Rustify - Grupo 11

Implementación de un Nodo Bitcoin

Integrantes:
Gabriel Carniglia
Alejo Fábregas
Camilo Fábregas
Tomás Yavicoli

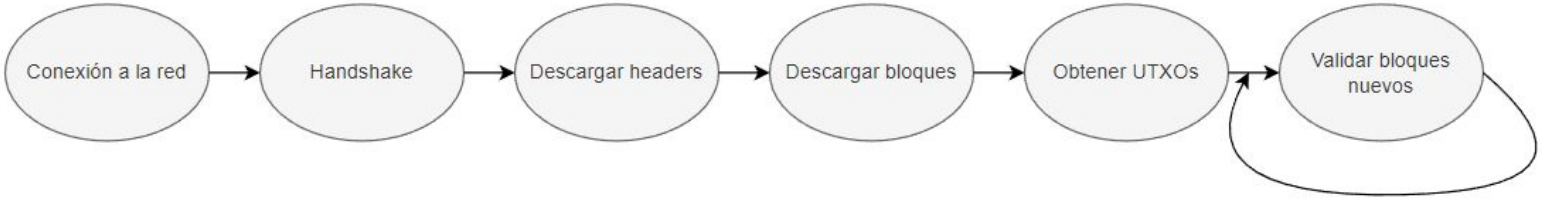
Agenda

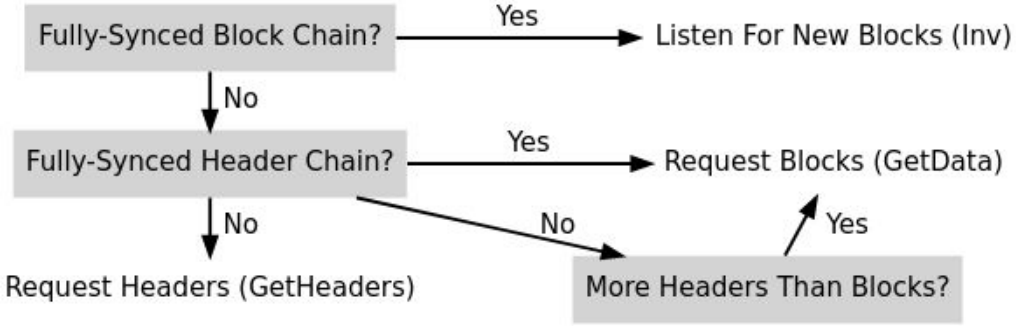
- Introducción
- Diagrama General de Funcionamiento
- Conexión a la Red y Handshake
- Descarga de Headers
- Descarga de Bloques
- Obtención de UTXO
- Validación de nuevos bloques

Introducción

1. **Conexión a la Red y Handshake**
   TcpStream con conexión a la testnet de Bitcoin. Mensajes version y verack.
2. **Descarga de Headers**
   Se guardan en disco y memoria objetos BlockHeader. Mensaje getheaders.
3. **Descarga de Bloques**
   Método headers-first. Descarga con multithreading, se guardan en disco. Mensajes getdata e inv.
4. **Obtención de UTXO**
   Se obtienen las UTXO matcheando los inputs y outputs de las transacciones.
5. **Validación de nuevos bloques**
   El nodo se queda escuchando nuevos bloques. Proof of Work y Proof of Inclusion.

