Framework for Distributed Application Development

Sanskar Amgain, Sandesh Ghimire
Pranjal Pokharel, Tilak Chad

Supervisor: Asst. Prof. Dr. Babu Ram Dawadi

What our project is?

A framework for programmers to write distributed applications.

The framework must be evaluated from a programmers's perspective, as in the level of **abstraction and control** it offers programmers.

Two sample programs written using the features of framework.

To demonstrate the features of this framework, we have developed a **distributed whiteboard** and distributed **mandlebrot computation**.

Academic research on distributed computing, networking, and more.

The primary focus of our project is on academic study and research, hence more focus on the framework internals and code over application development.

What our project is NOT?

NOT a demonstration of application development at industrial scale.

Testing results for the framework at industrial scale (1000s of clusters) has been infeasible **due to limited time, budget and resources** at our hands.

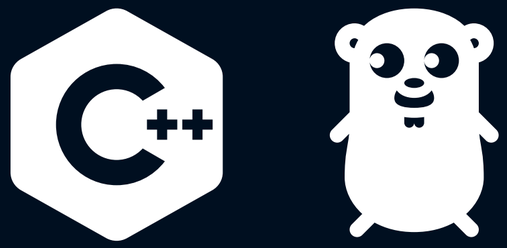
NOT an end-user product that can be used out of the box.

This is not an application but a *framework to develop* applications. The programmers using it must have astute knowledge of its API and networking knowledge.

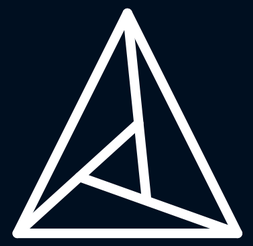
NOT a collection of 3rd party libraries and dependencies.

Everything from making connections to entire filesystem is original with no external dependencies. **We've built everything from scratch** (research).

Tools

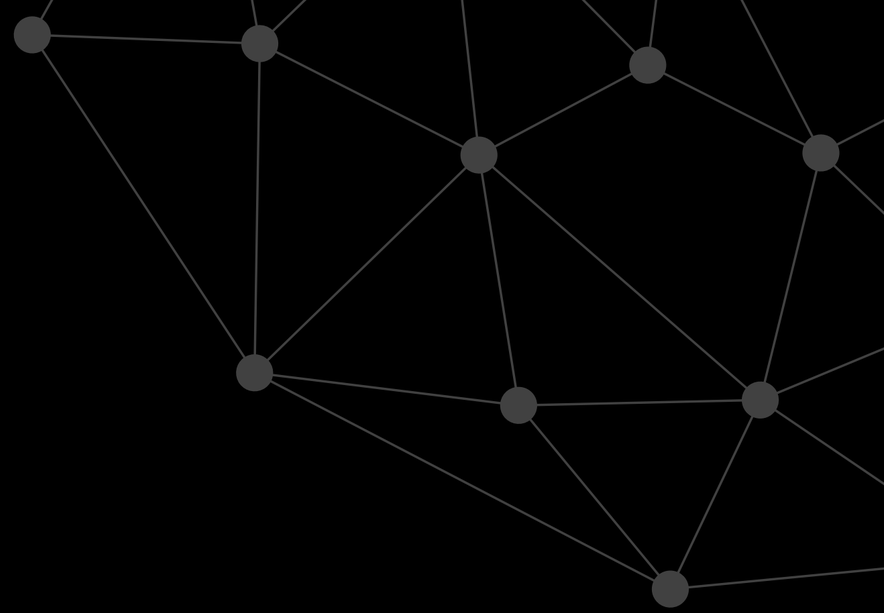


Languages: C++, Go



Build System: CMake

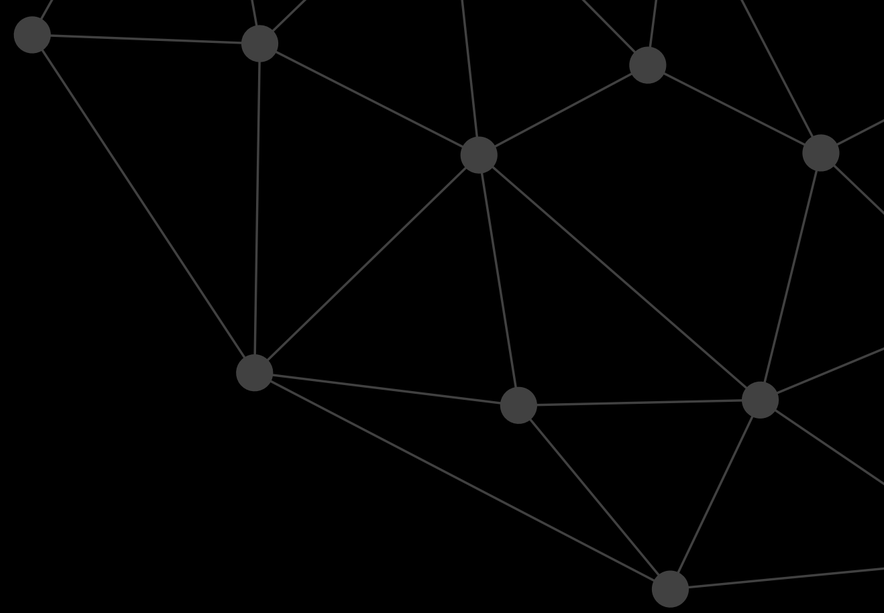
Platforms: Linux, Windows



Framework Overview

Overview of what the feature offers to a distributed system programmer

(and how it does so)



Connecting to the Network

The first requirement for any distributed application

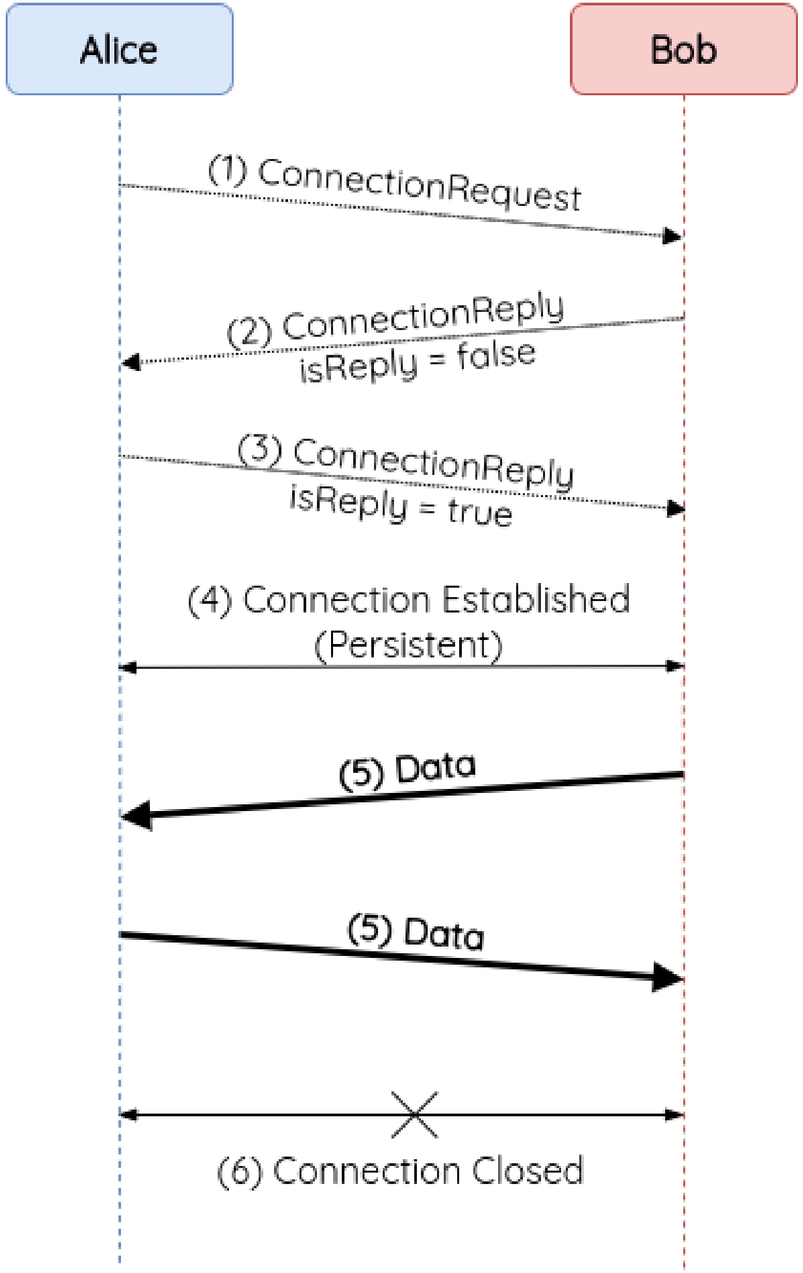
Connecting to Node

(1) (2) (3) Initiate connection and send each other acknowledgements.

(4) Establish persistent connection (re-use this TCP connection).

(5) Send data (bi-directional).

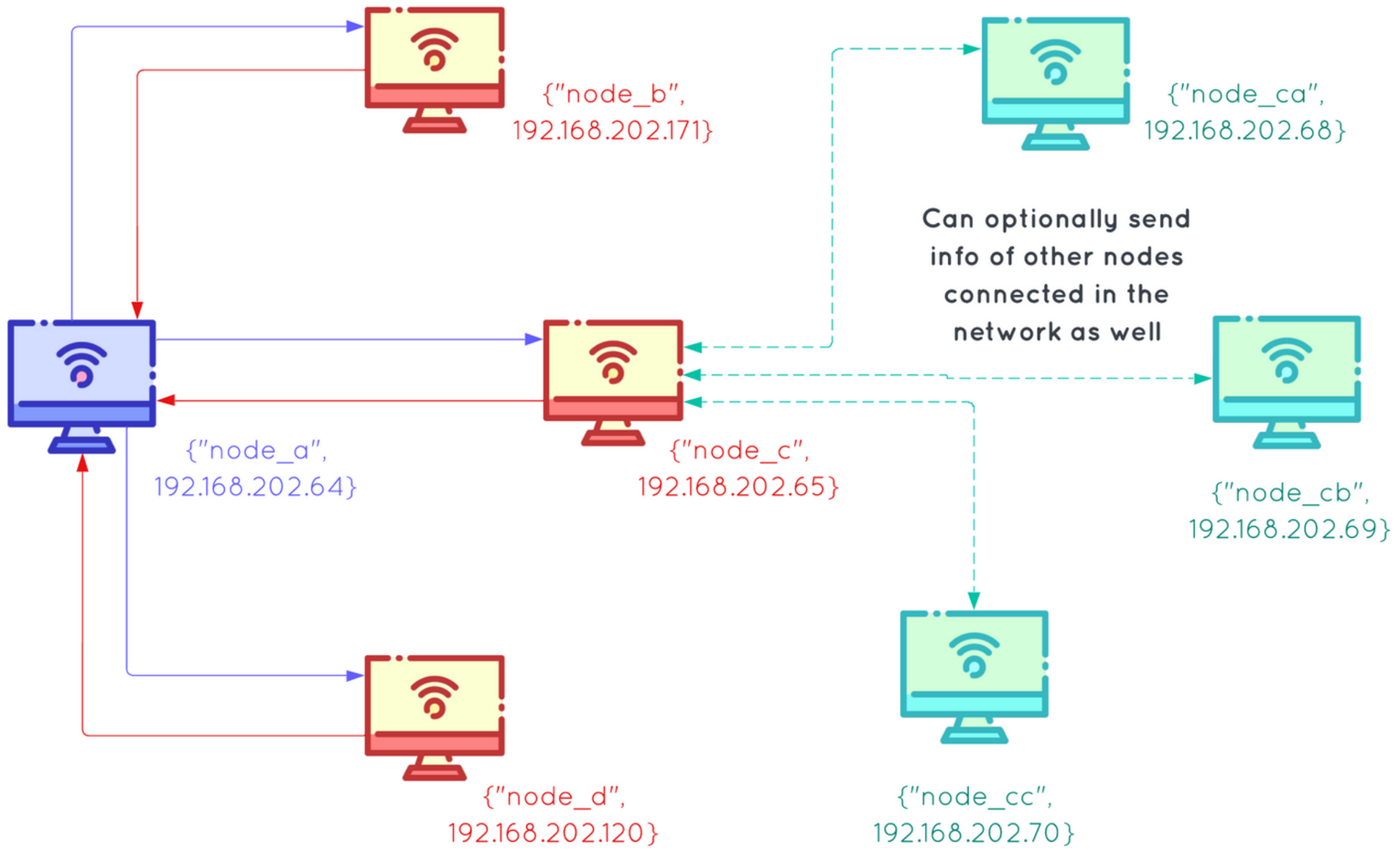
(6) Close connection.



Connecting to Network

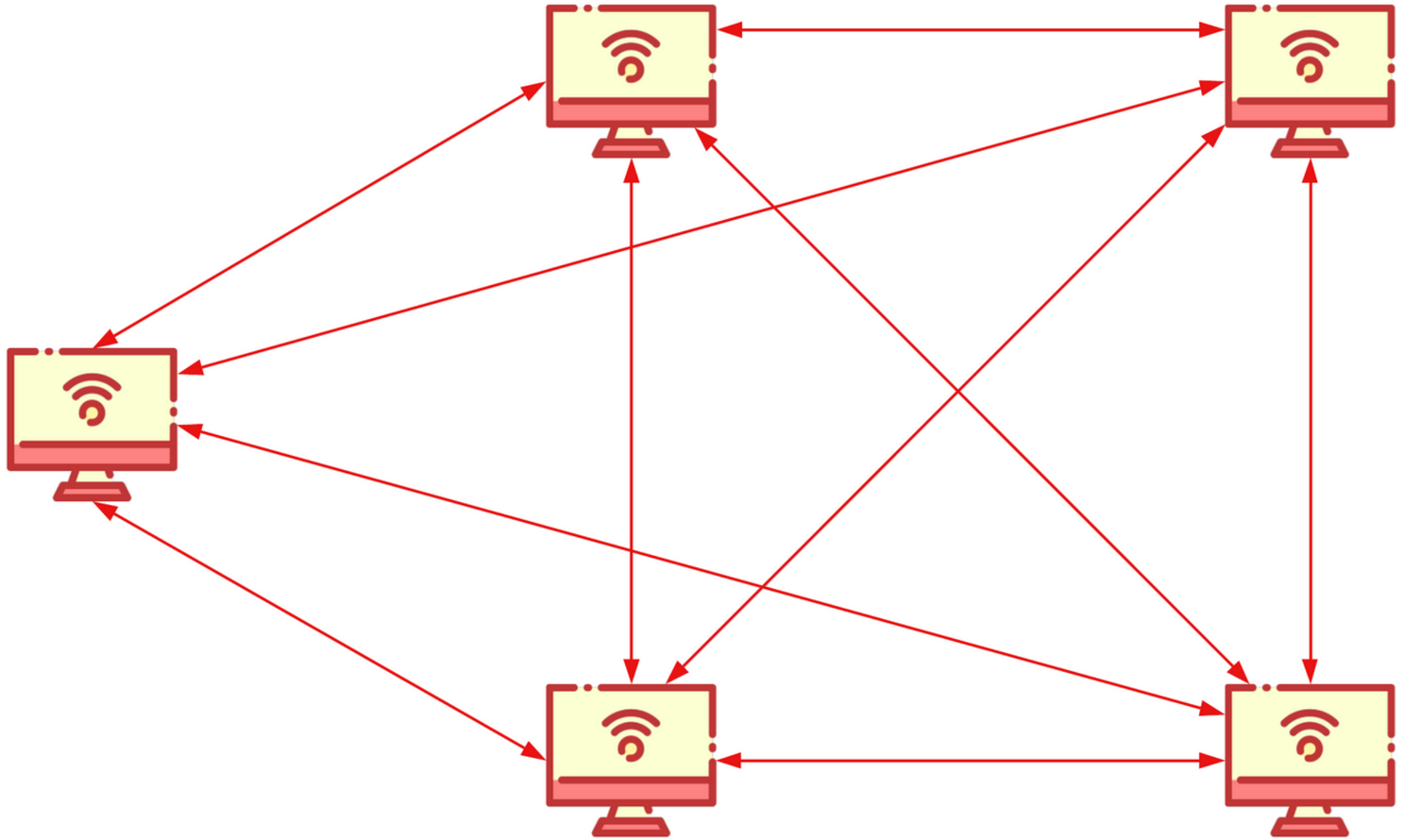
Every node is provided a **list of nodes** it can connect to (hard-coded).

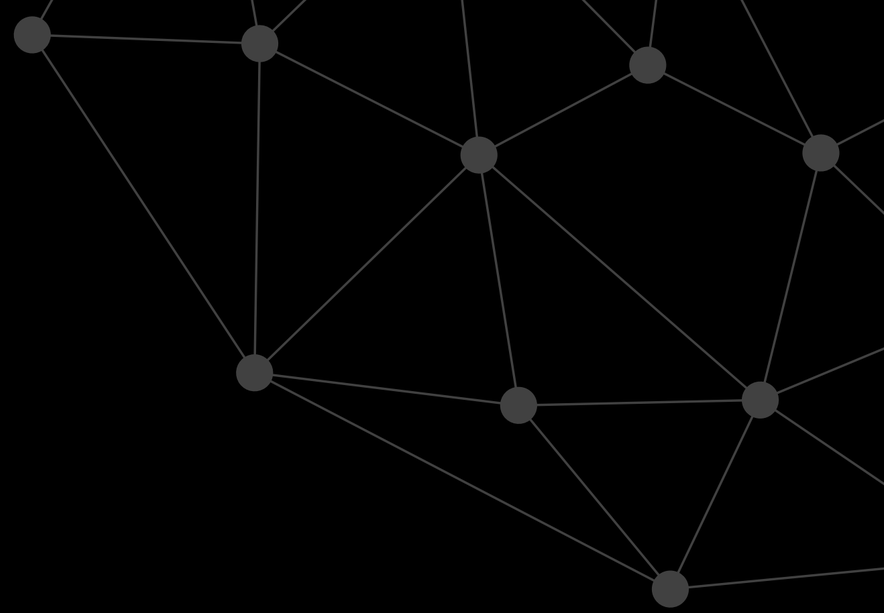
Optionally, the programmer can decide to **automatically fetch network information** when connected to a node in the network.



