

Kaspa Innovation Summit

Hong Kong, October 27th 2024

- **Overview of Rust**
Overview of the Rust programming language
- **Building Modern Multi-platform SDKs in Rust**
Overview of the Kaspa SDK
- **Smart Contract Design in Kaspa L1**
Overview of Kaspa as a transaction sequencer & zero knowledge proofer
- **L2 Research & Prototyping**
Overview of the Sparkle Project

Special thanks to the
Kaspa Ecosystem Foundation

Special thanks to

@hashtag, @starkbamse, @143672, @coderofstuff, @Someone234,
@Elichai2, @misutton, @tiram, @smartgoo, @OneBananaGirl,
@KaffinPX, @Ds/jwj, @matoo, @supertypo and many others.

About Myself

<https://aspectron.org>

@aspect

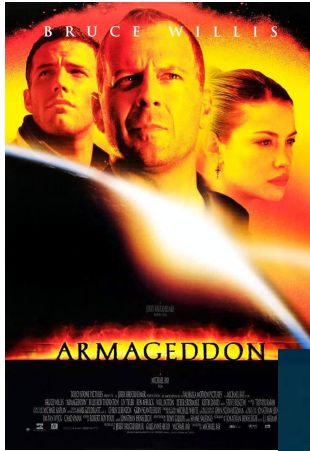
Software Architect 1988 ... present :/

Background & interests includes:

- Computer Graphics and Visual Effects
- 3d Graphics and Virtual Reality
- High-performance GPU processing (CG)
- Embedded & Mobile systems
- Content delivery systems
- Virtual Machines & Operating Systems
- Industrial automation
- Financial data processing
- Cryptocurrency technologies
- Smart Contract technologies
- Sharing knowledge and experience

Over 3 decades of C++ & software engineering...

1990s



2000s



Using consumer gaming hardware.

30mm USD of advertising revenue can be lost in case of a technical problem.



Scalingbitcoin



Bitcoinedge

<https://scalingbitcoin.org>

<https://bitcoinedge.org>





I've seen things you people wouldn't believe...



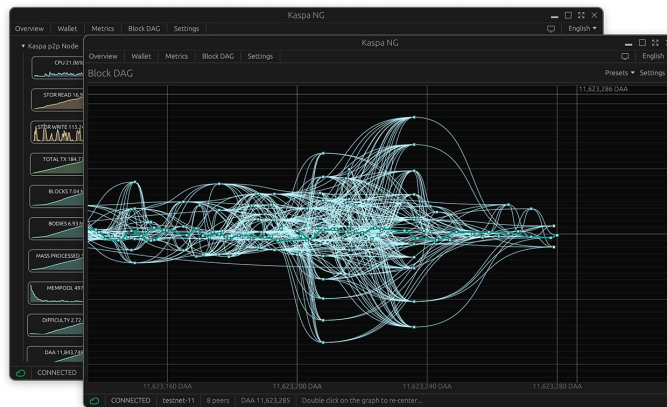
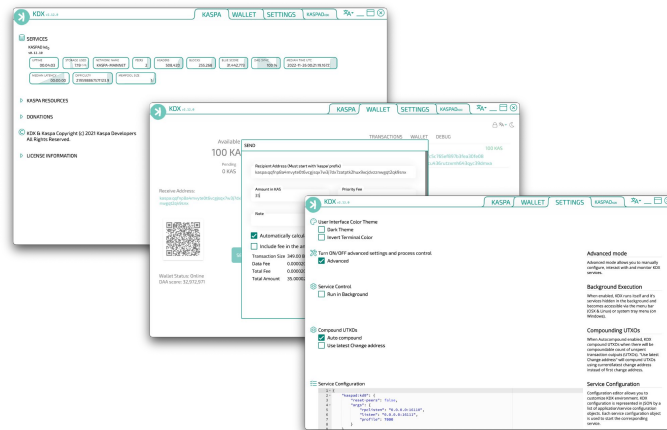
Advisor and Contributor to Kaspa Core

Legacy

- KDX (accessibility)
- Legacy TypeScript SDK

Rusty Kaspa

- Co-designed RPC infrastructure
- Consensus infrastructure testing
- Rusty Kaspa framework structure
- Rusty Kaspa Rust SDK + WASM32 SDK
- Core ecosystem support & coordination
- Kaspa NG (security)



Building Modern Multi-platform SDKs in Rust

Key Rust features and the Rusty Kaspas Framework SDK

Kaspa Innovation Summit
Hong Kong, October 27th 2024

A dark blue diagonal gradient bar that starts from the bottom left and extends towards the top right, covering the lower half of the slide.

A bit about the Rust programming language

Rust Origins

Rust began as a personal project in 2006 by Mozilla Research employee Graydon Hoare, named after the group of fungi that are "over-engineered for survival".

Mozilla began sponsoring the project in 2009, and would employ a dozen engineers to work on it full time over the next ten years.

Around 2010, work shifted from the initial compiler written in OCaml to a self-hosting compiler based on LLVM written in Rust. The new Rust compiler successfully compiled itself in 2011.

Rust Origins

Rust is a relatively young language and is a bit "late to the game".

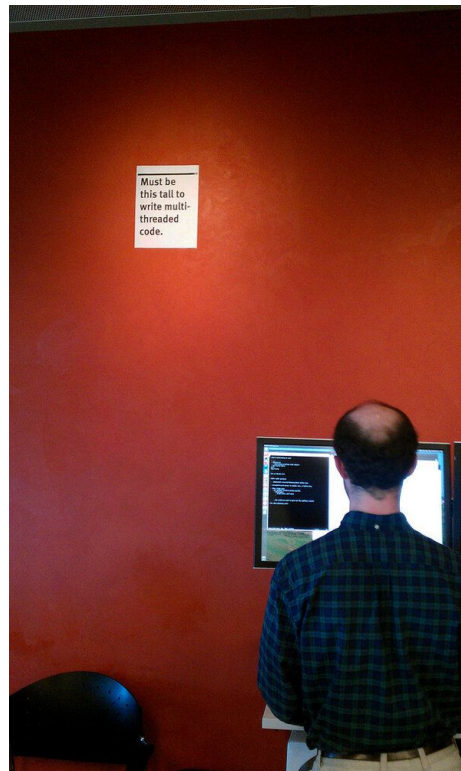
Kind of like Kaspa (in a grand scheme of things)...

Being late gives one advantages as one can learn from mistakes of others and work on avoiding them.

Rust Origins

Writing safe concurrent code is a "rocket science".

Picture taken in the Mozilla San Francisco office.



What Rust offers

(from the project management standpoint)

- Software Reliability - no unexpected behavior.
- Long-term sustainability - ease of long-term codebase management in large projects.
- Integrated documentation
- Ability to have the same codebase that targets different environments via different build targets (or via bindings): Native, WASM, Python, Dart, Swift, etc.
- Ability to bind to existing C-style APIs using FFI

Rust Toolchain

Toolchain is focused on Compiler Guided Development

- Rustc teaches you
- cargo check & cargo clippy
- Rust Analyzer

Various IDE Extensions

In Visual Studio Code these include:

- Rust Analyzer
- Error Lens
- Even Better TOML
- Dependi

Autocomplete & AI (If you have geographical constraints, get a VPN)

- Tab9
- Github Copilot
- **Cursor** (w/Claude)
- Claude (by Anthropic is better than ChatGPT as of Q4 2024)

Crate ecosystem

<https://crates.io>

Rust has an amazing crate ecosystem

It is a "Magic Wonderland" (as described by Michael Sutton) of stable, reliable, well-documented solutions, especially those focused on algorithmic processing.

