Cloud-based distributed intelligence of robot team in complex accident situation

Novak Zagradjanin, Aleksandar Rodic

Abstract - **Multi-robot systems can be considered as a suitable platform for dangerous mission in accident situations. In this paper we propose a multi-robot system with distributed intelligence that can perform complex mission faster and more reliable. Robots are connected to the cloud and can benefit from the powerful computational, storage and communication resources of the cloud, which processes and shares information. Our robot team consists of one robot scout and two robot manipulators with different sensory capabilities. The mission goal is to, in conditions of limited resources and insufficient information, collect enough data about accident situation, identify the most critical points and calculate appropriate path of robots in known or unknown environment where accident occurred.**

Index Terms **– multi-robot system; cloud; distributed intelligence; path planning.**

I. Introduction

In modern robotics robots are supposed to be as much as possible simple, energy efficient, low-cost, but still smart enough to carry out individual and collective intelligence. These two goals are normally contradictory to each other.

Cloud computing is a key enabler for solving these challenges. The idea is to use the advantages of the cloud technologies and to design and create architecture that will increase the effectiveness of a multi-robot system.

Cloud robotics is a field of robotics that attempts to incorporate cloud technologies centred on the benefits of converged infrastructure and shared services for robotics [1 - 4]. Cloud platform presented in this paper can communicate and control multiple types of robotic agents in complex accident situation. In order to perform a mission cooperatively, robots have to share the knowledge about the accident and environment through the cloud [5].

II. System overview

The over-all architecture of the system and its components is depicted in Figure 1. The intention of this architecture is to provide powerful processing capabilities to a multi-robot system, resource efficiency, interoperability and robustness [6].

The robot team is heterogeneous and consists of one robot scout and two robot manipulators. Robots are equipped with sensors, communication, actuation and manipulation modules, leaving all high level algorithms to the cloud.

Cloud is used as the support for multi-robot system and enables sparing resources and sharing data among robots. It is responsible for analyzing of accident situation, determining the weight distribution of accident classes, making decision about the most critical points, setting target destinations of robot manipulators to the weight distribution of accident classes and path planning of robots in complex and dynamic environment. For path planning D* Lite algorithm is used.

Data base feeds cloud with information specific to a particular accident and environment, as and other data important for mission planning.

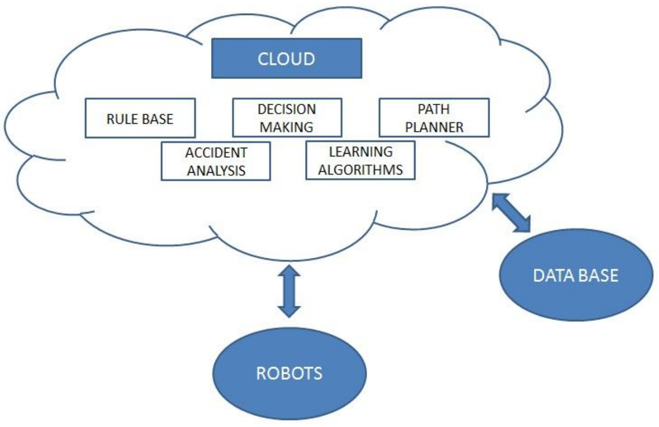

Fig. 1. The architecture of cloud-based multi-robot system.

III. D* Lite algorithm

D* Lite belongs to a group of graph-based path planners. It is an incremental algorithm that is efficient in dynamic environments. It works by performing an A* search to generate an initial solution. Then, when the map of terrain is updated, it repairs its previous solution by reusing as much of their previous search efforts as possible. As a result, it can be orders of magnitude more efficient than replanning from scratch every time the world model changes, as in the case of A* algorithm. Pseudocode of D* Lite according to which we implemented this algorithm in Matlab, can be found in [7].