Peer-to-Peer (P2P) Model

There is no single master server and all nodes communicate with each other through p2p connection.



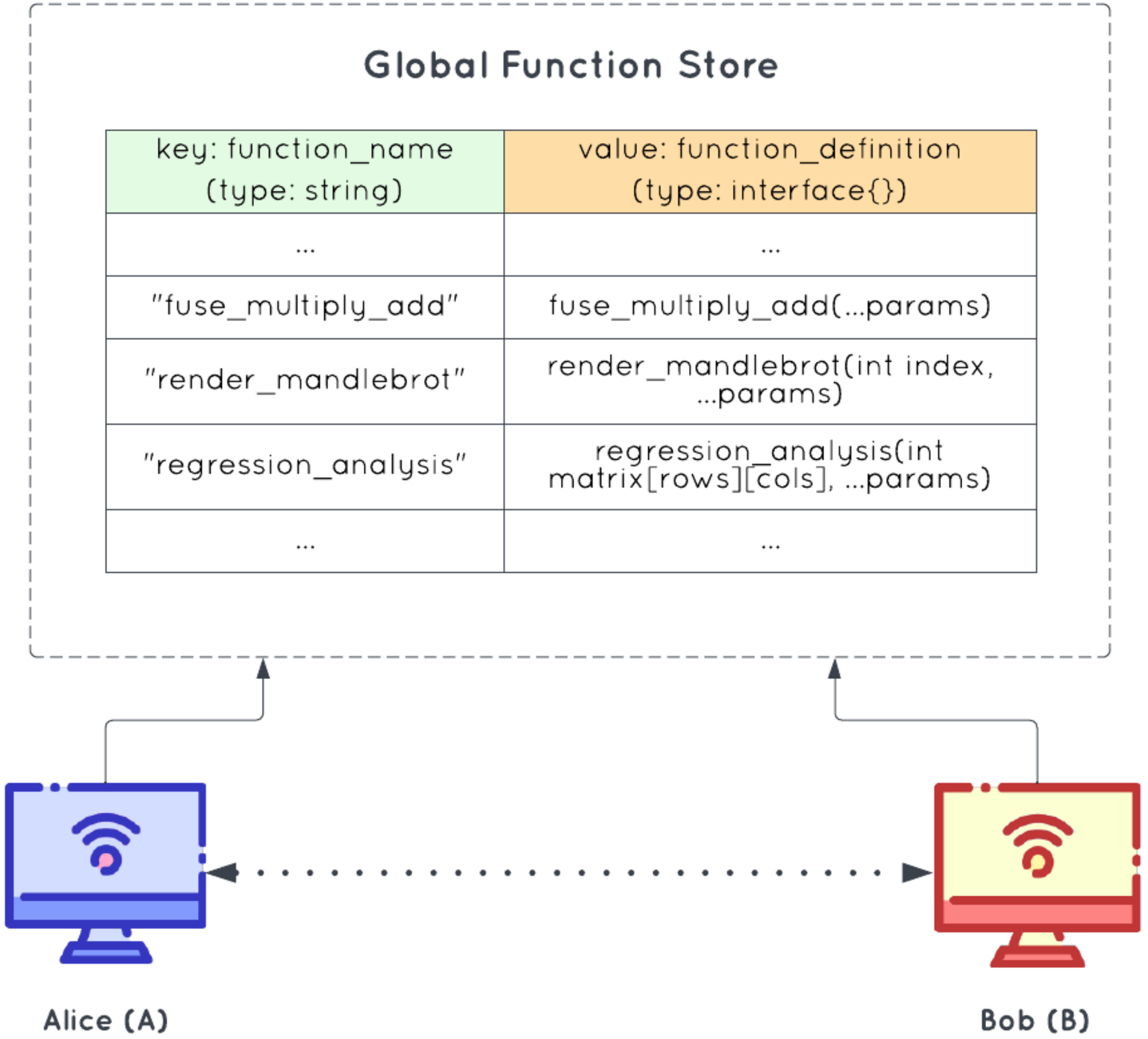


Inter-node Function Call

Distributing **procedures (processing)** across nodes

Global Function Store

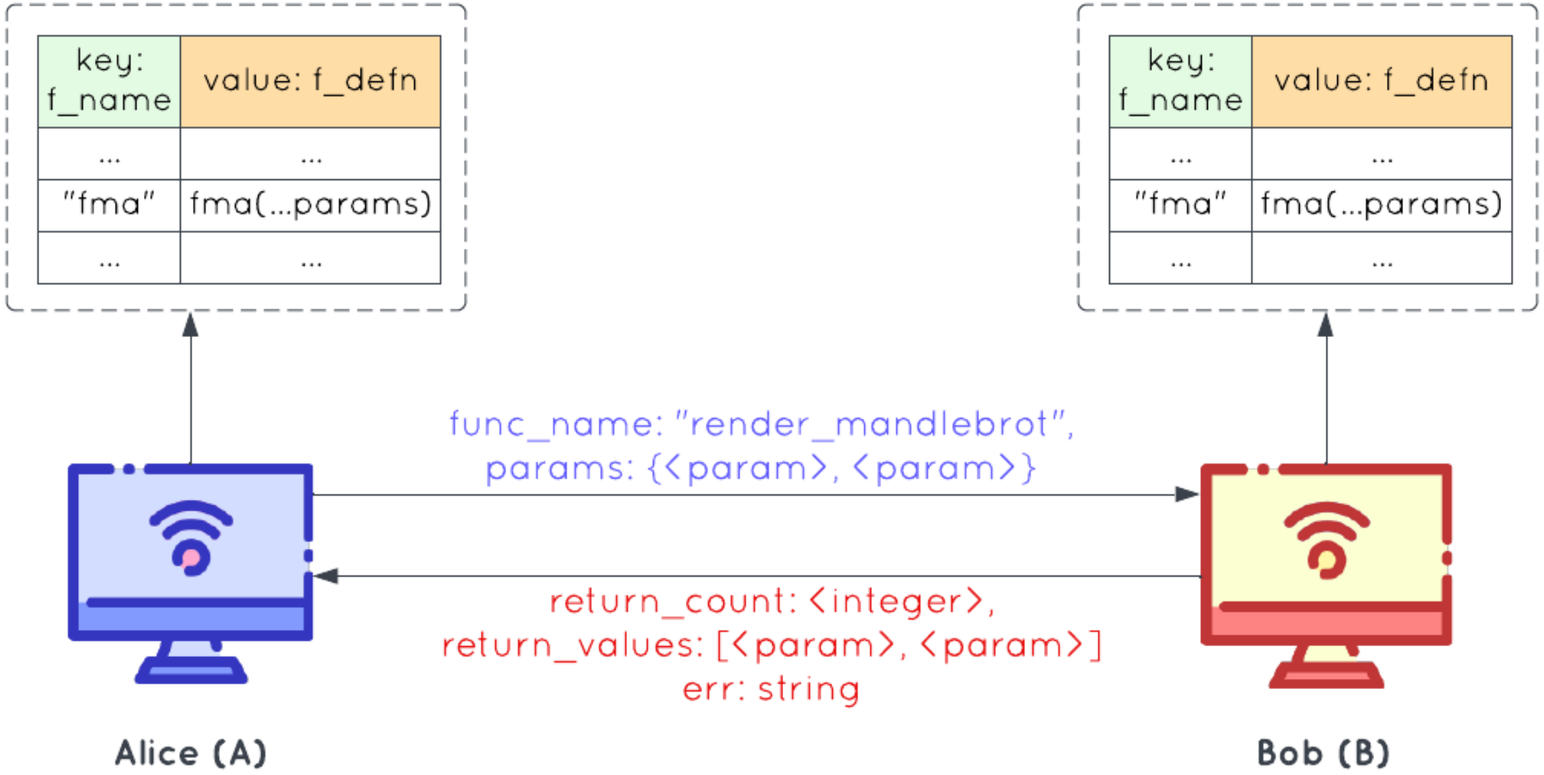
Store callable functions in a **global function store** accessible by all nodes.



Global Function Store (contd.)

....is not actually global!

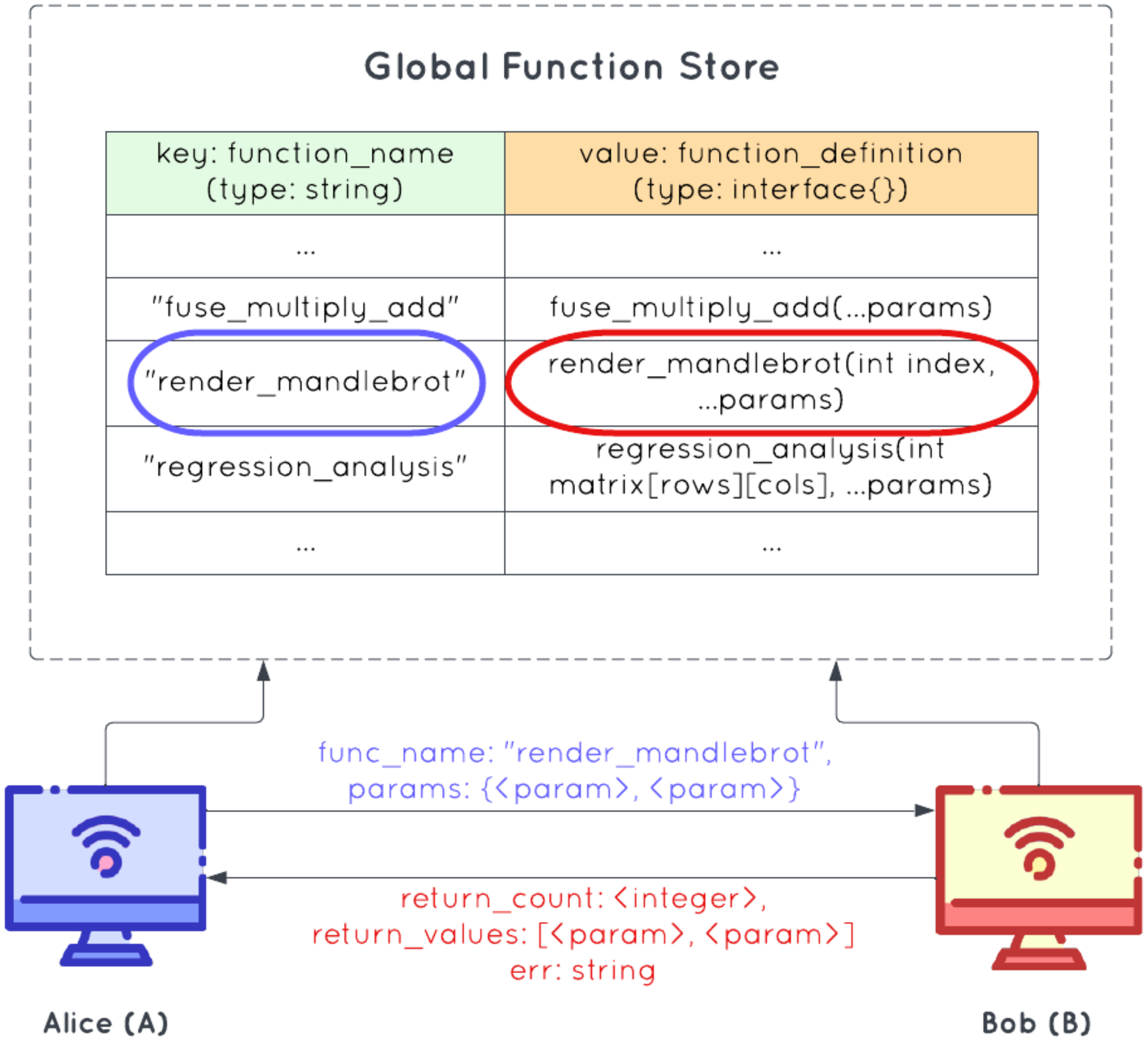
Each node has it's **own store** that is **synced up globally**.



Call and return

1) Alice calls "render_mandlebrot" with required parameters; Bob has function definition.

2) Bob returns result of processing; Alice aggregates the result.

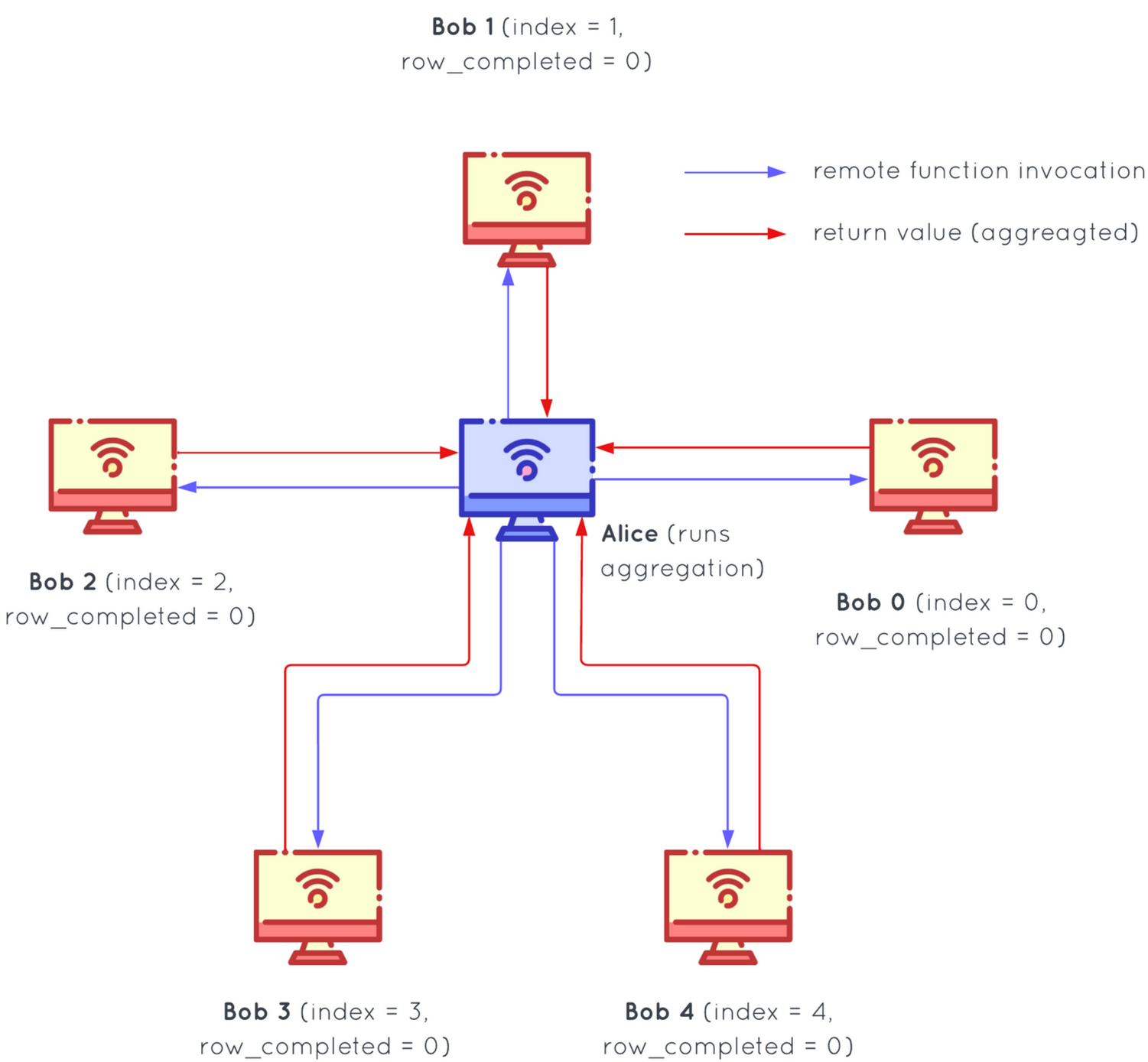


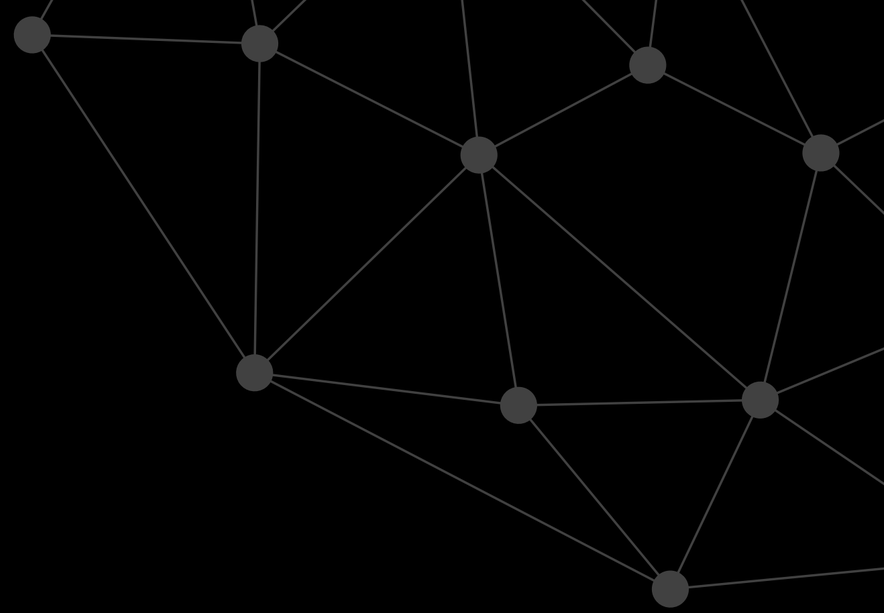
Expanding...

