# Methods

## High-Level System

The self-assembly planning problem for MSRR is trying to figure out how to construct an arbitrary configuration given multiple modules. And the Self-Reconfiguration Planning problem is to figure out how the MSRR transforms from one arbitrary configuration to another. In our method, we researched an extended version of this problem, the Self-Reconfiguration Planning for the MSRR group which means transform from a set of arbitrary configurations to another set of arbitrary configurations while the number of modules and configurations in each set could be different.

***Related works***

Casal and Yim [5] firstly published a divide-and-conquer approach for this problem. Two algorithms were presented which are based on a wireframe depiction of modular robot configurations and a substructure set resulting in a hierarchy construction of initial and goal configurations.

Hou and Shen [6] presented a configuration string to represent a robot configuration. Common and different substructures are detected, and all different substructures can be reconfigured into an intermediate structure and then into goal substructures. However, the way to compare two configurations is with respect to the center of graphs so that it is very likely to not find common substructures, resulting in many redundant reconfiguration steps.

A graph-based optimal reconfiguration planning approach [7] was developed later. The reconfiguration problem is converted into a distributed constraint optimization problem (DCOP) which is solved by existing DCOP algorithms whereas solving DCOP takes exponential time. Thus, a greedy algorithm was introduced to solve the configuration matching problem, but the optimal solution is not guaranteed. All these works require that all modules are connected during the reconfiguration process due to the limited motion capabilities of their hardware.

A configuration recognition algorithm using distributed information for modular robots was presented in [4]. A graph representation for modular robots was presented, including the definition of root module and connections. A matching and mapping algorithm is developed for configuration recognition by computing the maximum common subconfiguration (MCS) with respect to root modules.

Our method also incorporated the idea of MCS and based on the graph representation of configurations. However, instead of finding one common subconfiguration (CS) with the maximum number of modules, we are finding the Maximum set of common subconfigurations (MSCS) which include as more modules as possible in order to reduce the number of docking and undocking. And [8] also based on the reconfiguration for one MSRR robot and assumed that the number of modules is not changed. However, in our methods, we could handle the reconfiguration between the configurations with a different number of modules and even reconfiguration on a group of MSRR robots. Moreover, [8] cannot work on the configurations with the cycle while our method could.

***System Architecture***

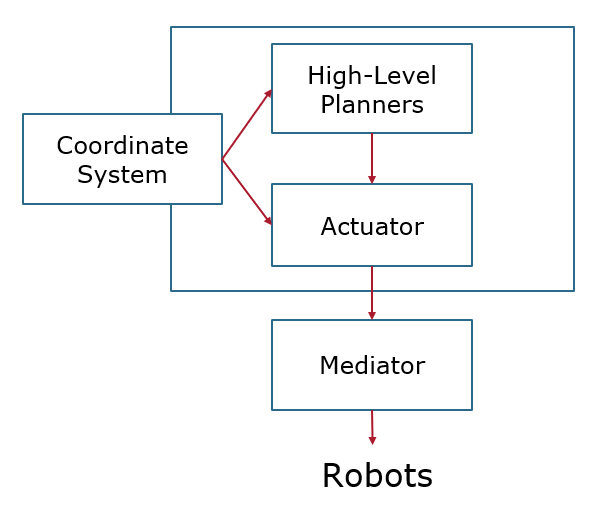


Fig. 2 The high-level system architecture

The subsystems in our high-level system are shown in Fig. 2. TheHigh-level planners and Actuators are the center of the high-level system. The high-level planners are composed of the planners that planning for high-level behaviors and output the plans composed of primitives. In our robot, the primitives are *Dock, Undock, and Move*, and the high-level planners include two high-level behaviors, self-reconfigure, and self-assembly. The details of these two planners will be discussed in the following parts. The actuator is composed of a plan processor that processes the plans from different planners and the compilers for the primitives that could explain the primitives as the actual robot action sequence for the controller. For example, a dock primitive *[0, 1, 1, 2, 1, 0]*, which means dock the left face of module 0 to the right face of module 1 with default direction and 0 orientation, primitive will be compiled as a path of the module 0 to move from its current position and rotation to the position and rotation that enable it to finish the dock. And with the desired path and rotation, then these signals will be sent to a low-level path tracking controller in the robot.