- All sorts of networking (Axum HTTP benched to handle 500k req/s)
- Cryptography & Encryption
- Bindings and interop
- High-efficiency algorithms (especially caching, lock-free high-performance implementations etc.)
- Bindings to existing popular C++ libraries

Huge growing ecosystem of contributors and builders.

Default "go to" for cryptocurrency & cryptography projects.

Diving into Rust internals

Just some highlights that I find valuable.

Enums

Enums in Rust

- **Multiple Variants:** Enums allow a type to be one of several different variants, each potentially holding different data.
- **Strong Type Safety:** Each variant can store different types of values, offering strong compile-time checks.
- **Flexible Data Representation:** You can mix and match variants with or without data in a single enum.
- **Pattern Matching:** Rust's powerful pattern matching works seamlessly with enums, making it easy to work with their variants.
- **Clear and Readable Code:** Enums provide a clear and concise way to represent state or data that can take multiple forms.

Example

```
enum Message {  
    Quit, // No data  
    Move { x: i32, y: i32 }, // Struct-Like data  
    Write(String), // Tuple-Like data  
    ChangeColor(i32, i32, i32), // Tuple-Like data  
}  
  
fn process_message(msg: Message) {  
    match msg {  
        Message::Quit => println!("Quit message"),  
        Message::Move { x, y } => println!("Move to coordinates: ({}, {})",  
x, y),  
        Message::Write(text) => println!("Write message: {}", text),  
        Message::ChangeColor(r, g, b) => println!("Change color to RGB({},  
{}, {})", r, g, b),  
    }  
}
```

Pattern Matching

```
enum Message {
    Quit, // No data
    Move { x: i32, y: i32 }, // Struct-Like data
    Write(String), // Tuple-Like data
    ChangeColor(i32, i32, i32), // Tuple-Like data
}

fn process_message(msg: Message) {
    match msg {
        missing match arm: `ChangeColor(_, _, _)` not covered
        Message::Quit => println!("Quit message"),
        Message::Move { x, y } => println!("Move to coordinates: ({}, {})", x, y),
        Message::Write(text) => println!("Write message: {}", text),
    }
}

error[E0004]: non-exhaustive patterns: `input::Message::ChangeColor(_, _, _)` not
covered
--> consensus/client/src/input.rs:20:11

20 |         match msg {
    |             ^^^ pattern `input::Message::ChangeColor(_, _, _)` not covered

note: `input::Message` defined here
--> consensus/client/src/input.rs:13:6

13 | enum Message {
    | ^^^^^^^
...
17 |     ChangeColor(i32, i32, i32), // Tuple-like data
    | ----- not covered
= note: the matched value is of type `input::Message`
help: ensure that all possible cases are being handled by adding a match arm with a
wildcard pattern or an explicit pattern as shown

23 ~         Message::Write(text) => println!("Write message: {}", text),
24 ~         input::Message::ChangeColor(_, _, _) => todo!(),
    |
```

Conditional Compilation

`#[cfg()]`

(we call them gates)

- **Conditional Compilation:** The `cfg` attribute in Rust allows you to conditionally include or exclude code based on compilation flags, target architectures, operating systems, or custom configuration options.
- **Cross-Platform Compatibility:** It enables writing platform-specific code that is only compiled on certain operating systems (e.g., Windows, Linux, macOS).
- **Custom Build Features:** You can enable or disable features at compile time using Cargo's feature flags, providing flexibility in building different versions of the same crate.
- **Performance Optimization:** Helps in optimizing builds by excluding unnecessary code for specific platforms or configurations, leading to faster compile times and smaller binaries.

Example:

```
#[cfg(target_os = "windows")]
fn windows_only_function() {
    // Windows-specific code here
}
#[cfg(feature = "my_feature")]
fn feature_specific_function() {
    // Code for when `my_feature` is enabled
}
```

Also important to note the `cfg_if` crate.

Borrow Checker

- **Memory Safety Without Garbage Collection:** Rust's borrow checker ensures memory safety by enforcing rules around references and lifetimes, preventing common issues like dangling pointers, data races, or double frees, without needing a garbage collector.
- **Ownership Enforcement:** The borrow checker enforces Rust's ownership model, where data can have either one mutable reference or any number of immutable references at a time, preventing simultaneous modification and access to data, reducing bugs.
- **Compile-Time Error Detection:** Instead of discovering memory issues at runtime, the borrow checker catches them at compile time, reducing the possibility of crashes or undefined behavior.
- **Zero-Cost Abstraction:** This safety is achieved without runtime overhead, ensuring high performance while still guaranteeing memory correctness.

Example

```
fn borrow_example() {  
    let mut x = 5;  
  
    {  
        let y = &x; // immutable borrow of `x`  
        println!("{}", y); // OK, only reading `x`  
    }  
  
    let z = &mut x; // mutable borrow of `x`, possible only when no  
    // other borrows exist  
    *z += 1;        // OK, modifying `x`  
}
```


Absence of "NULL"

- **Eliminates Null Reference Bugs:** In Rust, there are no null pointers by default. Instead of nullable pointers, Rust uses the `Option<T>` enum to represent values that can either be `Some(T)` or `None`, eliminating null reference bugs (a common source of crashes in other languages).
- **Safer Error Handling:** By using `Option<T>` and `Result<T, E>`, Rust enforces explicit handling of missing values or potential errors, ensuring developers consciously deal with “null-like” situations instead of causing runtime exceptions like null pointer dereferencing.
- **Compile-Time Safety:** The Rust compiler enforces handling of `Option<T>` or `Result<T, E>`, preventing accidental dereferencing of null-like values, catching potential issues at compile time instead of runtime.
- **More Predictable Code:** Since null values are explicitly represented and handled, the absence of null pointers in Rust leads to more predictable, reliable, and maintainable codebases.

Example Using `Option<T>`

```
fn find_user(id: u32) -> Option<&str> {  
    if id == 1 {  
        Some("Alice")  
    } else {  
        None // Explicitly handles "null" case  
    }  
}  
  
fn main() {  
    let user = find_user(1);  
    match user {  
        Some(name) => println!("Found user: {}", name),  
        None => println!("No user found"), // Explicitly handle "null" case  
    }  
}
```

Explicit Mutability

- **Explicit Mutability Declaration:** In Rust, variables are immutable by default, and mutability must be explicitly declared using the `mut` keyword. This makes it clear when a variable is intended to be changed, improving code readability and intent.
- **Prevents Unintended Modifications:** Since mutability is explicit, it prevents accidental changes to variables that are meant to remain constant, reducing bugs related to unintended state changes.
- **Enhanced Code Safety:** By enforcing clear mutability, Rust allows the compiler to optimize code better and ensures that developers are more aware of when data can be modified, reducing data races in concurrent programming.
- **Immutable by Default:** This approach aligns with functional programming paradigms where immutability is encouraged, promoting safer and more predictable code behavior.

Example

```
fn main() {  
    let x = 5; // Immutable by default  
    // x = 6; // Compile-time error: cannot assign to immutable variable  
    let mut y = 10; // `mut` makes the variable mutable  
    y = 20;         // OK, mutable variable can be modified  
    println!("y is now {}", y);  
}
```

- **Function Arguments:** You can also mark function parameters as mutable if you intend to modify the arguments, further promoting clarity in API design.

Strict Type System

- **Prevents Type Errors at Compile Time:** Rust's strict static typing system ensures that all type mismatches are caught at compile time, preventing many classes of runtime errors and improving overall code correctness.
- **Better Code Safety:** With strict type enforcement, Rust prevents unsafe operations such as passing the wrong type to a function or misusing data, enhancing memory and type safety.
- **No Implicit Type Conversions:** Rust avoids implicit type conversions (coercions) between types, such as between integers and floating points, preventing unexpected behavior and increasing clarity in how types are handled.
- **Enhanced Readability and Maintainability:** A strict typing system makes code more predictable and readable by enforcing clear contracts for function parameters and return types, making it easier to reason about what a function does.
- **Type Inference:** Despite its strictness, Rust offers type inference, meaning that the compiler can infer the type in many cases, reducing verbosity while still maintaining safety.
- **Generics:** Rust allows for strict yet flexible typing using generics, enabling code reuse while maintaining type safety.

