COP 701

Software Design Document
Distributed Ledger

Team 0:

Harish Chandra Thuwal - 2017 MCS 2074

Shadab Zafar - 2017 MCS 2076

Programming Language: Python 3.6
Development Environment: Ubuntu

Table Of Contents

Software Architecture

Handling Concurrency

The problem statement had an inherent requirement for concurrency, since each node needs to be able to perform multiple tasks at once - like handling incoming connections, broadcasting messages etc.

Traditionally, this is handled via Threads, but we decided to use asynchronous tasks & coroutines. This allowed us to only have a single thread per (node) process.

In Python, this is achieved via the `asyncio` module from the stdlib which provides an event loop that runs in a thread and executes all tasks and callbacks in the same thread.

Distributed Hash Table

For the DHT, we implemented the Kademlia Protocol which is also used in BitTorrent's DHT.

Some key features of Kademlia are:

- Kademlia's brilliance is its relative simplicity.
- NodeIDs are binary numbers of length B = 160 bits.
- Data being stored in or retrieved from a Kademlia network must also have a key of length B.
- Kademlia's operations are based upon the use of exclusive OR, XOR, as a metric.
- Parallel lookup/routing mechanism.
- The natural affinity of routing buckets to prefer reliable, long-standing nodes over new ones.

We consulted the following resources during our implementation:

- Kademlia - Wikipedia
- Kademlia Paper by Petar Maymounkov and David Mazieres,
- Kademlia Design Specification

Inter Node Communication

We used a custom UDP based protocol to implement remote procedure calls (RPC.)

Any node can execute an RPC of any other node, provided it knows the socket address of that node.

The elementary structure of an RPC message looks like:

`message_type`	`message_identifier`	`procedure_name`	`procedure_arguments`

`message_type` can be of three types:

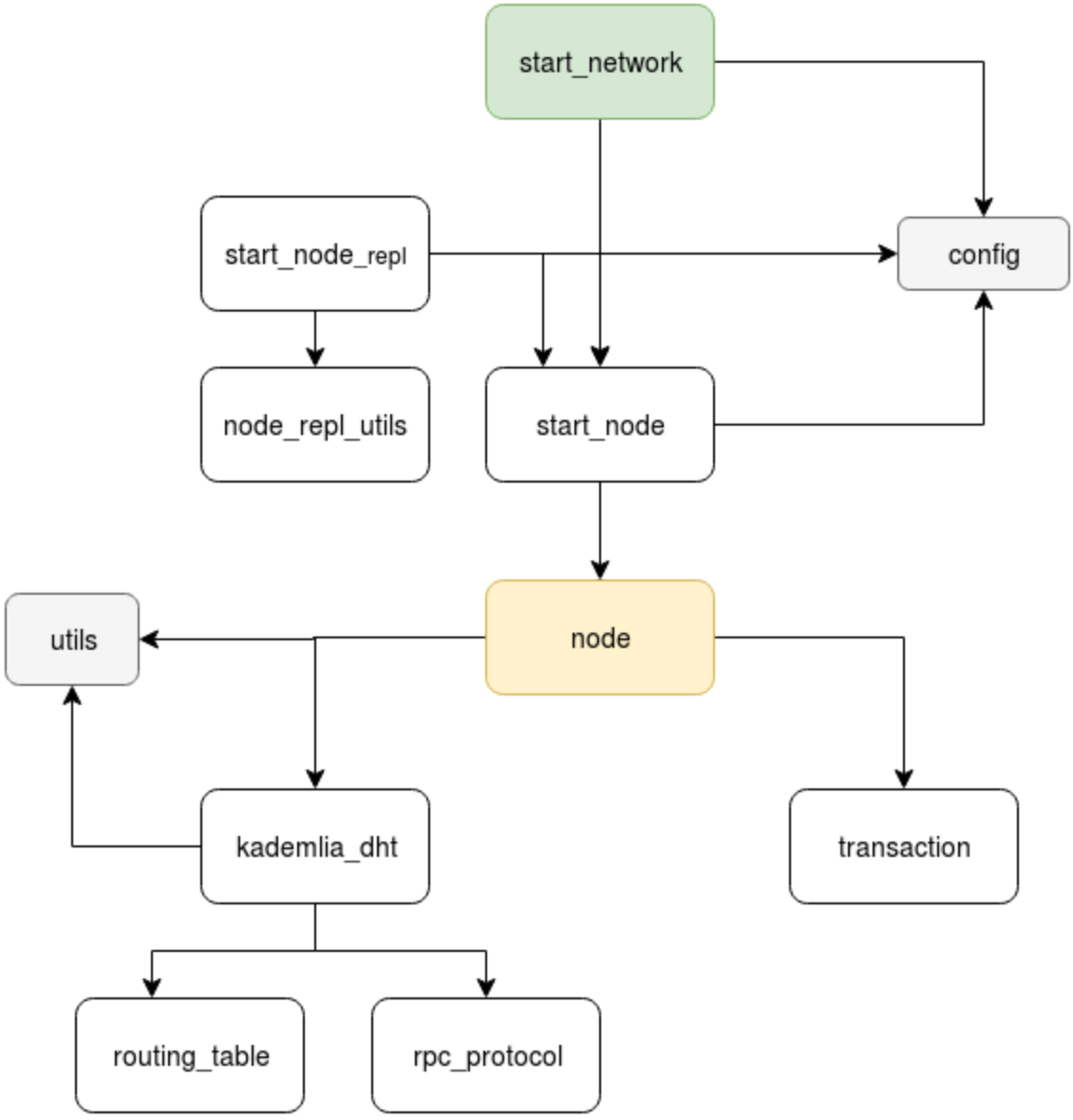
- Request
- Reply
- Broadcast

`message_identifier` is a 160 bit random integer used to associate a request message with a reply

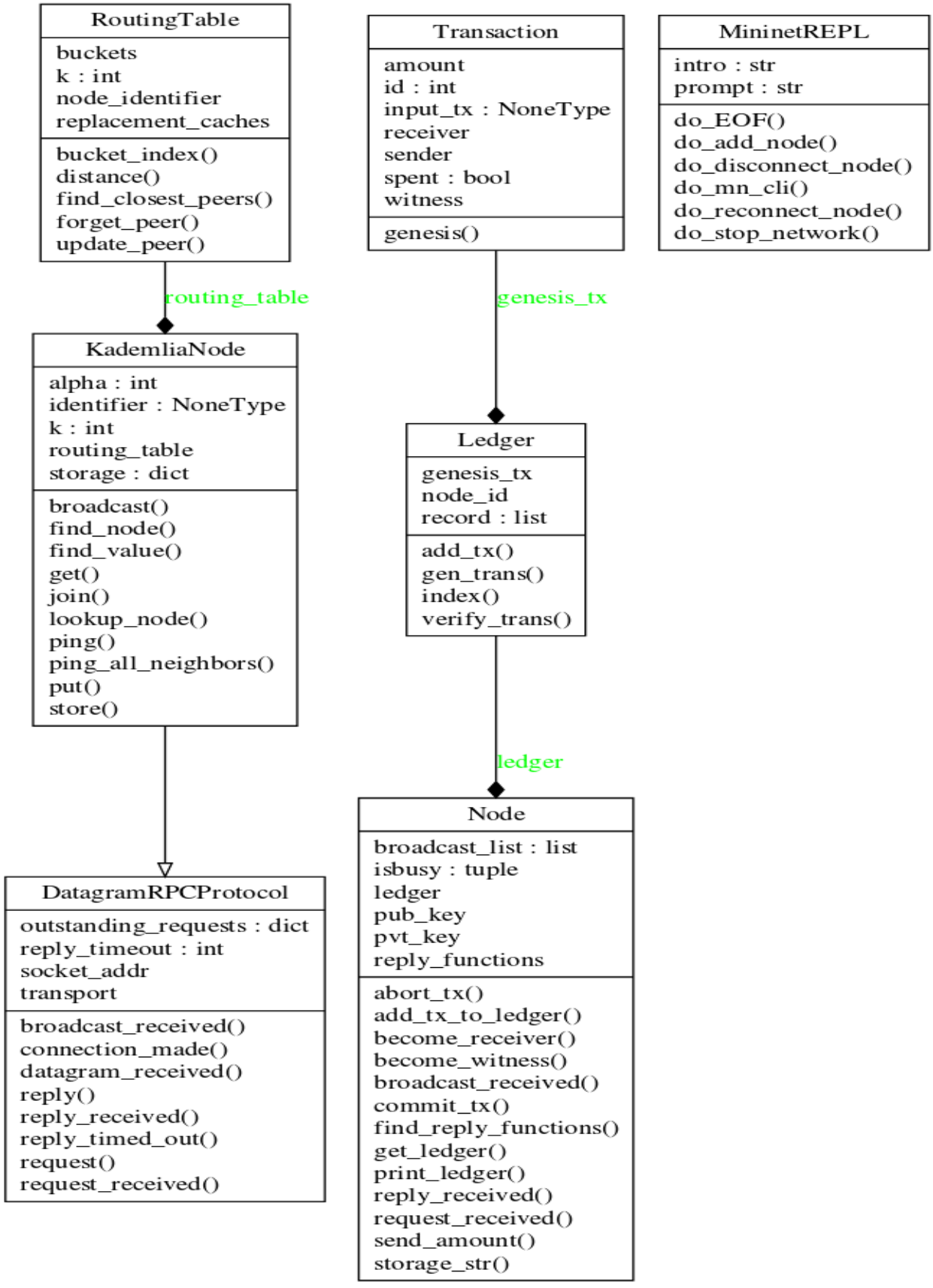
`procedure_name` is the name of the RPC which needs to be executed

`procedure_arguments` are the parameters sent to the RPC

Module Level Design



Class UML Diagram



Modules

start_network

Module to start & modify the mininet network.

