3.4 PRT-RRT* Software Design

We developed the PRT-RRT* implementation on the 7-DoF Franka Emika Panda robot arm using open-source software. The Robot Operating System (ROS) framework was utilized for managing communication between the planning, collision detection, control, and sensing processes. Each of these four processes will be referred to as a component. The controller component utilizes ROS MoveIt for high-level control within the ROS franka, libfranka for low-level control of the Panda robot, and Franka ROS to bridge the gap. The path planning process was implemented in the Open Motion Planning Library (OMPL) framework. The collision detector and the planner components both use the Bullet Continuous Collison Detection (CCD) library for collision detection. Figure 3.5 illustrates the software dependencies of each component.

The software package and documentation for installing and implementing PRT-RRT* for Panda or any other robot with a URDF can be found in [2]. Note that while the planner and collision detector components will work with other robots given a URDF, in order to implement this process for a new setup the sensor and controller processes will need to be designed for the robot and sensor stack in use. We use the ROS Joint Trajectory Controller interface for PID control of the Panda robot arm. A PID controller can be easily set up and tuned for a new robot as described in the documentation in [45]. The PID control gains for the Panda controller are reported in Table 1 in Appendix A. Note that any control may be used provided it is capable of reaching the neighborhood of goal configurations within a known finite time.

3.5 PRT-RRT* Component

There are four distinct components running in parallel that make up the PRT-RRT* process: the planner component, the controller component, the collision checker component, and the sensor component. See Figure 3.6 for a diagram of the communication between components.

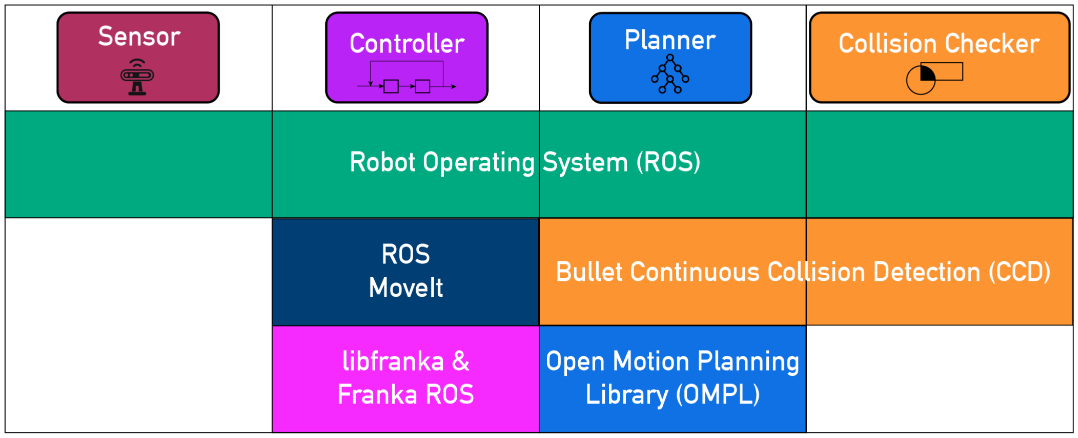

Sensor
Controller
Planner
Collision Checker
Robot Operating System (ROS)
ROS MoveIt
Bullet Continuous Collision Detection (CCD)
libfranka & Franka ROS
Open Motion Planning Library (OMPL)

Figure 3.5: The PRT-RRT* components are laid out in the top row with the software packages used in the 3 rows below. The components were designed using the software packages within their individual columns.

3.5.1 Planner Component

The planner component is made up of four primary process states and the transitions between them. The states and state transitions are illustrated in Figure 3.7 and described below.

Tree Expansion. The planner component seeks solution paths to the planning problem, starting from the current state of the agent $q(t)_{curr}$ and ending at the current target state $q(t)_{target}$ $\forall t$. The planner process employs the RRT* algorithm discussed in Section 3.2.2 to expand and rewire the tree as it searches for solution paths. Once a solution path is found, the planner enters the communication state, publishes the path out to the other components, and waits to hear back.

