DFINITY

The Internet Computer for Systems Researchers

Stefan Kaestle
Ulan Degenbaev, Adam Bratschi-Kaye, Andriy Berestovskyy

June 2022

We are hiring: dfinity.org/careers

Agenda

1) What is the IC?
2) Interesting Systems problems
3) Numbers
4) Q&A

DFINITY

What is the Internet Computer?

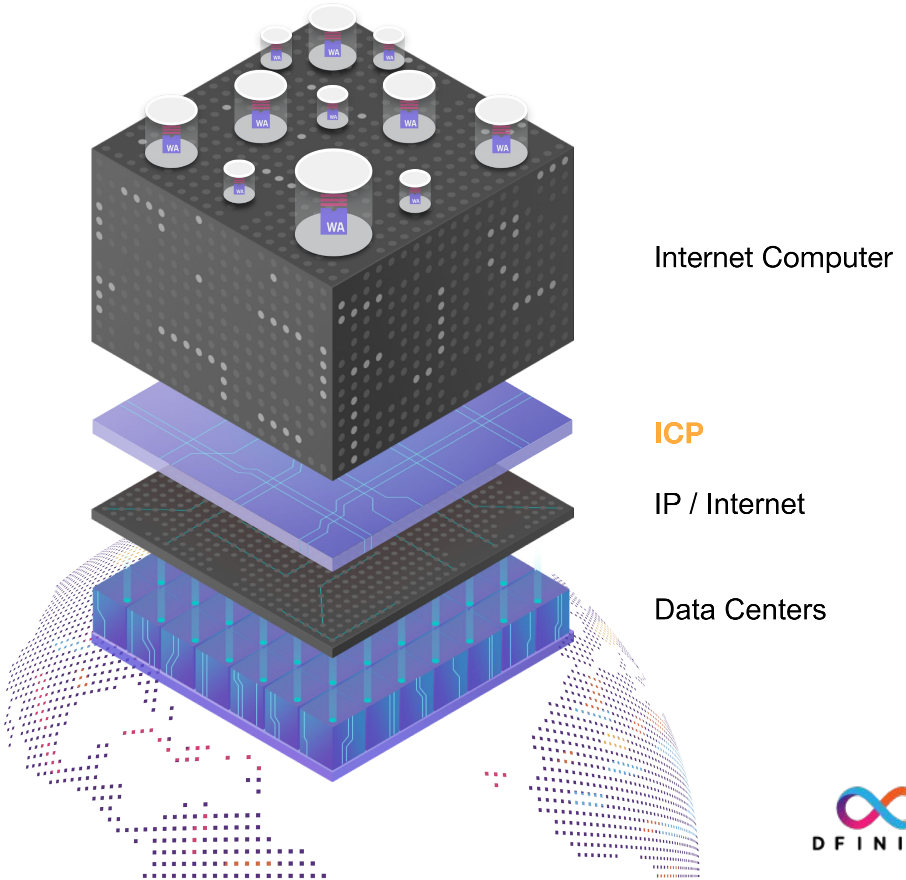
What is the Internet Computer?

Vision:

Platform to run any computation
in a decentralized and secure manner

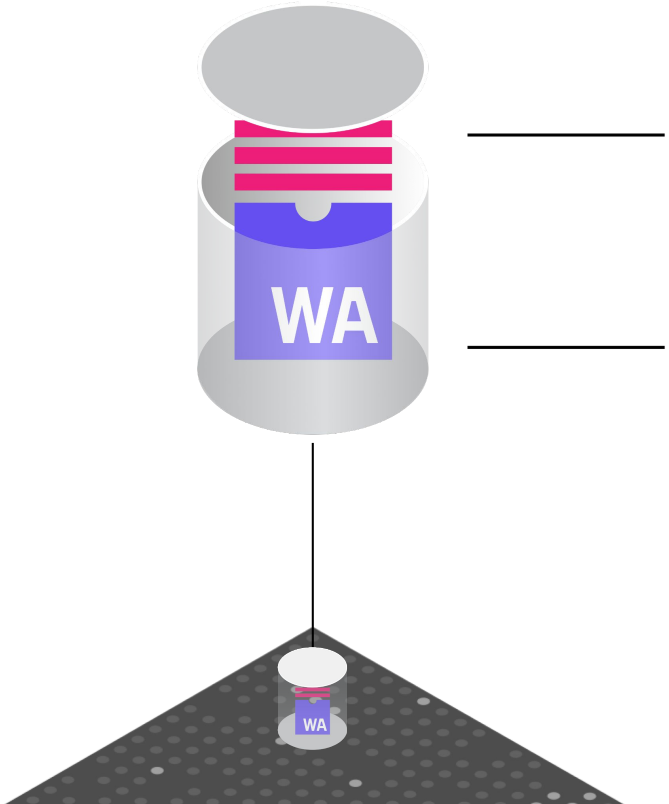
What's different about the Internet Computer

- **Byzantine** fault tolerance
  - Up to f out of 3f + 1 malicious nodes
  - Individual **nodes cannot be trusted**
- Geo replicated
- Decentralized
  - DFINITY cannot access most nodes
- Self governing
  - No single person in control of the IC
  - Votes to apply changes



