Communications Node

Requirements and High Level Design (HLD) Document

This is the high-level design and requirements specification document for the Communications Node problem. The goal of this document is to provide a list of requirements and to propose an architecture solution that can meet said requirements.

1.1 Introduction

The proposed problem is to write a simple base "Communications Node" (CN) for a distributed communications system running across multiple machines on a mixed (fast, slow, wired, WiFi, etc.) "local" network. In its most basic form, each CN should have a unique ID and should come up and continuously enumerate all the other nodes it can see on its network. No IP addresses, port numbers, or node IDs will need to be set manually.

The solution will be written with C++11 on Ubuntu Linux. It will be POSIX compliant.

1.2 Requirements

- Each CN must have a unique ID
- Update every 10-20 seconds
- List all other CNs on the reachable LAN
- Show round trip comms latency to every other node
- Show an estimate of to-from bandwidth with each other CN
- Show an arbitrary number of CNs
- Allow multiple CNs to exist on a single machine

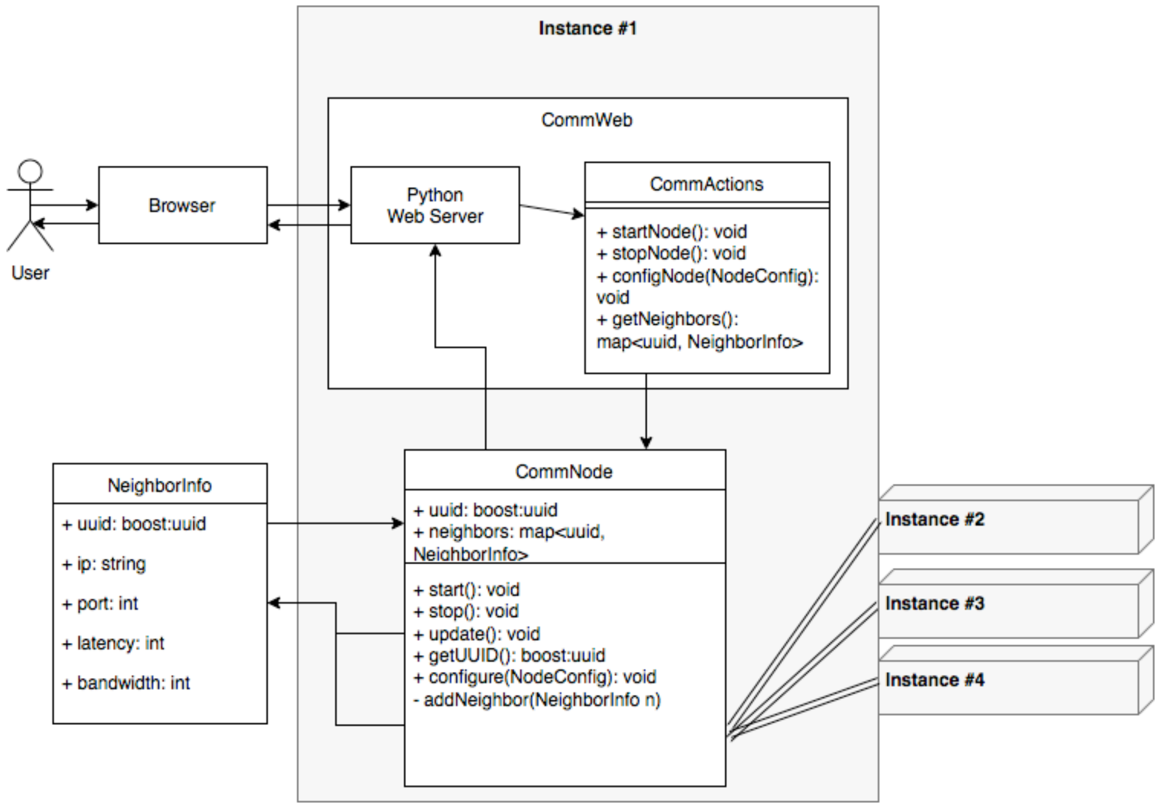
Definition of Objects:

CommNode- This is the primary C++ class for handling sockets, heartbeats, and metrics. It polls its neighbors and stores information about them in a Set.

NeighborInfo- This is a C++ struct that stores some information about a CommNode object and is built to work with STD library containers.

CommNodeLog- A thread-safe logger used by the entire application.

Architecture Diagram
Eventually, a thin-client will be added to allow the user to view and manage network status through a browser. There will not be time to execute this in the initial release, but the architecture diagram will still reflect it.



1.3.2 Requirements Table:

Requirements	**Description**	**Comments**
Each CN must have a unique ID	Each instance of the CommNode object will be given a UUID at runtime.	We will use the Boost.UUID library for this.
Update every 10-20 seconds.	Each CommNode instance will send a heartbeat and poll its neighbors every 10 seconds by default. This interval will be configurable	We'll use Boost.PropertiesTree to parse an .ini file for configuration.
List all other CNs on the reachable LAN	On every update, the CN will listen for heartbeat messages being broadcast on the LAN. For each such heartbeat it receives, it will create a TCP connection to the originator of the heartbeat.	These heartbeat packets will contain both the uuid and the tcp port that the remote Node is listening on.
Show round trip comms latency to every other node	As part of the update, the CN will poll all of its neighbors to determine round-trip latency which will be cached	This latency will be measured std::chrono and ping-pong messages.

