Rust China Conf 2020

Shenzhen, China

2020conf.rustcc.cn

Lessons Leaned

Building decentralized mission critical system in Rust

Introduction

Illia Polosukhin, 一龙

Co-founder @ NEAR

Previously

- Machine Learning for 10 years
- Engineering Manager @ Google Research
  - Deep Learning Research
  - Major TensorFlow contributor

10年机器学习领域的行业经验，作为工程师经理，在谷歌带领深度学习小组负责搜索业务的开发研究， 是TensorFlow人工智能开源项目的主要代码贡献者

About NEAR

- Sharded blockchain protocol
- Built to make developers life easy and allow to build usable apps
- Launched 2 months ago, already secures ~$1B of value
- Build globally available and community owned apps with Rust and AssemblyScript
- Novel business models for open source developers and entrepreneurs

```
# Make sure you have Node.js installed.

# Run this in your CLI to initialize a web app and
follow instructions from there

npx create-near-app --contract rust name
```

Introduction

孙元超

Architect @ Cdot

About Cdot

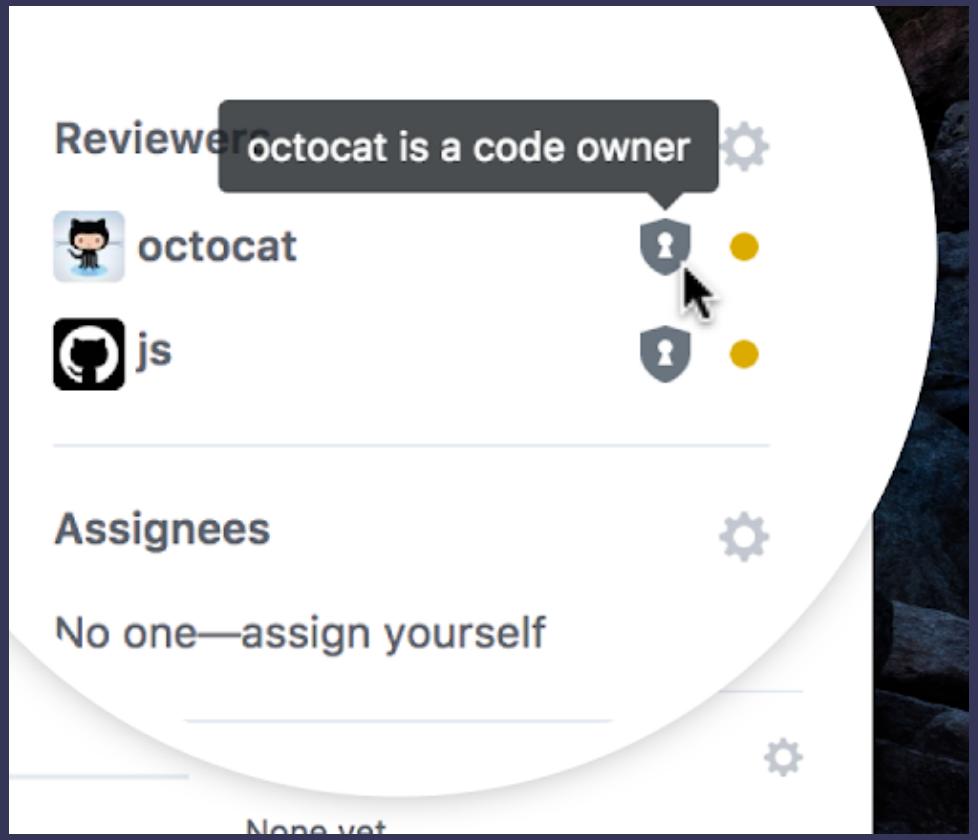
- Cdot is a Web3 infrastructure builder focused on blockchain interoperability.
- Be known for delivering cross-chain bridges and open-source components.
- Cdot build IBC protocol and Tendermint consensus in Rust.
- Granted by Near, Cosmos, Flow, PlatON, Oasis, Solana and Chainlink.

Development Process

It’s critical to set ground rules.