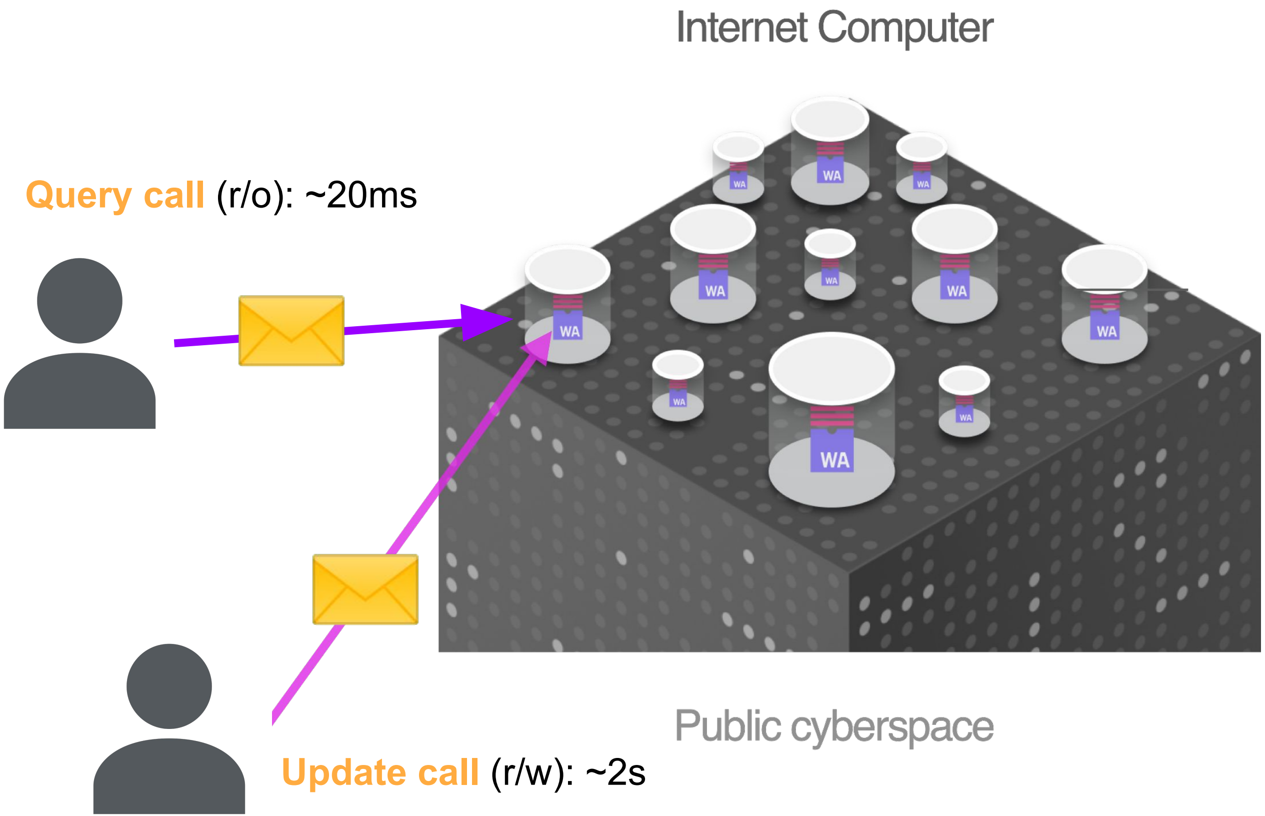
DFINITY

Canister Smart Contracts



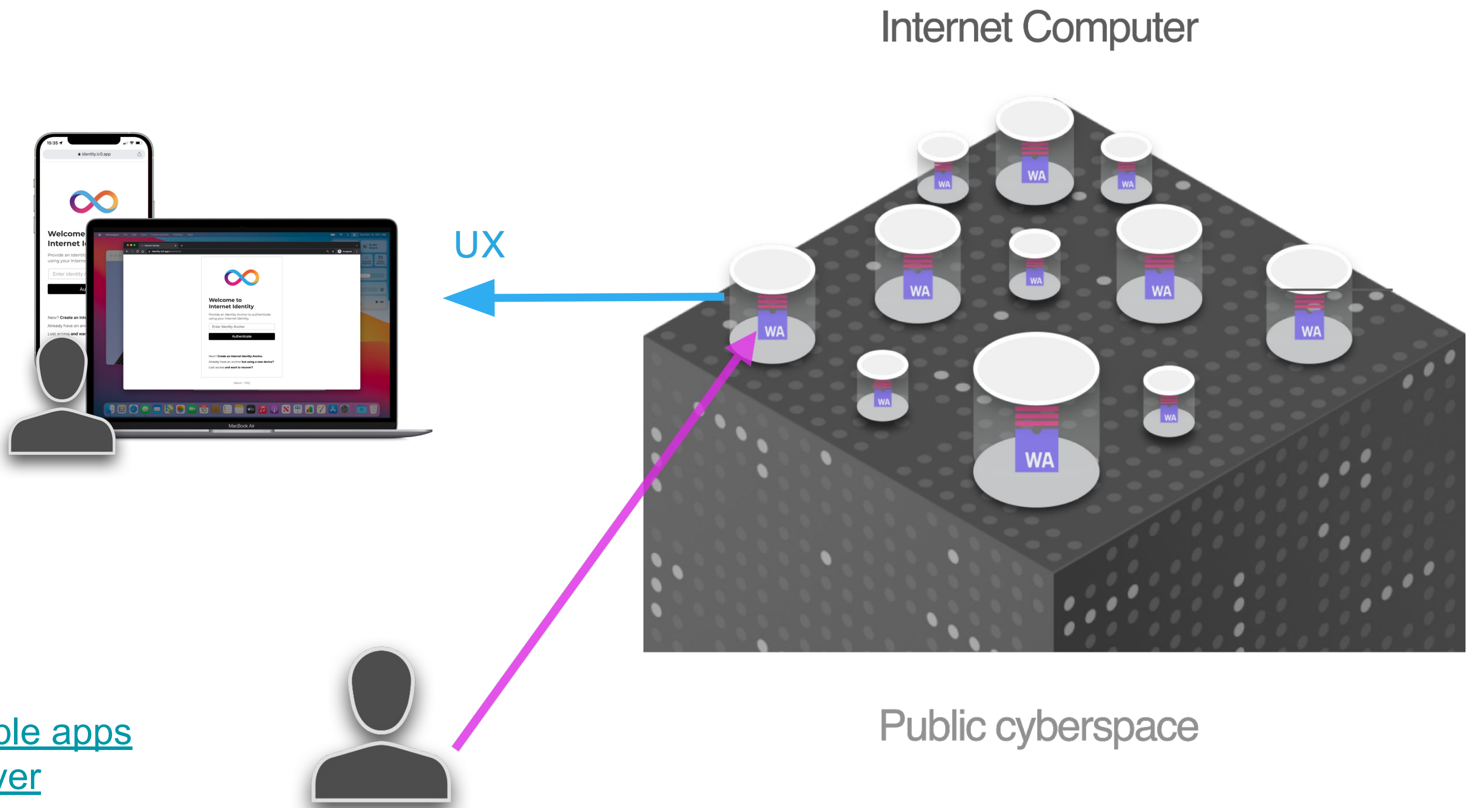
DFINITY

Users interact directly with Canisters: raw calls



DFINITY

Developers and users interact directly with Canisters



[Example apps](#)