We use S to denote the finite set of nodes in the graph, $succ(s)$ denotes the set of successor nodes of node $s \in S$, and $pred(s)$ denotes the set of predecessor nodes of s. For any pair

Novak Zagradjanin – University of Belgrade, Faculty of Electrical Engineering, Ministry of Defence, Department for Defence Technologies, Nemanjina 15, 11000 Belgrade, Serbia (e-mail: zagradjaninnovak@gmail.com).

Aleksandar Rodic – University of Belgrade, Faculty of Electrical Engineering, Institute Mihailo Pupin, Robotics Laboratory, Volgina 15, 11060 Belgrade, Serbia (e-mail: aleksandar.rodic@pupin.rs).

of nodes s, $s' \in S$ such that $s' \in succ(s)$ we require the cost of transitioning from s to s' to be positive: $0 < c(s, s') \leq \infty$. Given such a graph and two nodes s_{start} and s_{goal}, the task of a search algorithm is to find a path from s_{start} to s_{goal}, as a sequence of nodes $\{s_0, s_1,\ldots, s_\kappa\}$ such that $s_0 = s_{start}$, $s_\kappa = s_{goal}$ and for every $1 \leq i \leq k$, s, $s_i \in succ(s_{i-1})$. This path defines a sequence of valid transitions between nodes in the graph, and if the graph accurately models the original problem, a robot can execute the actions corresponding to these transitions to solve the problem. The cost of the path is the sum of the costs of the corresponding transitions. For any pair of nodes s, $s' \in S$ we let $c^*(s, s')$ denote the cost of a least-cost path from s to s'. For $s = s'$ we define $c^*(s, s') = 0$.

The goal of shortest path search algorithms such as A* search is to find a path from s_{start} to s_{goal} whose cost is minimal, i.e. equal to $c^*(s_{start}, s_{goal})$. Suppose for every node $s \in S$ we knew the cost of a least-cost path from s_{start} to s, that is, $c^*(s_{start}, s)$. We use $g(s)$ to denote this cost. Then a least-cost path from s_{start} to s_{goal} can be re-constructed in a backward fashion as follows: start at s_{goal}, and at any node s_i pick a node $s_{i-1} = \arg\min_{s' \in pred(si)}(g(s') + c(s', s_i))$ until $s_{i-1} = s_{start}$.

In particular, A* maintains g-values for each node it has visited so far, where $g(s)$ is always the cost of the best path found so far from s_{start} to s. If no path to s has been found yet then $g(s)$ is assumed to be ∞ (this includes the nodes that have not yet been visited by the search). A* starts by setting $g(s_{start})$ to 0 and processing (hereinafter this process we call as *node expansion*) this node first. The expansion of node s involves checking if a path to any successor node s' of s can be improved by going through node s, and if so then setting the g-value of s' to the cost of the new path found and making it a candidate for future expansion. This way, s' will also be selected for expansion at some point and the cost of the new path will be propagated to its children.

A* maintains four functions from nodes to real numbers:
- $g(s)$ is the the minimum cost of moving from the start node to s found so far;
- $h(s)$ (heuristic value) estimates the minimum cost of moving from s to s_{goal} of a search;
- $f(s) = g(s) + h(s)$ or *key* value is an estimate of the minimum cost of moving from the start node via s to the goal node;
- The parent pointer *parent*(s) points to one of the predecessor nodes s' of s from which is derived $g(s)$ and s' is called the parent of s. The parent pointers are used to extract the path after the search terminates.

