2020 IEEE International Conference on Robotics and Automation (ICRA)
31 May - 31 August, 2020. Paris, France

SwarmMesh: A Distributed Data Structure for Cooperative Multi-Robot Applications

Nathalie Majcherczyk *Student Member, IEEE* and Carlo Pinciroli *Member, IEEE*
Robotics Engineering, Worcester Polytechnic Institute, MA, USA
Email: {nmajcher,cpinciroli}@wpi.edu

***Abstract*— We present an approach to the distributed storage of data across a swarm of mobile robots that forms a shared global memory. We assume that external storage infrastructure is absent, and that each robot is capable of devoting a quota of memory and bandwidth to distributed storage. Our approach is motivated by the insight that in many applications data is collected at the periphery of a swarm topology, but the periphery also happens to be the most dangerous location for storing data, especially in exploration missions. Our approach is designed to promote data storage in the locations in the swarm that best suit a specific feature of interest in the data, while accounting for the constantly changing topology due to individual motion. We analyze two possible features of interest: the data type and the data item position in the environment. We assess the performance of our approach in a large set of simulated experiments. The evaluation shows that our approach is capable of storing quantities of data that exceed the memory of individual robots, while maintaining near-perfect data retention in high-load conditions.**

I. Introduction

Recent work studies the integration of multi-robot systems with centralized computation platforms, such as databases or cloud computing systems [1]. This approach enables one to aggregate information in a central location and perform efficient map merging, task allocation, and global state estimation — in other words, combining data storage with computational capabilities. This approach is particularly effective in indoor environments, such as warehouses, production chains, and hospitals, in which communication with a central system can be expected to be reliable.

However, many applications are not easily amenable to this approach. Mapping in remote locations, space applications, and disaster recovery are examples in which access to a centralized infrastructure is problematic, limited, or even impossible. In these applications, rather than envisioning a multi-robot system as *part* of a larger infrastructure, it would be desirable for it to *be* the infrastructure. These applications also entail the collection of large amounts of data, whose storage might exceed the capacity of any individual robot.

As a step in this direction, we study the realization of a decentralized data structure for storing, managing, and performing computation with shared data. We make three basic assumptions:

- Every robot devotes a quota of memory and bandwidth to storing and routing data. The amount of memory can change across robots;
- The amount of data that the robots must store is larger than the memory capacity of any individual robot;
- The network topology is dynamic due to robot motion.

Given these assumptions, we study how to distribute the data across the swarm. In designing a solution, we realized that in many applications certain features of the data play an important role for mission success. For example, mission-critical data should be stored in well-connected robots—in case of a temporary disconnection this data would be as widely available as possible. Analogously, the physical location of the data might suggest that certain robots are more suitable for storage than others.

The rest of this paper is organized as follows. In Section II we discuss related work. The design of our data structure is presented in Section IV. We report the results of our performance evaluation in Section V, and conclude the paper in Section VI.

II. Related Work

In peer-to-peer networks, common implementations of data sharing involve Distributed Hash Tables (DHTs). DHTs couple a distributed key partitioning algorithm and a structured overlay network to provide a self-organized data storage service. Information is abstracted in the form of tuples which are `(key, value)` pairs. The fundamental problem is to decide how to distribute tuples between nodes for storage. The key partitioning algorithm assigns ownership of a set of keys to each node in the network. The overlay network imposes a routing structure that makes for efficient search across the nodes. Comparative surveys [2],[3] highlight the main features of these protocols. These distributed data structures provide self-organizing, scalable and addressable storage. However, node additions and removals are costly as the topology needs to be maintained through reorganization. Furthermore, they can cause local network failures. Because they form relations between nodes randomly, unstructured overlay networks such as Gnutella and BitTorrent [4] provide alternatives when the network participant turnover is high. These protocols offer robustness to node removals at the cost of increased degree of centralization or loss of guarantees when locating data.

In the above mentioned protocols, the selection of neighboring nodes in the overlay network lacks physical grounding. This means that neighbors in the network could be

978-1-7281-7395-5/20/$31.00 ©2020 IEEE

far away from each other. Since routing information over longer geographical distances increases energy consumption and latency, there has been an effort to incorporate node location into overlay networks. Three main trends exist within this body of research: (1) Geographic layout, which constructs the overlay network so that neighbors are close in the physical space [5], [6], [7], [8]; (2) Proximity routing, which considers node proximity while routing in the existing overlay network [9]; (3) Proximity neighbor selection, which weighs in proximity between neighbors when constructing the overlay network [10]. These methods add a notion of node locality. However, the network topology only changes to accommodate node additions and removals but not motion. Therefore, they fail to capture the inherent dynamicity of robotic systems.