This **calling remote functions and aggregating results** is spread out through multiple nodes, where parameter to each node is differentiated by,

1) index
2) row_completed

This process is analogous to *MapReduce*.





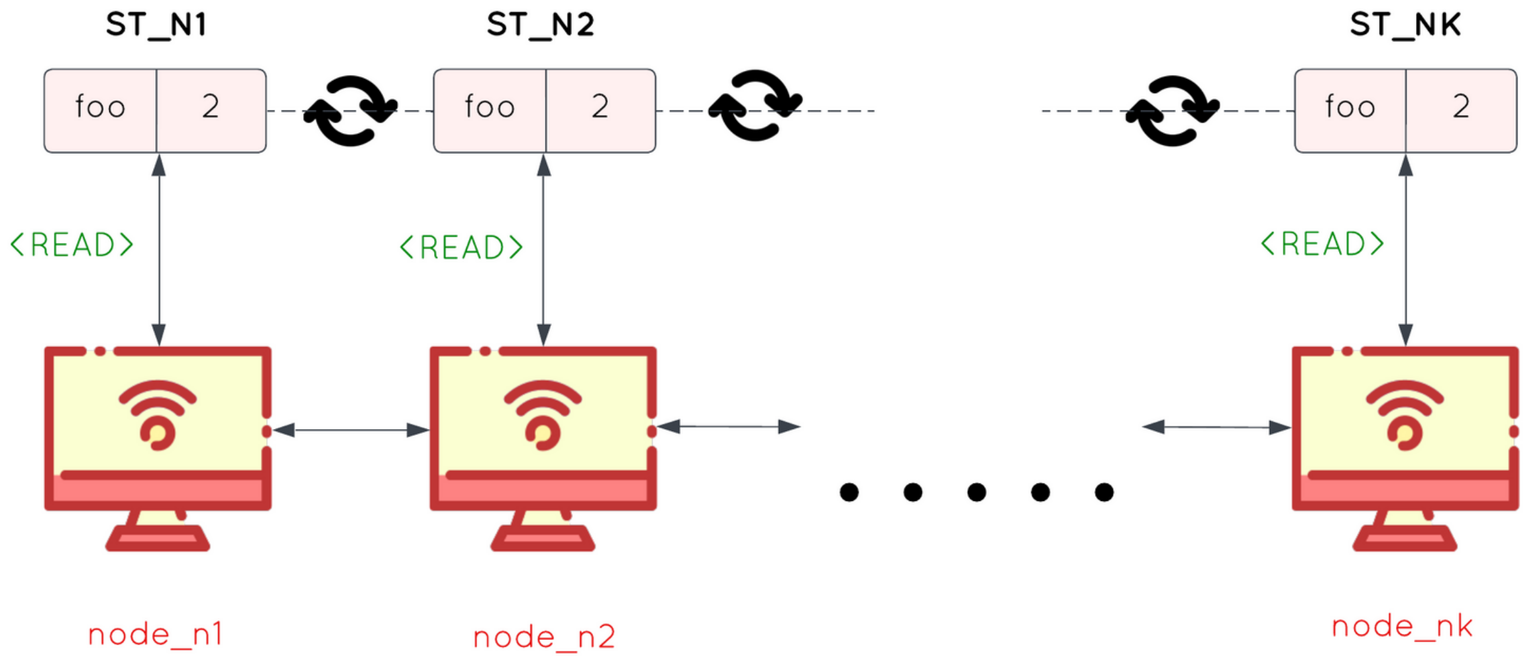
Distributed Variable

Distributing **value (state)** across nodes

Sync Values Across Nodes

Distributed variable, once created in an arbitrary node, can be **accessed by all nodes** in the network.

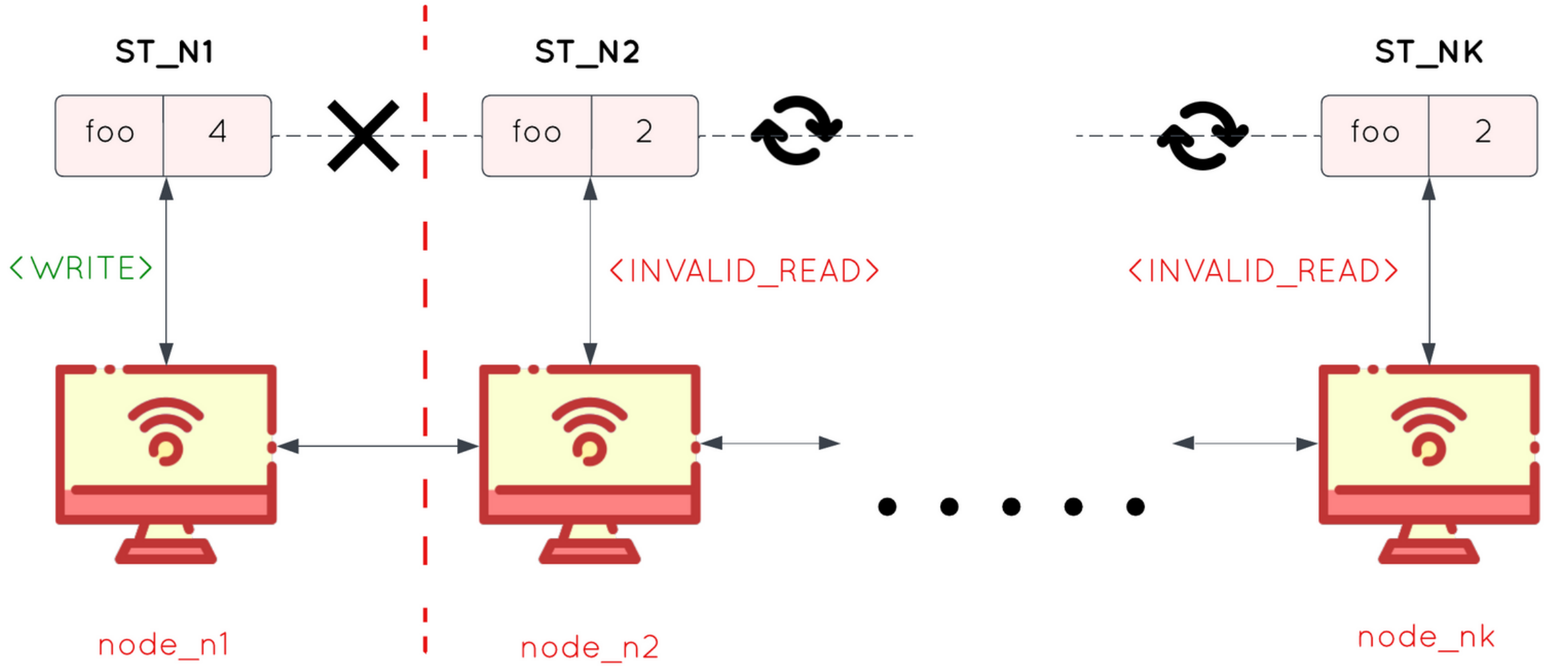
All nodes have a **symbol table** of distributed variables.



Updating Value?

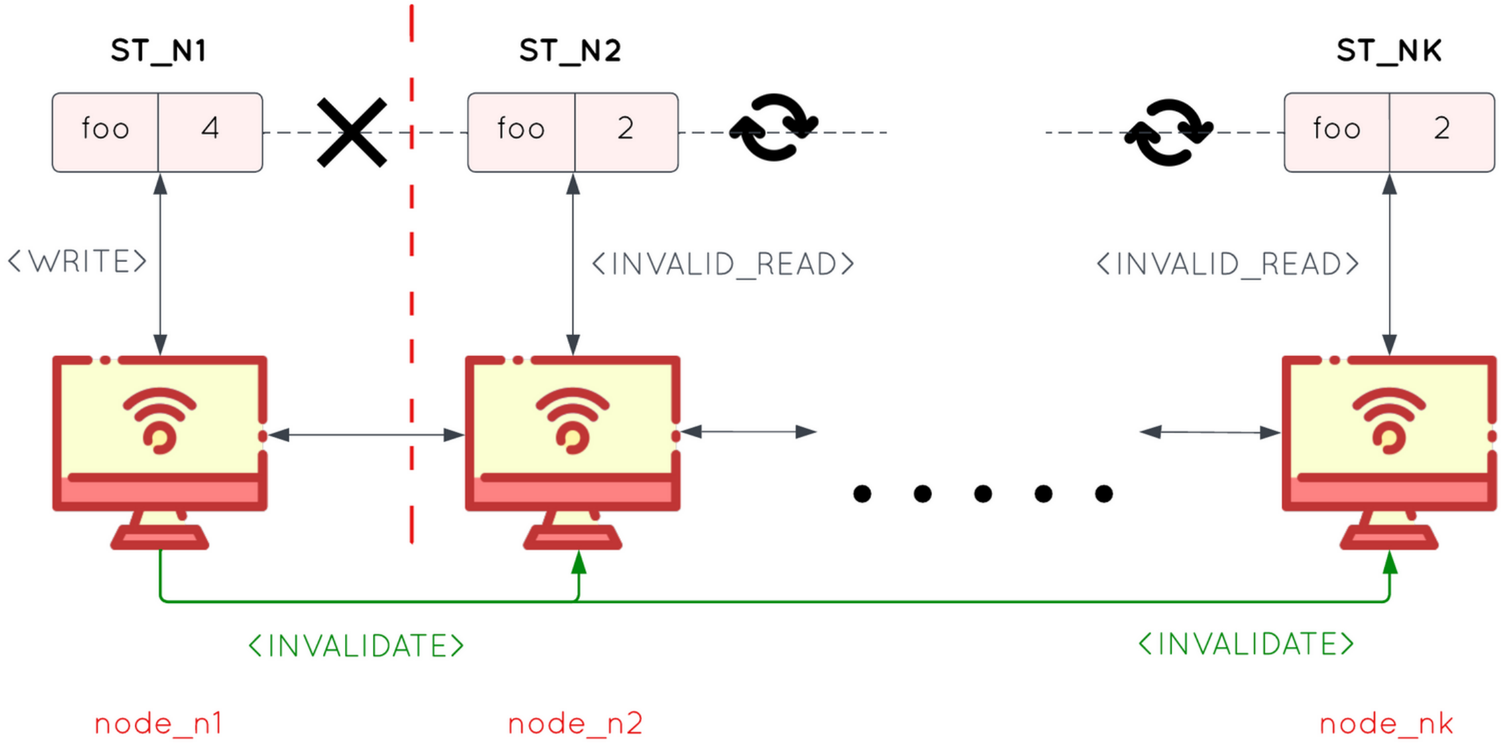
Q: *What happens when the variable is updated?*

A: The node that updates the variable locally has the correct value but other nodes have no idea the value has been changed.



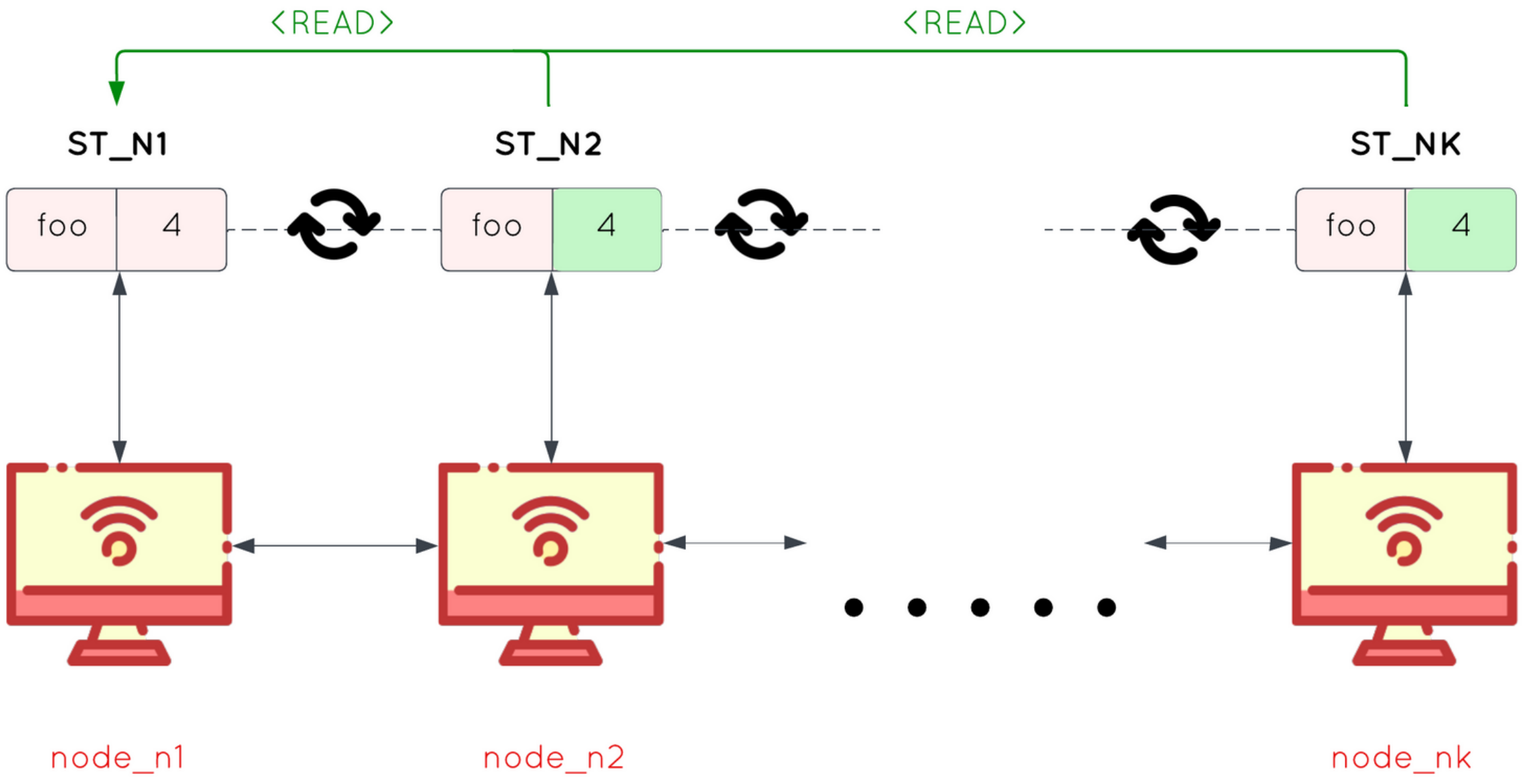
Solution: Write Invalidation Protocol (WIP)

The node that updates the value of the distributed variable sends a value **invalidation signal** to all nodes in the network.



Synchronize

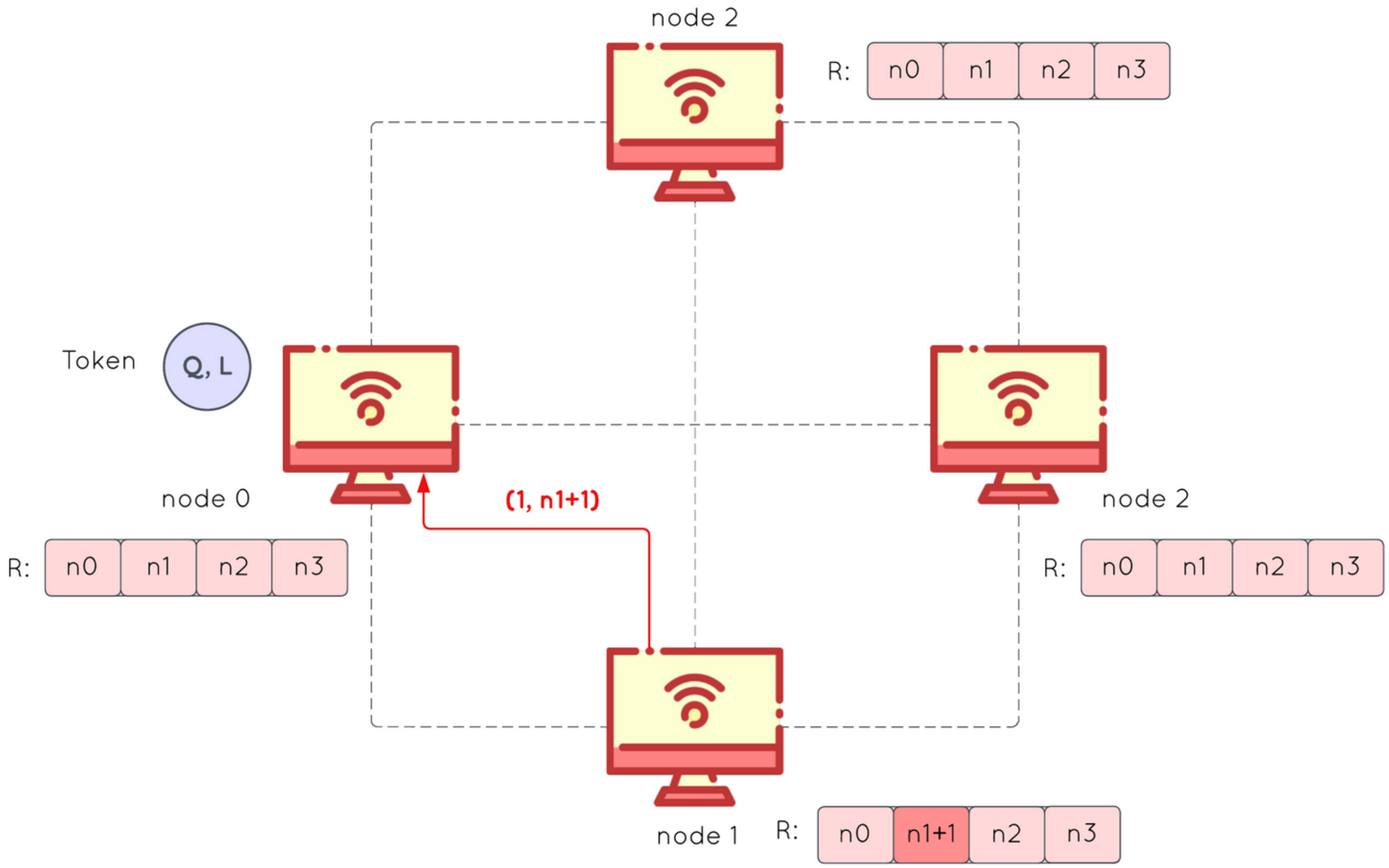
If a node receives an invalidation signal, the node requests the source of the signal to send the new value (state) of the distributed variable.