Communication. This short waiting period is essential to keep the solution path information aligned between the parallel processes. For example, if the planner continues updating the tree and finds a better solution path, say $\hat{E}(t)_{target}$ after $E(t)_{target}$ was published, then the path to be executed by the controller and checked for collision by the collision checker will be misaligned with the path held by the planner ($\hat{E}(t)_{target} \neq E(t)_{target}$). Thus, the planner must wait to continue operating on the tree until it receives an update that the controller is executing the next step in the

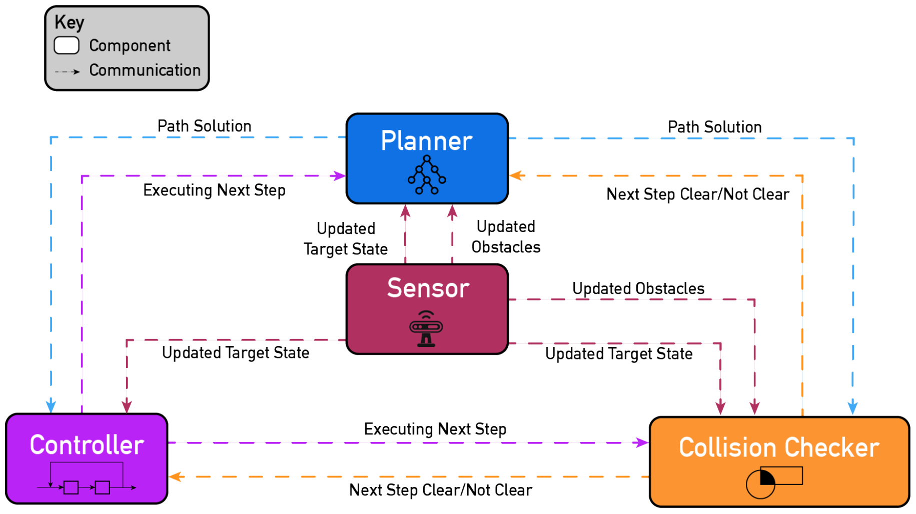


Figure 3.6: PRT-RRT* Communication Between Components

path or that the collision checker found the next step obstructed.

Tree Maintenance. Once the planner is notified that the control has begun executing to the next state in the solution path, the planner will transition to maintaining the planning tree. To maintain the planning tree, the planner component first advances the root node and then rewires from the root node similar to the RT-RRT* routines discussed in Section 3.2.2. Recall that in RT-RRT*, the root node is advanced when the agent state gets sufficiently close to the root node. In contrast, the PRT-RRT* planner does not advance the root node until the controller has started executing to a new state. The planner component will output an updated solution path if a better path is found during rewiring. The planner spends the majority of the maintenance period rewiring the tree, but also allocates some time to expand the tree using RRT* so that exploration of the space continues throughout the process. This exploration is useful in case the target state changes in the future.

Rerouting. If the next edge in the solution path becomes obstructed then the planner needs to seek an unobstructed path to the target. The collision checker component will alert the planner component to reroute. The reroute routine attempts to find a better parent for each of the nodes along the currently obstructed solution