The high-level behaviors could be simply expanded by adding new high-level planners. When adding new high-level behavior, the actuator should also be updated to make sure the plan could be correctly explained. The granularity of the primitives could be controlled by expanding the compilers for different primitives, it could be fine-grained like the direct control signals or coarse-grained like pattern recognition.

The Coordinate system is a supporting system that supporting the high-level planners and the compilers in the actuators. For example, when planning a path for compiling a move primitive, the coordinates of each module need to be obtained by the coordinate system. And the supporting system could be expanded for supporting more high-level behaviors or compiling more primitives, for example, adding a SLAM system could enable the robots to coordinate themselves without a supervising camera or a drone, a computer vision system enables the robots to executed visual-based behaviors.

The mediator is a server that solves the problem of the high-level system to communicate with other systems such as other robots or other platforms. For example, the ROS is running on Python 2, while our system is running on Python 3, which are running in a different environment, so our system could not directly publish messages in ROS, using a mediator could relay our signals to ROS.

***Graph Representation***

The configuration is represented as an undirected graph in our method. Modules are represented as the vertices in the graph. And the edges represent the connection between the modules. A connection could be represented as *e= {, , , , Ori, Rot}*, where and means the modules of this connection and , means the connector being connected respectively. Connector *{Top, Left, Right, Bottom}*. *Ori* means the orientation of the connection, for SMORES-EP, only the connection between two BOTTOM face connectors have different orientations. The connection between other faces could be rotated respective to each other, so the connection orientation makes no influence. *Rot* means the rotation between two faces. In our experiments, we assumed that the configuration lies on a 2D lattice. So, the rotation between to face should be 0 or 180 degrees. The reason for adding a rotation term in the connection is to calculate the topology of the configuration which is useful for planning the mapping from modules in current configuration to the target configuration in the following steps. It should also be noticed that our method is not restricted to 2D lattice, it could easily be extended to a 3D lattice, however, the 2D mapping planning algorithm in our method should be replaced by a 3D planner when this method is applied to the 3D lattice.

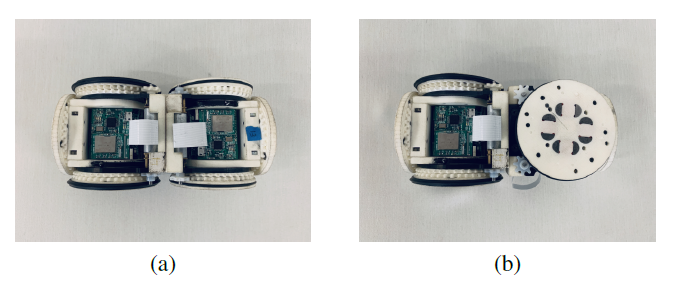


Fig. 3 The BOTTOM faces cannot rotate respective to each other. So, there are two ways or orientations to connect. The *Ori* value for (a) is 0 and 1 for (b).

One advantage of our method is the supporting of the self-reconfiguration planning of a group of MSRR robots which means when multiple robots in a group are required to reconfigure to new configurations, they can help each other by finding CSs in multiple robots. An MSRR group } is represented as the set of configuration graphs *= {, , ,……}* of the robots in this group. And the target configurations represented as the graph set *= {, , ,……}*.G represents the configuration graph (see Fig.2) . The edges are connections and the vertices are modules.

***Cycle Detect and Break***

Since most of the algorithms in our methods cannot work when cycles existed in the graph. So, the first step is to detect and break the cycle in the configurations. There are many classic methods to detect the cycles in an undirected graph such as DFS and Union-Find. For simplicity, we used a brute-force search to find the sub-graph which degrees of all vertices are 2 in our method. It’s not an efficient way but this part could be easily replaced by other quicker algorithms.

The cycle detects algorithm will output the edge set of each circle. In order to break these cycles, we need to choose one non-repeating edge from each cycle to remove from the graph. And we’ll record these edges since, in self-assembly and self-reconfiguration, we need to recover these edges. However, for the self-reconfiguration planning, break the cycle will also change the topology of the configurations which may reduce the number of potential CSs. So, we’ll enumerate each break plans, and used the MSCS search algorithm to find the size of its MSCS. And we’ll choose the break plan with a maximum size of MSCS. We discuss the MSCS algorithm in the following.