Code Owners

- Define Code Owners per repo / component
- At least 2 people who are both familiar with design and reasoning behind
- Require one of Code Owners approval to merge PRs for this component



Test Plan

- Before reviewing PR, check **Test Plan**
- Continuous Integration is all green
- If **Test Plan** is not sufficient -- ask to extend tests before continuing reviewing
- Review tests to make sure they match Testing Plan

Test plan

Changes in the old Doomslug tests:

1. `test_endorsements_and_skips_basic` : all the cases in which an approval for the next height which was not an endorsement was produced now produce no approval, since without NFG there's no need for approvals for the very next height which are not endorsements. Otherwise the test is unchanged.
2. `test_doomslug_approvals` I left the structure of the test the same, but designed new scenarios, since the change in the logic of when a block is ready to be produced is pretty drastic.

Changed `test_fuzzy_doomslug_liveness_and_safety` to ocasionally produce multiple blocks per height by two out of eight participants, and made the check that ensures final blocks are not reverted to use hashes, not heights. Collectively that catches the bug with skips and endorsements handled the same way. It stalls if only hash is used to include approvals in blocks, and catches safety violation if only height is used.

https://github.com/near/nearcore/pull/2390

Serialization

Seriously? Isn’t a solved problem?

Serialization, seriously?

- Most of the serialization methods are not byte to byte deterministic across platforms/ implementations
- Or very slow
- Need something simple, deterministic, lightning fast, minimum dependencies
- End up building “Binary Object Representation Serializer for Hashing” or **Borsh**

Check it out at https://borsh.io/

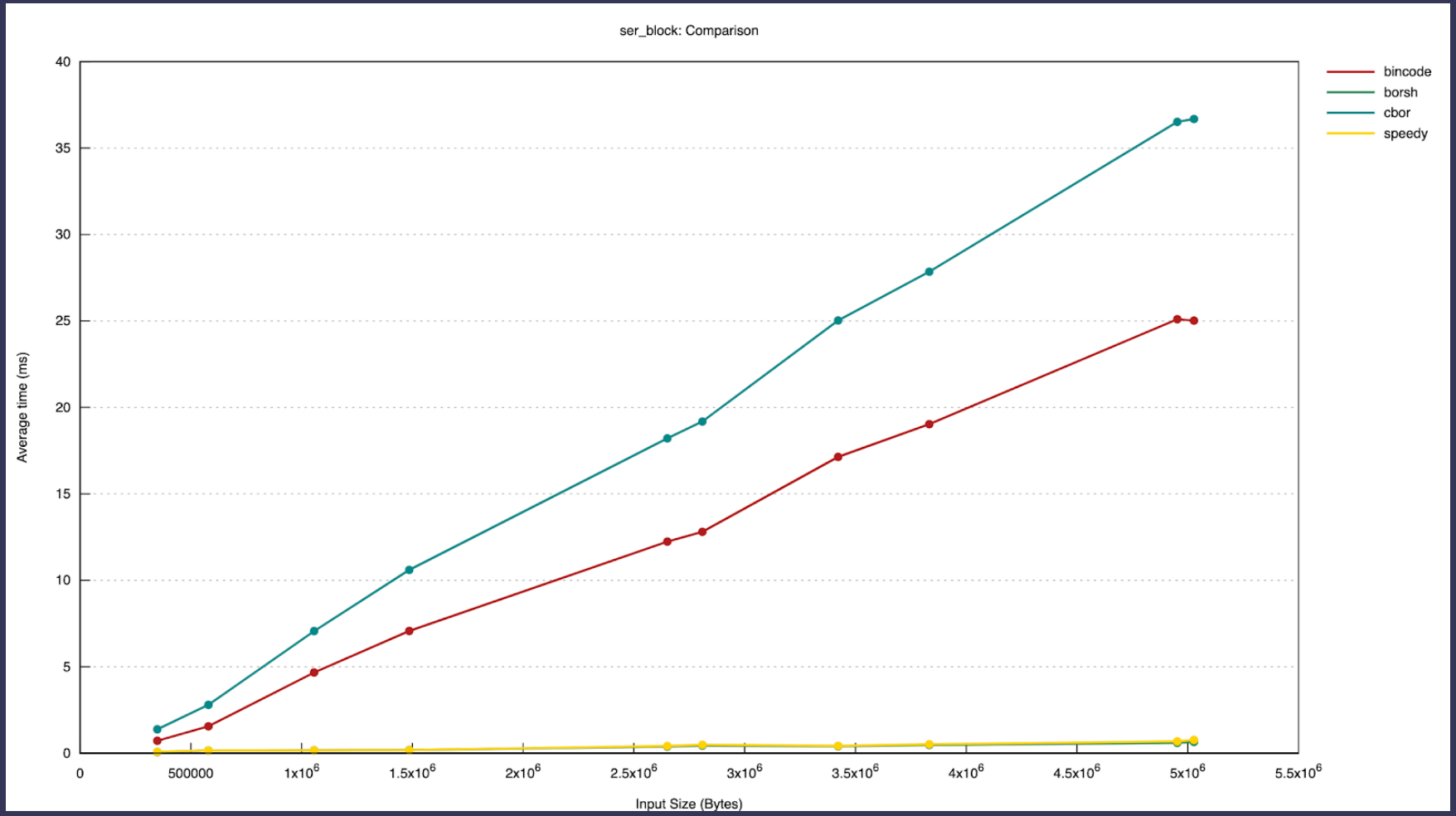
Borsh Usage

```rust
use borsh::{BorshSerialize, BorshDeserialize};

#[derive(BorshSerialize, BorshDeserialize, PartialEq, Debug)]
struct A {
    x: u64,
    y: String,
}

#[test]
fn test_simple_struct() {
    let a = A {
        x: 3301,
        y: "liber primus".to_string(),
    };
    let encoded_a = a.try_to_vec().unwrap();
    let decoded_a = A::try_from_slice(&encoded_a).unwrap();
    assert_eq!(a, decoded_a);
}
```