A* also maintains a priority queue, *OPEN*, of nodes which it plans to expand. In order to explain the purpose of forming priority queue, let us first introduce a notion of *local inconsistency*. A node is called locally inconsistent every time its g-value is decreased and until the next time the node is expanded. Suppose that node s is the best predecessor for some node s': that is, $g(s') = \min_{s'' \in pred(s')}(g(s'')) + c(s'', s')) = g(s) + c(s, s')$. Then, if $g(s)$ decreases we get $g(s') > \min_{s'' \in pred(s')}(g(s'')) + c(s'', s'))$. In other words, the decrease in $g(s)$ introduces a local inconsistency between the g-value of s and the g-values of its successors. Whenever s is expanded, on the other hand, the inconsistency of s is corrected by re-evaluating the g-values of the successors of s. This in turn makes the successors of s locally inconsistent. In this way the local inconsistency is propagated to the children of s via a series of expansions. Given this definition of local inconsistency it is clear that the *OPEN* list consists of exactly locally inconsistent nodes. The *OPEN* queue is sorted by $f(s)$, so that A* always expands next the node which appears to be on the shortest path from start to goal. A* initializes the *OPEN* list with the start node, s_{start}. Each time it expands a node s, it removes s from *OPEN*. It then updates the g-values of all of s's neighbors; if it decreases $g(s')$, it inserts s' into *OPEN*.

```
Main()
01. for all $s \in S$
02.    $g(s) = \infty$;
03. $g(s_{start}) = 0$;
04. OPEN = $\emptyset$;
05. insert $s_{start}$ into OPEN with value $f(s_{start}) = (g(s_{start}) + h(s_{start}))$;
06. ComputeShortestPath();
ComputeShortestPath()
07. while ($s_{goal}$ is not expanded)
08.    remove s with the smallest value f(s) from OPEN;
09.    for all $s' \in succ(s)$
10.       if ($g(s') > g(s) + c(s, s')$)
11.          $g(s') = g(s) + c(s, s')$;
12.          parent $(s') = s$;
13.          insert s' into OPEN with new value $f(s') = (g(s') + h(s'))$;
```

Fig. 2. A* algorithm (simplified forwards version)

The challenge for shortest path search algorithms is to minimize the amount of processing. The A* algorithm whose simplified pseudocode is presented in Figure 2 expands all nodes from *OPEN* while s_{goal} is not expanded. Using h-values focuses the search on the nodes through which the whole path from s_{start} to s_{goal} looks promising. It can be much more efficient than expanding all nodes whose g-values are smaller than or equal to $g(s_{goal})$, which is required by Dijkstra's algorithm to guarantee that the solution it finds is optimal.

The above approaches work well for planning an initial path through a known graph or planning space. However, when operating in real world scenarios, robots typically do not have perfect information. Rather, they may be equipped with incomplete or inaccurate planning graphs. In such cases, any path generated using the robot's initial graph may turn out to be invalid as it receives updated information. For example, the robot may be equipped with an onboard sensor that provides updated environment information as the robot moves. It is thus important that the robot is able to update its graph and replan new paths when new information arrives. One approach for performing this replanning is simply to replan from scratch: given the updated graph, a new optimal path can be planned from the robot position to the goal using A*, exactly as described above. However, replanning from scratch every time the graph changes can be very computationally expensive.

D* Lite is extension of A* able to cope with changes to the graph used for planning. D* Lite initially constructs an optimal solution path from the initial node to the goal node in exactly the same manner as backwards A* (i.e. search is performed from s_{goal} to s_{start}). D* Lite maintains for each node two values estimating path cost to the goal, which it uses to propagate path costs between neighboring nodes in the space.

Each node has a base path cost estimate *g*, along with another estimate called *rhs* that represents the path cost estimate derived from looking at the *g* values of its neighbors: $rhs(s) = \min_{s\rightarrow t\text{ exists}} (c(s, t) + g(t))$ or zero if *s* is the goal node. In implementation, each node maintains a pointer to the node from which it derives its *rhs* value.