Traits

Traits

- **Define Shared Behavior:** Traits are similar to interfaces in other languages. They define a set of methods or behaviors that types can implement.
- **Polymorphism:** Traits allow for polymorphism by enabling functions to accept any type that implements a particular trait, making your code more flexible.
- **Trait Bounds:** Functions can specify that they only accept types that implement certain traits, enabling compile-time checks for correctness.

Example

```
trait Speak {  
    fn speak(&self);  
}  
  
struct Dog;  
impl Speak for Dog {  
    fn speak(&self) { println!("Woof!"); }  
}  
  
struct Cat;  
impl Speak for Cat {  
    fn speak(&self) { println!("Meow!"); }  
}  
  
fn animal_speak<T: Speak>(animal: T) {  
    // The animal generic is restricted by trait  
    animal.speak();  
}  
  
fn main() {  
    let dog = Dog;  
    let cat = Cat;  
    animal_speak(dog); // Outputs: Woof!  
    animal_speak(cat); // Outputs: Meow!  
}
```

Explanation: The Speak trait defines a speak method. Both Dog and Cat structs implement this trait, and the animal_speak function can accept any type that implements Speak.

async Rust

async Rust

- **Concurrency without Threads:** Async Rust allows for concurrent programming without the overhead of creating multiple OS threads. Instead, it uses **async tasks** that can be run on a single or limited number of threads.
- **Non-blocking I/O:** Asynchronous code allows I/O operations (such as network requests) to be non-blocking, which means they can be suspended while waiting for results, freeing up resources for other tasks.
- **async and await Keywords:** These keywords are used to define and run asynchronous functions, allowing Rust to pause the execution of tasks and resume them when the awaited operation completes.
- **Executor Required:** Rust's async code does not automatically run in parallel; it requires an **executor** (e.g., tokio or async-std) to poll and run tasks concurrently.

Benefits

- **Highly Efficient Concurrency:** Async Rust enables you to perform many I/O-bound tasks concurrently without the cost of thread switching.
- **Memory Efficiency:** By using lightweight tasks instead of threads, async Rust reduces memory overhead and improves performance in systems with high I/O needs, like web servers and network clients.

Example

```
use async_std::task;
async fn fetch_data() -> String {
    // Simulate a non-blocking I/O operation

    task::sleep(std::time::Duration::from_secs(2)).await;
    String::from("Data fetched")
}
#[async_std::main]
async fn main() {
    println!("Fetching data...");
    let result = fetch_data().await; // Await the async
function
    println!("{}", result);          // Outputs: Data
fetched
}
```

async Rust

async Rust is very important in the Browser (WASM32 browser) environment.

Non-blocking execution on fine-granular tasks means no blocking on user input.

Special considerations:

``yield_executor()`` implemented as
``requestAnimationFrame()``

This forcefully suspends async Rust executor providing relief for Browser UI updates during time-consuming tasks.

Channels

Channels

- **Concurrency Communication:** Channels provide a way for different threads to communicate by sending messages between them.
- **Thread-Safe:** Rust channels are designed to be thread-safe, ensuring no data races occur when transferring data between threads.
- **Two Types:** Rust provides **bounded** and **unbounded** channels for controlling how much data can be sent before blocking occurs.
- **Ownership Transfer:** Channels allow moving data from one thread to another, ensuring that only one thread owns the data at any given time.

Example

```
use async_std::task;
use async_std::channel;
#[async_std::main]
async fn main() {

    let (tx, rx) = channel::unbounded();

    // Spawn an asynchronous task
    task::spawn(async move {
        let val = String::from("Hello from async task!");
        tx.send(val).await.unwrap(); // Send value asynchronously
    });

    // Receive value asynchronously
    let received = rx.recv().await.unwrap();
    println!("Received: {}", received); // Outputs: Received: Hello from async task!
}
```

Concurrency Without Shared State: Channels enable communication without needing shared mutable state.

Rayon

Rayon

- **Data Parallelism:** Rayon allows for parallel iteration over collections (like arrays, vectors) with minimal changes to your code.
- **Ergonomic API:** You can turn sequential iterators into parallel iterators with a simple method call (`.par_iter()`), making the transition to parallelism easy.
- **Safe Parallelism:** Rayon ensures safe concurrency by leveraging Rust's ownership and type system, preventing data races and ensuring thread safety.
- **Automatic Load Balancing:** Rayon dynamically balances the workload across threads, ensuring optimal usage of system resources.

Example

```
use rayon::prelude::*;
fn main() {
    let numbers: Vec<i32> = (1..1000).collect();

    // Parallel iteration using `.par_iter()`
    let sum: i32 = numbers.par_iter().sum();

    println!("Sum: {}", sum); // Sum is calculated in parallel
}
```

Key Features

- **Parallel Iterators:** Convert any iterator into a parallel iterator using `.par_iter()` or `.par_iter_mut()` for mutable access.
- **Parallel Processing:** Operations like `map`, `filter`, `reduce`, and `for_each` can be parallelized easily using Rayon.
- **Parallel Sorting:** Rayon also supports parallel sorting with `.par_sort()` and `.par_sort_by()` methods for efficient multi-threaded sorting.

Declarative Macros

Declarative Macros (`macro_rules!`)

- **Pattern-Based Code Generation:** Uses pattern matching to generate repetitive or complex code at compile time.
- **Zero Runtime Cost:** Macros are expanded at compile time, so they introduce no overhead during execution.
- **Improved Code Reusability:** Reduces code duplication by abstracting common patterns.
- **Flexible and Powerful:** Can handle complex inputs and expand to various Rust constructs.

Example

```
macro_rules! say_hello {  
    () => {  
        println!("Hello, world!");  
    };  
}  
  
fn main() {  
    say_hello!(); // Expands to `println!("Hello,  
world!");`  
}
```

Derive Macros

```
#[derive(Describe, Eq, PartialEq, Debug, Clone, Copy)]
#[caption = "Main menu"]
pub enum Main {
    /// Status and logs
    #[describe("Service status")]
    Status,
    /// Configure services
    #[describe("Configure")]
    Configure,
    /// Software updates
    #[describe("Updates")]
    Update,
    /// Uninstall services, delete data, etc.
    #[describe("Advanced")]
    Advanced,
    /// Exit the program
    Exit,
}
```

```
impl Main {
    pub fn caption() -> &'static str {
        "Main menu"
    }
    pub fn iter() -> impl Iterator<Item = &'static Self> {
        [Main::Status, Main::Configure, ...].iter()
    }
    ...

    pub fn describe(&self) -> &'static str {
        match self {
            Main::Status => "Service status",
            Main::Configure => "Configure",
            ...
        }
    }
    pub fn rustdoc(&self) -> &'static str {
        match self {
            Main::Status => "Status and logs",
            Main::Configure => "Configure services",
            ...
        }
    }
    ...
}
```

Attribute Macros

- Similar to `#[derive()]` macros
- Allow modification of the underlying AST (abstract syntax tree)

Example: `#[wasm_bindgen]`

- Detects underlying syntax.
- For structs, creates needed scaffolding, implements getters.
- For `impl` blocks, creates bindings for each function and provides automatic argument and return value conversion.
- For type declarations wrapped in `extern "C"` calling convention, creates corresponding TypeScript types.