Borsh Performance



Upgrades

Decentralized protocols are inherently hard to upgrade

Details https://nomicon.io/ChainSpec/Upgradability.html

Backward compatibility

- NEAR decided to maintain 1 version compatibility: X will work on top of X - 1
- It’s critical to check that new version is compatible with old version
- Check that new version can run on top of old nodes data
- Check that new version can operate with old nodes in the network

Updating Data Structures

- Nodes still should be able to process older version of data
- Leveraging Borsh's *enum* as version number
- Borsh Serializer/Deserializer handles all the versioning automatically
- Can easily add new versions by extending enum and adding new version of the data structure
- Version up-casting via impl From<BlockHeaderV1> for BlockHeaderV2

```rust
/// Versioned BlockHeader data structure.
/// For each next version, document what are the changes between versions.
#[derive(BorshSerialize, BorshDeserialize, Serialize, Debug, Clone, Eq, PartialEq)]
pub enum BlockHeader {
    BlockHeaderV1(Box<BlockHeaderV1>),
    /// V1 -> V2: Remove `chunks_included` from `inner_reset`
    BlockHeaderV2(Box<BlockHeaderV2>),
}
```

Protocol Versions

```rust
/// The protocol version that enables reward on mainnet.
pub const ENABLE_INFLATION_PROTOCOL_VERSION: ProtocolVersion = 36;

/// Fix upgrade to use the latest voted protocol version instead of the current epoch protocol
/// version when there is no new change in protocol version.
pub const UPGRADABILITY_FIX_PROTOCOL_VERSION: ProtocolVersion = 37;

/// Updates the way receipt ID, data ID and random seeds are constructed.
pub const CREATE_HASH_PROTOCOL_VERSION: ProtocolVersion = 38;

/// Fix the storage usage of the delete key action.
pub const DELETE_KEY_STORAGE_USAGE_PROTOCOL_VERSION: ProtocolVersion = 40;
```

Feature Flags

```rust
/// New Protocol features should go here. Features are guarded by their corresponding feature flag.
/// For example, if we have `ProtocolFeature::EVM` and a corresponding feature flag `evm`, it will look
/// like
///
/// #[cfg(feature = "protocol_feature_evm")]
/// EVM code
///
#[derive(Hash, PartialEq, Eq, Clone, Copy, Debug)]
pub enum ProtocolFeature {
    #[cfg(feature = "protocol_feature_forward_chunk_parts")]
    ForwardChunkParts,
    #[cfg(feature = "protocol_feature_rectify_inflation")]
    RectifyInflation,
    #[cfg(feature = "protocol_feature_evm")]
    EVM,
}
```

```rust
if checked_feature!("protocol_feature_evm", EVM, runtime_ext.protocol_version())
    && is_account_evm(&account_id)
{
```