***Self-Assembly Planning***

A basic ability of an MSRR system is to assembly the robots in the system with configurations in from a set of modules. Firstly, we need to choose a root for the configuration, the root will keep fixed in the assembly, other modules will be connected start from the root. One way to make the self-assembly time shorter is by letting the number of modules connected on each direction of the root balanced. Connection number (CN) [9] described the number of modules connected to each connector of a module. The method we calculate the CN is from [9]. After the CN of all modules is calculated, we could choose the module with all terms of its CN less than half of the number of modules in this configuration as the root.

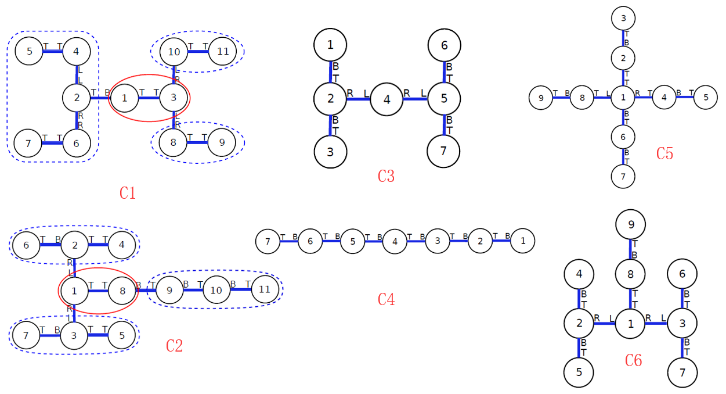


Fig. 4 Example of configurations.

After we found the root of the configuration, we’ll calculate the height of each module’s respect to the root. And then the self-assembly sequence could be determined based on the height of the module from low to high. For each configuration in , we’ll generate its self-assembly plan.

For example, the self-assembly plan for configuration C6 in Fig. 2 is:

*Root Module: 0*

*Stage 1*

*[[0, 1, 1, 2, 1, 0], [0, 2, 2, 1, 1, 0], [0, 7, 0, 0, 1, 0]]*

*Stage 2*

*[[1, 3, 0, 3, 1, 0], [1, 4, 3, 0, 1, 0], [2, 5, 0, 3, 1, 0], [2, 6, 3, 0, 1, 0], [7, 8, 3, 0, 1, 0]]*

Notice that the module index in our implementation started from 0 while in Fig. 4 it started from 1. The algorithm chose module 1 as the root module since the CN of module 1 is *[2,3,3,0]* (the number of modules connected to Top, Left, Right and Bottom face respectively). In stage 1, module 2,4,8 could connect to the module 1 parallelly, in stage 2 other connection could be executed.

***Maximum Set of Common Subconfigurations (MSCS)***

In order to reduce the docking and undocking times, we want to keep as more connections to the current configurations as possible. So, we need to search for the CSs between and . And find a set of CSs that contains the maximum number of modules.

Firstly, we need to search for all potential CSs between the configurations and . We compared a pair of configurations from and respectively at one time and compare all combinations between the configurations from and . In the search of one pair, we initiated the potential CS list *C* by comparing every pair of edges from and . If it meets the isomorphic condition, we could add a new branch of CS set with an initiated mapping (means mapping from in an edge from to in an edge from ) to list *C*. The isomorphic condition used in our method is:

*For two edges: :{} from ,*

*:{} from*

*If equal to , then*

*if equal to , return condition 3*

*else if not equal to , return condition 1*

*else if equal to return condition 2*

*else return False*

Condition 1 means the two edges are isomorphic, we’ll add this mapping to the CS. Condition 3 means the two edges are isomorphic and symmetry, we’ll add this mapping to the CS and adding a new branch of CS that adding the mapping from the inverse of to instead. Condition 2 means the inverse of is isomorphic to the and we’ll add the mapping from the inverse of to to CS. In some MSRR system, there are symmetries exist such as in SMORES-EP, the connection between to Left faces and two Right faces is mutually replaceable, by modifying the isomorphic condition, we could simply incorporate the symmetries to simplify the search.