... and much much more.

Proc Macros in Rusty Kaspas RPC

```
impl RpcApi for KaspasRpcClient {
```

```
    build_wrpc_client_interface!(
```

```
        RpcApiOps,
```

```
        [
```

```
            Ping,
```

```
            AddPeer,
```

```
            GetBlock,
```

```
            GetBlockCount,
```

```
            GetBlockDagInfo,
```

```
            GetBlocks,
```

```
            GetHeaders,
```

```
            GetInfo,
```

```
            ...
```

```
    ---
```

```
    Xxx -> XxxRequest, XxxResponse, xxx_call + fn body
```

```
    (Ident manipulation, snake case conversion)
```



```
impl RpcApi for KaspasRpcClient {
```

```
    async fn ping_call(&self, request : PingRequest) -> RpcResult<PingResponse> {
```

```
        let response: ClientResult<PingResponse> = self.inner.rpc.call(RpcApiOps::Ping, request).await;
```

```
        Ok(response.map_err(|e| e.to_string()))
```

```
    }
```

```
    async fn add_peer_call(&self, request : AddPeerRequest) -> RpcResult<AddPeerResponse> {
```

```
        let response: ClientResult<AddPeerResponse> = self.inner.rpc.call(RpcApiOps::AddPeer, request).await;
```

```
        Ok(response.map_err(|e| e.to_string()))
```

```
    }
```

```
    async fn get_block_call(&self, request : GetBlockRequest) -> RpcResult<GetBlockResponse> {
```

```
        let response: ClientResult<GetBlockResponse> = self.inner.rpc.call(RpcApiOps::GetBlock, request).await;
```

```
        Ok(response.map_err(|e| e.to_string()))
```

```
    }
```

```
    async fn get_block_count(&self, request : GetBlockCountRequest) -> RpcResult<GetBlockCountResponse> {
```

```
        let response: ClientResult<GetBlockCountResponse> = self.inner.rpc.call(RpcApiOps::GetBlockCount, request).await;
```

```
        Ok(response.map_err(|e| e.to_string()))
```

```
    }
```

```
    ...
```

Rust Ownership & Bindings

When working with bindings, Rust ownership model exhibits itself in unexpected ways.

If you have something in Rust, to give it elsewhere, you have to clone the data, ...or otherwise provide some type of a reference.

If you clone the data and give it up, you no longer own it - this creates problems with features like "getters" and "setters" in JavaScript

Rust Ownership & Bindings

```
#[wasm_bindgen]
#[derive(Clone)]
struct A {
    // 'pub' makes automatic getter on 'b'
    pub b : B;
}
```

```
#[wasm_bindgen]
#[derive(Clone)]
struct B {
    pub c : C;
}
```

```
#[wasm_bindgen]
#[derive(Clone)]
struct C {
    pub value : u32;
}
```

```
// Exposed to JavaScript
class A { b : B }
class B { c : C }
class C { value : number }
```

```
let a = new A();
a.b.c.value = 42;
console.log(a.b.c.value); // 0 :(
// a.b (clone) b.c (clone) :(
// even a.b.setCValue(42) won't help as 'b' in this example is also a clone
```


Rust Ownership & Bindings

```
#[wasm_bindgen]
#[derive(Clone)]
struct A { pub b : Arc<Mutex<B>>; }

#[wasm_bindgen]
impl A {
    #[wasm_bindgen(getter)]
    pub fn b(&self) -> B {
        self.b.lock().unwrap().clone()
    }
}

#[wasm_bindgen]
#[derive(Clone)]
struct B { pub c : Arc<Mutex<C>>; }

// same for 'impl B'...
```

```
struct InnerC { pub value : u32; }

#[wasm_bindgen]
#[derive(Clone)]
struct C { inner : Arc<Mutex<InnerC>>; }

#[wasm_bindgen]
impl C {
    #[wasm_bindgen(getter)]
    pub fn value(&self) -> u32 {
        self.inner.lock().unwrap().value
    }

    #[wasm_bindgen(setter, js_name = value)]
    pub fn set_value(&self, value: u32) {
        self.inner.lock().unwrap().value = value;
    }
}

// In JavaScript
a.b.c.value = 42;
console.log(a.b.c.value); // 42 :)
```

Rusty Kaspas Node vs Client layers

CONSENSUS TRANSACTION

```
/// Represents a Kaspas transaction
#[derive(Debug, Clone, PartialEq, Eq, Default, Serialize, Deserialize, BorshSerialize,
BorshDeserialize)]
#[serde(rename_all = "camelCase")]
pub struct Transaction {
    pub version: u16,
    pub inputs: Vec<TransactionInput>,
    pub outputs: Vec<TransactionOutput>,
    pub lock_time: u64,
    pub subnetwork_id: SubnetworkId,
    pub gas: u64,
    #[serde(with = "serde_bytes")]
    pub payload: Vec<u8>,

    #[serde(default)]
    mass: TransactionMass,

    // A field that is used to cache the transaction ID.
    // Always use the corresponding self.id() instead of accessing this field directly
    #[serde(with = "serde_bytes_fixed_ref")]
    id: TransactionId,
}
```

CONSENSUS CLIENT TRANSACTION

```
/// Inner type used by [`Transaction`]
#[derive(Debug, Clone, Serialize, Deserialize)]
#[serde(rename_all = "camelCase")]
pub struct TransactionInner {
    pub version: u16,
    pub inputs: Vec<TransactionInput>,
    pub outputs: Vec<TransactionOutput>,
    pub lock_time: u64,
    pub subnetwork_id: SubnetworkId,
    pub gas: u64,
    pub payload: Vec<u8>,
    pub mass: u64,

    // A field that is used to cache the transaction ID.
    // Always use the corresponding self.id() instead of accessing this field directly
    pub id: TransactionId,
}

#[derive(Clone, Debug, Serialize, Deserialize, CastFromJs)]
#[wasm_bindgen(inspectable)]
pub struct Transaction {
    inner: Arc<Mutex<TransactionInner>>,
}
```

Rust Ownership & Bindings

Why Arc<> and Mutex<>?

We must use async Rust in the browser environment, which does not need to be thread safe.

At the same time, async Rust in native targets needs to be thread safe...



Send & Sync Markers

Send and Sync in Rust

- Send: A type is Send if it is safe to transfer ownership of values across threads. Essentially, it means that the type can be moved to another thread. Most primitive types in Rust are Send by default.
- Sync: A type is Sync if it is safe for the type to be referenced from multiple threads. A type T is Sync if &T (a reference to T) can be shared across multiple threads. Types that are Sync allow multiple threads to access the same reference safely.

Key Difference

- Send: Moves ownership between threads.
- Sync: Allows shared references between threads.

Send & Sync Markers

Examples Arc vs Rc

- **Rc<T> (Reference Counted)**: Used for single-threaded scenarios. It is not Send or Sync, meaning it cannot be shared across threads. It is used when you need shared ownership within a single thread.
- **Arc<T> (Atomic Reference Counted)**: Safe to use across multiple threads and is both Send and Sync. It provides thread-safe reference counting using atomic operations, allowing you to share ownership between threads.