[Discover](#)

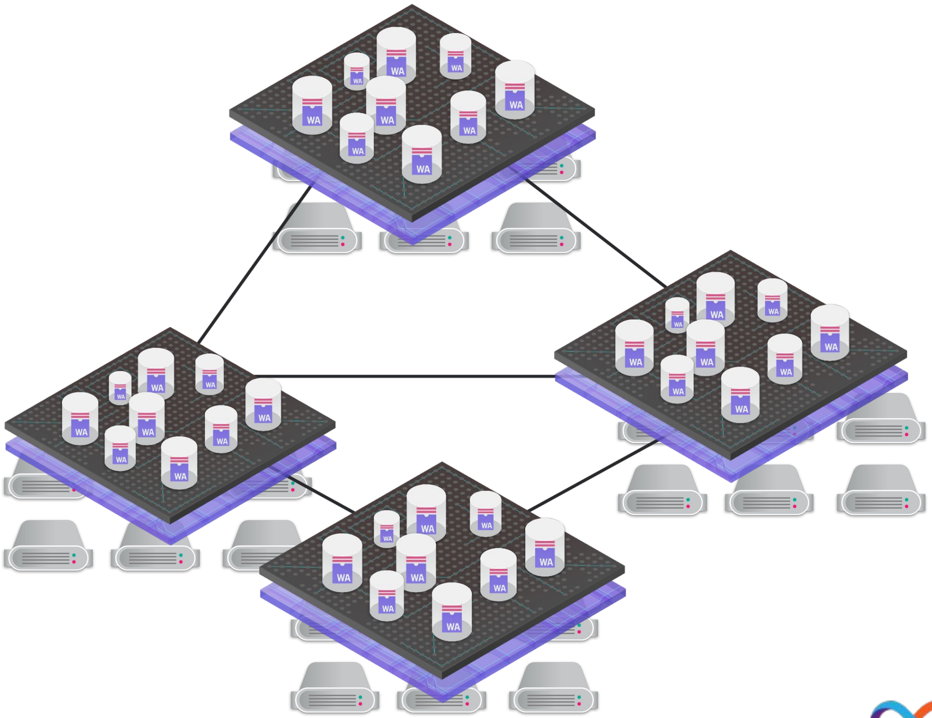
DFINITY

State Machine Replication (SMR)

Nodes must have same state

1. State on all nodes is identical
2. Deterministic state transitions
3. Ordered input

→ State still the same after executing inputs



IC state:

- Canister code, data and queues
- System state

DFINITY

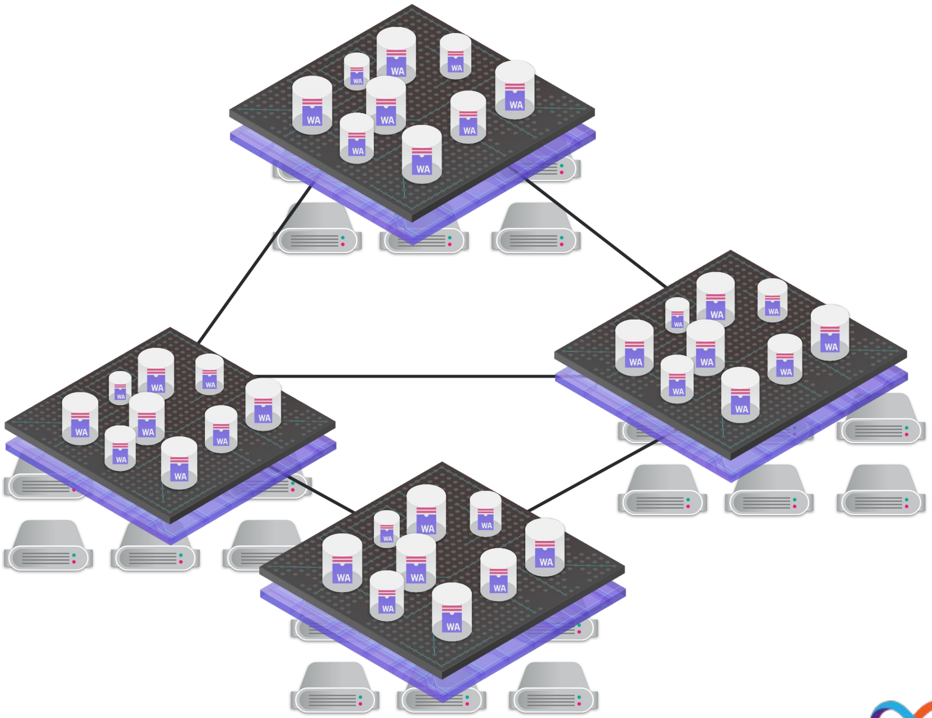
Scalability: Nodes and Subnets

Nodes are partitioned into subnets

Each subnet runs instance of SMR

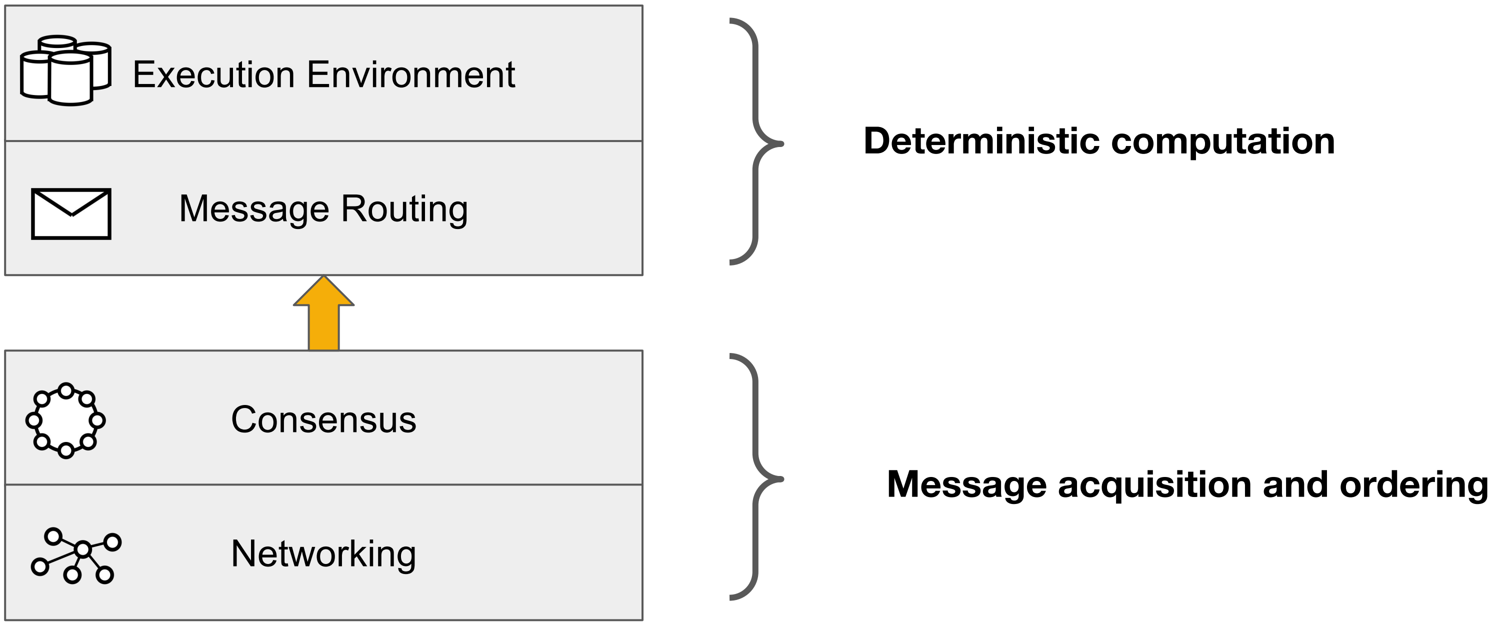
Each subnet hosts a subnet of canisters

Communication across subnets possible



DFINITY

ICP Layers



Execution Environment
Message Routing
Consensus
Networking
Deterministic computation
Message acquisition and ordering

Execution Environment

Hello World example app

```
#[query]
fn greet(name: String) -> String {
    format!("Hello {}", name)
}
```

- Canister code: wasm
  - Official support: Rust and Motoko
- Install: get canister ID
- Call via canister ID
  - Raw calls
  - HTTP calls

App with state: Orthogonal persistence

- Illusion: programs run forever
- Program state (incl. heap) is persisted/restored automatically

App with state: Orthogonal persistence

```
thread_local! {
    static STATE: RefCell<State> = RefCell::new(State::default());
}

#[derive(Default)]
pub struct State {
    owner: Option<Principal>,
    ledger: Option<Principal>,
    exchange: Exchange,
}
```

Sampel Defi

Note:
Programming is significantly simpler in Motoko

App with state: Orthogonal persistence

```
#[query(name = "getBalance")]
#[candid_method(query, rename = "getBalance")]
pub fn get_balance(token_canister_id: Principal) -> Nat {
    STATE.with(|s| s.borrow().exchange.get_balance(token_canister_id))
}
```

