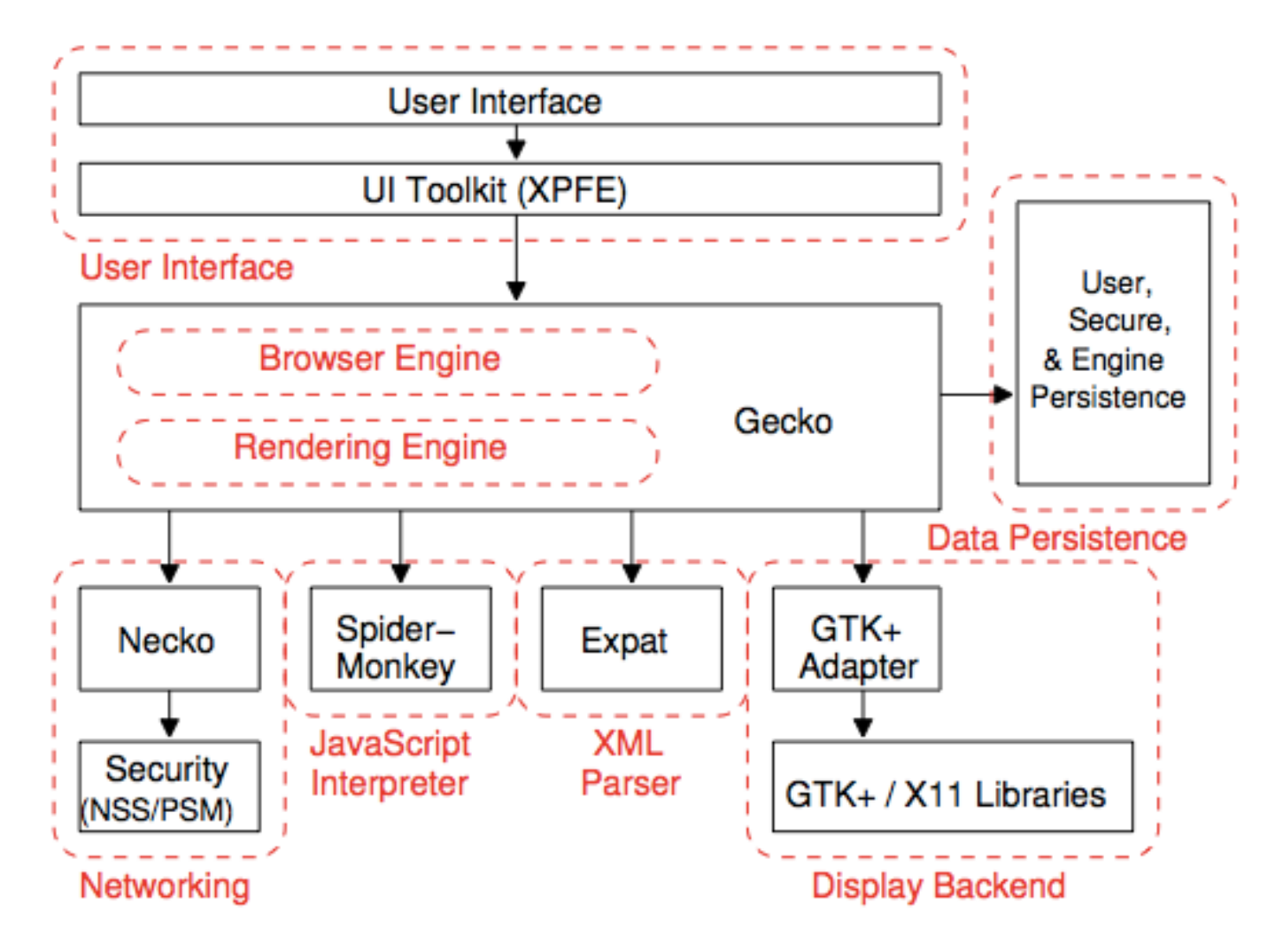


Figure 12: FireFox Architecture

Servo's Architecture

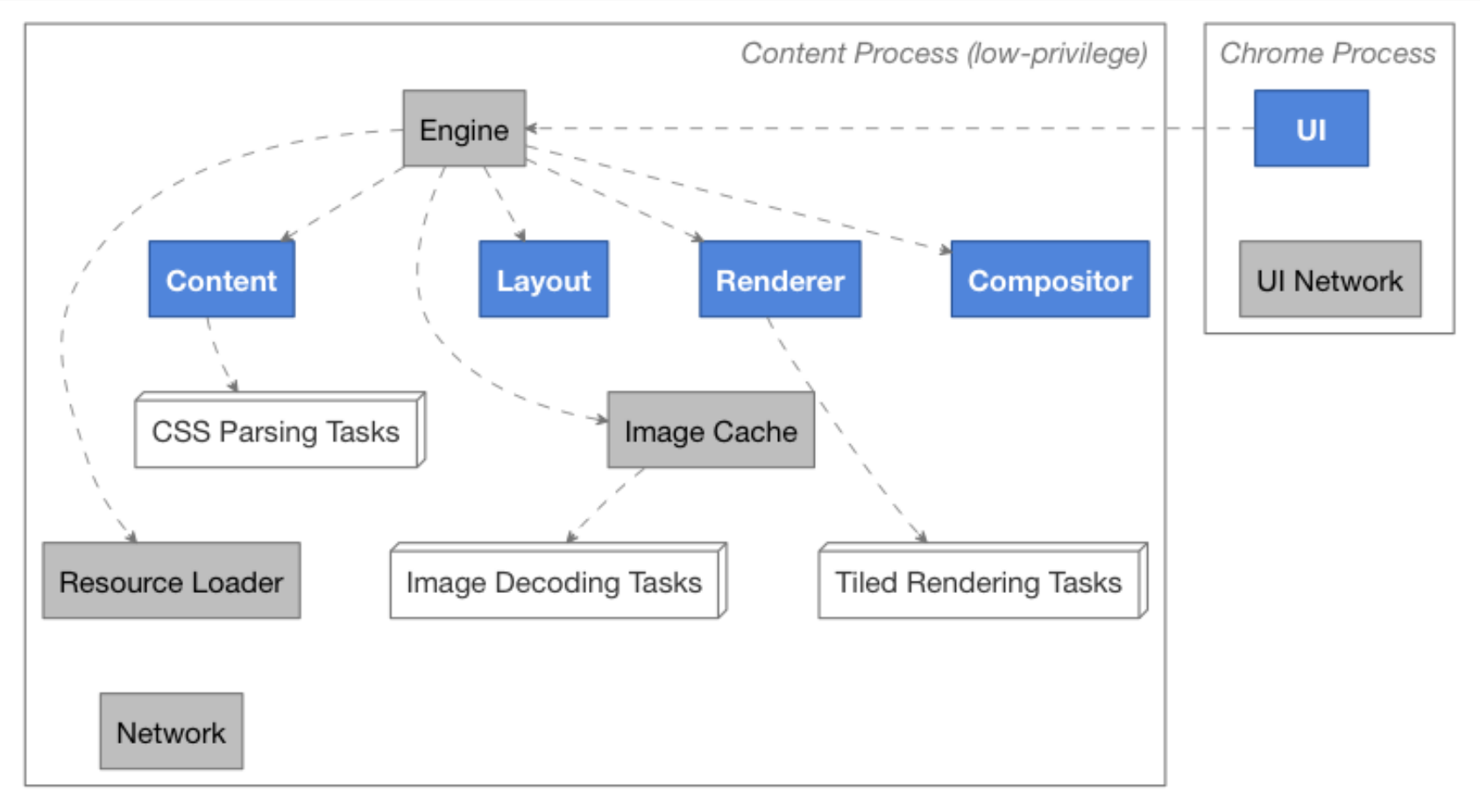


Figure 13: Task supervision diagram

Servo's Architecture (Cont.)

Each constellation manages a *pipeline* of tasks:

- Script
- Layout
- Renderer
- Compositor

`https://github.com/servo/servo/wiki/Design`

Servo's Strategies for parallelism and concurrency

- Task-based architecture
- Concurrent rendering
- Tiled rendering
- Layered rendering
- Image decoding
- GC JS concurrent with layout

Personal Experience of Hacking on Servo

- Very mature infra, easy to use tools
- You can learn a lot
- Nice and supportive people in Mozilla
- Very friendly to new contributors

Roadmap

- Ship one Rust component in Firefox Nightly, riding the trains
- Experiment with the uplift of a major piece of Servo into Gecko
- June tech demo