Rc

```
use std::rc::Rc;
fn main() {
    let shared_value = Rc::new(5);
    let shared_value_clone = Rc::clone(&shared_value);
    println!("Value: {}", shared_value);
    println!("Clone: {}", shared_value_clone);
    // Cannot send Rc across threads (will result in a compile-time error)
    // std::thread::spawn(move || {
    //     println!("From another thread: {}", shared_value_clone);
    // }).join().unwrap();
}
```

Arc

```
use std::sync::Arc;
use std::thread;
fn main() {
    let shared_value = Arc::new(5);
    let shared_value_clone = Arc::clone(&shared_value);
    // Spawn a new thread and move the `Arc` clone into it
    let handle = thread::spawn(move || {
        println!("From another thread: {}", shared_value_clone);
    });
    // Wait for the thread to finish
    handle.join().unwrap();
    println!("Original value: {}", shared_value);
}
```

Rust Ownership & Bindings

```
let a = A::new(); // 'a' must be Send
spawn(async move {
    a.b.c.value = 42;
    println!("{}", a.b.c.value); // 42
});
```

Spawning a tasks in Tokio (native executor) requires data exchange to be ``Send``.

Spawning a task in Browser (wasm_bindgen executor) does not require data to be ``Send`` (JavaScript single-threaded).

Given that our code is multi-platform, it all has to be ``Send`` capable.

Data Representation (WASM32)

```
let x1 = new X();  
x1 is now {                                     // Rust Object 'X' representation  
  __wbg_ptr : 1234,                             // Rust Memory Pointer to X  
  propertyA : "hello",                          // getter  
  propertyB : 42,                               // getter  
}  
  
let x2 = {                                     // JavaScript Object 'X' repr  
  propertyA : "hello",                          // value  
  propertyB : 42,                               // value  
};
```

Example of two identical objects where one is created in Rust (WASM32 runtime) and exported to JavaScript and the second one created in JavaScript. Now we call:

```
doSomething(x1);  
doSomething(x2);
```

Data Representation (WASM32)

```
impl TryCastFromJs for TransactionInput {
    type Error = Error;
    fn try_cast_from<'a, R>(value: &'a R) -> std::result::Result<Cast<Self>, Self::Error>
    where
        R: AsRef<JsValue> + 'a,
    {
        Self::resolve_cast(value, || { // <- first, pass to resolve_cast to check for Rust type
            if let Some(object) = Object::try_from(value.as_ref()) {
                let previous_outpoint: TransactionOutpoint = object.get_value("previousOutpoint")?.as_ref().try_into()?;
                let signature_script = object.get_vec_u8("signatureScript").ok();
                let sequence = object.get_u64("sequence")?;
                let sig_op_count = object.get_u8("sigOpCount")?;
                let utxo = object.try_cast_into::<UtxoEntryReference>("utxo")?;
                Ok(TransactionInput::new(previous_outpoint, signature_script, sequence, sig_op_count, utxo).into())
            } else {
                Err("TransactionInput must be an object".into())
            }
        })
    }
}
```


WASM32 bindings

wasm-bindgen

web-sys

js-sys

wasm-bindgen

- Provides binding automation for various primitives including structs and functions.
- Handles type conversions + allows custom TypeScript type declarations

web-sys

- Provides access to *all Browser JavaScript APIs*

js-sys

- Provides access to JavaScript primitives (JsObject, Array, Number, Map etc.)

Multi-platform layer

workflow-rs

workflow-rs

- General-purpose crate developed for multi-platform applications.
- This adventure started due to lack of WASM32-compatible async WebSockets (there was literally nothing at the time in the browser environment - now still not much but the ecosystem is gradually evolving)
- Provides framework abstractions that function uniformly in all target environments
- Supports native, WASM32 (browser, nodejs, bun, deno, electron, NWJS + Browser Extension environments)

By restricting SDK I/O to this crate, this compatibility is projected on the Rusty Kaspas SDK.

Multi-platform layer

workflow-rs

workflow-core

- async Rust toolkit: (spawn(), sleep(), intervals, timeouts)
- Timer proxying via background workers in browsers (inactive tab handling)
- File I/O (native, NodeJS 'fs' module, localStorage, chromeStorage)
- 'runtime' tools (target platform/runtime identification)
- dirs (home and data folder resolution)

Everything we ever needed for general-purpose application development went in this or the accompanying crates.

Multi-platform layer

workflow-rs

workflow-log

- Just logging...
 - stdio in native
 - console (log/warn/err) in WASM32

workflow-wasm

- General purpose JS utilities
- JsValue & JsObject extensions
- Callback handling
- Panic handling (browser & node hooks)
- TryCast (TryCastFromJs trait)

workflow-serializer

- High-performance binary serialization based on Borsh (auxiliary serialization traits and macros)

workflow-dom

- DOM injection utilities, loading JS scripts via Blobs

Multi-platform layer

workflow-rs

workflow-websocket

Foundational layer for WebSocket I/O - provides the WebSocket struct (class) that functions in all above-mentioned targets.

On the backend:

WASM32

- W3C-compatible WebSockets

Native

- Tungstenite (tokio-tungstenite)

... it is fascinating how months of work and refactoring can be summarized into a single presentation with two key points ...

Multi-platform layer

workflow-rs

workflow-rpc

Foundational layer base on workflow-websocket crate for RPC management.

- Request
- Response
- Notifications

Similar to JSON RPC in structure (but websocket-framed).

Supports 2 encodings:

- Borsh (high-performance binary encoding)
- JSON (serde-json)

JSON encoding is similar to JSON RPC but uses large integers (not directly deserializable in JavaScript).

Multi-platform layer

`workflow-rs`

`workflow-rpc`

Rust implementation is meant for IPC-based functionality.

The methodology is to ensure that an RPC method call is a simple async function call. The application should not know that it is connected to RPC.

The goal is to provide a single set of Rust data structures available in the server and client allowing RPC layer to become a transparent.

Kaspa RPC

RpcApi trait

- Central trait declaring all RPC methods including their Request and Response data structures.
- To implement a server or a client, one simply needs to implement the RpcApi trait.
- Doesn't have to be RPC (KNG can use the trait directly by spinning up Rusty Kaspa daemon as a thread and latching onto the ``RpcCoreServices`` as a trait - in KNG this is known as "Integrated Node" mode where Rusty Kaspa and Kaspa NG run in a single process).

Kaspa gRPC

`kaspa-grpc-client`

Standard approach, primarily focused on server-side interfacing using gRPC .proto protocol definitions.

Provides conversions between messages defined in .proto definitions and Rusty Kaspa RPC Request/Response data structures.

gRPC is compatible with the legacy Golang Node and on it's own, gRPC has a huge client adoption.

Unfortunately, no ability to easily connect from web browsers.

Kaspa wRPC

`workflow-rpc/client -> kaspa-wrpc-client`

wRPC is based on workflow-rpc

- Native to Rust / Rusty Kaspa
- Native + WASM32-compatible client
- Compatible with bindings (Python etc.)

wRPC currently powers the majority of the Kaspa web application ecosystem.

Kaspa wRPC Resolver

Kaspa Resolver (application)

A simple high-availability load-balancer (v2).

Resolver is comprised of 2 primary components:

- wRPC client that connects to multiple wRPC endpoints and monitors connection health, node sync state and connection load.
- HTTP server for querying available wRPC endpoints.

Support for Resolver is integrated in wRPC client where if enabled it polls a resolver on each connection.

Public Node Network (PNN) initiative

Public Node Network is a contributor-maintained network of resolvers and Rusty Kasper nodes.