[Sampel Defi]

App with state: Orthogonal persistence

```
#[update]
#[candid_method(update)]
pub async fn deposit(token_canister_id: Principal) -> DepositReceipt {
    let caller = caller();
    // Stuff happens here ..
    STATE.with(|s| {
        s.borrow_mut()
            .exchange
            .balances
            .add_balance(&caller, &token_canister_id, amount.to_owned())
    });
    DepositReceipt::Ok(amount)
}
```

[Sampel Defi]()

Orthogonal persistence: Track changes + accounting

Challenge: Need to track changes to memory

Current solution (simplified): Map memory pages on demand

Example: Canister call

1. Initially: no page is mapped
2. Read access: page fault → map r/o, *increase read counter*
3. Write access: page fault → (re-)map r/w, *increase write counter + remember page*
   a. Query call: throw away dirty pages
   b. Update call: store changes in heap delta

Note: 95% of message executions change at most seven memory pages.

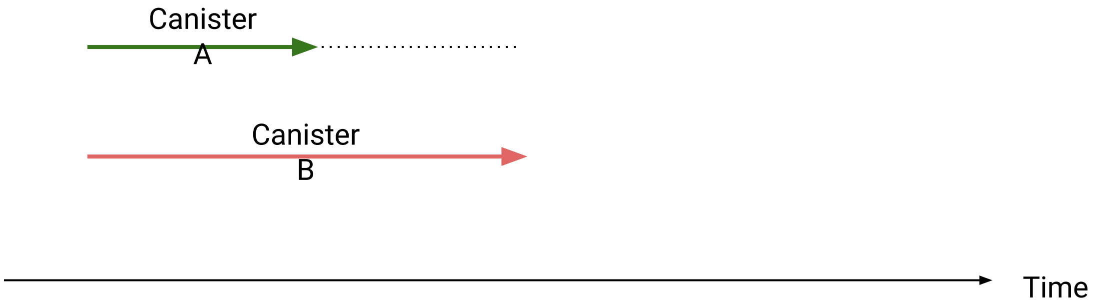
Orthogonal persistence: Performance

Naive solution quite slow.

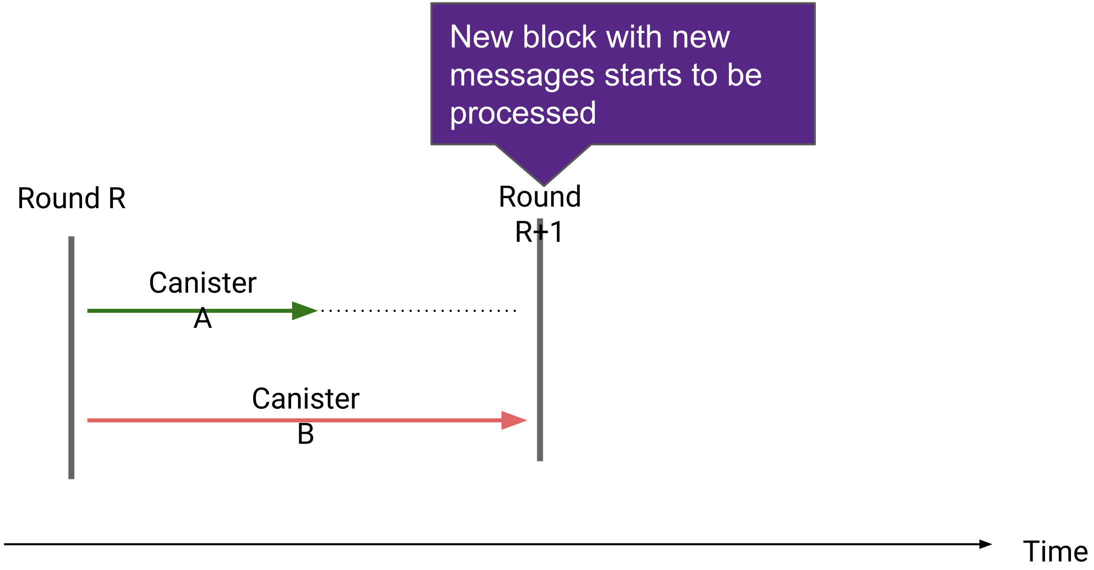
- Can **speculatively map** multiple consecutive pages: → trade accuracy for speed
    - diff on speculatively mapped r/w pages
- Map **r/w for query calls** (we throw changes away anyway)

Future: might explore modifying the wasm runtime to compile in profiling instructions