In the context of swarm robotics, Pinciroli *et al.* proposed a distributed tuple space called *virtual stigmergy* [11] that copes with frequent topology changes. In this approach, each robot maintains a local time-stamped copy of the data which is only accessed upon read and write operations. This mechanism works well with node mobility and limited bandwidth but it leads to full data duplication. This means that the collective memory of the system is under-utilized. The SOUL file sharing protocol [12] builds on virtual stigmergy and unstructured overlay networks to enable sharing of larger-size data in the form of `(key, blob)` pairs. SOUL involves locally storing blob meta-data on each node and splitting blobs into datagrams across different nodes. This decomposition uses a bidding mechanisms that minimize the reconstruction cost at so-called processor nodes. This method addresses the problem of managing data files with a focus on how to split, distribute and recombine them. Memory usage is improved but meta-data is still fully duplicated across nodes for each of the files. Various update and bidding processes increase latency in the network.

In this paper, we take inspiration from existing methods and propose a novel approach to distributed data sharing. Our design embraces the decentralized nature of robot swarms and the constant change and volatility in the network topology that results from robot motion. We organize our data flow based on instantaneous local properties at each node so as to get a memory efficient, consensus-free approach with a low communication overhead.

III. Problem Setup and Challenges

In this section, we describe the fundamental assumptions imposed both on the multi-robot system and on the nature of the events to record in the physical environment. We proceed by describing challenges of the distributed storage problem in this context.

A. Ad-hoc Robotic Network

We consider an autonomous and decentralized system of N robots which act as both the infrastructure for storing information and sole users of this information. We define the system across the following features:

a) Communication Modalities: We assume that the robots have the ability to exchange data within a communication range C. This implies the existence of an ad-hoc network with each robot acting as a node. We further assume that robot communication is limited to gossiping, i.e., broadcasting messages to all neighbors within C. Because we also desire to route some messages from one robot to another using the point-to-point communication modality, we assume that the robots have a constant unique identifier $i \in [1, N]$ and a variable node identifier δ_i made known to their neighbors. The knowledge of i singles out a specific robot, while δ_i enables the selection of a suitable storage node for a specific tuple.

b) Finite Resources: We impose a realistic finite bandwidth on outgoing messages. We also limit the memory capacity M_i of each robot allotted for the self-organizing data management process. Variable $m_i(t)$ records the amount of memory used by robot i at a given time.

c) Dynamic Topology: The robots are moving according to a logic defined by the developer. Robot motion follows linear dynamics and has a limited speed. The number of neighbors $ngbrs_i(t)$ of a robot changes over time.

B. Inputs

We consider inputs to the data structure to stem from events which have a position $\mathbf{x} \in \mathbb{R}^d$ and happen at a time $t \in \mathbb{N}$. Such events can be, e.g., records of a physical phenomenon sampled at a particular time and place, records of an internal robot state, or records of swarm-wide state. To implement a data structure in which robots can retrieve and update tuples encoding some events, each specific tuple needs a unique identifier τ meaningful to all network nodes. In particular, for updates, tuples need to have a notion of version. For convenience, we achieve versioning by time-stamping tuples with a global time. Distributed synchronization algorithms such as vector clocks [13] can be used to implement this aspect.

IV. Methodology

A. Overall Architecture

We describe our design following the structure depicted in Figure 1. SwarmMesh provides algorithms across three levels of abstraction.

1) User-level querying: As stated in Section I, robots are at the same time the networking infrastructure and users of the data stored by the network. As a user, a robot can execute different querying commands on the data structure. These operations are meant to enable modifying and retrieving information stored globally as required by the robot behavior. This behavior is defined by the developer and independent from SwarmMesh.

2) Queried data propagation: Another layer of SwarmMesh handles the dissemination of user read and write queries throughout the data structure. Read queries are flooded across the network. This type of query requires replies from certain nodes to be routed back to the robot

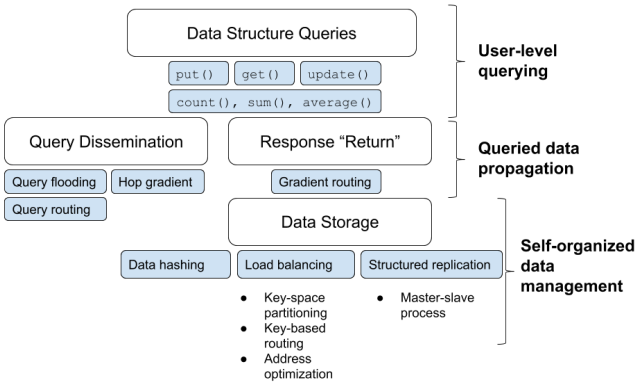


Fig. 1: Overall architecture.

emitting the query. We route write operations to a suitable node for storage in a point-to-point fashion.

