EECS 376
Andrew Wood
aew61
Homework 4

This assignment was more difficult than I anticipated. The difficulty lies mostly with subtle assumptions about the nature and status of paths generated by the clients, as well as the architecture that I used. This assignment was about using an action server and client to control a robot (in stdr simulator) that moved from waypoint to waypoint (as implemented in Homework 3) but needed to avoid obstacles using an onboard lidar. More precisely, the action client was to submit a plan for asynchronous execution to the server, and was then to listen for lidar alarms (as implemented in Homework 2). When a lidar alarm was detected, the client would cancel the plan in action (assuming the plan was not already fully executed), and was to submit another plan. After talking with Professor Newman, he specified that this "secondary" plan did not have to be different from the first plan, it just had to show the asynchronous dispatch properties of an action server.

My implementation was built around the ideas of layers. The action server was built in three layers, with each successive layer handling more "low-level" functionality. The first layer was comprised of the "main" function, and was responsible for initializing parameters of movement, publication / subscription / action topics, and creating the server instance. The second layer was the sever implementation itself, and it was solely responsible for accepting requests and supplying the cancel, feedback, and success interface. The third layer was responsible for generating the commands necessary to execute the plan given to it by the second layer, and for executing that plan (when instructed), instruction by instruction. As will be shown, this type of architecture, while robust in the sense of code reuse and abstraction, actually constrained me when lidar alarms were introduced.

When the client cancels an action (path execution), this implies that the path is not safe to execute, and there are obstacles in the way. Therefore, that information must propagate its way down through the layers of the action server until they reach the currently executing movement command (the interface was to determine the speed, yaw rate, and movement amounts (either a distance or a theta), and the movement command would execute until that amount had been moved) so that the robot can be halted. This implies that there must be some hook built into the system that could be used to break the loop of the movement command. Since the server is not multithreaded, then class member variables are not a suitable option as changes will not propagate down since the thread will be occupied by the movement loop, and therefore the only suitable option is to use the actionlib::SimpleActionServer instance itself that is generated by the second layer.

The problem with this is that now ANY movement commands will be halted if a SINGLE cancel request is made, so it is not up to the client to determine when it should cancel an action. If the client cancels every time it receives a lidar alarm, then no secondary path can be executed since once a lidar alarm is received, that alarm will always prevent further movement since its existence stops movement from occurring in the third layer of the server. The question then became, is there a reasonable way to assume that a newly uploaded path should be executed before the client takes lidar alarms seriously again? I could not come up with a satisfactory answer, so therefore my implementation halts forever once a single lidar alarm is detected.