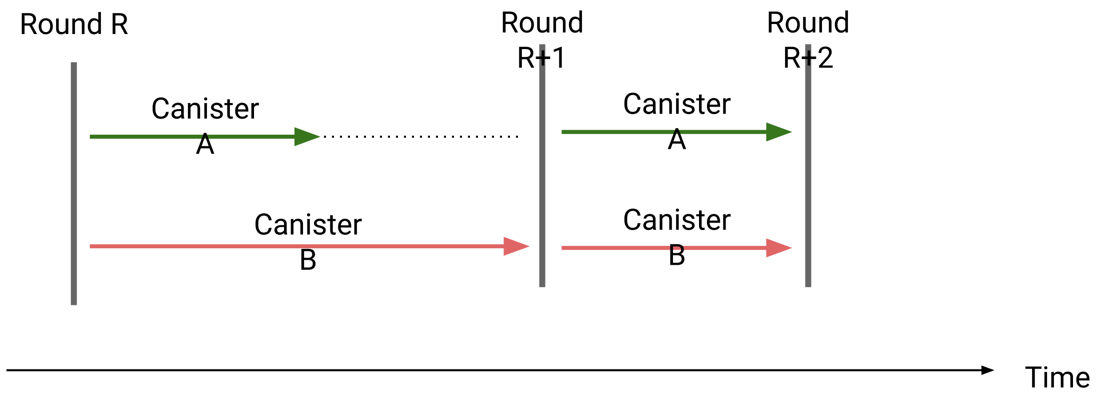
Multiple concurrent canister executions



Multiple concurrent canister executions



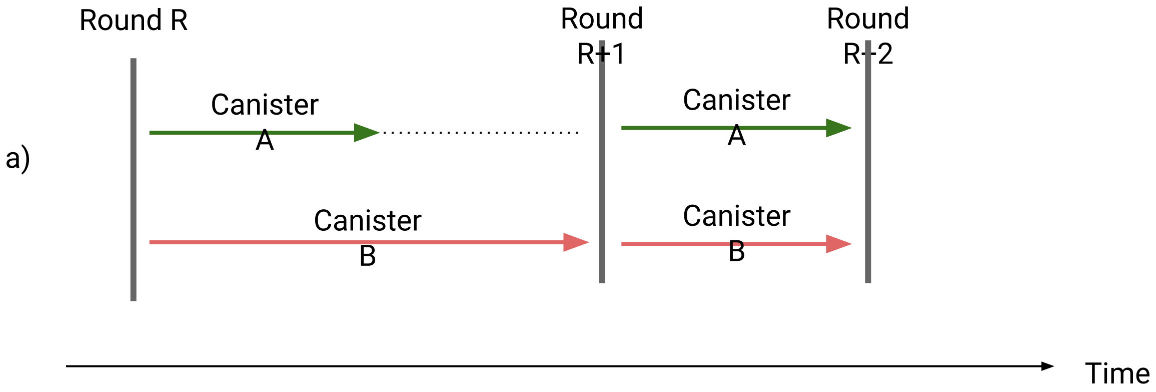
Multiple concurrent canister executions



Canister A might suffer from Canister B

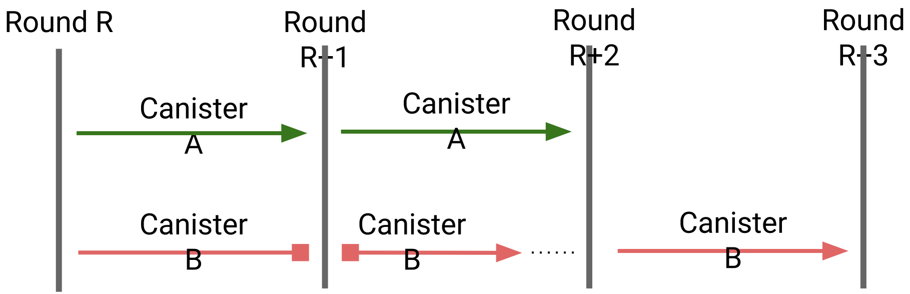
Multiple concurrent canister executions

- Want block ~1s → Execution has to process messages in ~1s



- Limit number of instructions per message
  a. But: Some messages take longer
     - E.g. canister upgrade, with expensive pre- and post-hooks
     - Garbage collection

Scheduling: time slicing



- Has to be **deterministic**
    a. Load balancing etc. gets harder

- Reservations (compute allocations)
- Good resource usage
    a. Fill with best-effort, fairness
- Intermediate state must not be observable
    a. Atomicity: Roll-back on error + Isolation

Time slicing and checkpointing

- Checkpoint to disk every 500 rounds (~500s, ~8min)
- Contains all state required to resume computation

- Partially executed messages at checkpoint?
    a. Nodes have be able to resume from checkpoints
    b. What to do with incomplete message executions?

Scheduling & time slicing

- Quite challenging
- Still ongoing discussion

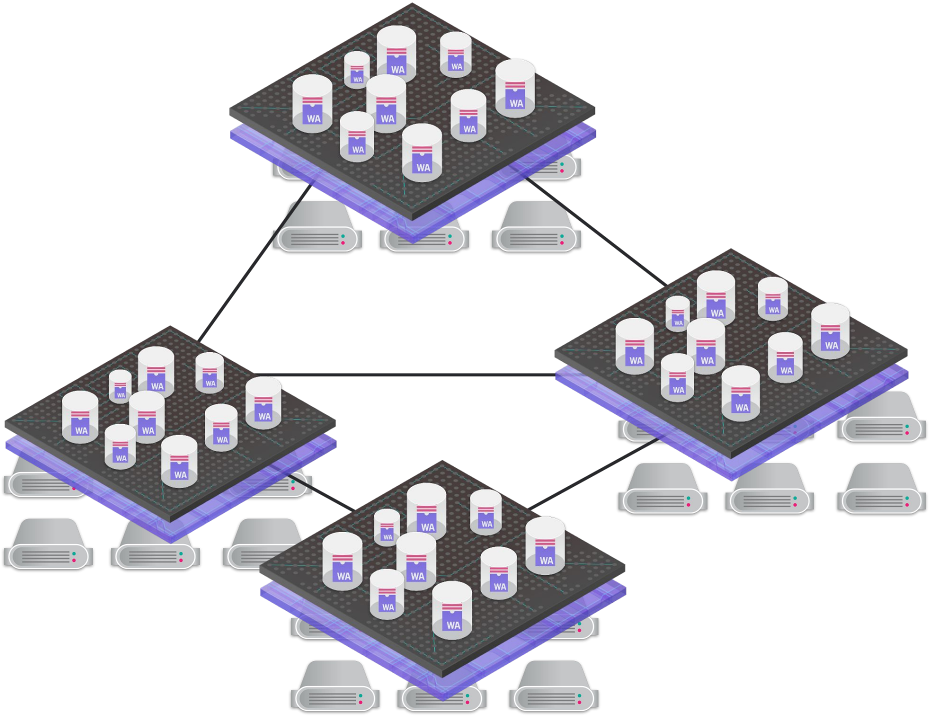
- Come talk to us if you are interested in working on things like this