3) Self-organized data management: The bottom layer determines how the tuples get distributed across nodes. It also ensures a certain degree of robustness by creating inactive replicas in other nodes in a controlled way. The main design intuitions driving the data distribution are that: (1) some events/tuples are more important than others (hierarchy in data hashing); (2) some nodes are better suited to hold more valuable tuples than others (hierarchy in key-space partitioning); (3) the hierarchy of nodes changes very often and should be updated based on local information only.

B. User-level Querying

A robot user can perform the following operations:

- `put(k,v)`: writes a tuple into the data structure. It performs an `erase(k)` to remove any potential outdated version of the tuple and a `store(k,v)` of the new tuple.
- `store(k,v)`: assigns a tuple to a particular node in the data structure.
- `erase(k)`: removes a tuple from the data structure.
- `get(k,`Δ`)`: returns all the values corresponding to keys $\in [k-\Delta; k+\Delta]$.
- `get(x,y,r)`: returns all the values for tuples located within a radius r of the point (x, y) expressed in a global reference frame. To use this feature, we need the added assumption of a global reference frame and the ability to locate events in this reference frame.

Robots can also perform in-network computation:

- `count(k,`Δ`)` or `count(x,y,r)`: returns the number of tuples with keys $\in [k-\Delta; k+\Delta]$ or located within a radius r of the point (x, y).
- `sum(k,`Δ`)` and `sum(x,y,r)`: returns the sum of values corresponding to keys $\in [k-\Delta; k+\Delta]$ or located within a radius r of the point (x, y).
- `average(k,`Δ`)` or `average(x,y,r)`: returns the result of the corresponding `count()` and `sum()` operations as a pair.
- `min(x,y,r)` or `min(k,`Δ`)`: returns the minimum value in the associated spatial range or key range.
- `max(x,y,r)` or `max(k,`Δ`)`: returns the maximum value in the associated spatial range or key range.

The return values of the `count()`, `sum()` and `average()` operations percolate across nodes. The user robot which emitted the initial query must combine the intermediate return values into the final result. Monotonic (i.e., commutative and associative) operations such as `min()` and `max()` do not require combining intermediate results.

The performance of spatial queries, i.e., operations with arguments (x, y, r), and that of queries by key, i.e., operations with arguments (k, Δ), depends heavily on the way we distribute the data in the network. Our approach is meant to be modular and we present two possible data hashing functions in Section IV-D.1.

C. Queried Data Propagation

1) Read-operation flooding: Queries that aim to retrieve data from the data structure are flooded to all nodes. Each robot emitting a read query computes the query's unique identifier by concatenating the value of its query counter and its robot unique identifier.

2) Hop gradient: While flooding the network with a read query, we opportunistically create a gradient to the source of the query. Upon reception, every robot increments a hop counter included in the query message and broadcasts it further along. For each received request, the robot stores the query unique identifier and hop count in a circular buffer.

3) Reply gradient routing: The hop gradient gives us a convenient way to route replies back to the source node by forwarding replies from nodes with a higher or equal hop count. For this, we rely on the assumption that the motion of the robots preserve a gradient path to the source for long enough, which is a realistic assumption in most settings when comparing motion speed and information propagation speed.

4) Write-operation routing: When a robot writes the result of some local information processing to the data structure, the tuple may be routed to a different robot for storage based on its key. This algorithm is described in Section IV-D.3.

D. Self-organizing Data Management

1) Data Hashing: When writing a tuple using `put(k,v)`, the robot must compute the key `k`. In our protocol, a key should be in the format $k_\tau = (\tau, \rho_\tau)$ where τ is a tuple unique identifier and ρ_τ is a value that maps to one or multiple nodes which can store the tuple.

The robot assigns τ by concatenating its robot unique identifier and the count of tuples it has written into the data structure. Each field has a set number of digits so that every τ is unique. As stated previously, our design considers that events vary in importance and we use this property to distribute them across nodes.

Read queries described in Section IV-B can either use `k` $= \rho_\tau$ or `k` $= (\tau, \rho_\tau)$ for tuple addressing. Queries for

[1]The key argument `k` has an uniquely identifying part τ and a content-dependent part (or hash) ρ_τ.

ρ_τ can yield multiple tuples while queries in (τ, ρ_τ) relate to a specific tuple.

The robot computes ρ_τ using a function mapping a characteristic of the event to its relative importance. We select the function such that the higher ρ_τ, the more valuable the piece of information. We propose two hashing functions:

- Category-based: A robot can register different types of events. For example, it can mean that the robot has several different on-board sensors and determines the event type by the triggered sensor. We use a ranking function $R_T(s_\tau)$ that assigns higher values to event types that we consider most important: $h_C : type_\tau \mapsto \rho_\tau = R_T(type_\tau)$.
- Spatial: We decide that in a global reference, tuples further away from the origin are the most desirable because they are difficult to discover by robots. This idea can be generalized to specific areas in any reference frame: $h_{SP} : (x_\tau, y_\tau) \mapsto \rho_\tau = \sqrt{x_\tau^2 + y_\tau^2}$.