Diagrama de funcionamiento

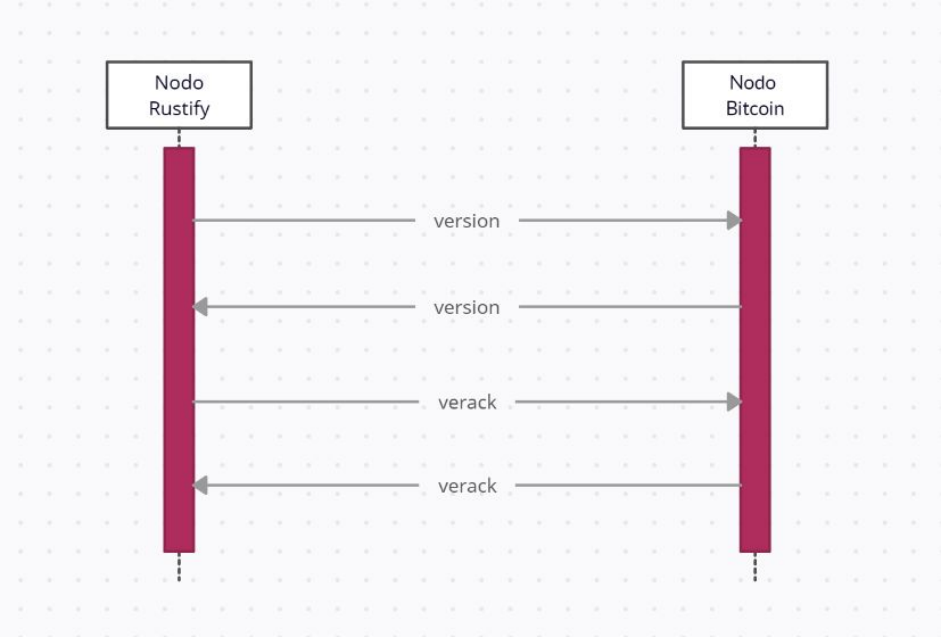




Overview Of Headers-First Initial Blocks Download (IBD)

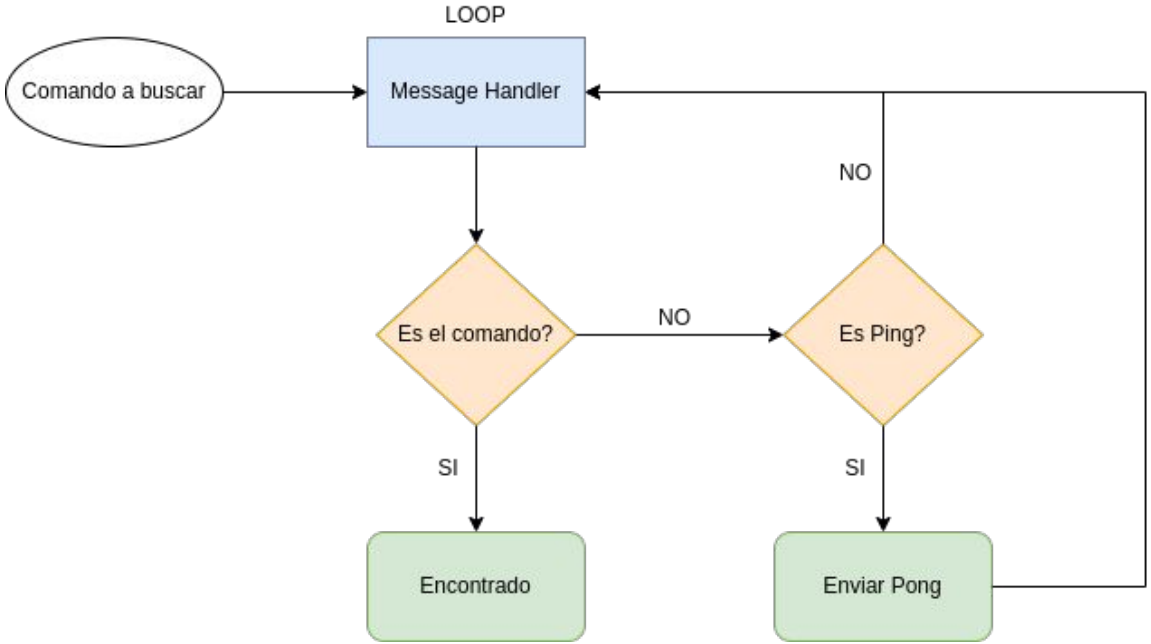
- Modalidad Headers-First
- Bloques -> Multi-threading