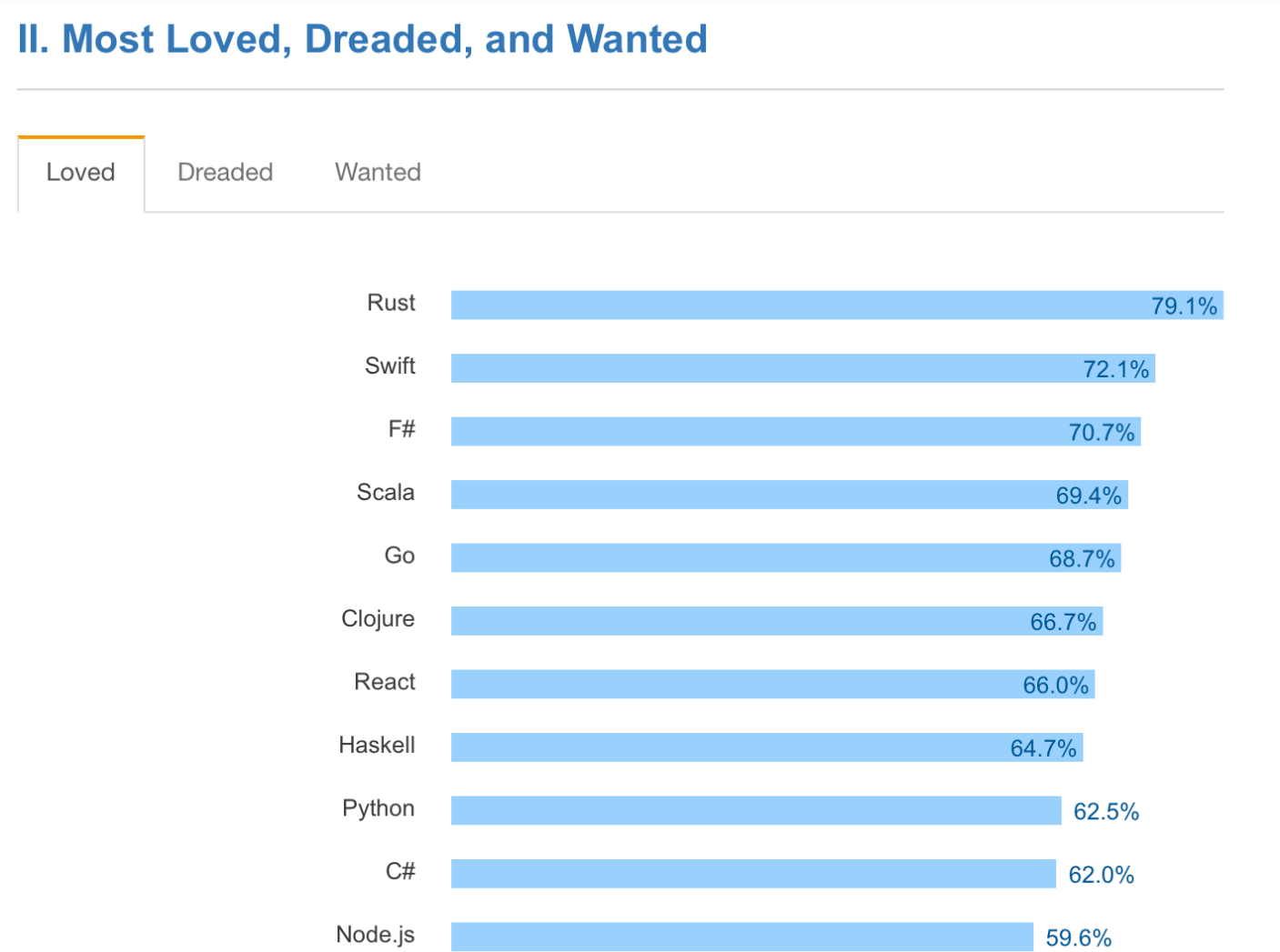
Servo needs you

- Step 1: Learn Rust
- Step 2: Try to solve an essay issue
- Step 3: First PR landed!

Backup Slides

Most Loved PL in 2016 is Rust

https://stackoverflow.com/research/developer-survey-2016



Rust *just* turns one year old

Rust 1.0 was released at May 15, 2015. Here is a post on the past year.

One year of Rust

May 16, 2016 • Aaron Turon

Rust is a language that gives you:

- uncompromising performance and control;
- prevention of entire categories of bugs, including classic concurrency pitfalls;
- ergonomics that often rival languages like Python and Ruby.

It's a language for writing highly reliable, screamingly fast software—and having fun doing it.

And yesterday, Rust turned one year old.