After initiation, we’ll search each branch in *C*, we iteratively compare all connected edges for the leaf vertices in an SC and the SC mapped by it and adding all edge mappings that meet the isomorphic condition to the SC and expand again until no edge mapping meets the condition. Then we’ll execute a clear for *C* to move out the same CS from the list. And the resulting *C* contains all potential SC between a pair of configurations from and respectively. And we’ll merge the *C* of each pair of configurations to get our final *C*.

Then, we need to choose the set of CSs from *C* to get the set of CS that contain the maximum number of vertices. We firstly initiated a list subconfiguration combination *SCC* as a list with an empty list. And a list that represents the CS that could be added to the current combination *available* as a list contains a list of the index of all CS in *C*. Then iteratively to create a new branch of combination by adding one CS to a combination in *SCC* from its corresponding *available* list and then update the *available* list. Creating a new branch of combination means adding the combination to *SCC* and adding its available CSs to *available*. The way of generating the available list for a combination is to find all CSs in *C* that have the mapped vertices in that are not contained in that combination of CSs. After that, we could get all potential combinations of CSs in *C*. Then we keep all the combinations that covered the maximum number of vertices as the *MSCS*.

***Self-Reconfiguration Planning***

The information we need in the self-reconfiguration of MSRR group robot includes the root modules which means the modules that do not need to move, the modules used as the center of the configurations in , the pairs of modules needed the be un-docked, and the new connections need to be formed. Here, we’ll use a cluster graph to describe our configuration graph. The vertices in the cluster graph include SCs and the vertices that are not in SC.

Firstly, we need to choose the centers of target configurations. We enumerate all combinations of centers from all clusters and the mappings of modules then evaluate its potential cost of moving in the reconfiguration. And we choose the centers that have the least cost of moving. For each set of centers, we choose the root cluster from these center clusters by finding the cluster with a maximum number of vertices. We simply choose the root cluster from the center clusters may cause us to fail to find the global minimization of moving cost. However, it’s a trade-off between the computational cost and the accuracy of the result, this simple selection strategy helps us reducing the computational cost significantly while the result is still near perfect. The way we evaluate a set of centers is as follows.

With root clusters and the center clusters and then the mapping of each module. We firstly generate an estimated coordinate for modules the configurations in and and then estimated the moving cost based on it. Also, we need to consider the topology of the configurations. The topology contains the location of the centers of each configuration and their rotation respect to the configuration with the root cluster. For configurations in , the centers are the root of center clusters. The root is calculated based on CN as above. For configurations in , the centers are the root modules of each configuration except the configuration with the root cluster. For that configuration, the center is the root of the root cluster. Ideally, the topology of is given by sensor coordinates. However, if it’s not provided, the algorithm will assume that all configurations are aligned, this line is the x-axis for the estimated coordinate, and the origin of the coordinate is the center of root cluster. Then, we regard the Top face of the root module is +y, Bottom is -y, Left is -x, Right is +x. And regard the MSRR modules as cubes which length of edges are unit length. Then, we calculate the rotation of each module with respect to the center of each configuration. And calculate their coordinates iteratively. Then, transform the coordinates of modules in each configuration to the root frame to get all coordinates in and . Then we merge the coordinates of and to simulate the worst case.

Based on the mapping and the new connections required to calculate the cost of moving. The cost of one move from one coordinate to another is calculated by multiply the moving distance and the number of modules. Different metrics for moving costs could influence the result of planning. We used two way as the metrics of moving distance. The Euler distance and a rectangular distance. The rectangular distance defined as the large side (with respect to the two-point) of the smallest rectangle cross the two-point and one vertex of the smallest rectangle that could conclude all points in the coordinate. And the rectangular distance performed better in our experiments. Finally, we’ll choose the centers set and mapping with the lowest cost.

Then the docking sequence could be generated based on the mapping and new edges. New edges are the edges in the cluster graph of which are the edges that connect each cluster by connect the vertices of each cluster. If the new edges contain the vertices that are not mapped in the mapping which means the modules of are inadequate. The algorithm will add a virtual vertex noted by a negative index to the map that vertex. Similarly, the docking sequence could be obtained by the connecting edges in the cluster graph of .