Show an estimate of to-from bandwidth with each other CN	As part of the update, the CN will poll all of its neighbors to determine the maximum bandwidth between the two Nodes.	All messages in the prototype implementation are 128 bytes, so that figure will be used to calculate bandwidth.
Show an arbitrary number of CNs	CNs need to be able to handle an arbitrarily large number of peers.	This will be handled by storing information about each CN's neighbor in a map to ensure uniqueness and fast lookup times. These information objects are a fixed size, so space won't be a concern.
Allow multiple CNs to exist on a single machine	Machines need to be able to run multiple CNs with no clashing.	Each instance will have a UUID, so there shouldn't be a clash there. Port binding and the web server will be handled with a master/slave relationship (see 1.3 Desired Behavior)

1.3 Desired Behavior

The goal is for the node and the web server to run as services/daemons. The web server does nothing until a request is made. The node makes connections and runs behaviors in the background. When a browser navigates to the server address, the server sends a page showing the current status of any CNs running on the machine. The user is able to control CNs locally or remotely using a simple interface.

This setup allows for the most flexibility and the smallest footprint possible within the time constraints of the project.

1.3.1 Startup

On startup, the CN will generate a UUID. This virtually ensures that each node will have a unique ID without needing to setup something like a DHCP server to hand out IDs or to maintain a shared memory registry. The odds of clashing are so low as to be negligible.

Additionally, as part of startup, the node will set up a UDP socket on a designated port to listen for heartbeats and a TCP socket on a random port to handle two-way communication between individual nodes.

1.3.2 Heartbeats

Nodes will advertise their presence (as well as some basic information) through heartbeat messages. The format for these messages are:

add uuid port

The first word in the string is the command to be executed. The UUID is the node id and the port is where the node is listening for TCP connections.

1.3.3 UDP broadcast

After starting up, the CN will start sending a heartbeat message at a set interval to its NIC's broadcast IP on a predetermined port number (hardcoded at 8000 for the prototype). All CNs will be listening at their localhost on that predetermined port number for heartbeats from other nodes.

1.3.4 TCP connections

When a heartbeat is received, the CN first tests whether that neighbor already exists in the map of neighbor nodes. If not, then it adds it to the map and establishes a TCP connection. The TCP server accepts new connections and spawns a new thread to handle further IO.

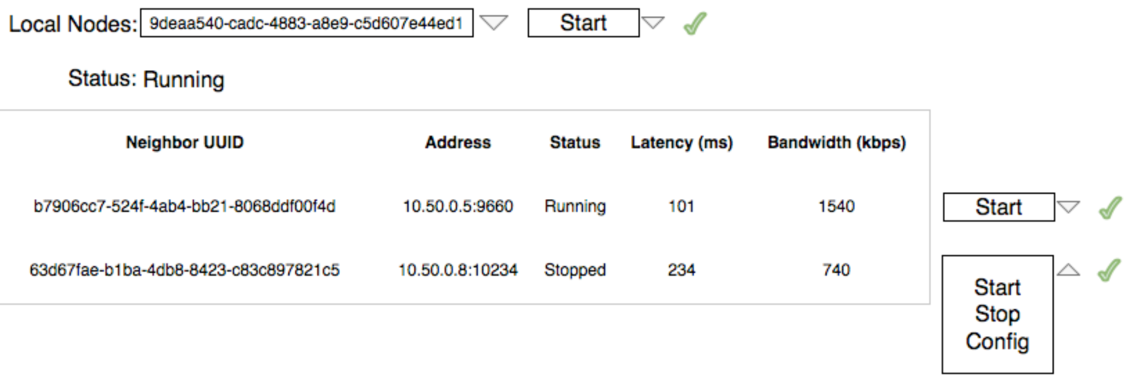
1.3.5 Multiple CNs on same machine

Any heartbeat that comes from the same IP as the CN's local IP but with a different UUID is treated as a local neighbor. These are treated the same as remote neighbors except they are also added to a second map.

When multiple CNs are started on the same machine, the first one to bind a socket to the UDP port will act as the master. Every other CN acts as the slave. When the master receives a UDP packet, it forwards that information to every node in the local neighbor list. This behavior will also extend to the python web server. The first CN to set up their python server will handle all HTTP requests and forward any required actions to the appropriate CN.

1.3.6 Web server (future releases)

The web server will be written in python and use a C++ module to send commands to the CNs on the network. To start, when a browser navigates to the appropriate web address the server will send a simple HTML page with the following layout:



Status: Running

Neighbor UUID	Address	Status	Latency (ms)	Bandwidth (kbps)
b7906cc7-524f-4ab4-bb21-8068ddf00f4d	10.50.0.5:9660	Running	101	1540
63d67fae-b1ba-4db8-8423-c83c897821c5	10.50.0.8:10234	Stopped	234	740

Information about node neighbors is displayed in the table with each neighbor on its own row. The Local Nodes dropdown contains a list of UUIDs of nodes running on the local machine. The dropdowns on the right hand side of the table send actions to that node. There are currently three actions planned (start, stop, config).

1.4 Future Improvements

- *Web Server*- The Python web server will be step one.
- *Node nicknames* – UUIDs can be cumbersome for users, so adding a nickname field and making it configurable through the web server makes sense.
- *Improved socket architecture*- Bandwidth is currently depressingly low due to the nature of how these sockets are implemented. Something more robust will need to be made.
- *C++ sockets* – C-style sockets are fine, but using something like Boost would make deploying cross-platform much easier.
- *Unit tests* – Originally unit tests were planned for the prototype, but were scrapped due to lack of experience with the testing architecture.
- *Generated documentation*- Using Doxygen or a similar tool to automatically generate code documentation would be pretty simple to add to the build process, but adding and reformatting appropriate comments would take too much time to do during the prototype.
- *Exception handling*- The application should have more robust exception handling, but it's a little too much overhead for a prototype.
- *Signal handling*- Responding to signals from the OS is a good way to ensure the program shuts down gracefully when possible, especially since the CN might want to let other nodes know when it is being stopped.