When the algorithm has updated the pointers, the formula for *rhs* indicates that the robot should follow the pointer from its current node to pursue an optimal path to the goal. A node is called *consistent* if its *g* and *rhs* values are equal. It is called *overconsistent* if $g > rhs$ and *underconsistent* if $g < rhs$. Each non-goal node starts with its *g* value equal to ∞. When the *g* values of neighboring nodes are lowered (to a finite path cost estimate), the *rhs* value of the node is lowered, making it overconsistent. If at some point the *g* values of neighboring nodes are raised (an increase in the path cost estimate from those nodes), the node may have its *rhs* value raised, which may make it underconsistent. Inconsistent nodes eventually trigger updates in their neighbors, hence overconsistent nodes propagate path cost reductions through a region, while underconsistent nodes propagate path cost enlargements through a region.

Like A*, D* Lite uses a heuristic to focus its search and to order its cost updates efficiently. D* Lite also maintains an *OPEN* list of inconsistent nodes to be expanded in the current search iteration. At each step it expands the node on the *OPEN* list with the minimum *key* value. The priority, or *key* value, of a node *s* in the queue is: $key(s) = [k_1(s), k_2(s)] = [\min(g(s),rhs(s)) + h(s_{start},s),\min(g(s),rhs(s))]$. A lexicographic ordering is used on the priorities, so that priority $key(s)$ is less than or equal to priority $key(s')$, denoted $key(s) \leq key(s')$, if $k_1(s) < k_1(s')$ or both $k_1(s) = k_1(s')$ and $k_2(s) \leq k_2(s')$.

When changes to the planning graph are made (i.e. the cost of some edge is altered), the nodes whose paths to the goal are immediately affected by these changes have their path costs updated and are placed on the planning queue (*OPEN* list) to propagate the effects of these changes to the rest of the node space. In this way, only the affected portion of the node space is processed when changes occur. Furthermore, D* Lite uses a heuristic to further limit the nodes processed to only those nodes whose change in path cost could have a bearing on the path cost of the initial node. As a result, it can be up to two orders of magnitude more efficient than planning from scratch using A*. Finally, it is important to point out that D* Lite can be pre-configured either to search for an optimal solution ($\varepsilon = 1$) or to search for a solution bounded by a fixed suboptimality factor ($\varepsilon > 1$).

IV. SIMULATED SCENARIO

Scenario discussed in this paper implies that there occurred a complex accident situation in partly known urban environment. At the time T_0 we do not have sufficient information about the accident. It is only known that there have occurred accident classes 1 and 2 over a wide area, with non-uniform weight distribution of both classes.

Since we have limited resources, there is no possibility to react appropriately to the entire area where the accident occurred. Consequently, the mission goal is to explore situation in detail, to analyze accident situation, to determine the weight distribution of accident classes, to make decision about the most critical points and to plan appropriate path of robot scout and robot manipulators.

For the purpose of the mission planning, at the level of the cloud the map of environment is discretized into cells of appropriate dimensions. Every cell is approximated with one node in graph. We model the graph as 8-connected: each node is connected to its horizontal, vertical and diagonal neighbors.

Robot scout is used primarily for exploring accident situation, but and environment near the accident. In general it is UAV. It is lighter and faster than robot manipulators. Also it has better sensory capabilities and it can to classify accidents in classes with the required spatial resolution in the area of interest, as well as to determine the weight of accident classes with the same spatial resolution.

Robot manipulators are UGVs equipped with appropriate equipment for manipulation, for instance one for fire extinguisher and other for clearing the rubble. They have sensors for detecting surroundings in order to enable obstacle avoidance and appropriate behavior related to the manipulation itself.

All three robots begin their mission at the time T_0, assuming that the robot scout is previously sent to the area of the accident (UAV). The robot scout perform exploring of the accident situation, while robot manipulators move toward the target destination assigned in relation to the weight distribution of accident by classes defined on data known in T_0.

When the robot scout completes its task, updating of weight distribution of the accident classes is done based on new information and after that updating of the target destination of robot manipulators is performed. We define this moment as T_{update}.