Some numbers

The IC in Current Numbers

Network Layer:

- 477 nodes
  - From 54 node providers
- 33 subnets



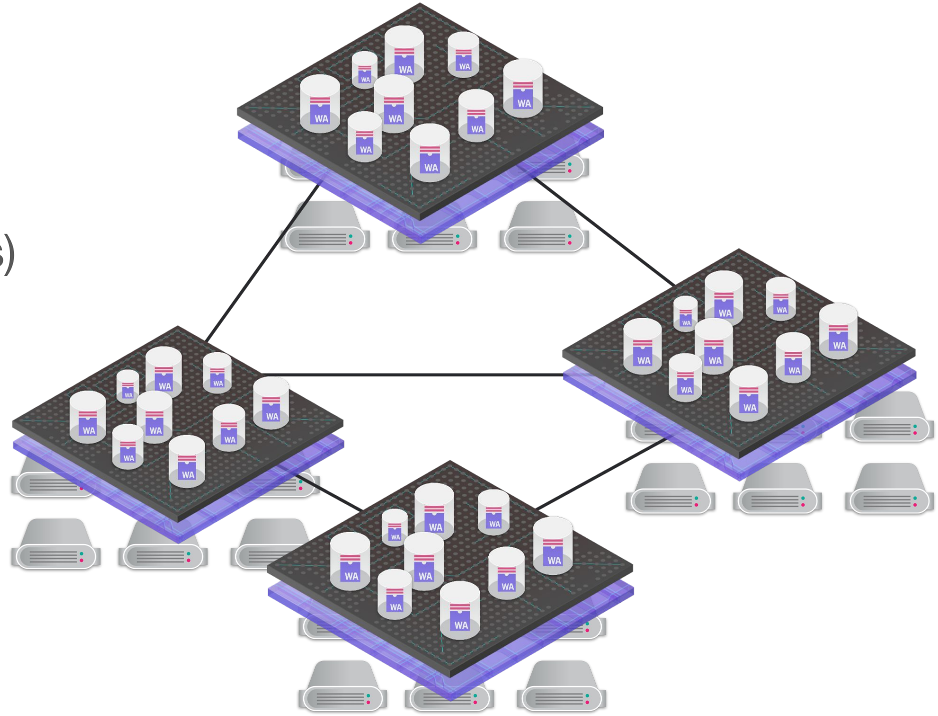
https://dashboard.internetcomputer.org/

DFINITY

The IC in Current Numbers

Application Layer:

- 75K+ canisters
- > 2 Mio registered identities (~users)
- ~1.1TB total state (and counting…)