Conexión a Red y Handshake



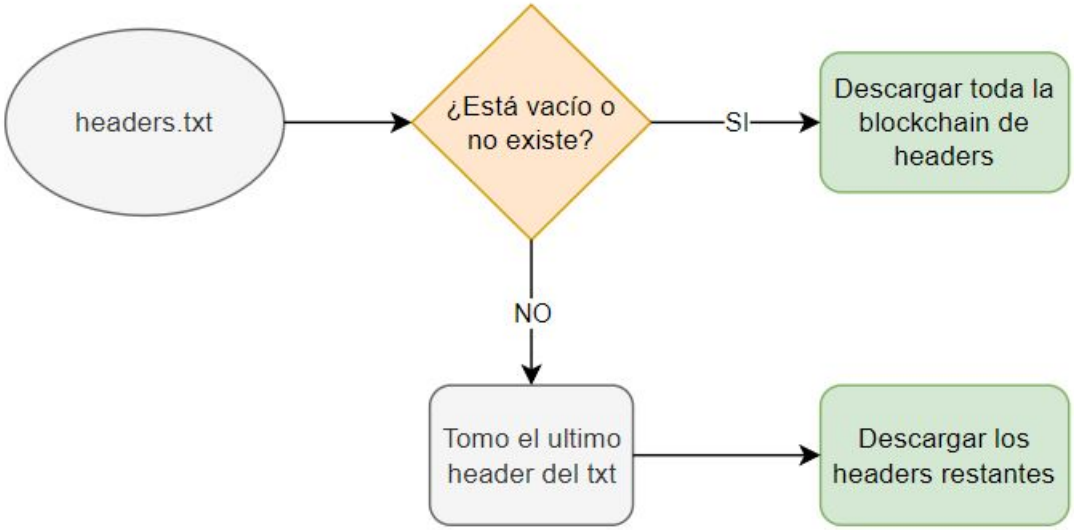
- Conexión por TCPStream a un nodo de la red de Bitcoin
- Handshake con el nodo enviando y recibiendo mensajes version y verack

Handler


LOOP
Comando a buscar
Message Handler
NO
Es el comando?
NO
Es Ping?
SI
SI
Encontrado
Enviar Pong

