Communications Node

Requirements and High Level Design (HLD) Document

This is the high-level design and requirements specification document for the Communications Node problem presented to Robert Miller by Uber ATC. 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 in C++ and be compiled to run on multiple platforms (OS X, Linux, and Windows at the least). A HTTP server will also be written using Python. This application will make heavy use of the Boost libraries to speed up production.

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
- The CN's port, ID, and heartbeat interval are all configurable.
- Allow startup and shutdown of nodes through the web

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.

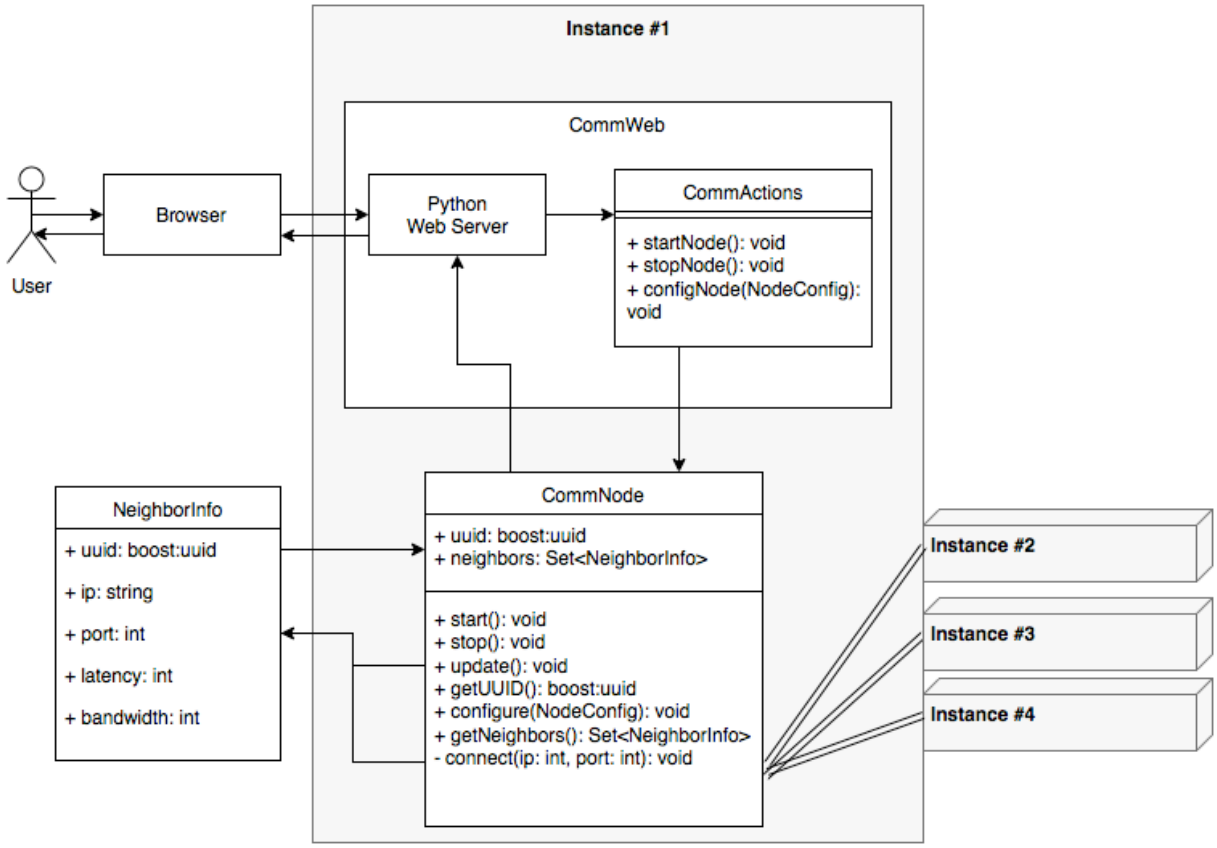
NeighborInfo- This is a C++ class/struct that stores some information about a CommNode object.

CommWeb- This C++ class launches a python HTTP server that can be accessed from a browser. When an action is made in the browser, an event is triggered by a handler in an instance of the CommActions class.

CommActions- This C++ class is used to handle events that are triggered in CommWeb. It makes for cleaner code to keep the event handler separate from the web server.

Web Server- The python web server will be run by the CommWeb class. It will be a simple http server that displays some information and data visualization from the nodes on the network. It will also allow basic actions such as stopping and starting nodes as well as sending configuration data.

Architecture Diagram



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	We'll use Boost.PropertiesTree for parsing a .ini file

	configurable	
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 IP address and the port to establish the connection to. The connections will be established asynchronously.
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 to be retrieved by the Python web server.	This latency will be measured by boost timers 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.	This will be calculated using the ping-pong messages used for the latency requirement.
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 Set 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. The problem here is with the web server. A perfect solution would be to have the first CN up stand up the python server. Other CNs would see that the port is currently occupied and attempt to register with the server. When the user makes a request through the browser, they are presented with a control that allows them to switch views between all of the CNs on the node.
The CN's port, ID, and heartbeat interval are all configurable	Some configuration options need to be made available to the user. There will be both a GUI option and a .ini file to allow the user to control at least CN IDs, heartbeat intervals, and port numbers.	As previously stated, we will use the Boost.PropertiesTree to parse the .ini file.
Allow startup and shutdown of nodes through the web	The user must be able to startup and shutdown both neighboring nodes and the current node through the web interface.	Closing the python server gracefully if the current node is shut down would be the biggest challenge here. Everything else could be handled with

		TCP socket calls.

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, sends heartbeats, and behaves as normal. When a browser is opened, 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.