The cloud support multi-robot system. It is responsible for centralized integration of information about the accident situation and the environment known in the T_0 and collected from all three robots during the mission, determining and updating of the weight distribution of accident classes, setting target destinations of robot manipulators to the actual weight distribution of accident classes and path planning and replanning for robots.

V. SIMULATION RESULTS

In Figure 3 example of mission profile for stochastically generated start positions of robot manipulators, environment (obstacles and free cells) and accident (accident classes and weight) situation is presented.

There are highlighted the positions of the robots in characteristic moments T_0, T_{update} and T_{final}. The dashed lines show the paths of the robots calculated by the planner in the cloud based on the limited information available at the time T_0, along which robots start their movement in order to minimize waste of time.

At the moment T_{update} robot scout has completed its mission and on the basis of information that it collected and forwarded to the cloud additional analysis of accident situations is carried out. The weight distribution of the accident classes now is updated, after which the positions of most critical points are corrected as and appropriate target destination of robot manipulators. The dotted lines show the paths of the robots calculated by the planner at the moment T_{update}.

Final paths reconsctructed at the moment T_{final} (when robots arrived at the target destinations) are showed with solid lines. Looking at the paths between the moments T_{update} and T_{final} it can be noticed that the solid lines do not coincide fully with the dotted lines. This is because the environment in that part is the most complex. Since the map of the terrain at the level of the cloud is formed on an incremental basis in accordance with the dynamics of collecting information with sensors of the robots, when changes are detected the planner repairs its previous solutions by reusing as much of its previous search efforts as possible.

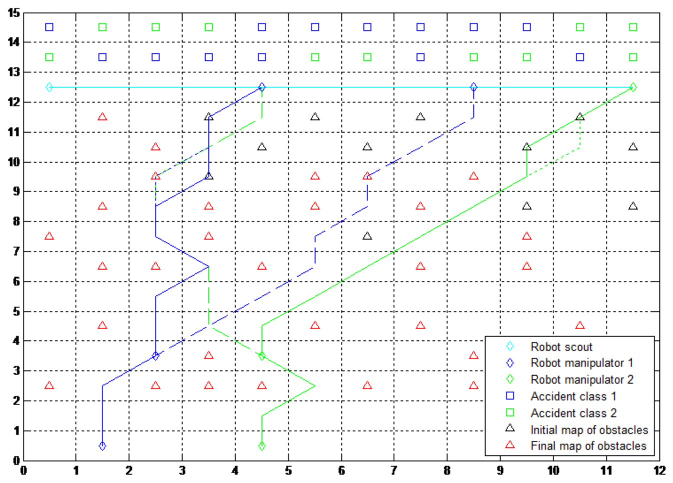

Fig. 3. Mission profile for stochastically generated start positions of robot manipulators, environment and accident situation.

In accordance with the above mentioned it is important to emphasize the importance of cloud in terms of fusion of information about the accident situation and the environment known at the T_0 and collected from all three robots during the mission. In this way, more reliable analysis of the accident situation is provided, as well as more effective mission planning and optimization of path of robots.

In Figures 4 and 5 two potential extreme situations of the environment are presented. It can be seen that, in all other equal conditions, planner makes the decision to correct the paths of robot manipulators based on the common knowledge base about the environment much earlier compared to the planning on the basis of knowledge only of the individual robots and thus significantly reduces the time of movement from the start to the target destination.