As a result, we could obtain our self-reconfiguration plan based on above information we get. For example, the self-reconfiguration plan from C3 and C4 to C1 and C6 by our algorithm is:

*I.Clusters*

*[[0, 1, 3, 2, 4, 5, 6], [12, 13]]*

*II.Root Modules*

*[0, 1, 3, 2, 4, 5, 6]*

*II.Graph Centers*

*[[11], [0, 1, 3, 2, 4, 5, 6]]*

*III.Undocking*

*[[7, 8], [8, 9], [9, 10], [10, 11], [11, 12]]*

*IV.Docking Sequence*

*Stage 1*

*[[12, -1, 0, 0, 1, 0], [13, -2, 1, 1, 1, 0], [13, -4, 2, 2, 1, 0], [3, 9, 0, 0, 1, 0]]*

*Stage 2*

*[[-3, -2, 0, 0, 1, 0], [11, -4, 0, 0, 1, 0], [-1, -5, 2, 1, 1, 1], [-1, 7, 1, 2, 1, 1], [9, 10, 3, 0, 1, 0]]*

*Stage 3*

*[[-5, -6, 0, 0, 1, 0], [7, 8, 0, 0, 1, 0]]*

where the vertices indexed by negative numbers means the modules lacked.

***Actuator***

The main part of our actuator is a compiler for compile the *Dock* primitive. After obtaining the coordinates and rotations from the coordinate system, the target position and target rotation could be influenced by the relative position and rotation of the module and the target module. The desired path could be computed by a path planner.

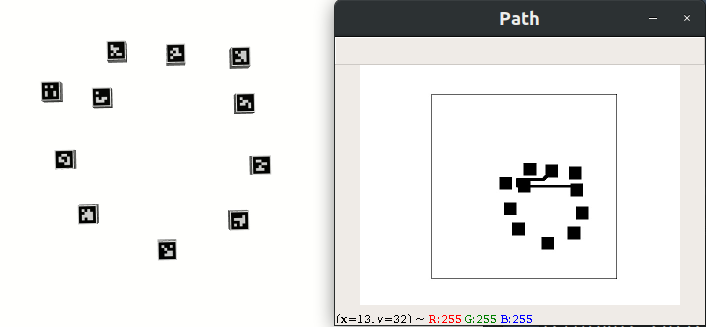


Fig. 5 Path Planner

The path planner is based on the Jump Point Search (JPS) [10] pathfinder. Compared to other search-based planner like A\* [11] which could provide a global-optimum path and Theta\* [11] which provides a smooth and optimum path, the JPS do not guarantee a global optimum but runs much faster than above ones. Compared to the sampling-based methods like RRT [12] and RRT\* [12], the RRT does not provide optimum results although it runs much faster. RRT\* could provide a near-optimum result while it requires a large amount of sampling points, however, doing this will make the performance advantage of RRT\* disappear, and the results from JPS is also more stable than it could always provide a near-optimum result.

Fig. 5 showed a sample of the path planning for docking. The data is represented as a bitmap. The path is started from the current position of the module and its docking position. It could be seen that the paths for multiple modules could be planned simultaneously. The paths for modules in the same stage will be planned at the same time. And the planner will also find the global shortest paths for multiple modules that the guarantee summation of the paths for all modules in a stage shortest.