- start_network()
    - Starts the mininet network used for the simulation.
    - The number of hosts can be controlled via a command line argument.
- cleanup()
    - Kills all nodes and performs other cleanup tasks.

class MininetREPL

This class provides a REPL-based user interface that can be used to tweak the network.

- do_stop_network()
    - Stops the network, kills all nodes, kills all xterms and performs standard mininet cleanup.
- do_add_node()
    - Add a new host to the existing network
    - Start the node process on that host
- do_disconnect_node()
    - Disconnects node from the network.

- do_reconnect_node()
    - Reconnect a previously disconnected node to the network.
- do_mn_cli()
    - Launches the standard Mininet CLI
    - From which all mininet commands can be executed

start_node

Deals with spawning a new node process on a mininet host.

- start_node
    - Spawns a new node process on a mininet host.
- setup_logging
    - Initialize a logger that logs all messages etc. onto stdout
- log_routing_table
    - Log the routing table at regular intervals
- log_dht
    - Log the distributed hash table at regular intervals
- log_ledger
    - Log the ledger at regular intervals

- two_phase_protocol

  Perform the two phase protocol required to commit a transaction.

    - Generates digital signature of the transaction
    - Handles both the phases of two phase protocol
    - Once the transaction is committed - broadcast it along with its digital signature

start_node_repl

Provides an interface on the first node to interact with all the nodes. Apart from the REPL the first node is similar to other nodes in every aspect.

- node_repl

  This function creates a REPL based user interface that can be used to interact with other nodes and execute RPCs on them

  Various Commands available on this REPL are:

  - id
    - Print id of a node
  - hash_table
    - Print hash table of this node
  - routing table
    - Print routing table of this node
  - put
    - Store a (key, value) pair on the network DHT
  - Get
    - Access a previously stored value by its key
  - send_bitcoins
    - Send bitcoins to a node
  - broadcast
    - Broadcast an RPC over the network

- ledger
    - Print the ledger of this node
- Help
    - List all available commands

- start_node_with_repl
    - Spawns a new node process with the above REPL capabilities on a host in the mininet.
    - For our simulation, we only have one REPL node, which is also used as a Bootstrapping node - the node all hosts initially connect to.

rpc_protocol

This module implements a UDP based protocol to deal with the communication between nodes and handling RPCs.

- rpc
    - A function decorator that converts every function to a rpc.
- connection_made
    - A callback indicating that the system has established a connection.
- datagram_received
    - The function to be executed upon receipt of a datagram packet.
- broadcast_received
    - Handles servicing of a broadcast request
- request_received
    - Handles servicing of a request (unicast)
- reply_received
    - Handles servicing of the reply to a request sent earlier.
- reply_timed_out
    - Schedules a timeout for each outbound request to enforce the wait timeout.
- request
    - Issues an RPC to a remote peer
- reply
    - Sends a reply to an earlier RPC call

kademlia_dht

This module provides procedures to deal with network initialization and the maintenance of the distributed hash table.

Before handling a remote procedure call, each node first updates its routing table - increasing the priority of the callee's node.

Remote Procedures

- ping
    - Test whether a node is alive
    - An alive node replies with its ID
- store
    - Store a (key, value) pair in local storage.
- find_node
    - Return the k-closest peers to a key that this node is aware of.
- find_value
    - Find the value corresponding to a key in storage of this node.
    - Gives the value of the key if the key is found else gives the k-closest peers to it.

Other Procedures

- put
    - Given a key and a value, store it on the dht and return the number of nodes who successfully accepted the value.

- get
    - Given a key, find the value corresponding to it from the distributed hash table.
- lookup_node
    - The iterative node lookup procedure to find either the nearest peers to or the value of a key
- ping_all_neighbors
    - Procedure to ping all the neighbors of this node.
    - Neighbors refer to the nodes whose information is present in the routing table of this node
- join

  Procedure that is called by a new node joining the network. It performs the following tasks :

    - Pings the known node / bootstrapper to update its routing table
    - Find all nodes close to itself and ping them
    - Send its genesis transaction to the known node
    - Sets its ledger equal to the ledger of known node
    - Broadcasts it genesis transaction to the network
- broadcast
    - Method that allows the current node to broadcast an RPC over the network.

routing_table

This module provides the variables and remote procedures for maintaining the routing table used by kademlia_dht.