Nightly Protocol

- Borrowing from Rust Compiler, use nightly for bleeding edge features
- Nightly protocol defines which protocol features to include
- Allows to merge unstable code into master without jeopardizing stable releases

```rust
/// Current latest stable version of the protocol.
#[cfg(not(feature = "nightly_protocol"))]
pub const PROTOCOL_VERSION: ProtocolVersion = 41;

/// Current latest nightly version of the protocol.
#[cfg(feature = "nightly_protocol")]
pub const PROTOCOL_VERSION: ProtocolVersion = 44;
```

Testing

Heavy and rich testing enables to move fast

Continuous Integration

- Critical that clean builds pass all unit and integration tests **on every commit** to give quick feedback
- *cargo check && cargo test --workspace*
- Clean Rust build on 1 CPU takes ~40 min.
- On demand 32vCPU cloud instance to reduce to 2-3 min.
- Includes basic backward compatibility tests as well

Expensive and Non-Hermetic Tests

- Some tests are way too complex for running on every commit
- Others tests require open ports or heavy on CPU to run in parallel

```rust
#[cfg(test)]
#[cfg(feature = "expensive_tests")]
mod tests {
```

```rust
lazy_static! {
    static ref HEAVY_TESTS_LOCK: Mutex<()> = Mutex::new(());
}

pub fn heavy_test<F>(f: F)
where
    F: FnOnce() -> (),
{
    let _guard = HEAVY_TESTS_LOCK.lock();
    f();
}
```

```rust
lazy_static! {
    static ref OPENED_PORTS: Mutex<HashSet<u16>> = Mutex::new(HashSet::new());
}
```

Cluster Tests

- Decentralized protocols are also should be tested in adversarial and complex networking environment
- Python tests help with orchestrating clusters of nodes
- Works both locally and using “MockNet”
- MockNet: network of pre-deployed 50 nodes which can be coordinated from test running machine
- Allows to start and kill nodes, pause network, rewrite messages between nodes and send arbitrary messages
- Nodes have adversarial mode that enables various malicious strategies

Nightly Tests

- Complex tests running on dedicated testing infra: Nayduck + MockNet
- Takes ~4 hours, runs on the latest master commit
- Developers can trigger custom build for their branch

Branch	Title	User	Status	x
master				
master 29fcaf3	disable performance-stats feature by default (#3761)	NayDuck	Debug SKIPPED 4 pending	x
master 29fcaf3	disable performance-stats feature by default (#3761)	NayDuck	Debug SKIPPED 1 passed 3 failed	x
master 29fcaf3	disable performance-stats feature by default (#3761)	NayDuck	Debug BUILD DONE 117 passed 9 failed 1 timeout 1 ignored Debug/Nightly BUILD DONE 118 passed 8 failed 1 ignored	x
master 84fec31	fix: shutdown neard when the client protocol version is too old (#3735)	NayDuck	Debug BUILD DONE 119 passed 8 failed 1 ignored Debug/Nightly BUILD DONE 117 passed 9 failed 1 ignored	x

Test Networks

- Two levels of test networks: **BetaNet** and **TestNet**
- **BetaNet** runs nightly_protocol:
  - Decentralized protocol testing with engaged node operators
  - Tooling developers start integrating new features
- **TestNet** runs pre-production version:
  - “Canary” testing of new versions for backward compatibility
  - Node operators test infra and new binaries
  - Developers test their applications

Go Slow To Move Fast

- Blockchain is a family of protocols that unlocks ease of coordination to build and adopt application specific protocols
- Speed of innovation in protocol can define its success
- Rust provides the best environment for fast development of mission critical protocols
- To keep up pace of innovation it's critical to have development processes and robust testing

NEAR

Thank you

cdot

We are hiring Rust devs!

near.org/careers

cdot.network