Descarga de Headers

- Genesis Block hash
- Ciclo getheaders()
- Vec<BlockHeader>

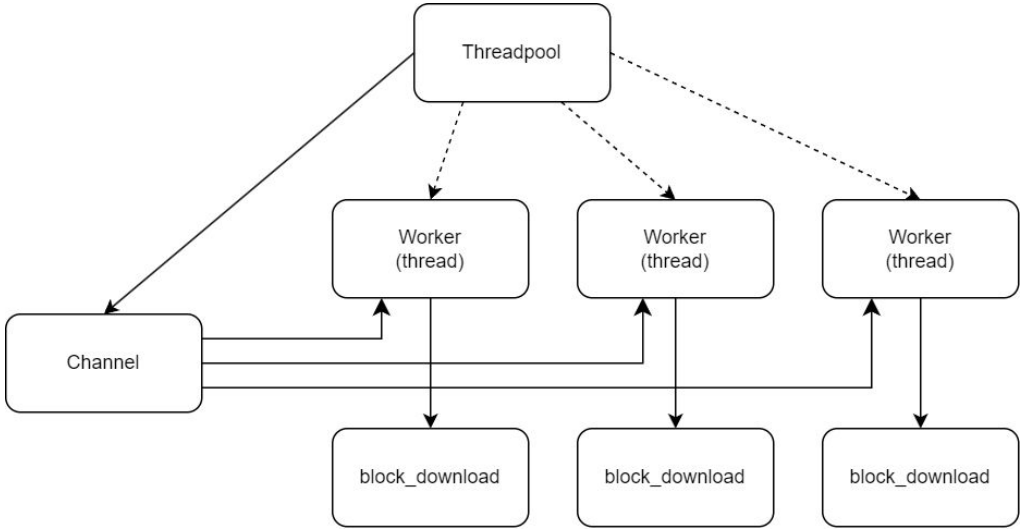


Descarga de Bloques

```rust
pub fn block_download(socket: &mut TcpStream, header: BlockHeader) -> Result<(), RustifyError> {
    getdata(
        socket,
        cant_bloques_en_msj: CANTIDAD_BLOQUES_POR_INV,
        headers: vec![BlockHeader::as_bytes(&header).to_vec()],
    )?;
    receive_block_data(socket)?;
    Ok(())
}
```

- Struct inv
- Guardado de un bloque por archivo (con nombre blockhash)
- Acoplamiento para multihilo

Multithreading



- Threadpool con Workers (threads) y un Channel para comunicarse.
- Cada thread va descargando los bloques de los headers que le llegan por el channel.

UTXO

```
let blocks: Vec<SerializedBlock> = leer_bloque_memoria()?;

let (inputs: HashMap<(String, u32), TxIn>, outputs: HashMap<(String, u32), TxOut>) = obtain_txin_txout(blocks)?;

let (utxos: HashMap<(String, u32), TxOut>, inputs: HashMap<(String, u32), TxIn>) = match_spent_txns(inputs, outputs)?;
```

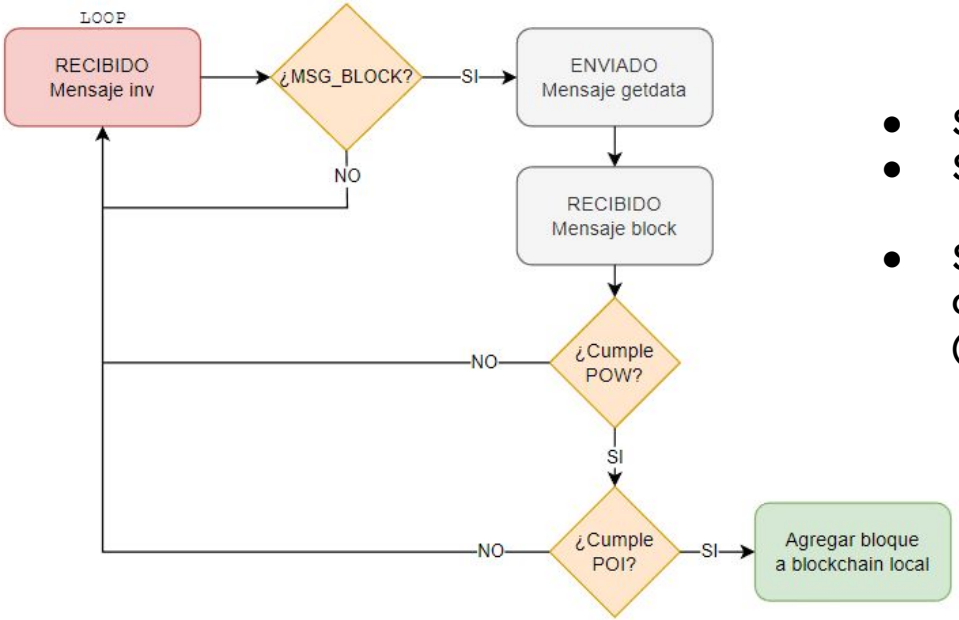
- structs de parseo de transacción:
  (Txn, TxIn, TxOut, LockTime, CompactSize)

UTXO

```rust
for possible_utxo: &(String, u32) in outputs.keys() {
    match inputs.get(possible_utxo) {
        Some(_input: &TxIn) => {
            //Hubo match!
            inputs.remove(possible_utxo);
        }
        None => {
            //Es un UTXO
            if let Some(valor_utxo: &TxOut) = outputs.get(possible_utxo) {
                utxos.insert(k: possible_utxo.clone(), v: valor_utxo.clone());
            }
        }
    }
}
```