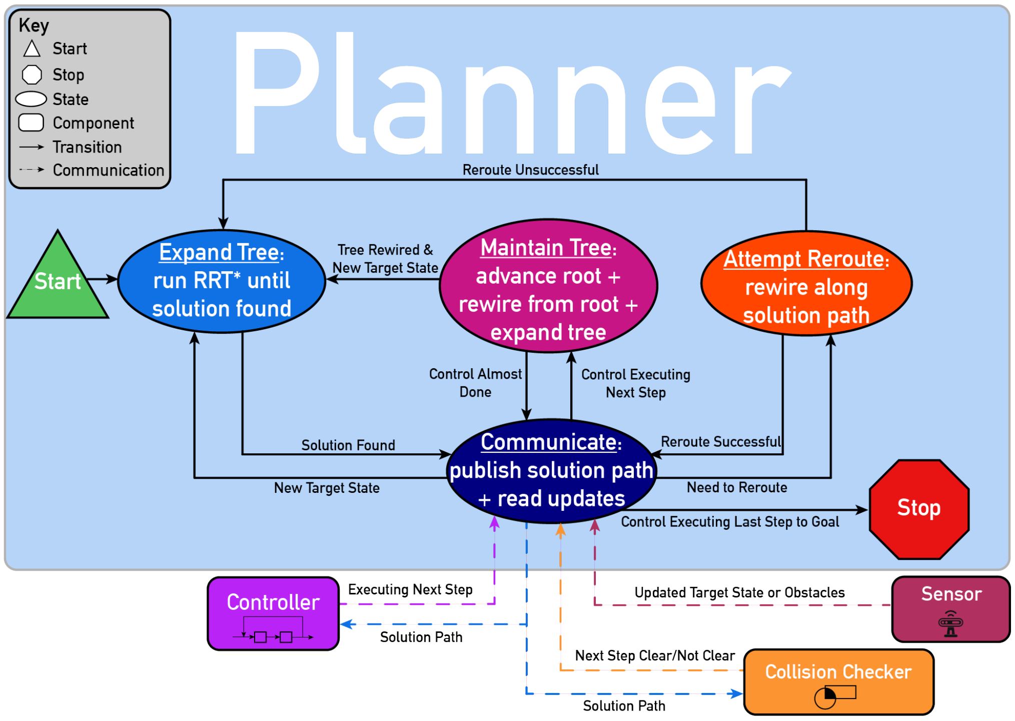

Key
Start
Stop
State
Component
Transition
Communication
Planner
Reroute Unsuccessful
Start
Expand Tree: run RRT* until solution found
Tree Rewired & New Target State
Maintain Tree: advance root + rewire from root + expand tree
Attempt Reroute: rewire along solution path
Control Almost Done
Control Executing Next Step
Solution Found
Communicate: publish solution path + read updates
Reroute Successful
New Target State
Need to Reroute
Stop
Control Executing Last Step to Goal
Controller
Executing Next Step
Solution Path
Updated Target State or Obstacles
Sensor
Next Step Clear/Not Clear
Collision Checker
Solution Path

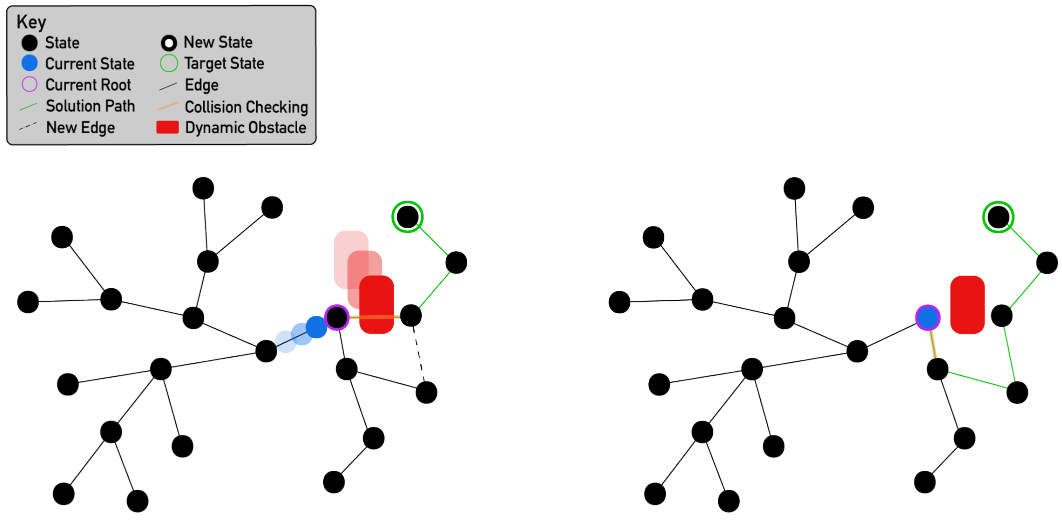
Figure 3.7: PRT-RRT* Planner States

path, $q_i \in Q_{target}$ in order to yield an unobstructed solution path to the target state among nodes already in the tree. If the reroute routine fails to find an unobstructed path to the target already in the tree, then the planner will expand the tree via RRT* until a solution path is found. The reroute routine is outlined in Algorithm 7 and an example of a successful reroute is depicted in Figure 3.8. This reroute routine is a novel contribution.