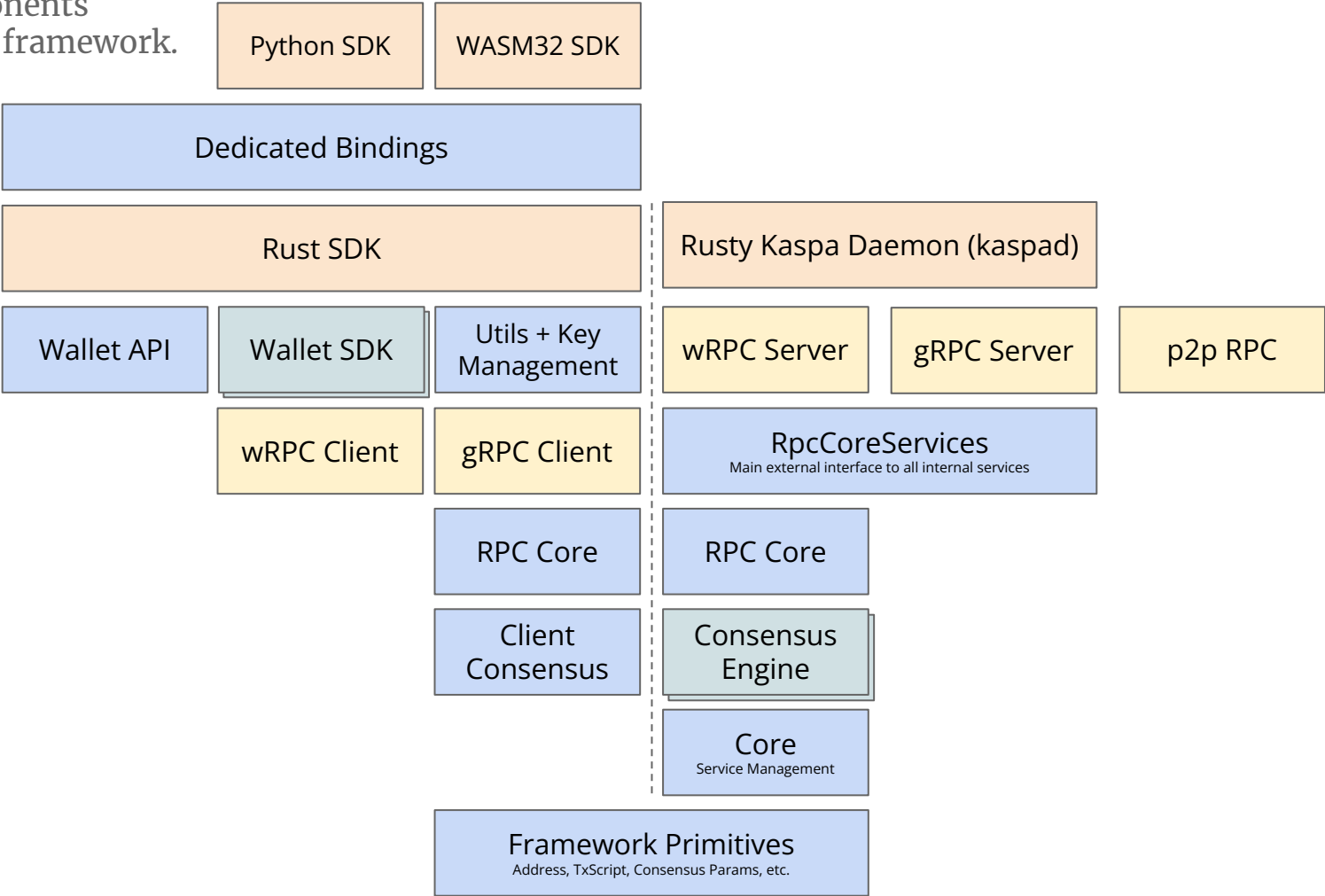
Primary goals:

- Provide developers with quick on-ramp for testing APIs and integration.
- Provide backend support for application developers who do not have ability/infrastructure to run their own nodes.
- Provide infrastructure developers with an out-of-the-box high-availability clustering solution (automatic node failover within a cluster)

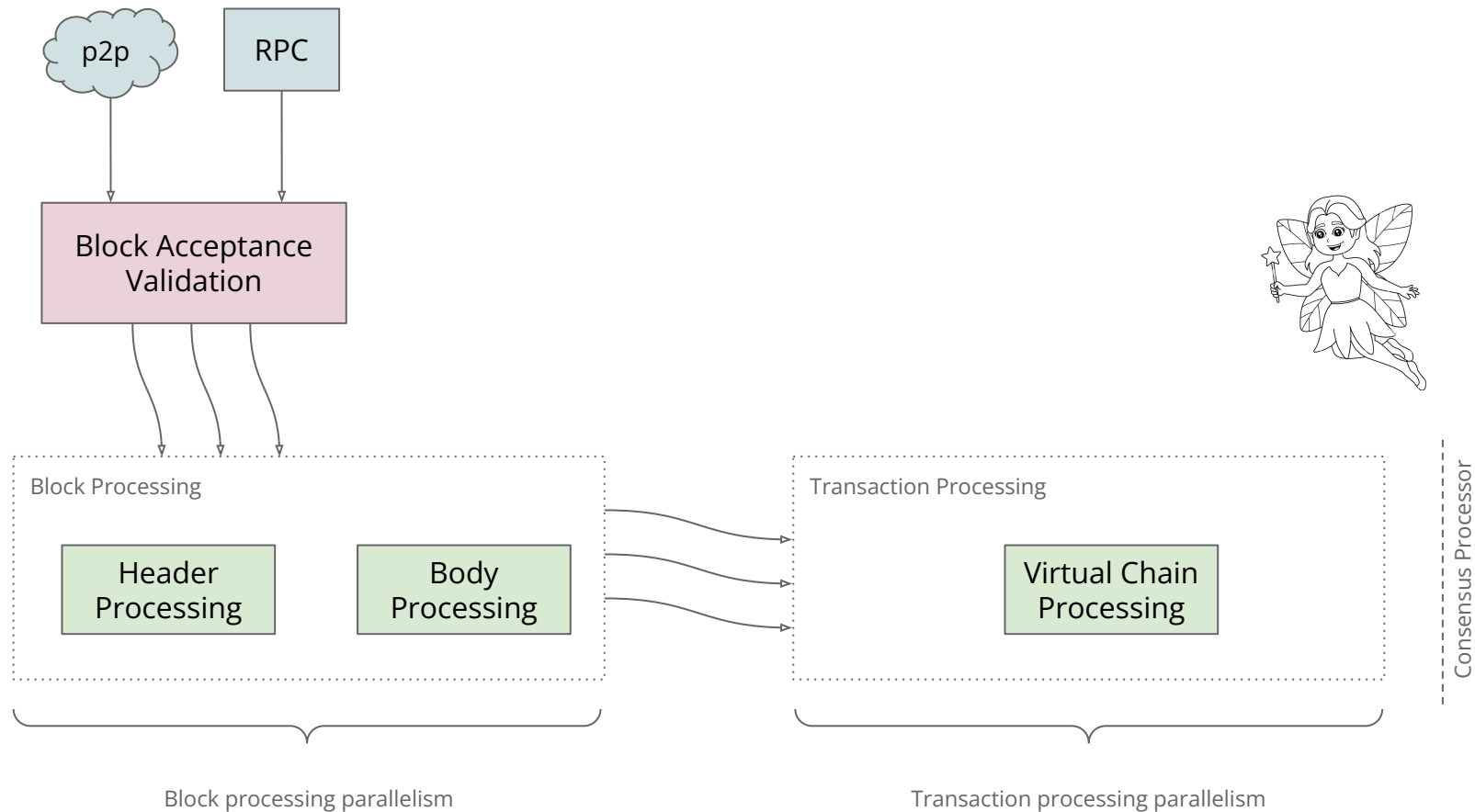
SDK & Rusty Kaspas Components

Key Logical Components
of the Rusty Kaspas framework.