Data Structures:

- buckets
    - The peers known to a kademlia node are organized in buckets which can hold a maximum k peers. These are known as k-buckets.
    - Peers in each bucket are ordered on the basis of last interaction.

Procedures:

- update_peer
    - Add or Update peer in the routing table.
- forget_peer
    - Remove peer from the routing table.
- find_closest_peers
    - Find the k closest peers to this node.
    - Where k is defined by the Kademlia Protocol.

transaction

This module provides classes for representing transactions & ledgers and performing operations on them.

class Transaction

Class transaction provides the data structure to represent a transaction.

Each transaction contains the following components

- id
    - A unique id used to identify a transaction
- input_tx
    - List of input transactions used
- sender
    - The id of the sender of the transaction
- receiver
    - The id of the intended receiver of the transaction
- witness
    - The id of the witness of the transaction
- amount
    - The amount of money being sent

class Ledger

This class implements the representation of the ledger of transactions to be maintained by each node

Ledger class basically consist of a list of transactions called record.

Procedures

- gen_trans
    - Procedure that if possible generates a valid transaction based on the amount to be sent.

- verify_trans
    - Procedure that on given a transaction verifies it validity as per the node's ledger.

node

This module provides remote procedures that are involved in handling transactions.

- send_amount
    - Used to initiate a transaction - transfer money from one node to another.
- become_receiver
    - Used by sender during the first phase i.e commit request phase of the two phase protocol to request the receiver node
    - Procedure checks and returns the status of the intended receiver i.e, whether it is busy in some other transaction or not.
    - If it is busy then the sender aborts the transaction.
- become_witness
    - Used by sender during the first phase i.e commit request phase of the two phase protocol to request the witness
    - Procedure checks and returns the status of the intended witness i.e, whether it is busy in some other transaction or not.
    - If it is busy then the sender aborts the transaction.
- add_tx_to_ledger
    - A remote procedure that when executed adds the passed transaction to the ledger.

- Procedure does not performs any kind of verification so verification of the transaction must be done before calling it.

- commit_tx

  - A remote procedure that when executed tries to verify a transaction and then add it to the ledger.

  - It performs the following tasks:
    - Verify Digital Signature of the transaction
    - Verify validity of the transaction based on the ledger.
    - Send **abort** to sender if any of the above verification fails.
    - On successful verification add the transaction to its ledger and set the spent status of the input transaction to true.

- abort_tx

  - A remote procedure that is used to abort a transaction.
  - If the transaction to be aborted was added in the ledger then removes it.

utils

- gen_pub_pvt
  - Generate a public, private key pair
  - Using Elliptic Curves

- sign_msg
  - Generate a digital signature for a message using
  - This uses the private key of the signer

- verify_msg
  - Verify that a message matches with a digital signature
  - This uses the public key of the signer

External Modules

- `asyncio`
    - An asyncio eventloop (per node) is used to perform tasks concurrently without needing to run multiple threads per node.
    - This module is a recent addition to Python and is the reason we want to stick to Python 3.6
- `aioconsole`
    - This handles asynchronous reading of input from stdin.
    - It was used to provide the node base REPL interface.
- `shlex`
    - Used to parse the REPL input entered by the user and break it into to tokens based on white spaces
    - This also takes care of nested quotes etc. in the input.
- `sockets`
    - Nodes talk to each other via sockets - which are UDP because no setup/teardown packets are required for functioning.
- `hashlib`
    - SHA 1 hashes are be used for ID of a node and as the key space for the Kademlia DHT
- `ecdsa`
    - Generation of public, private key pair & signing/verification of digital signatures.

- `logging`
    - Each node logs messages being received and handled to their stdout.
    - But it can also be written to a separate file named `<node_id>.log` in a common `logs` directory.

Testing

Environment

We tested the code on Ubuntu 16.04 machines running mininet version 2.2.1

The entire design is centered around asynchronous tasks using the asyncio module, so the code requires Python 3.6.

To install the required Python 3 packages, run:

```
pip3 install --user aioconsole==0.1.3 ecdsa==0.13
```

mininet is not available with Python 3, so part of the code also requires Python 2.

To install the required Python 2 packages, run:

```
pip2 install --user mininet==2.2.1
```

Running the code

Apart from this, ensure that mininet is properly setup.

In a terminal window, run:

```
sudo python2 start_network.py 10
```

This will spawn 10 nodes (each with a separate xterm.)

The start_network terminal provides a REPL that can be used to modify the network - add, remove nodes etc.

The first spawned xterm has another REPL that allows us to execute RPCs on a host. This can be used to demo money transfer etc.

Both REPL interfaces have a `help` command that lists all available commands and their usage.