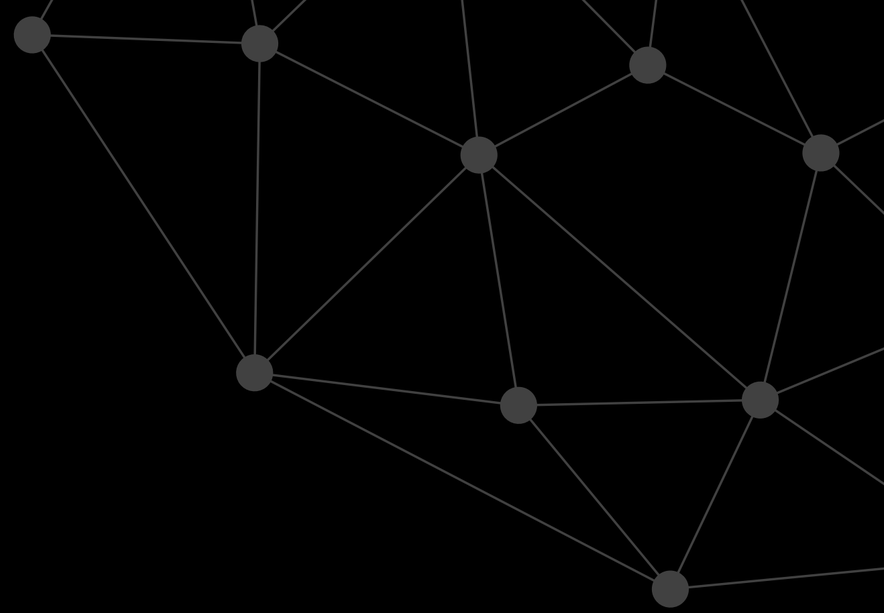


Distributed Mutex: Suzuki Kasami

To prevent race conditions, we have implemented a token-based locking **(Suzuki-Kasami)** algorithm.

This is a single, global mutex for entire network that **doesn't require time synchronization**.





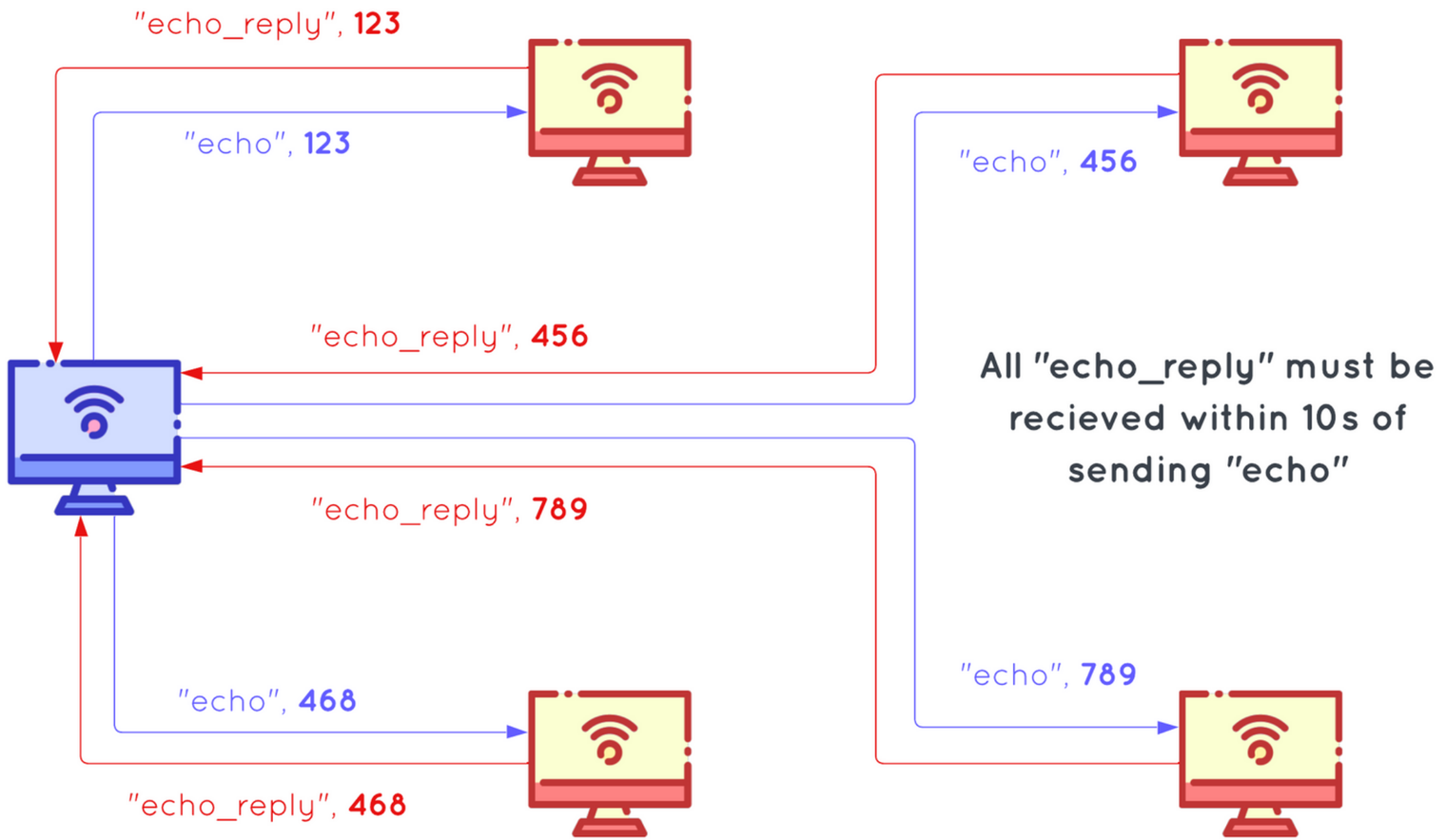
Node Failure Detection

How do we check the status of a node?

Echo Check (Polling)

Alice **requests** each node in the network she is connected to with an **"echo"** message.

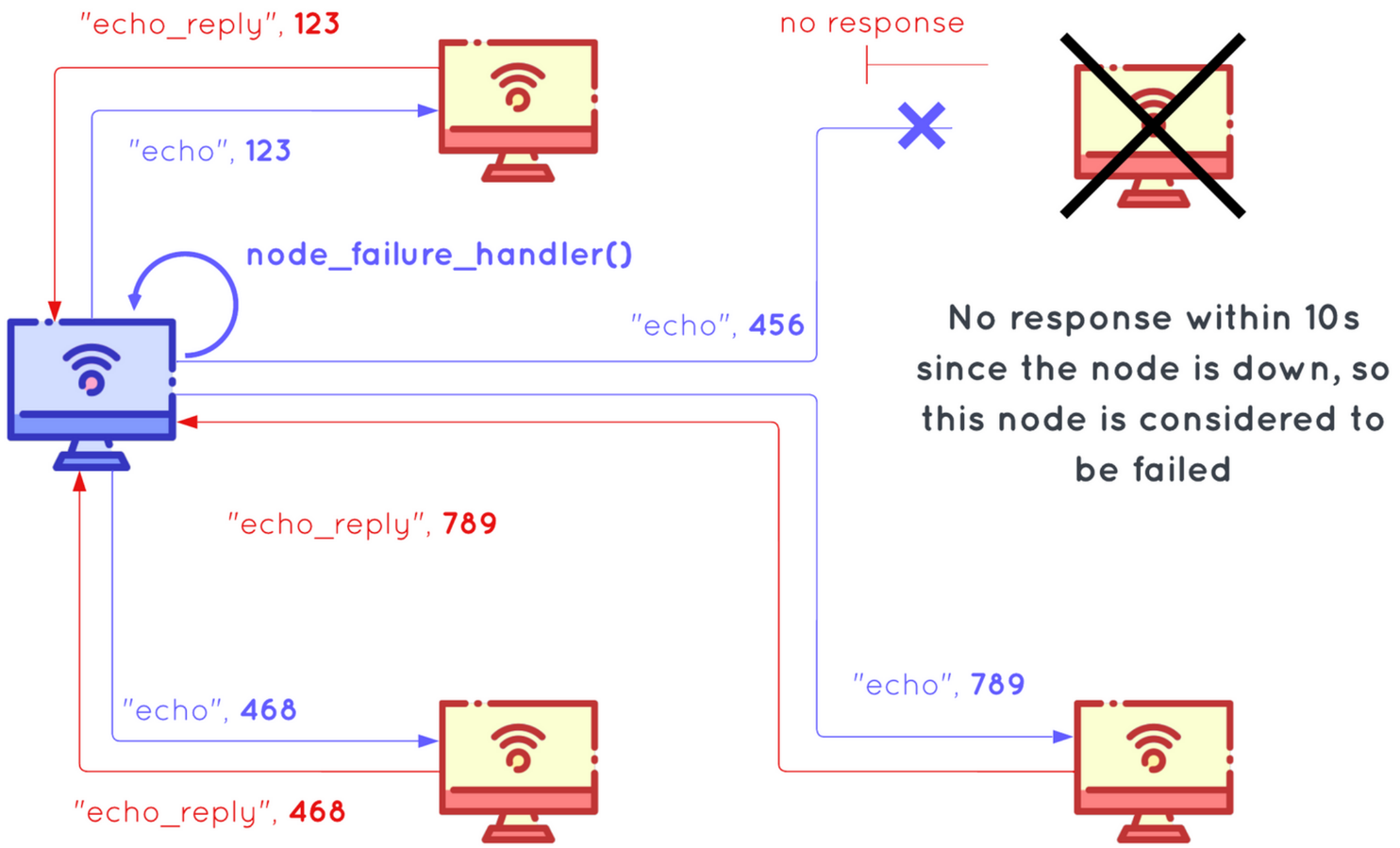
If an "echo_reply" **response** is **received from each node within 10 seconds** then node is marked as **available.**



Failure Detected!

A failed node doesn't reply with "echo_reply" in the given time frame, so the **framework doesn't send further requests** to it (*network partition*).

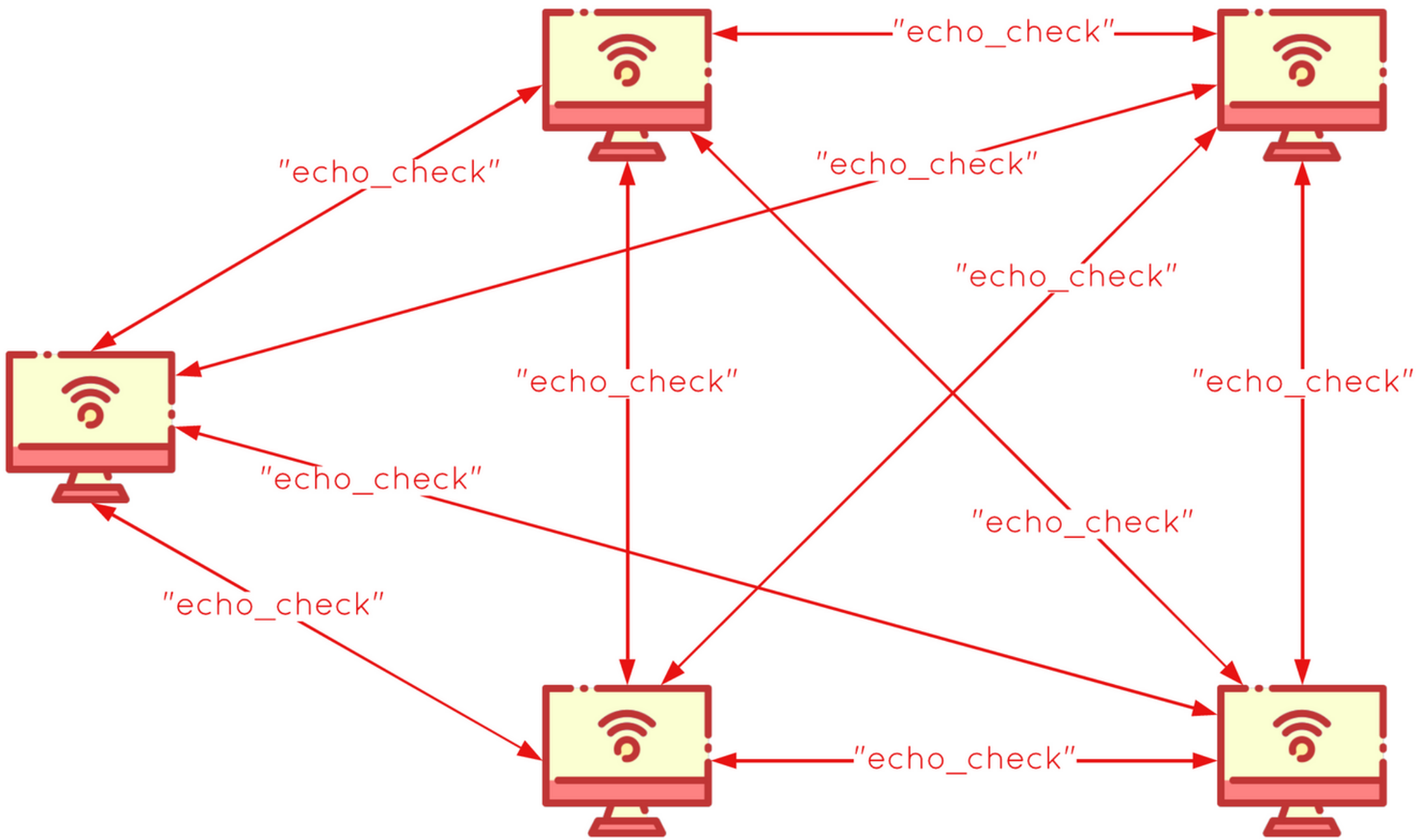
The programmer is allowed to define **callback function** that executes **when node failure is detected**.



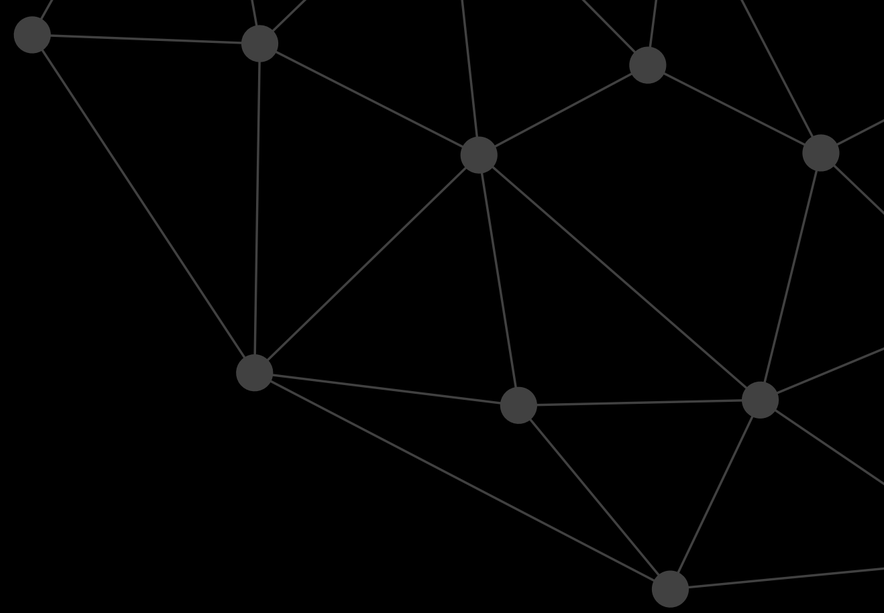
What actually happens...

Every node on the network keeps **polling** all currently connected nodes to check for node failure.

Each node can have it's own callback function defined.