To ensure solution paths account for up-to-date information about the environment, the planner component receives information about the current state of obstacles $Q(t)_{obs}$ and the current target state $q(t)_{target}$ from the sensor component.

Algorithm 7: PRT-RRT* Reroute Routine

```
  Input: $\mathcal{T}, Q(t)_{target}, Q(t)_{obs}$
1 for $q_i \in Q(t)_{target}$ do
2     if $q_i \in Q(t)_{obs}$ then
3         continue
4     else
5         $Q_{nn} = \text{GetNeighbors}(\mathcal{T}, q_i)$
6         $\text{AddToBestParent}(\mathcal{T}, Q_{nn}, p_i, q_i)$ using Algorithm 3
7     end
8 end
  Output: $True$ if reroute succeeded, else $False$
```



(a) The solution path (green path) becomes obstructed by a dynamic obstacle (red block) and the planner attempts to reroute the next state in the solution path.

(b) The reroute is successful and an unobstructed solution path (green path) is found to the target state (green circle).

Figure 3.8: Example of Successful PRT-RRT* Reroute.

Hyperparameters. The hyperparameters incorporated in the PRT-RRT* planner are listed below.

- Max distance (r_{max}): Maximum euclidean distance between two connected nodes in the tree.
- Goal Bias (α): Small number in the interval $[0, 1]$ that influences how often the goal is sampled. See Equation 3.3

- Max neighbors (k_{max}): Maximum neighbors allowed within r_{max} of a node. If a new sample exceeds this number, then it will be rejected. See Algorithm 5.

- Nearest neighbor (r_{min}): Minimum Euclidean distance allowed between two nodes in the tree. If a new sample is closer than r_{min} to a sample already in the tree, then it will be rejected. See Algorithm 5

- Prime Tree seconds (t_{prime}): Amount of time in seconds the PRT-RRT* algorithm is initially allowed to plan before returning a solution.

3.5.2 Controller Component

The controller component executes solution paths $E_{target}(t)$ step-by-step after they have been received from the planner. Parallelizing this process allows the control signal frequency to be independent of the planner process frequency. Each step in the path must be explicitly declared collision-free by the collision checker before the controller will begin execution of the step.

As the controller component begins executing the next step in the current solution path, it outputs an update that it is executing the next step, and how long it will take to execute. If the controller component receives a notification from the sensor component that the target state has been updated or a notification from the collision checker that the next step in the path is obstructed, it will disregard its stored solution path and wait for the planner to output an updated solution path to the current target state before executing any more steps. The controller process is illustrated in Figure 3.9.

3.5.3 Collision Checker Component

The collision checker component repeatedly checks for collisions along the first edge in $E_{target}(t)$ between the current root node, $q_{root}(t)$ and the second node in $Q_{target}(t)$ $\forall t$. When the collision checker component first checks for collisions along a new edge, or when the collision status changes along an edge already checked for collisions, it