***Coordinate system***

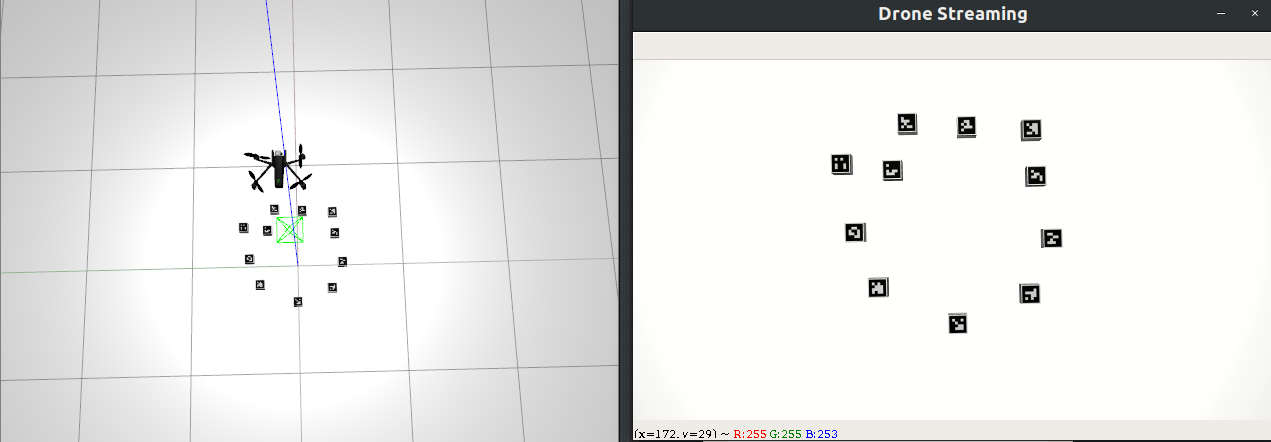


Fig. 6 Coordinate system

Fig. 6 showed our coordinate system. The coordinate executed by a drone. Using a drone to execute the coordinate could provide us an Orthographic Projection which makes the coordinate more accurate, moreover the horizon of the drone is larger, it could also decrease the heterogeneity of the robot system by not introducing a camera to the robot.