1) every node broadcasts an "echo" msg every 20s
2) every node checks for "echo_reply" msg from all other after 10s



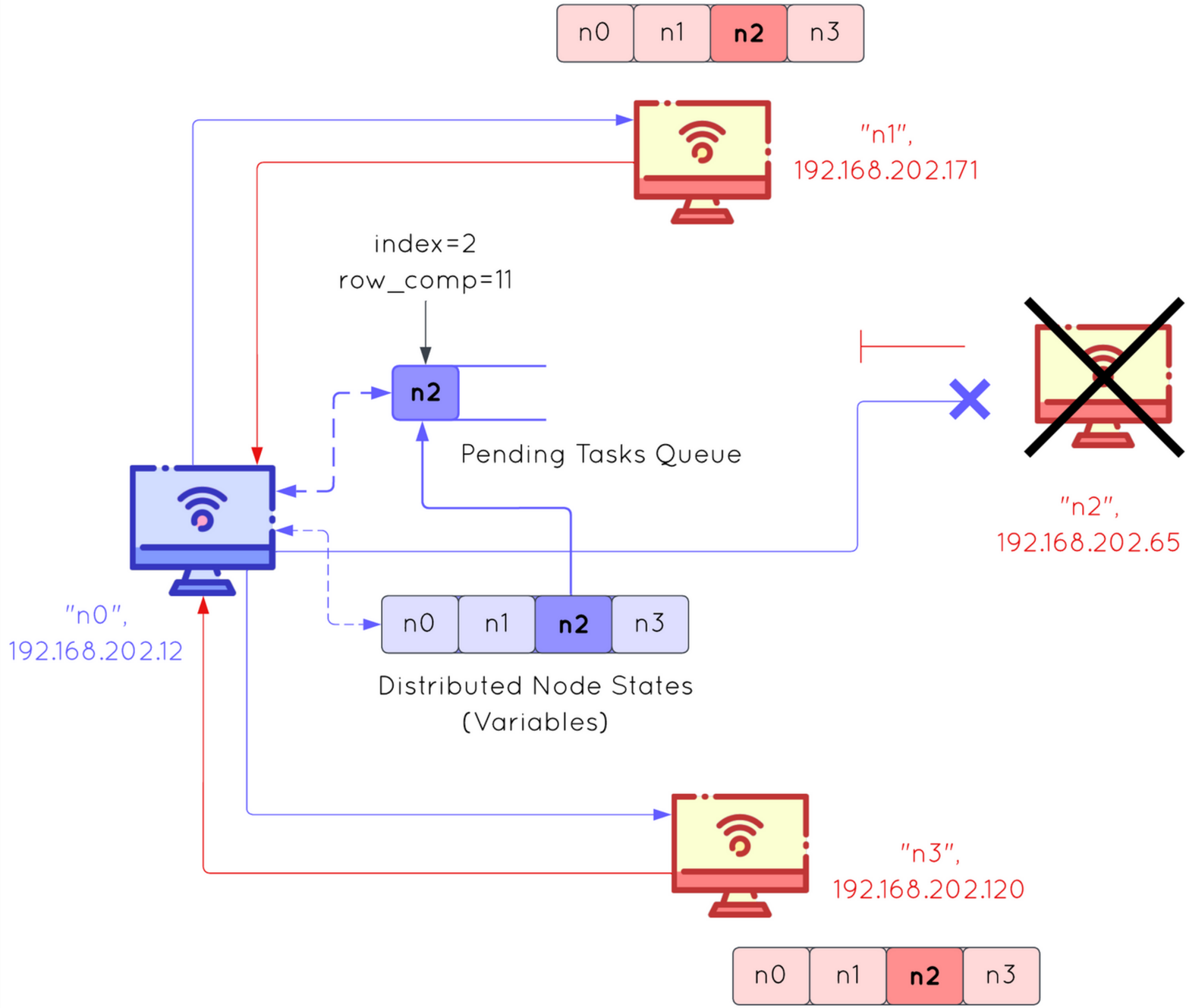
Handling Node Failure - Mandlebrot

Distributed Variable + Node Failure Callback

Pending Events Queue

Add the **latest state (distributed variable)** of computation from the failed node onto the **pending events queue**.

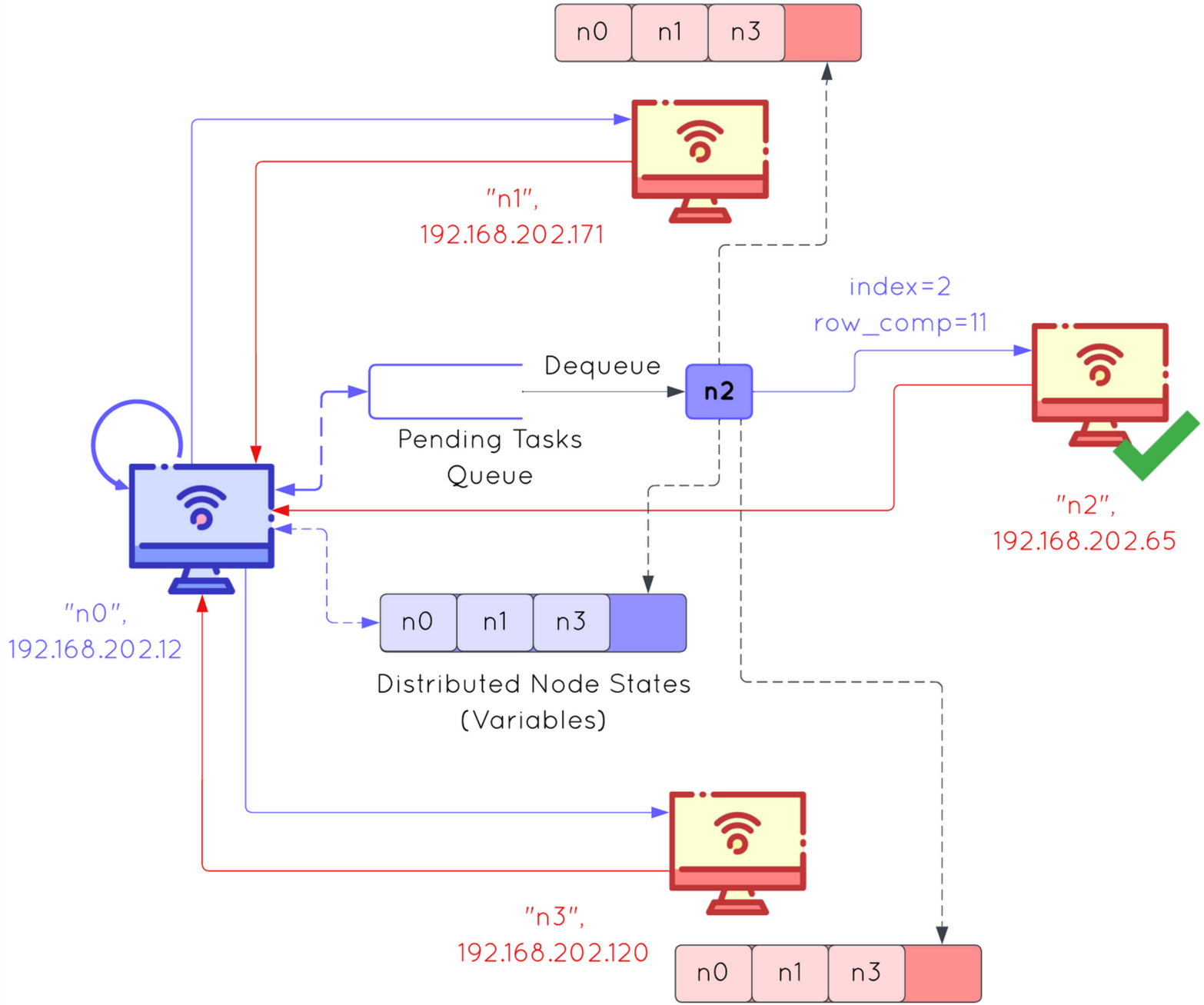
The queue is local to the node that initiates the mandlebrot computation.

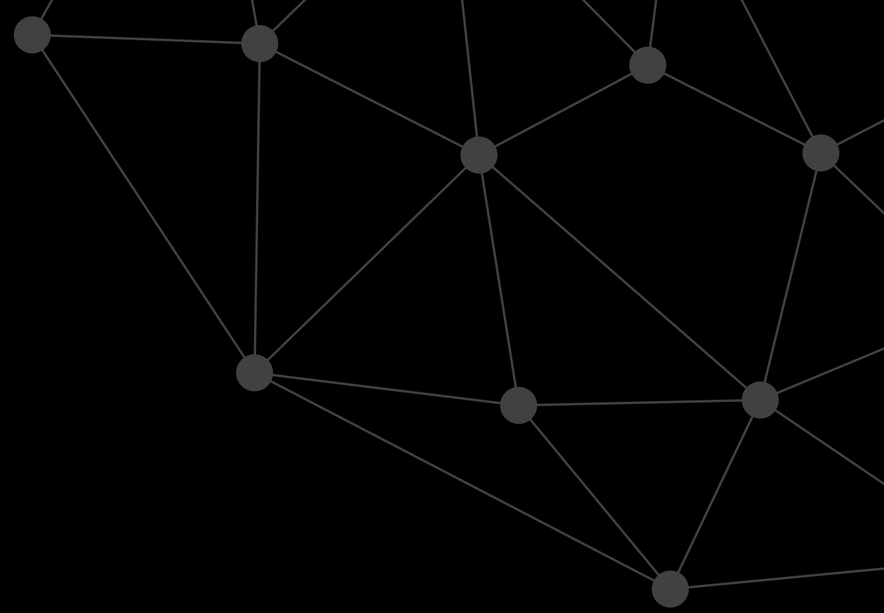


Callback on Node Failure

When the failed node or any other node is available for computation, **resume task from queue**.

The availability of another node is decided by the echo-reply mechanism (polling).





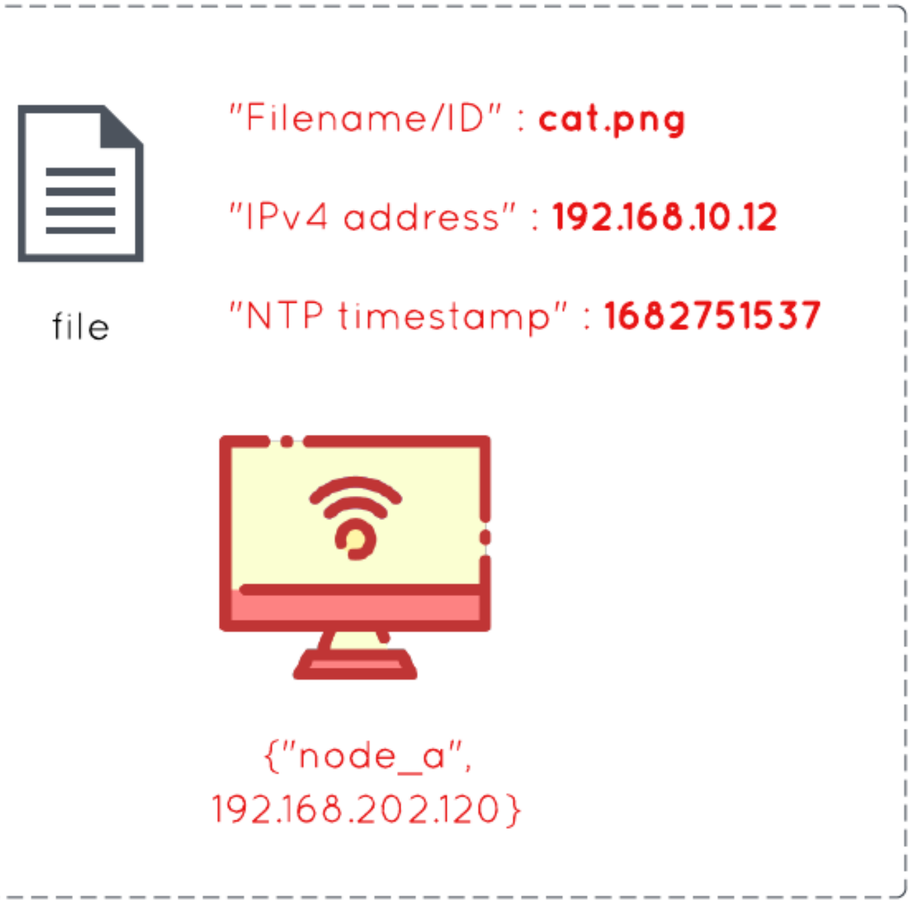
GutFS (Guthi File System)

For **persistence and file organization** within the network

File Structure

A file is uniquely identified in the filesystem through,

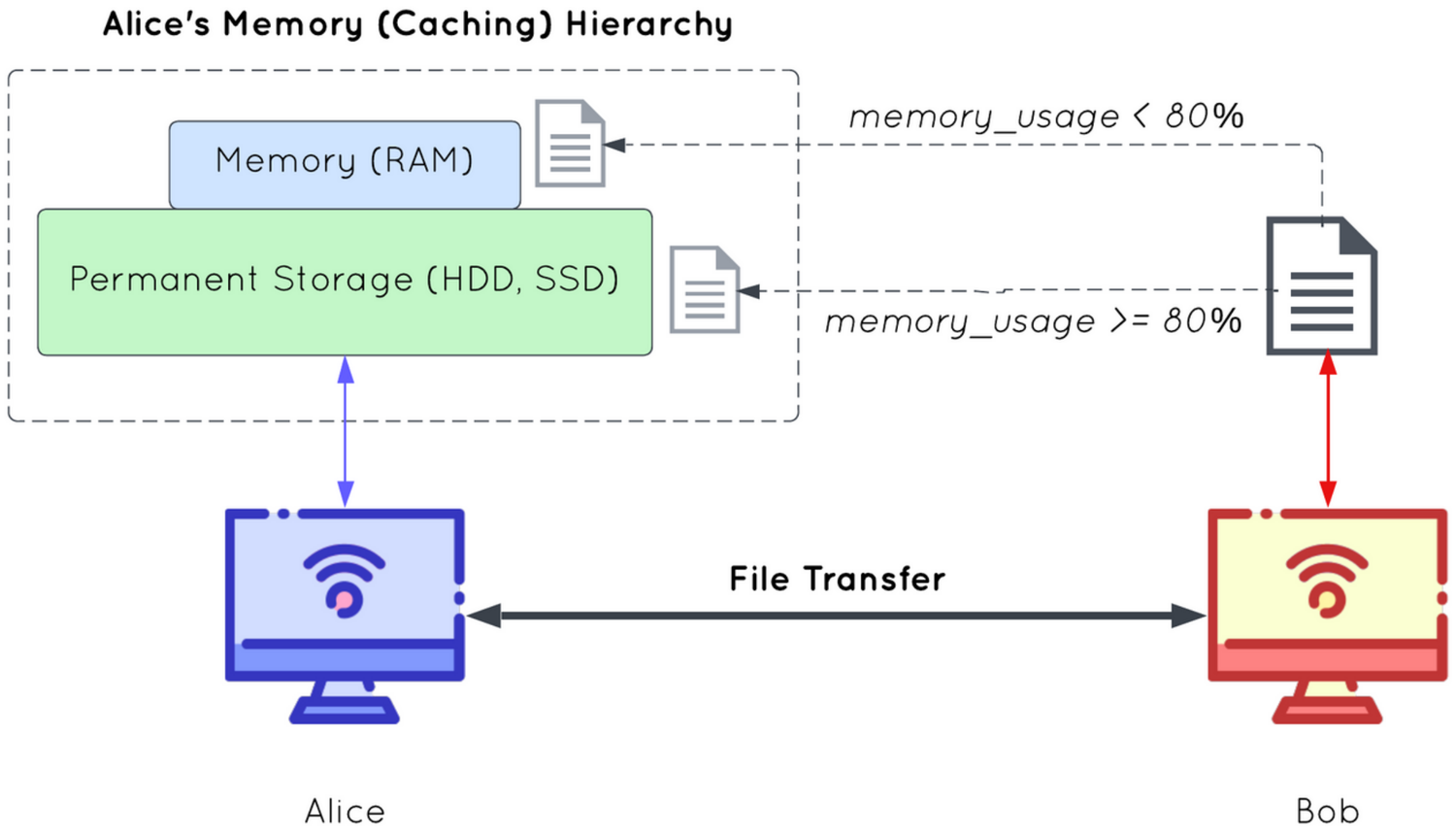
1) **File ID** (name)
2) **IPv4 address** (in which node is the file stored)
3) **NTP timestamp** (store latest modification date for caching)



Multi-Level Caching

If, *memory usage >= 80%*, files fetched from other nodes are cached on the **local disk (permanent storage)**.