- Tuvimos que cambiar de Vec a HashMap para optimizar tiempos (de ~28 días a 4 segundos)

Validación de nuevos bloques



- Se chequea Proof of Work
- Se chequea Proof of Validation

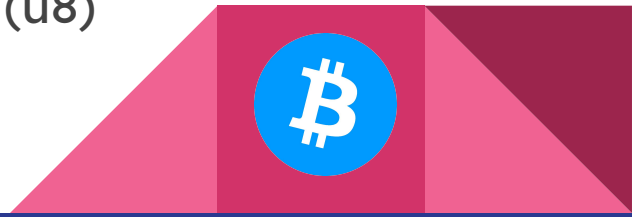
- Si se cumplen, el **bloque** se descarga a disco (/blocks), y el **header** a memoria (Vec<BlockHeader>) y disco (headers.txt).

Proof of Work

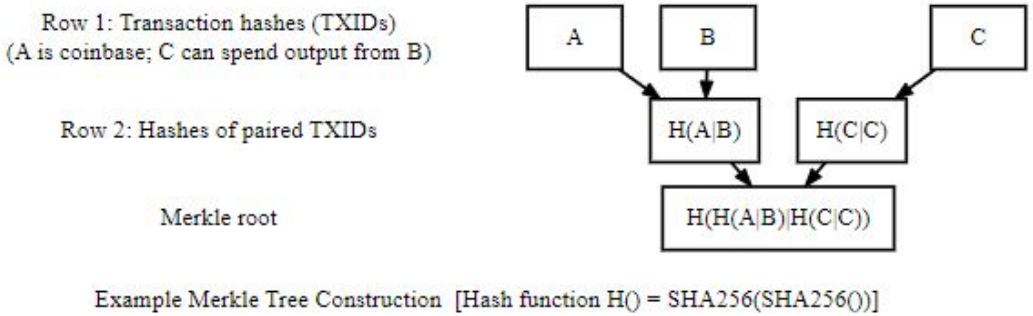
```
nBits:   18013ce9              Exponente: 18                    Mantisa: 013ce9

Target: 0000000000000000013ce9000000000000000000000000000000000000000000
Hash:   00000000000000000007e9e4c586439b0cdbe13b1370bdd9435d76a644d047523
```

- Hash bloque < Target (dificultad)
- nBits: representación compacta de un número de 256 bits
- Target a partir del exponente y mantisa del nBits (notación científica)
- Se comparan byte a byte en arreglos de bytes (u8)



Proof of Inclusion



Example Merkle Tree Construction [Hash function H() = SHA256(SHA256())]

- Generar Merkle Tree a partir de **Vec<Txn>**
- TXID = sha256d(transaccion_bytes)
- Comparar hash de **Merkle Tree root *generado*** vs ***real***

Anexo: Config y Logger

```rust
use std::fs;

#[derive(Debug)]
2 implementations
pub struct Config {
    pub version: i32,
    pub partial_node: u64,
    pub node_network: u64,
    pub user_agent_rustify: String,
    pub headers_path: String,
    pub blocks_path: String,
    pub cant_threads: usize,
    pub cant_blocks_por_inv: u32,
}
```

```rust
pub enum LogLevel {
    Info,
    Error,
}
#[derive(Debug)]
2 implementations
pub struct Logger {
    file: Arc<Mutex<File>>,
}
```

Bibliografía

- Documentación Bitcoin Developer: https://developer.bitcoin.org
- Protocol Documentation Wiki: https://en.bitcoin.it/wiki/Protocol_documentation
- Programming Bitcoin, Jimmy Song, O'Reilly 2019: https://www.oreilly.com/library/view/programming-bitcoin/9781492031482/
- The Rust Programming Language: https://doc.rust-lang.org/book

¿Consultas?