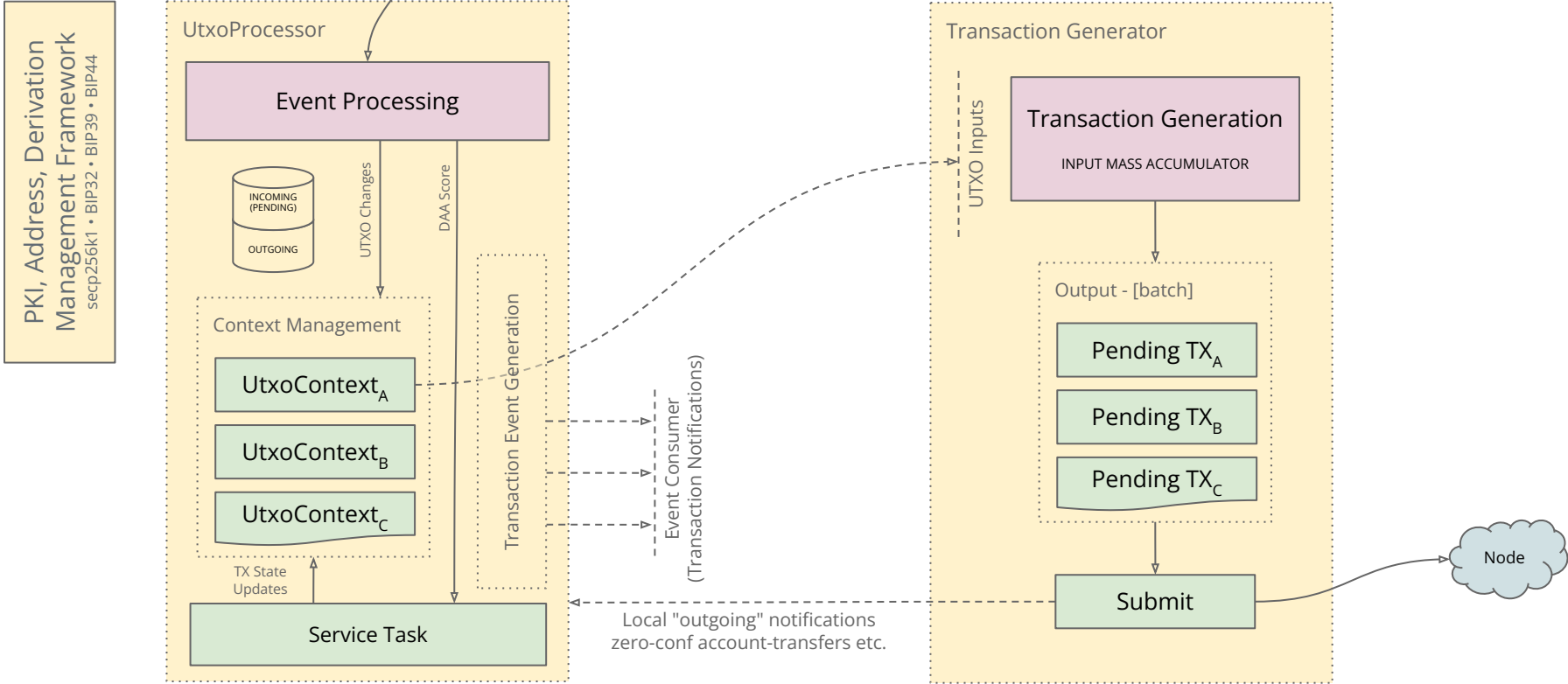
In-place and dedicated bindings





Consensus Engine (Bird's-eye view)



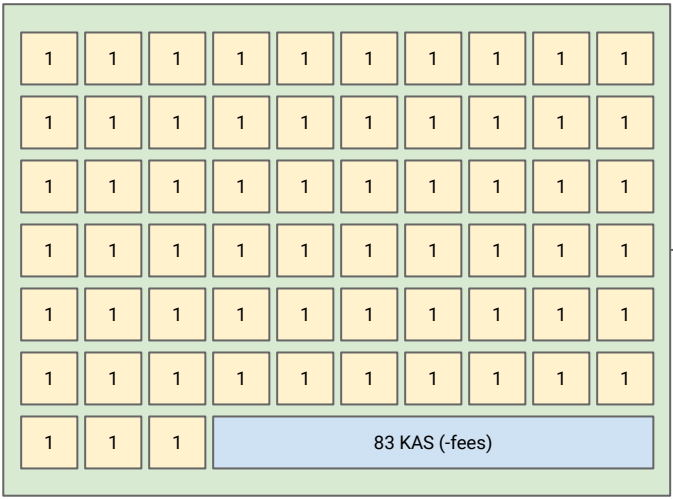
Wallet SDK - Key Logical Components



Transaction Mass Limits

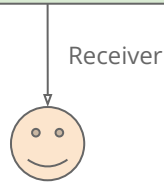
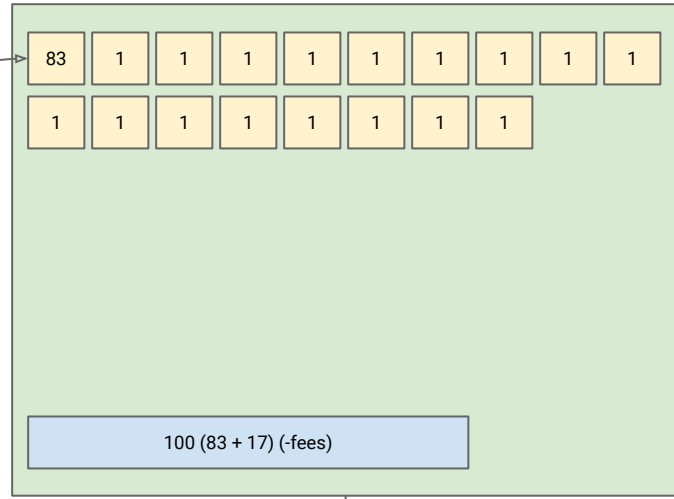
-  Inputs (UTXOs)
-  Outputs (UTXO wannabes)

Transaction

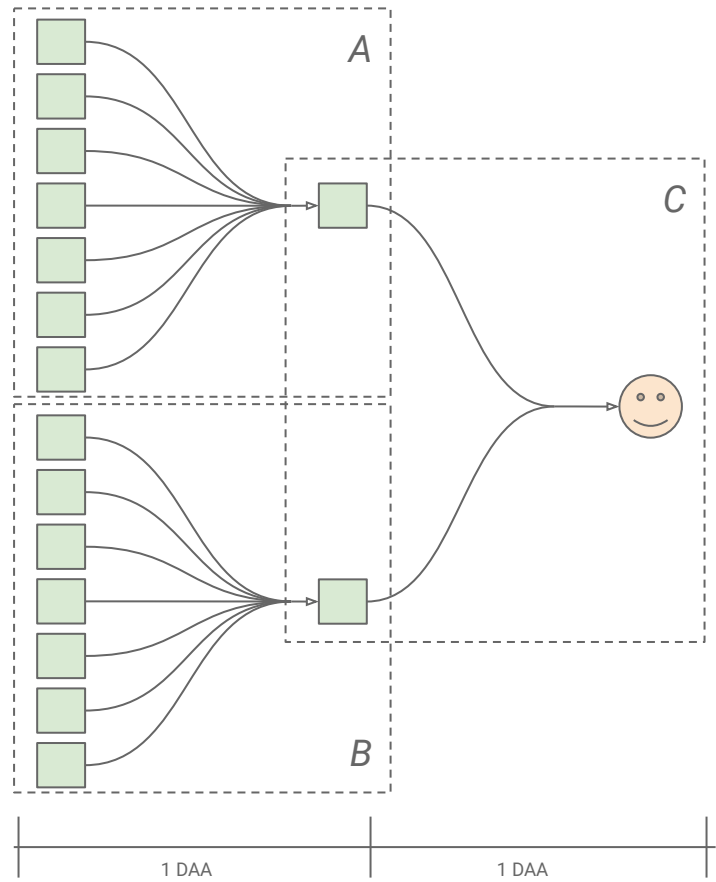


Transaction mass limit is 100,000.
For standard inputs & outputs this denotes 83 inputs + 1 output.