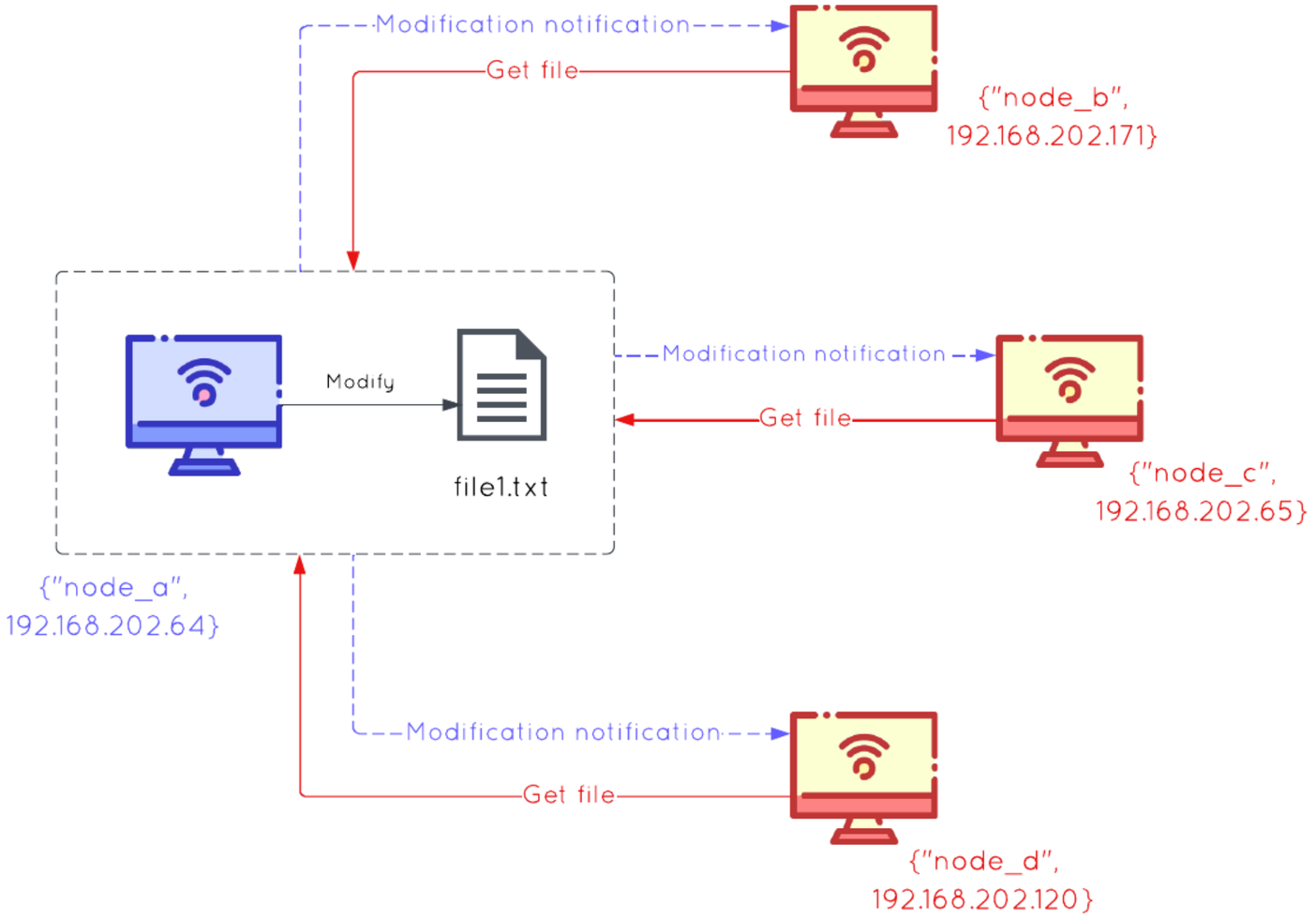
Else, fetched files are cached on the **main memory**.



Asynchronous File Tracking

Files and folders can be tracked for modification asynchronously (polling not required).

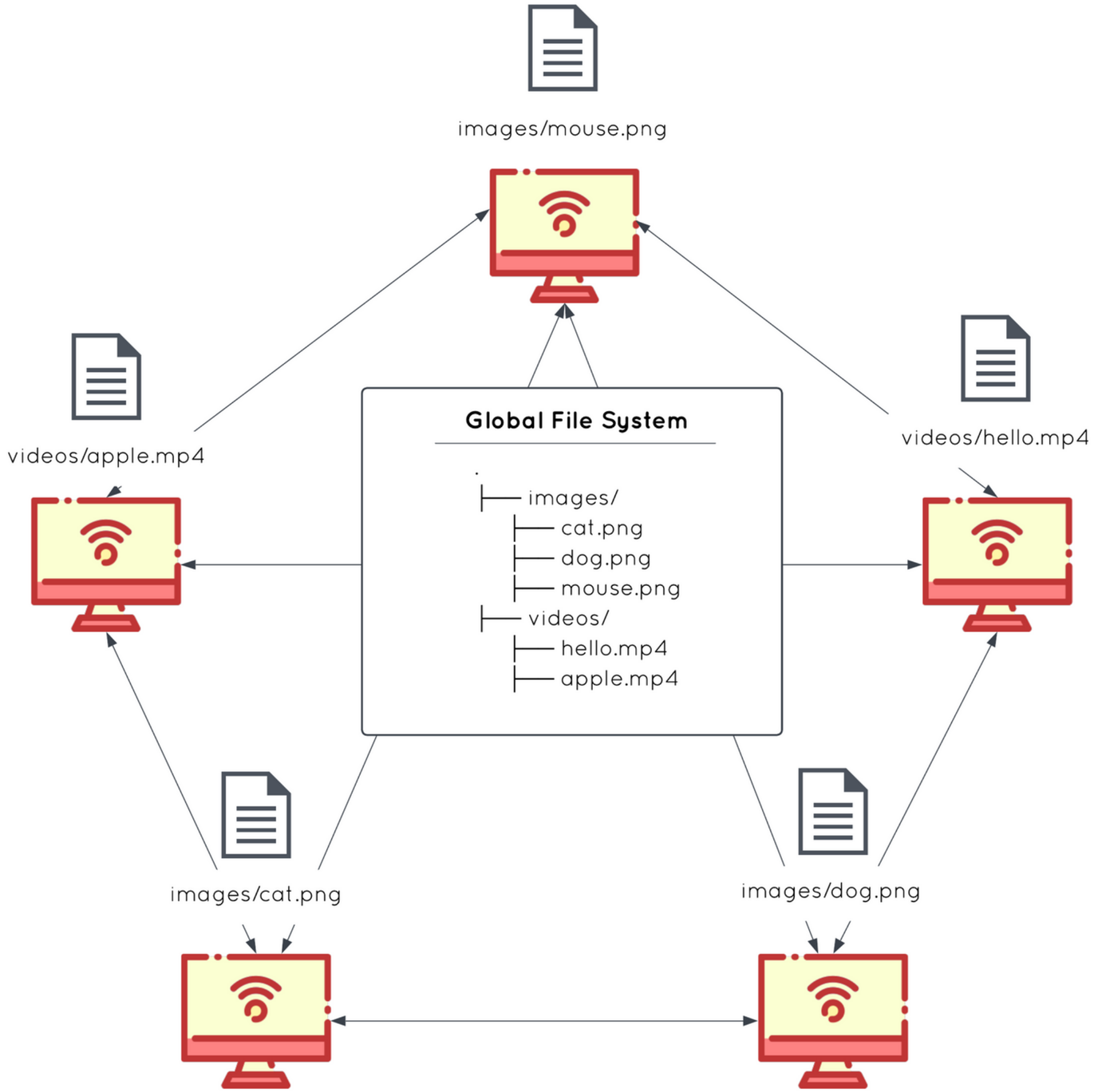
Modification event can be broadcasted to other nodes.



Global Filesystem

Each node queries other nodes for their local filesystem, eventually forming a network-wide **global file system**.

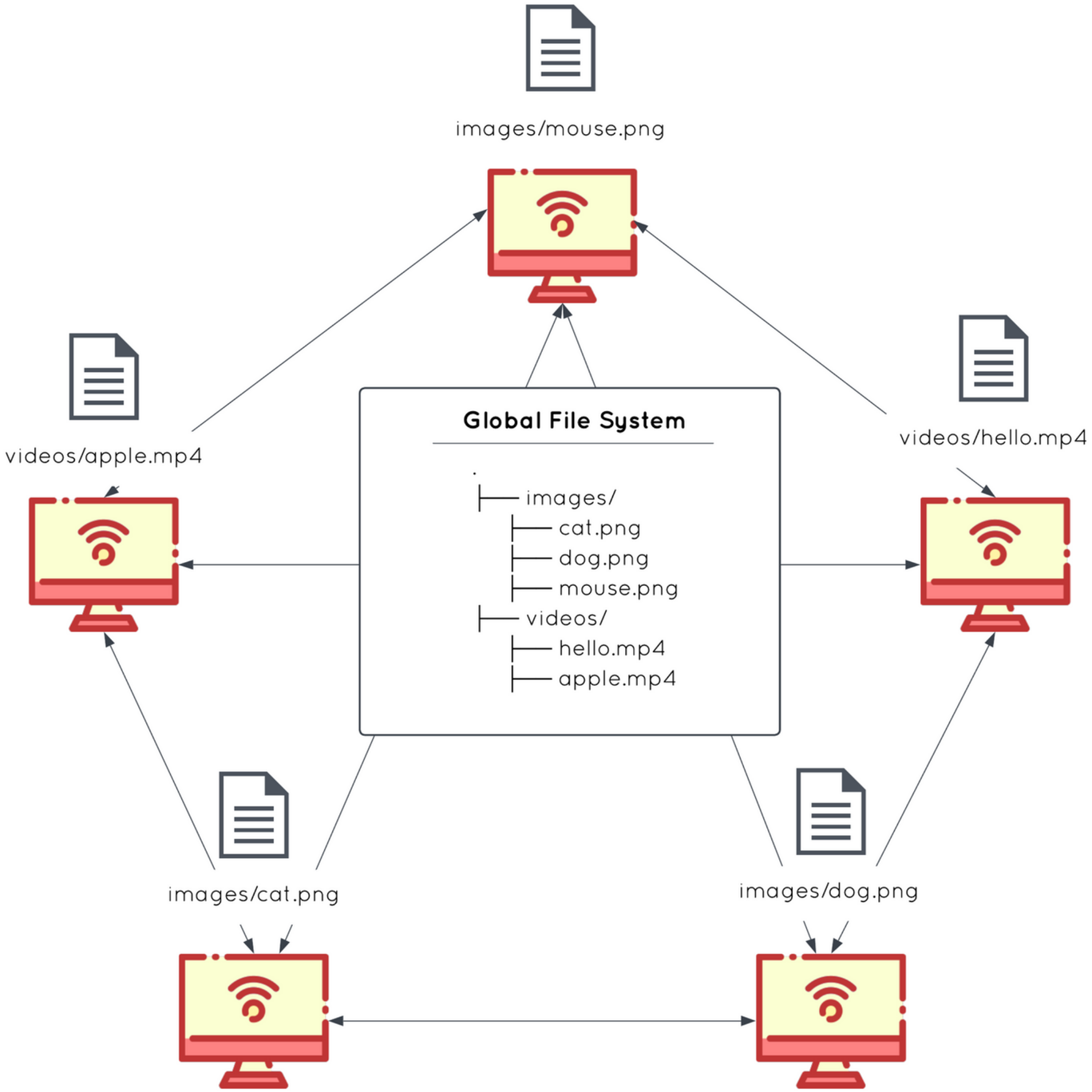
Since this is a P2P architecture, the global filesystem is visible fully for all nodes.



Global Filesystem

Each node queries other nodes for their local filesystem, eventually forming a network-wide **global file system**.

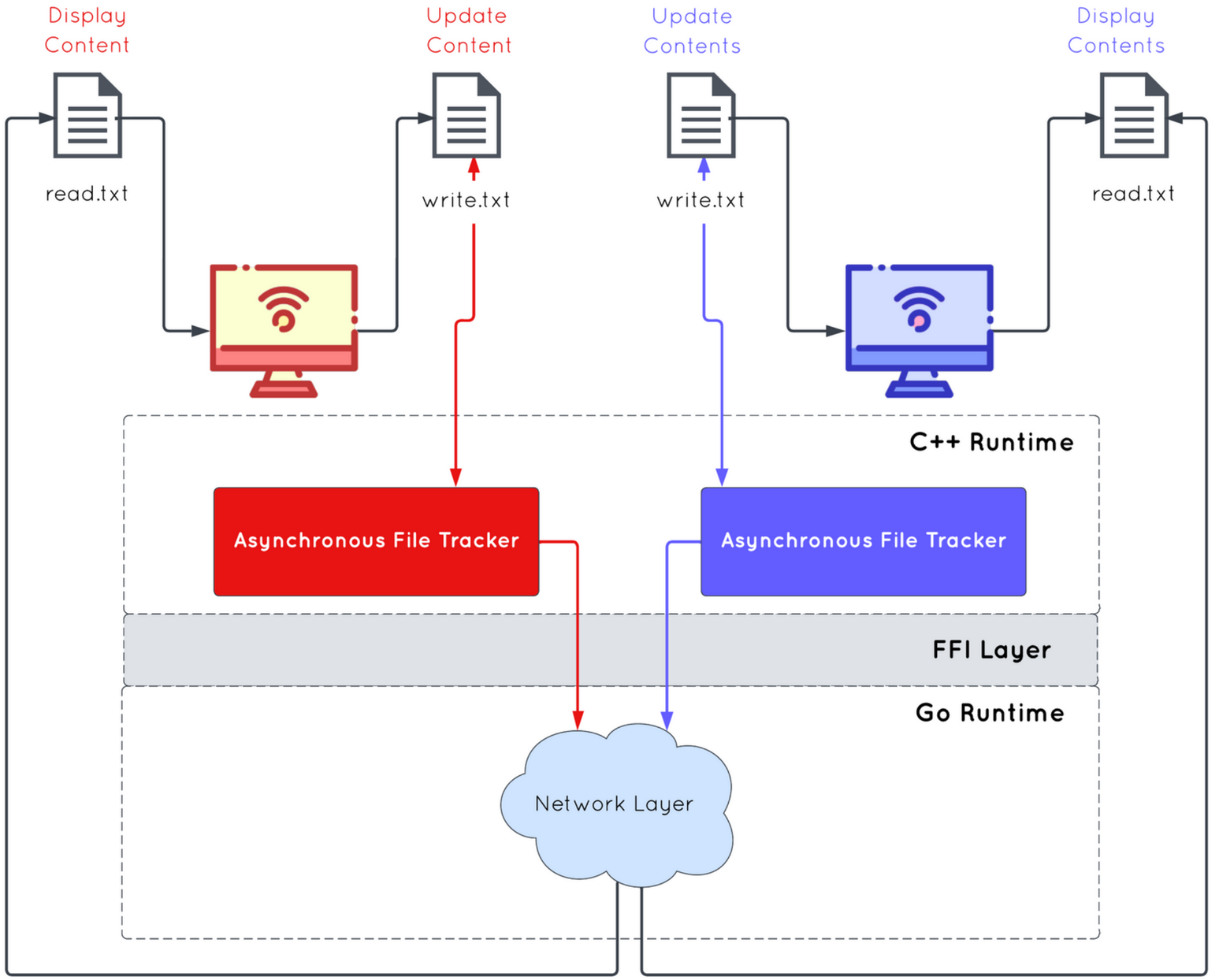
Since this is a P2P architecture, the global filesystem is visible fully for all nodes.

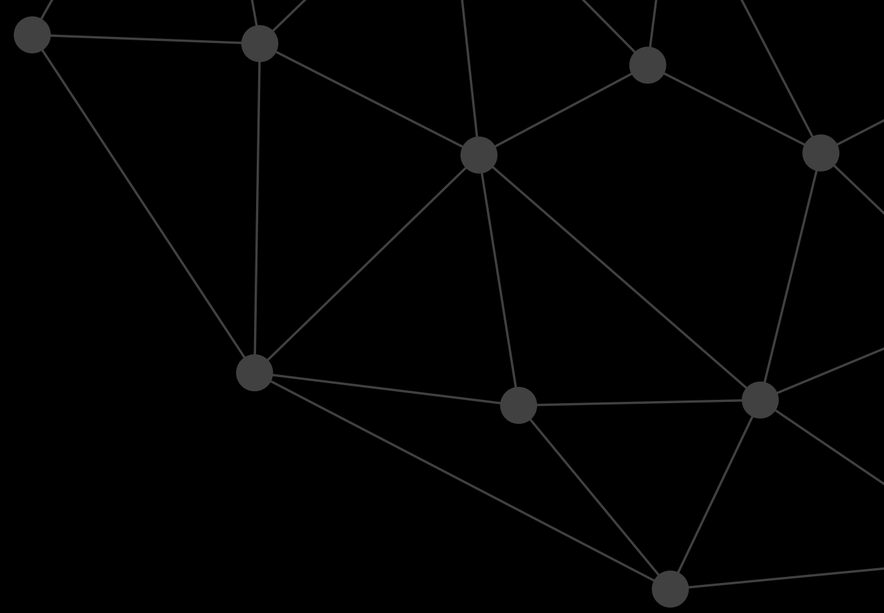


Study: Whiteboard

Use **one file for reading** and **another for writing** (concept: Unix pipes).

When one node writes on whiteboard, it is saved to file, the modification of which is tracked by the filesystem and notified to all other nodes for update.





Implementation Details

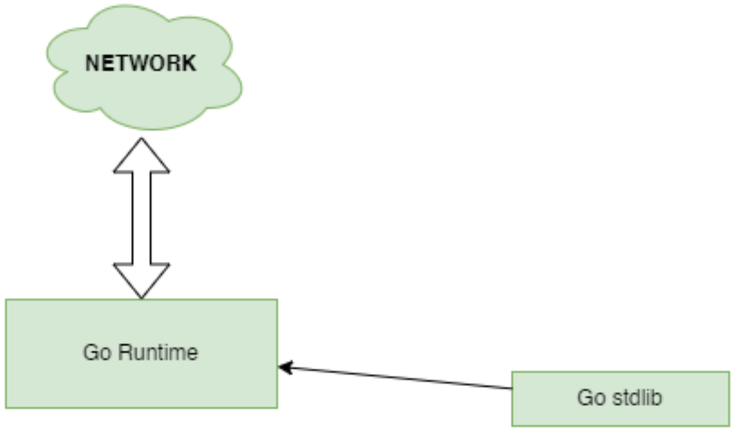
How the components of our framework interact internally with one another

i.e. the framework architecture

Go: Networking

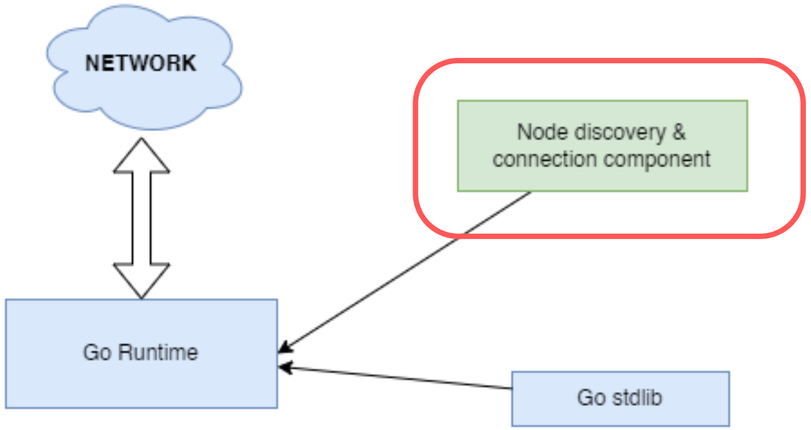
The Go Programming Language has outstanding support for concurrency, network-layer programming and error recovery.