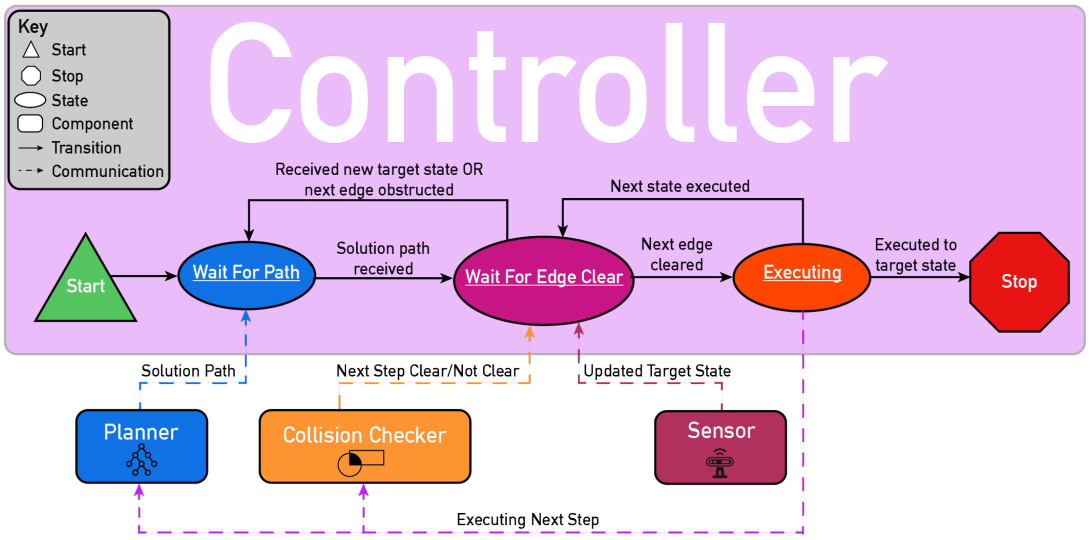


Figure 3.9: PRT-RRT* Controller States

will output whether the edge is collision-free or not. Recall that $q_{root}(t)$ is advanced to the next node in $Q_{target}(t)$ when the controller begins executing from the previous root. The collision checker process is illustrated in Figure 3.10.

3.5.4 Sensor Component

The sensor component observes when obstacles move or the target state changes and communicates this information to other components. The sensor component is purely an observer of the environment and does not take any input from other components.

A good example to illustrate the sensor component detection is a camera observing the space around a robot arm for a pick and place application. If the robot is tasked with picking up moving objects, then both the target state and the obstacle state related to the object change. The sensor component must detect this and notify the other components.

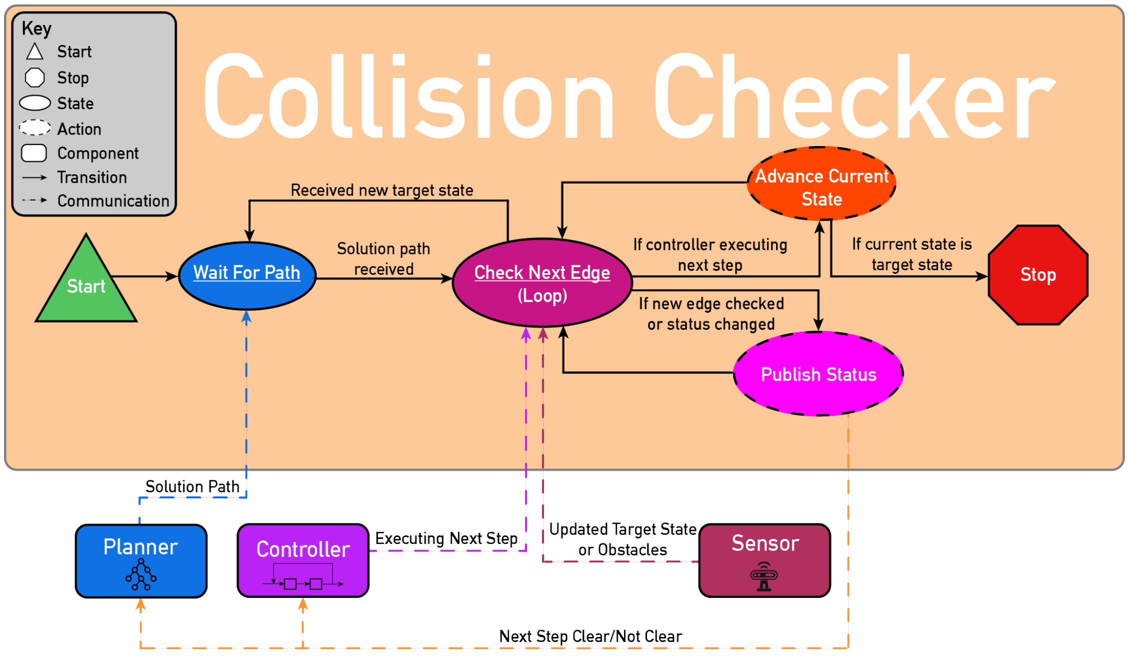


Figure 3.10: PRT-RRT* Collision Checker States