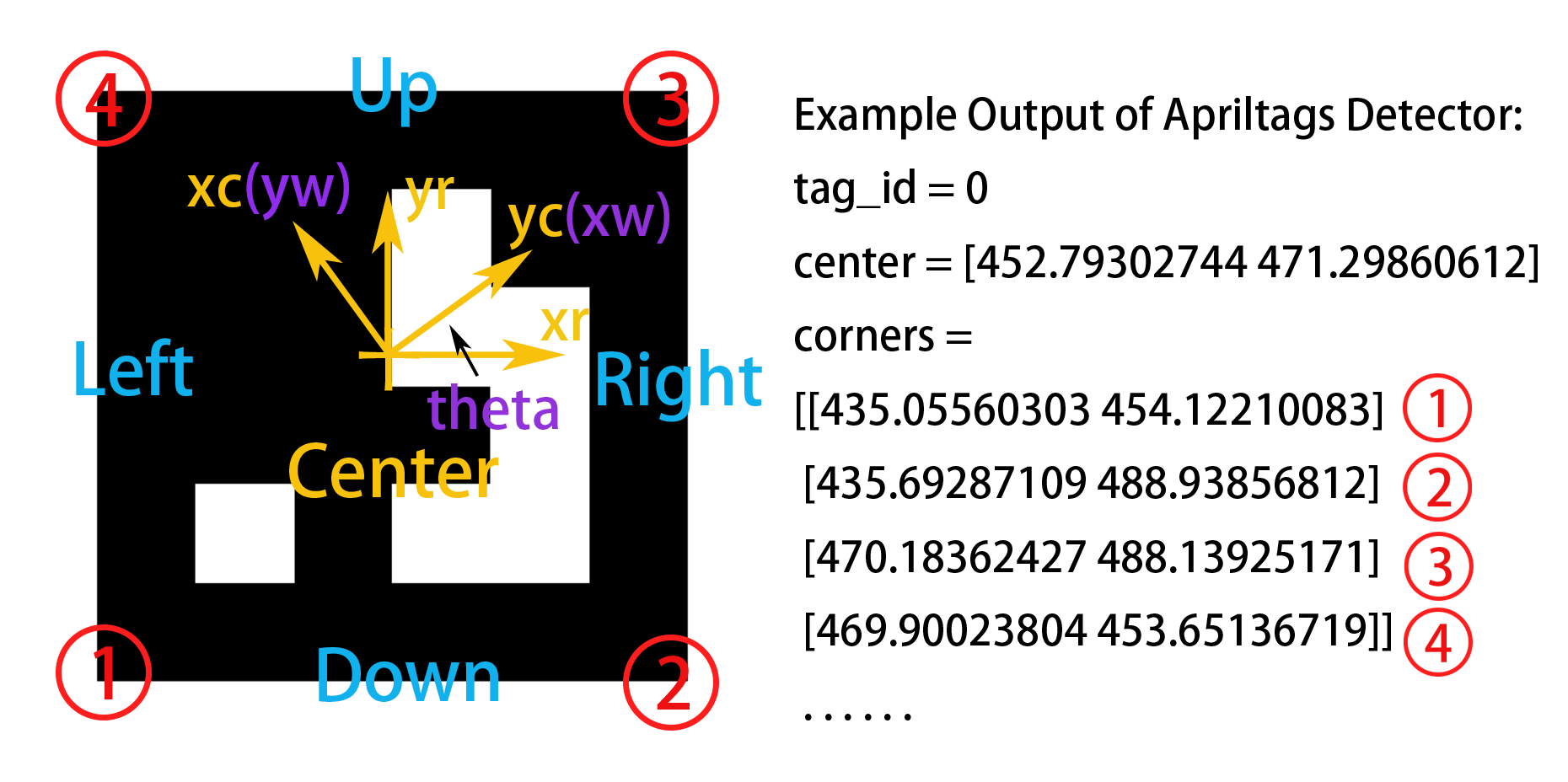


Fig. 7 AprilTag-based coordinates

The coordinate is based on AprilTag [13]. AptilTag detector could return the coordinates of the center and four corners of the tag, the orientation of each faces and the robot frame we used of the tag are described in the Fig. 7 by the blue labels and xr, yr. For a group of modules, we will choose a module as the center module, the choosing of the center is based on the root modules of the reconfiguration, xc, yc means the frame of the center module, it was also used as the world frame, the position of center and the orientation of each module theta is calculated in this world frame. Moreover, in order to eliminate the influence from the scaling when the relative height of the drone changed, the coordinates will be regulated by divided by the length of the edge of the tag, and then scaled in order to generate a finer bitmap used for path planning.

***Mediator***

The communication between the mediator and the high-level system is based on a socket. Then the signals are relayed to other systems like ROS. Fig. 8 showed the communication between the high-level system and the mediator server, the high-level system sends the signal include the current position, rotation, desired position, rotation, the derivative of desired position and rotation and the derivative of time to the module which shown in the left part of the figure. Then the mediator server relays the signal to a ROS topic while each module has a different topic as shown in the right part. By expanding the message processing program in the mediator server and providing corresponding interfaces, the message could be relay to other routes like a WiFi module in the real robot.