Our base network layer is written using Go and its standard library.



Node Discovery

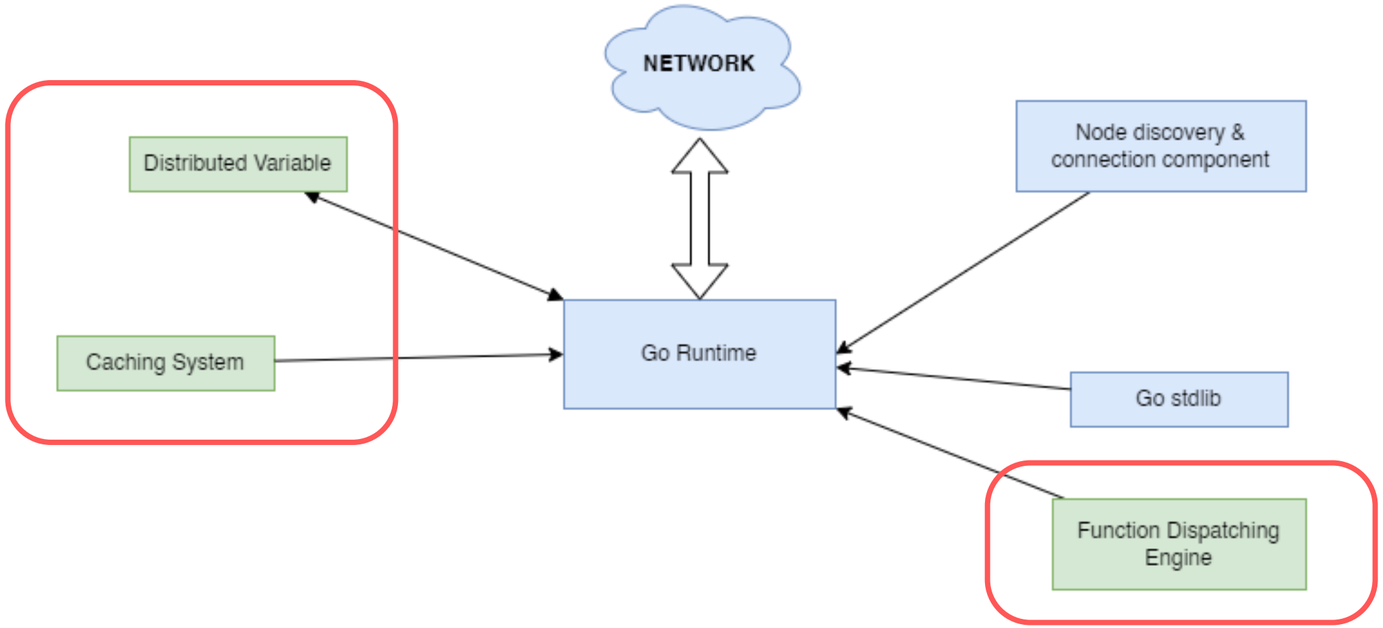
The node discovery and connection logic is written in Go, supported by the standard library of the language.



Distributed Variables, RPC and Caching

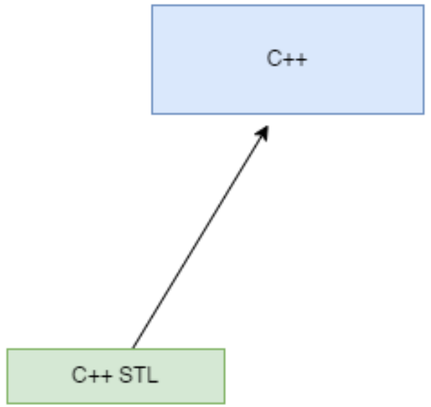
Distributed variables are a key feature to our framework that goes along with remote function dispatch (call).

Caching system in Go caches network connections for reuse.



C++: Base Layer

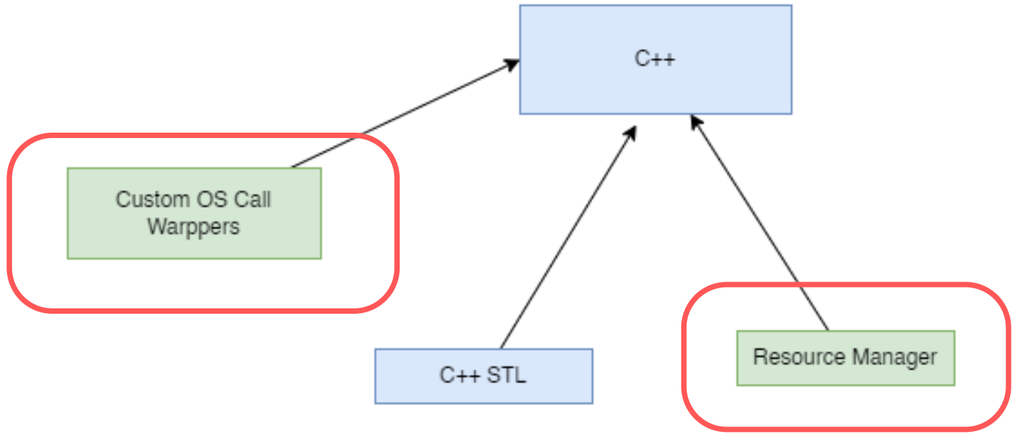
While Go works on the network layer, the C++ executable works close to the underlying file system and operating system (OS).



System Calls and Resource Management

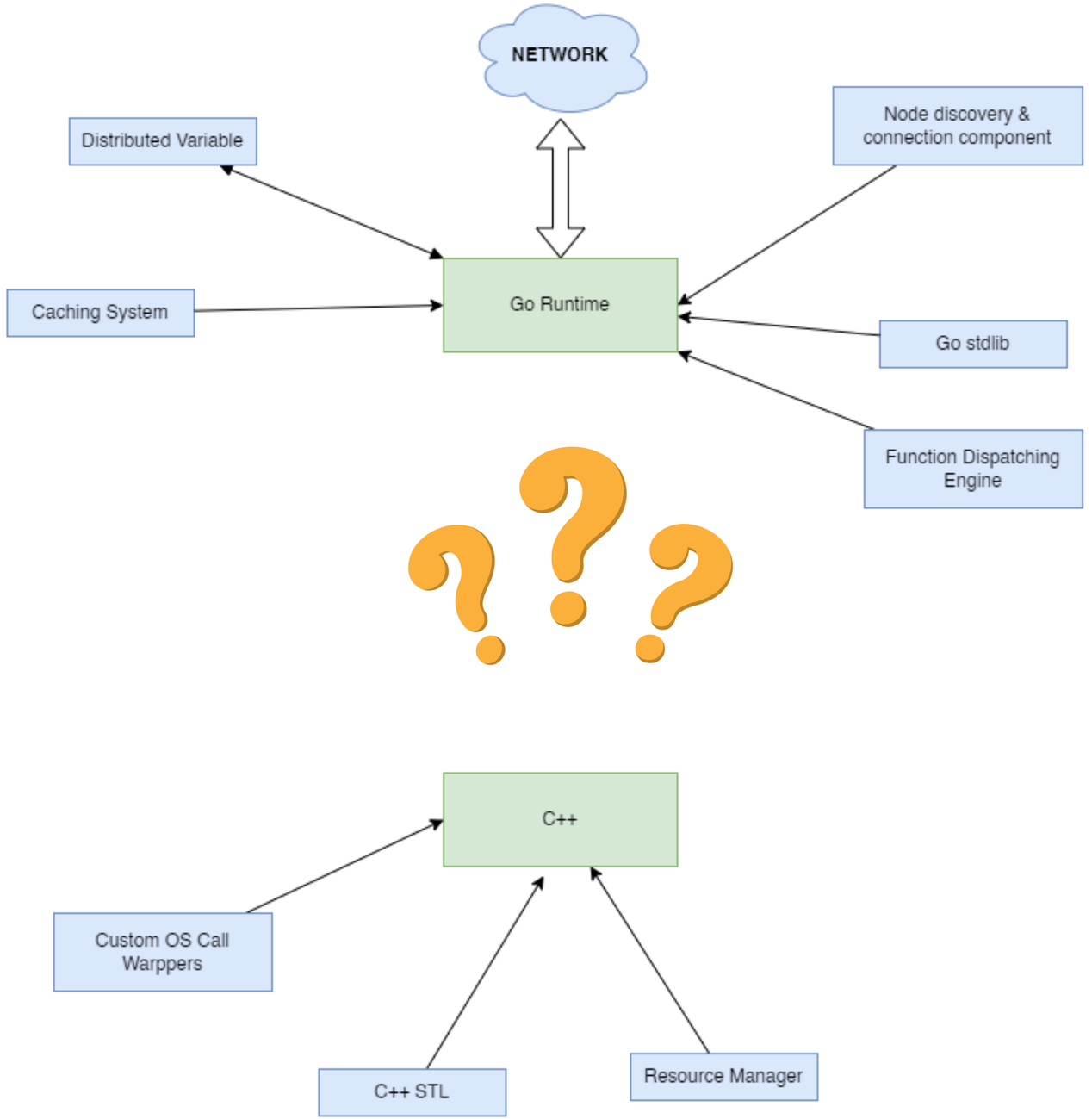
Resource manager is responsible for tracking resource usage across the nodes.

This works alongside system calls that have been provided a cross-platform abstraction.



Connecting the Dots

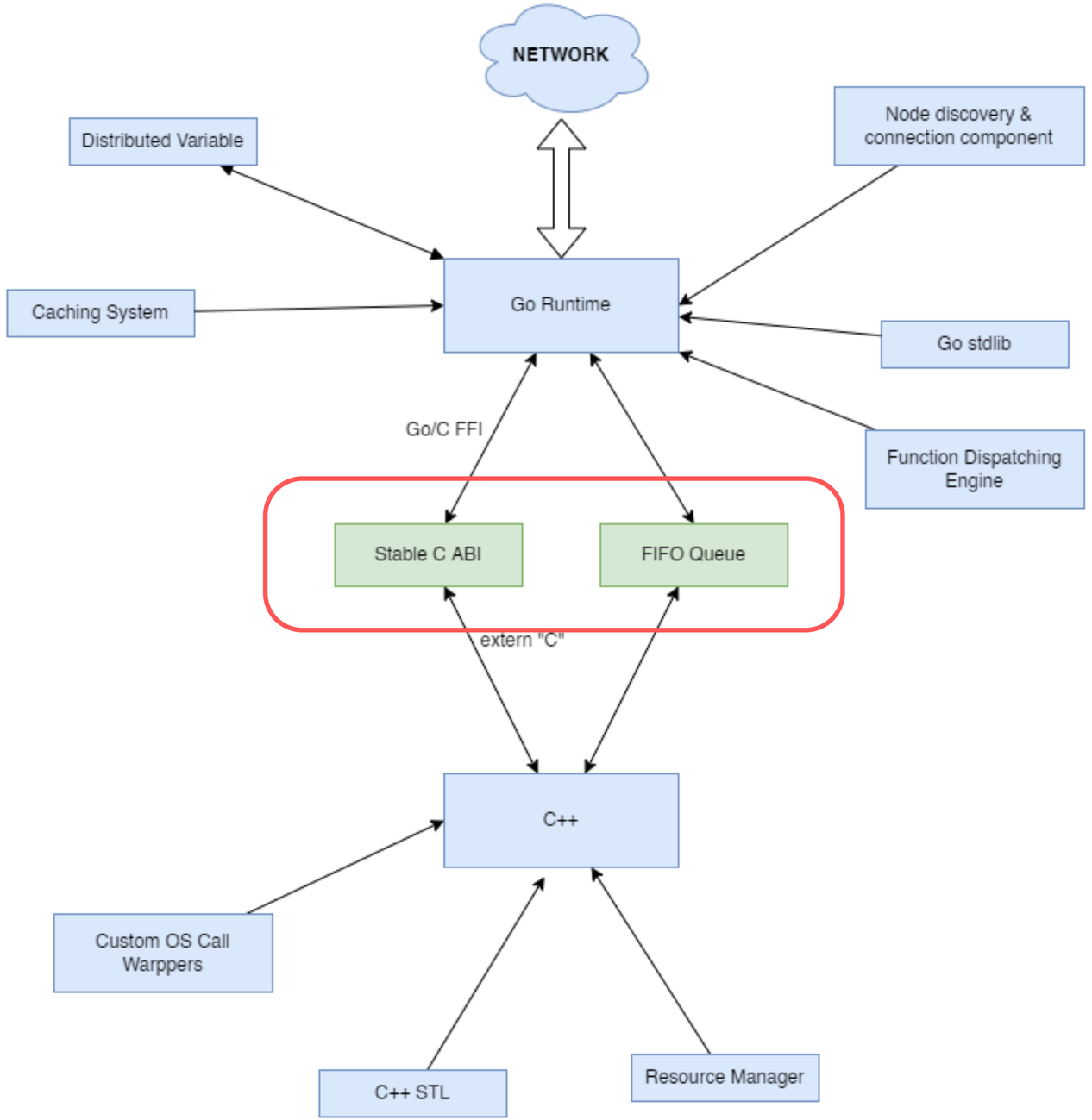
Here we can see our two core components of the framework - the network component and the resource (file) management component.



FFI & Queueing System

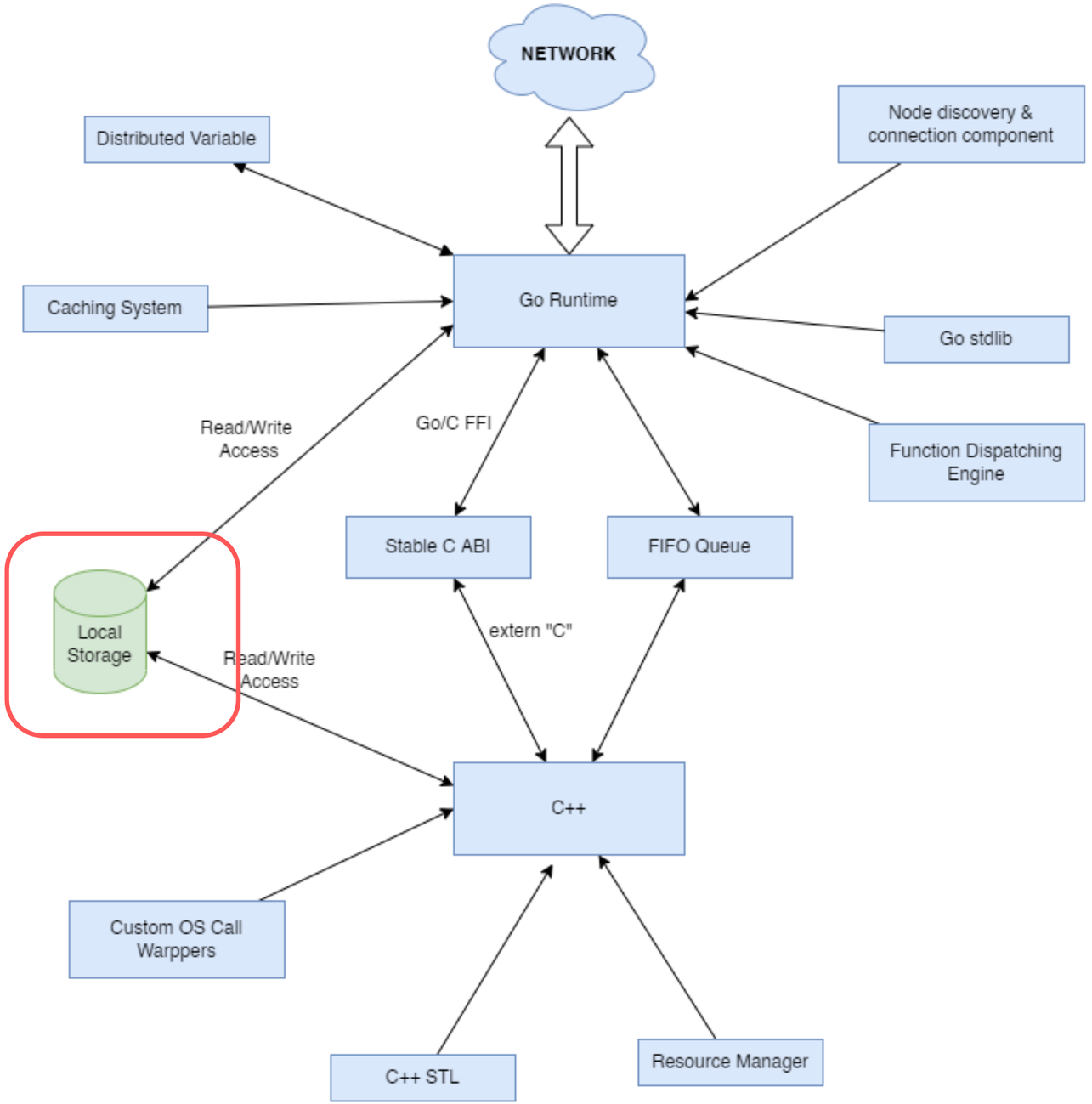
Foreign Function Interface (FFI) allows us to invoke C functions from Go using shared libraries.

