Robotic Manipulation
Anthony McNicoll, Andy Li, Alex Volkov
HW 1: The Robot Operating System

5. In order to debug our *sim_master* node during development, we made use of the *rosservice, rosnode, rostopic,* and *rosparam* ROS commands.

Note: *rosparam* was needed to set ROS parameter server variables when testing the node using *sim_master*; under normal conditions (using roslaunch) these parameters are automatically initialized in the launch file.

Example uses of these commands are shown below：

Set ROS parameter num_block to 10, then echo it back to make sure it was set correctly:
```
$ rosparam set num_blocks 10; rosparam get num_blocks
```

Get info on node *sim_master*, such as connections to topics and services:
```
$ rosnode info sim_master
```

Echo the activity on topic /status (sent from *sim_master*):
```
$ rostopic echo /status
```

Call the /move_robot service (that *sim_master* responds to) with Action=open and Target=0 arguments
```
$ rosservice call /move_robot ‘open’ 0
```

7. For tasks *scatter*, *stack_ascending* and *stack_descending*, we drive each arm per message for each single action. However, one does not need to wait the other gripper to finish the whole cycle of action (move to, close, move above, then open/drop). We can double the efficiency by shift their action sequence as shown in the table below.

Table 1. Example of alternative manipulation to scatter a stack of 7 blocks, or to ascend or descend 7 blocks from scattered pattern.

	Gripper 1		Gripper 2	
t1	Move To	Block 1	stand by	
t2	Close		stand by	
t3	Move Above		Move To	Block 2
t4	Open		Close	
t5	Move To	Block 3	Move Above	
t6	Close		Open	
t7	Move Above		Move To	Block 4
t8	Open		Close	
t9	Move To	Block 5	Move Above	
t10	Close		Open	
t11	Move Above		Move To	Block 6
t12	Open		Close	
t13	stand by		Move Above	
t14	stand by		Open	

Table 1. Example of alternative manipulation to scatter a stack of 7 blocks, or to ascend or descend 7 blocks from scattered pattern.

For the *odd_even* task, we will implement parallel manipulation by calling two arms to stack up each stacks with synchronized or shifted action sequence. Because in this case the grippers are much less likely to interact others' motion. On the other hand, compared to the three tasks above, grippers cannot perform synchronized action sequence. However, whether the motion is synchronized or not, their efficiency will be the same. But the synchronized manipulation will have more risk of physical interference between grippers/arms.

8. In our implementation of the *controller* and *sim_master* nodes, it was not necessary for the former to wait for an update on the /status topic to issue a new command on /move_robot service. Because the system is entirely simulated and therefore deterministic (in that the entire state of the system is captured in the nodes), we were able to have *controller* issue a burst of commands over /move_robot to *sim_master* at a time. Each such burst is initiated (if necessary) by an update arriving on the */status* topic, basically acting like an initial condition from which the *controller* node calculates the necessary next steps (commands) for the robot (*sim_master*) to take. In other words, we issue this burst of commands effectively open-loop, but because the entire system is defined and modeled in our software, there is no room for error that would cause the system to deviate from the *controller's* expectations. Of course, such a methodology would not work with a real robot (such as Baxter), as continuous feedback of state from the embodied and far more complicated system will be necessary to ensure the controller maintains an accurate model and will be able to issue the correct commands.

9. In our configuration, attempting to run a second instance of sim_master will kill the first instance. This is because the node is initialized without being anonymous and any second instance of the node with the same name will cause ROS to kill the first copy. Starting the second node with a different name will allow it to run in parallel with the first copy. It will receive the same commands over the common /move_robot service, and publish its own state on /state. Unfortunately for the controller, which assumes there is only one sim_master, the duplicate information (likely describing two totally different states, as the nodes might not both start in the same configuration depending when you run the second node) will not adhere to the expected state transitions of a sim_master node, thereby confusing the controller and if not causing a fatal error (due to some safety checks we have implemented in parsing the messages) then at least sending incorrect commands. Running a second instance of the controller will have a similar effect, except this time in the form of two separate sources of commands coming in on /move_robot, thereby also breaking the invariant model of the system assumed in our code.