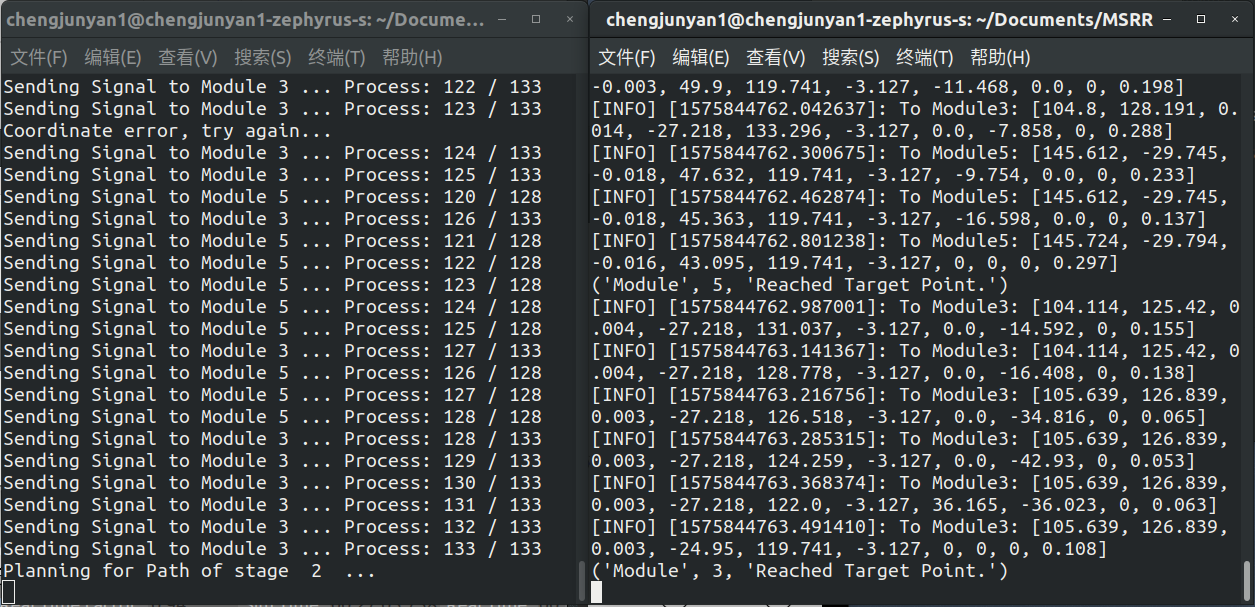


Fig. 8 Mediator server

Signals to different modules could be sent parallelly by multi-threads. For example, when sending the path for the modules in a same stage of reconfiguration, the paths of all these modules could be sent parallelly. When parallelly sending signals, a sync semaphore was used to enable the paths for the next stage will start to send only the signals for the current stage were completely sent out. And a mutex was used when sending signal for one module to keep the route of message sending is not interfered with by other messages.

# Conclusion

## High-Level Planners

*The self-assembly planner and the self-reconfiguration planner have been implemented in Python. Some improvements are made compared to the planner used by Moab. Including the cycle detect and break, finding a set of common sub configurations with a maximum number of modules instead of single maximum common sub configuration, self-reconfiguration for MSRR group instead of one MSRR, self-reconfiguration between the MSRR with different module number, minimizing the moving cost of modules in planning. However, our implementation is based on a centralized planner and cannot work sole using the distributed information. And the computational cost of our method is still high although self-reconfiguration planner does not require real-time. Moreover, using multiple common sub configurations also requires the mobile locomotion control for multiple modules which is not been accomplished yet.*

##### References

1. J. Daudelin, G. Jing, T. Tosun, M. Yim, H. Kress-Gazit, and M. Campbell, "An integrated system for perception-driven autonomy with modular robots," *Science Robotics,* vol. 3, no. 23, p. eaat 4983, 2018, doi: 10.1126/scirobotics.aat4983.
2. T. Tosun, J. Davey, C. Liu, and M. Yim, "Design and characterization of the EP-Face connector," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 9-14 Oct. 2016 2016, pp. 45-51, doi: 10.1109/IROS.2016.7759033.
3. C. Liu, M. Whitzer, and M. Yim, "A Distributed Reconfiguration Planning Algorithm for Modular Robots," *IEEE Robotics and Automation Letters,* vol. 4, no. 4, pp. 4231-4238, 2019, doi: 10.1109/LRA.2019.2930432.

[4] G. Jing, T. Tosun, M. Yim, H. Kress-Gazit, "Accomplishing high-level tasks with modular robots", *Autonomous Robots conditionally*

[5] A. Casal and M. Yim, “Self-Reconfiguration Planning for a Class of Modular Robots,” vol. 3839, 1999, pp. 3839–3839–12.

[6] F. Hou and W. Shen, “Distributed, Dynamics, and Autonomous Reconfiguration Planning for Chain-Type Self-Reconfigurable Robots,” pp. 3135–3140, May 2008.

[7] F. Hou and W. Shen, “Graph-Based Optimal Reconfiguration Planning for Self-Reconfigurable Robots,” Robotics and Autonomous Systems, vol. 62, no. 7, pp. 1047 – 1059, 2014.

[8] C. Liu and M. Yim, “Configuration Recognition with Distributed Information for Modular Robots,” in IFRR International Symposium on Robotics Research, 2017.

[9] F. Hou, “Self-Reconfiguration Planning for Modular Robots,” Ph.D. thesis, University of Southern California, Los Angeles, 2011

[10] D. Harabor and A. Grastien, “The JPS Pathfinding System”, in 26th National Conference on Artificial Intelligence. AAAI, 2012

[11] F. Duchon and A. Babinec, “Path Planning with Modified a Star Algorithm for a Mobile Robot”, in Procedia Engineering, vol. 96, 2014

[12] S. Karaman, E. Frazzoli, “Sampling-based Algorithms for Optimal Motion Planning”, The International Journal of Robotics Research, 30(7), 846–894., 2011

[13] Edwin Olson, “AprilTag: A robust and flexible visual fiducial system”, IEEE International Conference on Robotics and Automation, 2011