FIFO queue is provided by the OS for queueing filesystem requests from the network.



Local Storage (Memory, HDD/SSD)

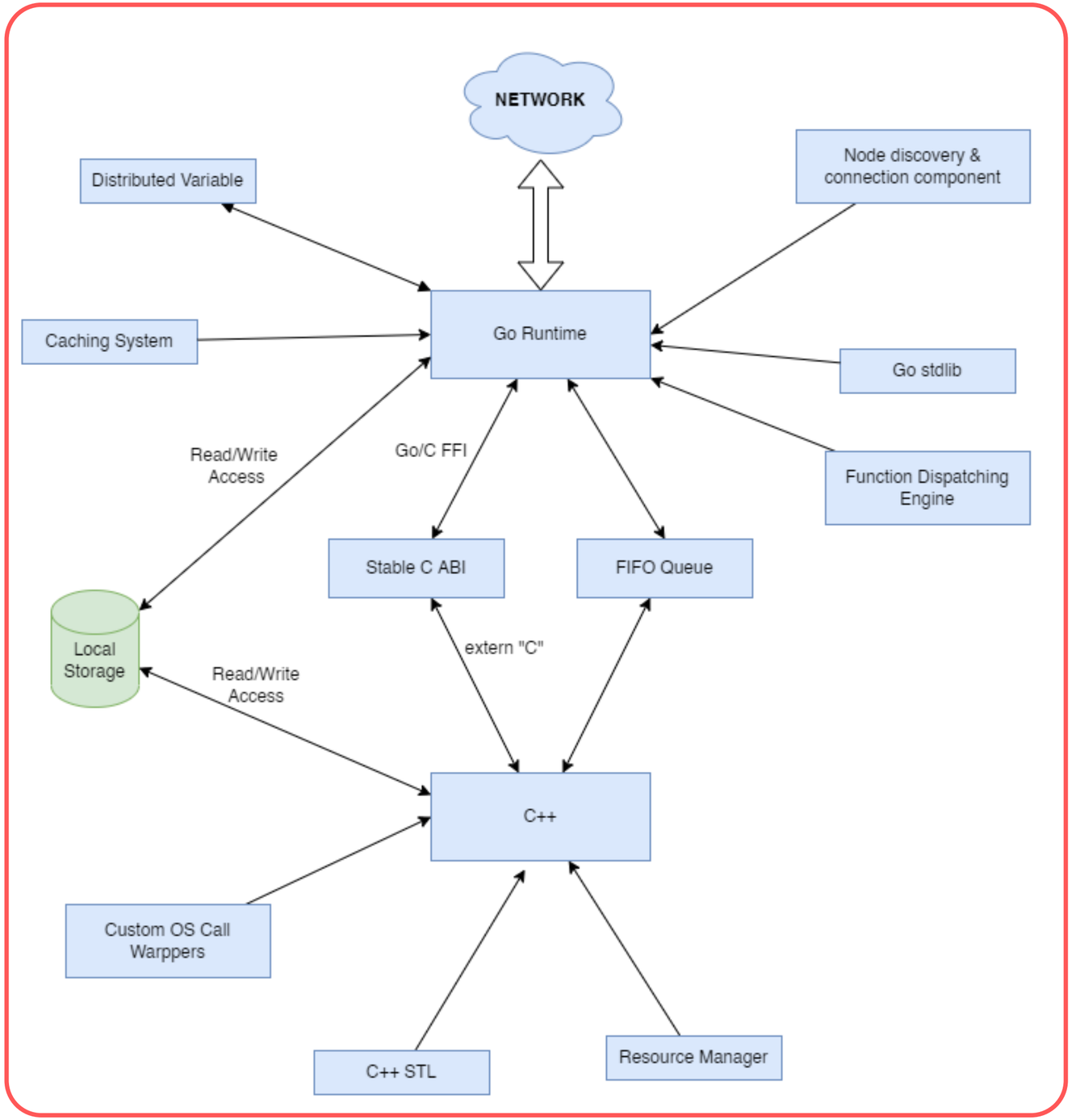
The two components additionally interact with one another through writing in/reading from memory or permanent storage (eg, file syncing).



All of this is a single node!

All of this implementation is the internals of a SINGLE NODE.

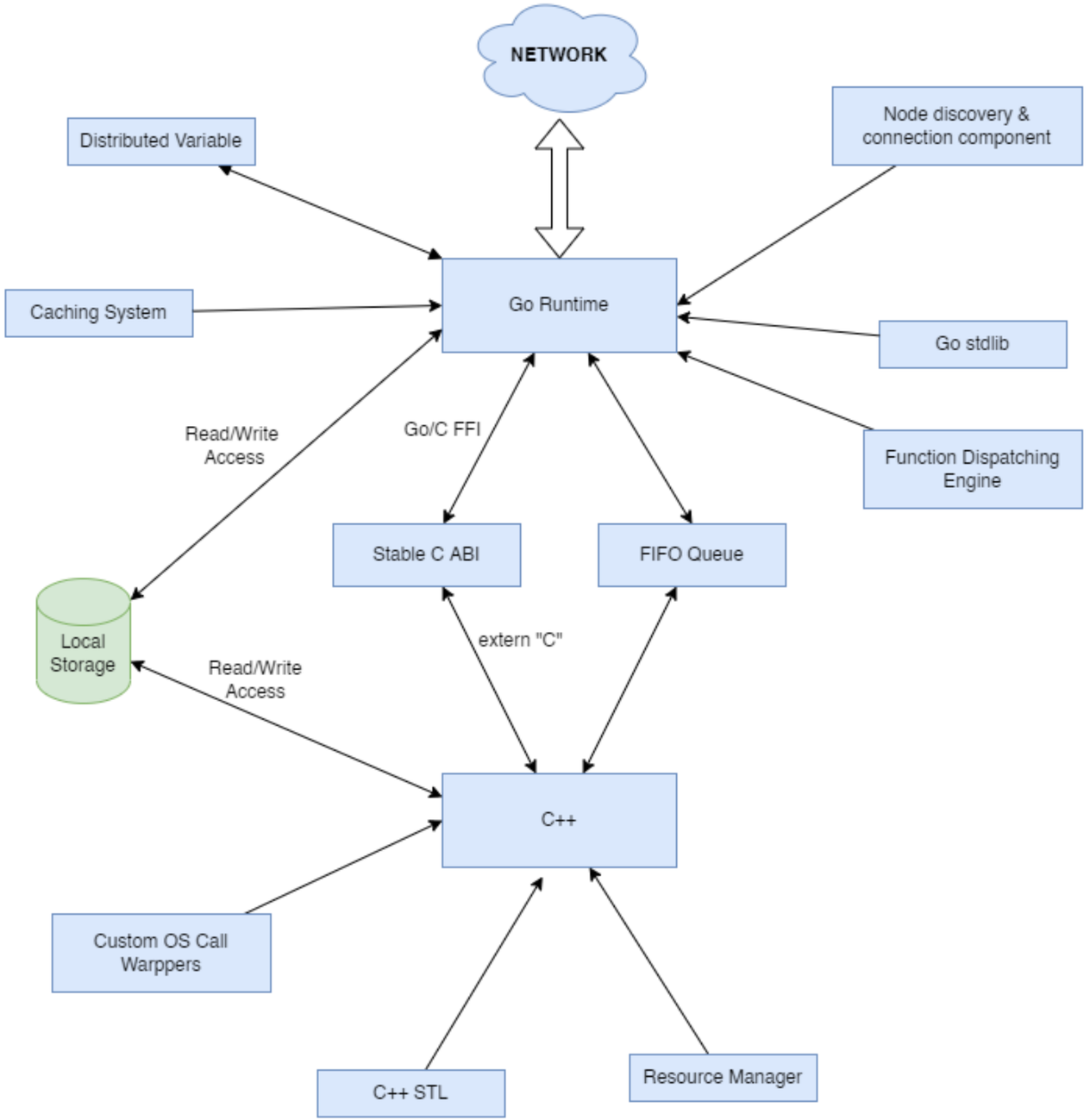
The C++ executable and the Go runtime work together to provide all the features to the programmer that we've discussed so far.

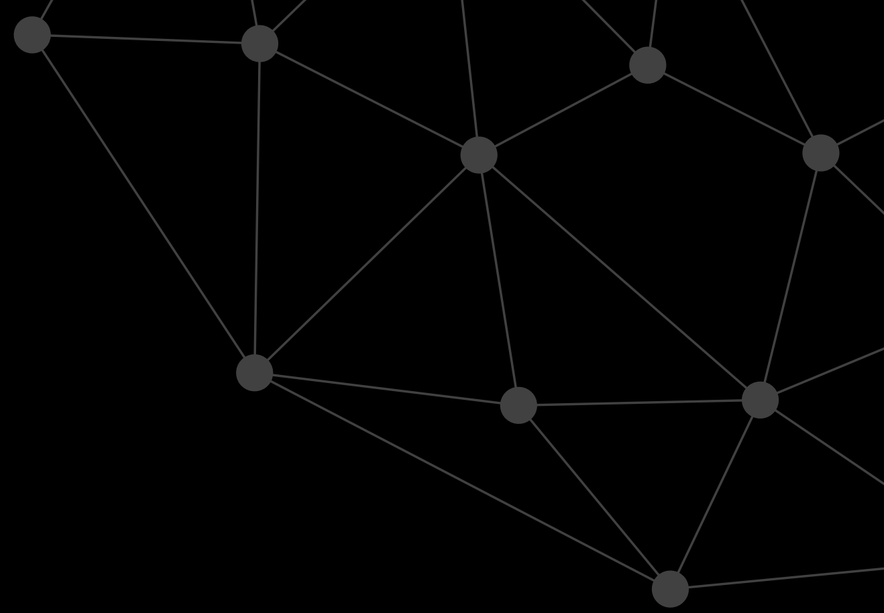


All of this is a single node!

All of this implementation is the internals of a SINGLE NODE.

The C++ executable and the Go runtime work together to provide all the features to the programmer that we've discussed so far.





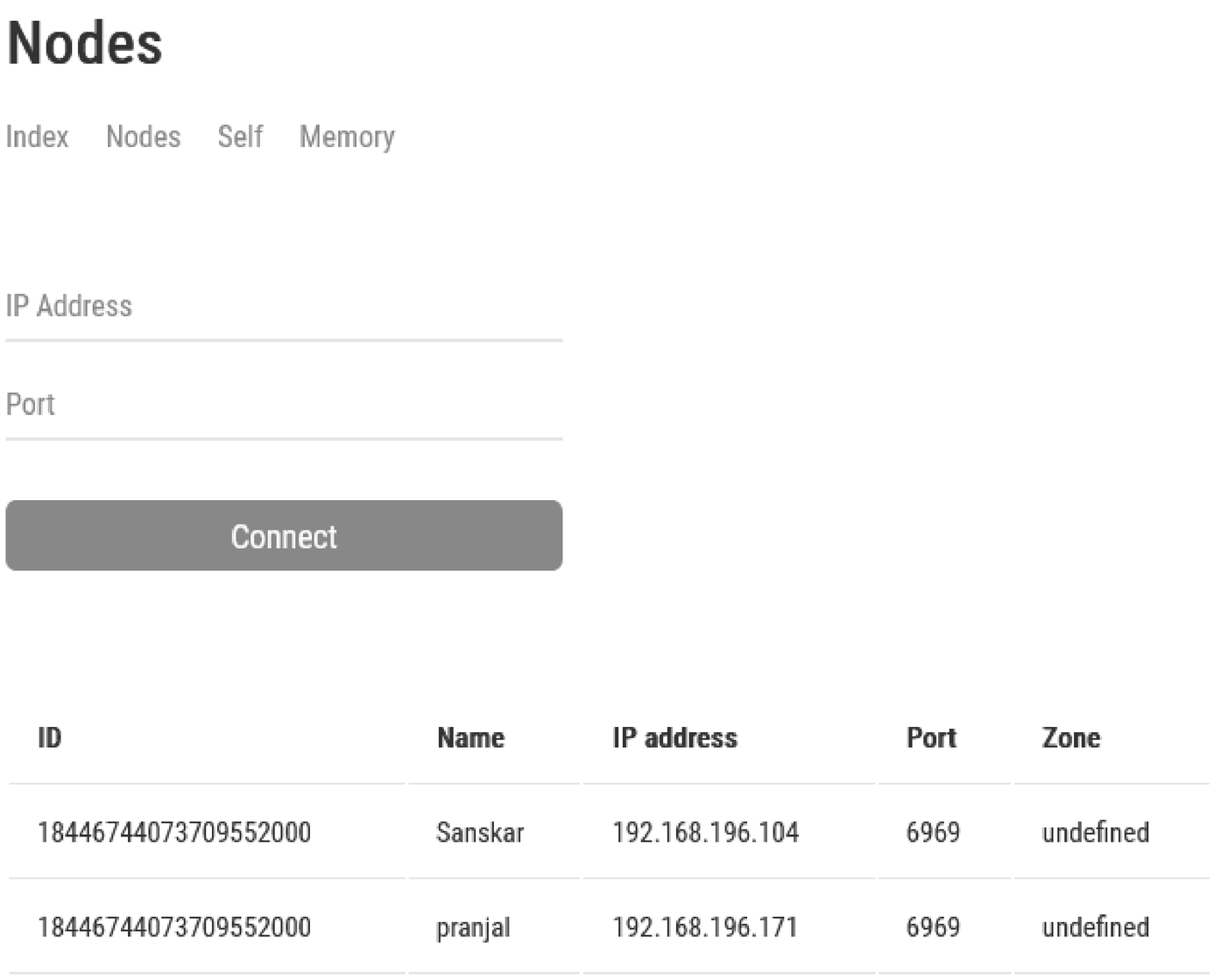
Network Interface & Visualization

UI for the end users

Nodes Tab

A **new node** enters the **IP address** and **port** to connect to the distributed network.

Each node can see **ID**, **name**, **address** and **port** of other nodes connected.



Resources Tab

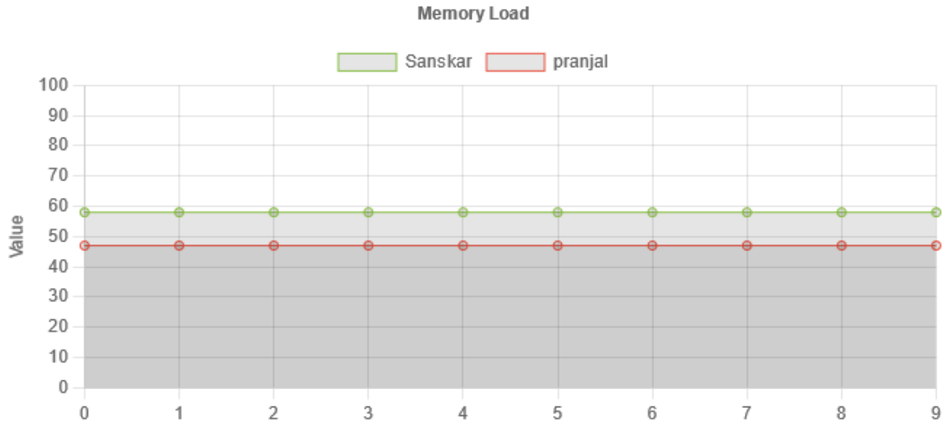
A **Resource Monitoring Graph** provides monitoring of resources of each node connected to the network.

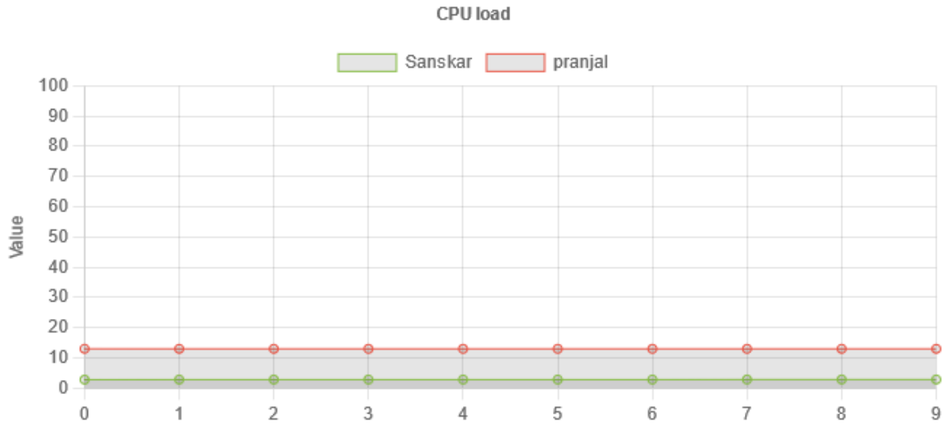
Memory Usuage and **CPU Usuage** can be observed from the Resources Tab.

Memory

Index Nodes Self Memory

ID	Name	Address	Installed	Available	Memory_Load
18446744073709552000	Sanskar	192.168.196.104	15411	6372	58
18446744073709552000	pranjal	192.168.196.171	11865	6213	47





Self Tab

Provides the node's own network information

ID, **name**, **IP address** and **port** information of the current node is provided.

Self

Index Nodes Self Memory

Attribute	Value
ID	18446744073709552000
Name	Sandesh
IP Address	192.168.196.68
Port	6969
Zone	undefined

~~Limitations~~

Possibilities!

- Multiple Mutex Algorithm

- Internal Logs

- Bindings for multiple languages

- Conflict resolution for distributed filesystem

Demo