Sweeping
a.k.a. Compounding
a.k.a. Batching



Parallel Transaction Processing



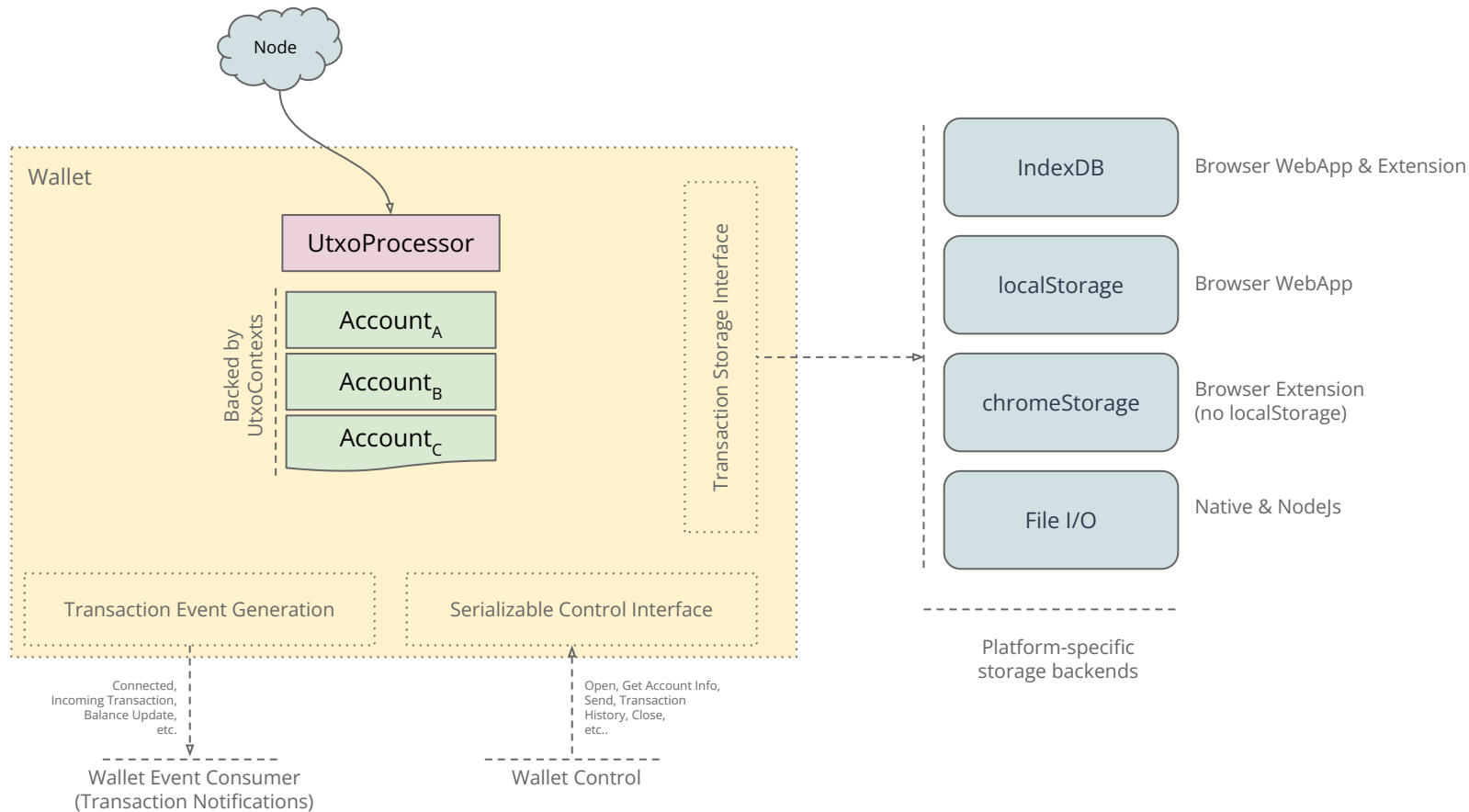
Unrelated transactions will process in parallel.

Transactions A and B will process in parallel within 1 DAA score.

Transaction C will process on the next DAA score.

Total cost is ~2 DAA.

Wallet API - Key Logical Components



Documentation

Rust

Automatically generated by RustDoc

WASM32

wasm-pack converts RustDoc to TypeScript, we then run TypeDoc on top of that to get TypeDoc documentation.

Python

TBD

Challenges

TypeScript "typings"...

While Rust ensures type integrity, it is very difficult to track all potential mistakes as type linkage breaks between Rust and TypeScript.

Recent Example:

```
`Address::setPrefix` was tagged as a setter :/
```

Solution to this is to have full test coverage in TypeScript, but this is very demanding.

Challenges

General TypeScript type acrobatics for complex functions...

```
export type RpcEventMap = {
  "connect" : undefined,
  "disconnect" : undefined,
  "block-added" : IBlockAdded,
  "virtual-chain-changed" : IVirtualChainChanged,
  ...
}

export type RpcEvent = {
  [K in keyof RpcEventMap]: { event: K, data: RpcEventMap[K] }
}[keyof RpcEventMap];
...

export type RpcEventCallback = (event: RpcEvent) => void;
...

interface RpcClient {
  addEventListener(callback: RpcEventCallback): void;
  addEventListener<M extends keyof RpcEventMap>(
    event: M,
    callback: (eventData: RpcEventMap[M]) => void
  )
}
...

#[wasm_bindgen(js_name = "addEventListener", skip_typescript)]
pub fn add_event_listener(&self, event: RpcEventTypeOrCallback, callback:
Option<RpcEventCallback>) -> Result<()> {
  ...
}
```

Outstanding Work

- RPC Pagination

Tolerable due to mainnet decentralization

- Python SDK release

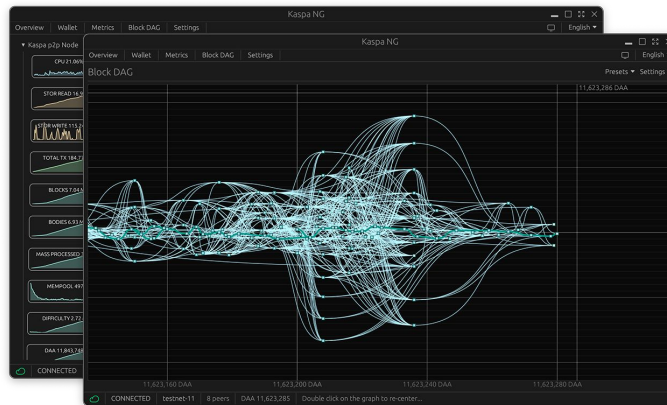
In final stages

- PSKT WASM bindings

In final stages

Case study

Kaspa NG



- Runs as a Native app on Windows, Linux, MacOS (embeds Rusty Kaspa p2p Node) - functions similar to Bitcoin Core.
- Runs as a Web application (WASM32) in a browser, accessible on mobile devices.
- Runs as a Web Extension.
- Runs as a command line (cli) wallet by embedding Rusty Kaspa CLI wallet crate (Windows requires 2 binaries for in terminal use).

Uses EGUI (<https://egui.rs>) to provide platform-neutral UX powered by OpenGL (glow) and WebGPU.

Resources

Rusty Kaspas - <https://github.com/kaspanet/rusty-kaspa>

Kaspa Integration Guide - <https://kaspa.aspectron.org>

Kaspa NG - <https://kaspa-ng.org>

workflow-rs - <https://github.com/workflow-rs/workflow-rs>