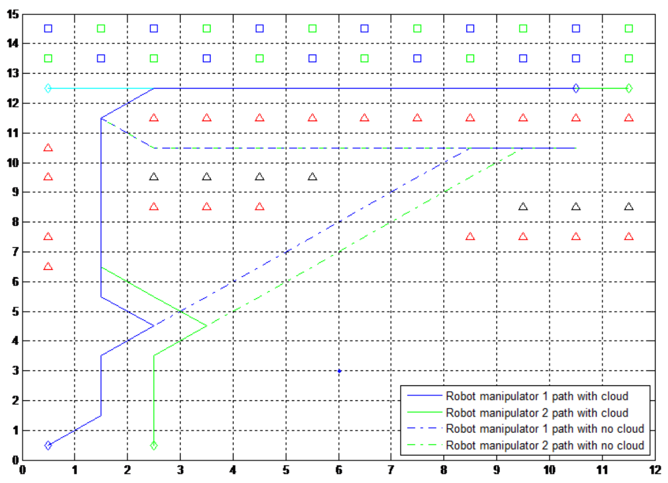

Fig. 4. Path planning of multi-robot system based on individual and common knowledge about the environment – example 1.

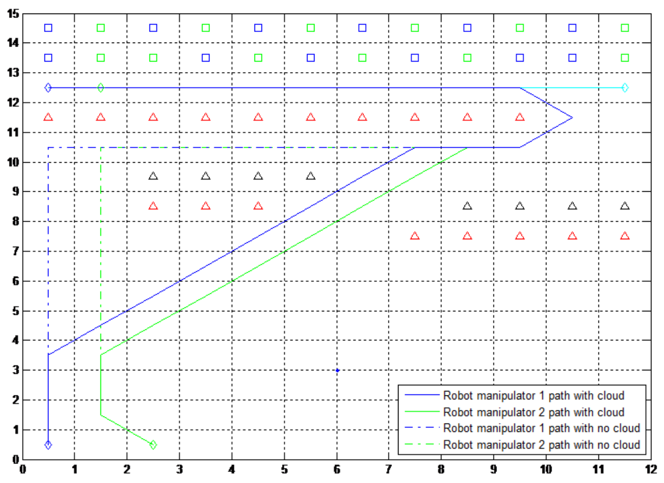

Fig. 5. Path planning of multi-robot system based on individual and common knowledge about the environment – example 2.

VI. Conclusion

In this paper we presented cloud-based distributed intelligence of robot team in complex accident situation.

Cloud platform shifts computation load from agents to the cloud and provides powerful processing capabilities to the multi-robot system. This allows the onboard systems of the robots are greatly reduced, keeping only sensors, communication, actuation and manipulation modules.

Proposed approach use path planning algorithm that can operate in environment that is unknown in advance. Besides, the algorithm is adapted for using the common knowledge base about the environment, enabling more efficient paths searching.

The main disadvantage of the system is sensitivity of the communication loss.

REFERENCES

[1] S. Choudhary, L. Carlone, C. Nieto, J. Rogers, Z. Liu, H.I. Christensen, and F. Dellaert, “Multi Robot Object-based SLAM”, Int. Symposium on Experimental Robotics (ISER), 2016.

[2] P. Benavidez, M. Muppidi, P. Rad, J. Prevost, M. Jamshidi, L. Brown, “Cloud-based realtime robotic Visual SLAM”, Proceedings of 9th Annual IEEE International Systems Conference, SysCon 2015.

[3] H. He, S. Kamburugamuve, G. Fox, W. Zhao, “Cloud based Real-time Multi-robot Collision Avoidance for Swarm Robotics”, International Journal of Grid and Distributed Computing Vol. 9, No. 6, pp.339-358, 2016.

[4] D. Hunziker, M. Gajamohan, M. Waibel and R. D’Andrea, “Rapyuta: The RoboEarth Cloud Engine”, Proceedings - IEEE International Conference on Robotics and Automation, 5/2013.

[5] M. Eich, R. Hartanto, “Fuzzy-DL Perception for Multi-Robot Systems”, Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS 2014), June 17-19, Montreal, Canada, 6/2014.

[6] I. Osunmakinde, R. Vikash, “Development of a survivable cloud multi-robot framework for heterogeneous environments”, International Journal of Advanced Robotic Systems 11, 2014.

[7] S. Koenig, M. Likhachev, “D* Lite”, Eighteenth National Conference on Artificial Intelligence (AAAI), pp. 476-483, 2002