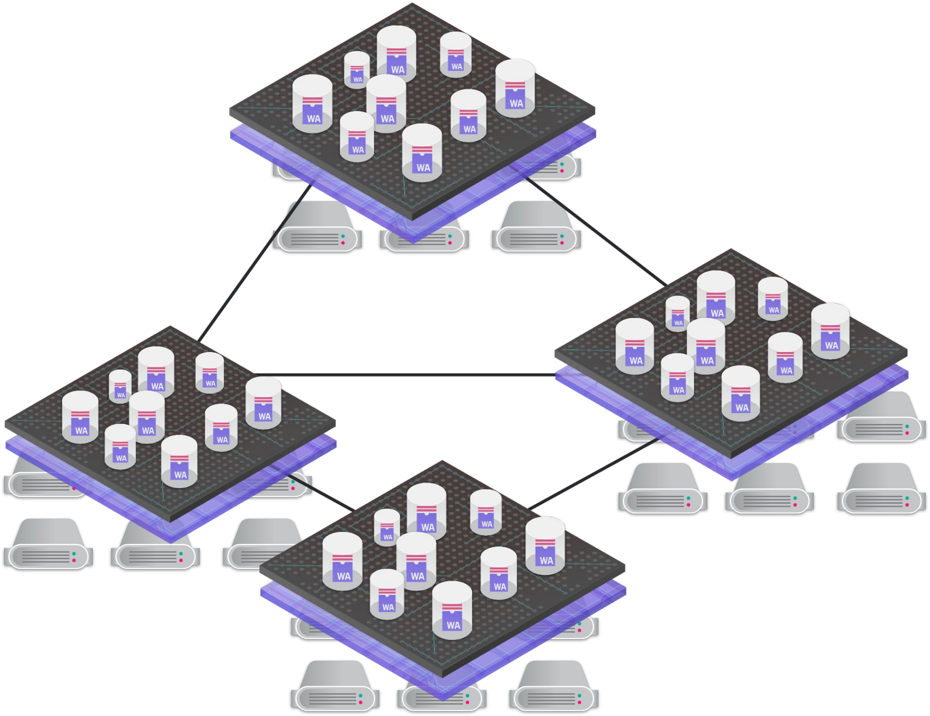
https://dashboard.internetcomputer.org/

DFINITY

The IC in Current Numbers

Consensus

- 850M+ blocks created
- ~34 blocks per second
- ~3500 messages per second



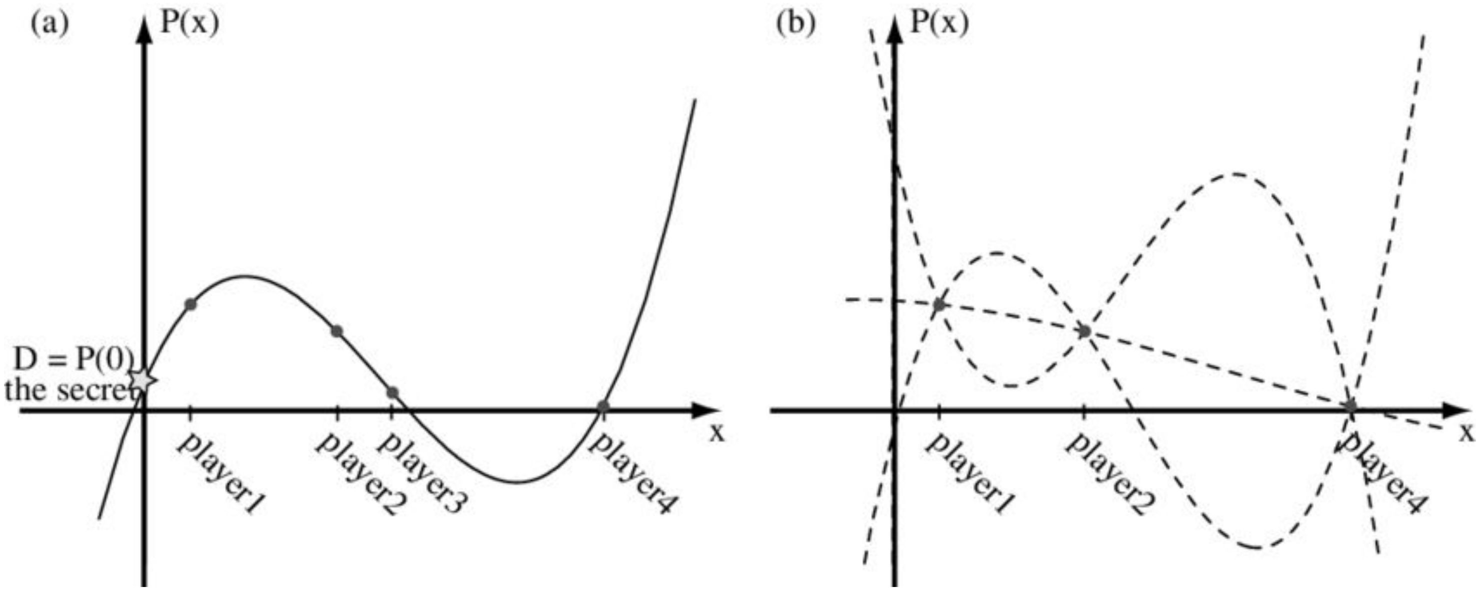
https://dashboard.internetcomputer.org/

DFINITY

Energy consumption of the IC

- Blockchains have a bad reputation
  - Mostly due to proof of work
- We don’t do that
- We have random beacon and threshold cryptography
  - Single public key that can be used to verify responses from IC
  - Can throw away old state (don’t need to maintain forever)

Threshold Cryptography in a nutshell



Shamir's polynomial of degree 4

DFINITY

Energy consumption of the IC

- Peak power consumption of node machines: 700W
- Power usage effectiveness (PUE): 2.33 (extremely conservative)
  - A PUE of 1: all power is spent on compute
  - A PUE of 2: as much power for cooling etc as for compute
  - 2.33 is quite conservative (e.g. Google closer to 1.1)
- With PUE: 1631W per IC node

- Number of machines: 518 + 11 boundary nodes (as of weekend)
- Total max power consumption of all nodes: ~863kW

- ~3300 transactions / s → 261.45 Ws per transaction = 261.45 Joule
- Conservatice: hardware currently is underutilized

Energy consumption of the IC

Activity	Energy Used, in Joules (J)
A single Google Search [1]	1,080 J
A single Solana transaction	*1,837 J*
Keeping an LED light bulb on for one hour [2]	36,000 J
Using a fully-charged iPhone 13 on battery [3]	44,676 J
Working for an hour with a computer and monitor [4]	46,800 J
One eth2 transaction [5]	126,000 J
Watching an hour of television on a 40 inch+ LCD TV [4]	540,000 J
Playing one hour of a PlayStation 5 game [6]	708,840 J
Running a refrigerator for one hour [4]	810,000 J
One hour of central air conditioning [4]	12,600,000 J
Using one gallon of gasoline [7]	121,320,000 J
One Ethereum transaction [8]	692,820,000 J
One Bitcoin transaction [9]	6,995,592,000 J

- ~3300 transactions / s → 261.45 Ws per transaction = 261.45 Joule
- Conservatice: hardware currently is underutilized

Solana Energy Usage Report

Questions? Reach out to:

stefan.kaestle@dfinity.org
ulan.degenbaev@dfinity.org
adam.bratschikaye@dfinity.org

We are hiring: dfinity.org/careers