2) Key Partitioning: Similarly to other distributed data structures such as DHTs, nodes partition the key space to decide which one of them needs to hold tuples corresponding to specific keys.

As stated in Section IV-A, we use the idea that some nodes are superior than others. Our intuition is that a robot with more neighbors $ngbrs_i$ is less likely to get disconnected from the swarm and is better positioned to dispatch tuples upon query. A second insight is that the more free data memory $m_i(t)$ a robot has, the less likely it is to overflow its memory and discard information. We also desire to have instantaneous self-organized partitioning completely based on local information. Therefore, we chose to make nodes assign themselves a node identifier δ_i as follows:

$$\delta_i(t) = \begin{cases} m_i(t) \cdot ngbrs_i(t) & \text{if } ngbrs_i(t) > 0 \\ 1 & \text{otherwise} \end{cases}$$

A node with node identifier δ_i can hold a tuple with key (τ, ρ_τ) if $\delta_i(t) > \rho_\tau$. We refer to this condition as (H) in the rest of this text. The free memory variable is in number of tuples. In order to store tuples in the data structure, we should match the frequency distributions of data hashes and node identifiers, i.e., there should be nodes with unique identifiers at least high enough to hold the hashed tuples. This has implication on the design of the hashing functions. They should map to values smaller than $\max_i(M_i) \cdot (N-1)$ and spread the data across likely node identifiers.

3) Key-based Routing: If a robot holds one or more tuples not satisfying (H), it places them in a routing queue. It then tries to send them starting with the highest ρ_τ to a robot with a high enough node identifier. If there are candidates satisfying (H) to receive the tuple, the sender picks one at random. If none of the neighbors satisfy the condition, the robot sends it to the neighbor of highest δ_i. We impose a limit on the memory capacity M_i and divide it into routing and storage capacities. In case of overloads on M_i, the robot discards the least important tuple, i.e., with lowest ρ_τ.

4) Address Optimization: When a robot has an empty routing queue and it stores a tuple with ρ_τ closer to the node identifier of a candidate neighbor, we let the robot evict the tuple to the corresponding neighbor. This is an optimization to ensure efficient access to a tuple by key. We further noticed that requiring at least a half full storage memory helps balancing the load between nodes.

5) Structured Replication: To ensure robustness to node failures, we make copies in neighboring nodes using a master-slave approach. The master is the robot holding the original tuple. The master picks a slave to hold an inactive copy of the tuple. Robots do not return inactive tuple copies upon queries; this ensures consistency. Master and slave exchange a heartbeat signal. If the master fails to receive the heartbeat signal within a time-out duration, it picks another slave. The master can also send a kill signal to cancel the inactive copy. The master cancels the copy if the slave gets outside of a safe radius of communication ($\ll C$) or if it decides to route the active tuple to another robot. If a slave fails to receive the heartbeat within the time-out duration from the master, it activates the tuple copy.

V. Evaluation

A. Metrics and Parameters

We evaluated different aspects of our approach such as scalability, memory-related performance, and routing protocol efficiency.

To study *scalability*, we performed our simulated experiments with different numbers of robot inside an arena sized to imposed different robot densities (see Table I). These densities imply that the ad-hoc network stays often connected even with diffuse robot motion. This enables us to study a system facing intermittent disconnections.

In hash tables, the *load factor* is the number of data items over the number of memory slots (buckets). This parameter indicates the load of the data structure and is typically used to decide when to partition of the memory into an increased number of buckets. For our distributed and self-organized approach, we define the load factor as l_f = number of events$/(N \cdot S)$, where S is the storage capacity. The memory capacity M includes both storage and routing capacities.

To understand the *performance* of the key-based routing algorithm, we track the number of hops and time steps for a tuple to be routed to a suitable node for storage and upon query. We implemented messaging with a queue and we imposed a limit on the bandwidth for outgoing messages. We only send one tuple at a time.

To study *availability*, we considered the fraction of tuples received over the expected tuples for a `get()` query. We checked for consistency by confirming that active copies of tuples were all unique.

B. Simulated Experiments

We tested our system using the ARGoS multi-robot simulator [14]. We ran simulations with and without robot motion. We picked a simple diffusion motion with a maximum

TABLE I: Simulation parameters

Parameter	Value
Number of Robots N	$\{10, 50, 100\}$ robots
Communication range C	2 m
Memory capacity M_i	20 tuples $\forall$ i
Storage capacity S	10 tuples
Routing capacity R	10 tuples
Time step	0.1 s
Bandwidth	5.7 kB/s
Robot density	$\{0.6, 1\}$ robot/m^2
Robot speed	$\{0, 5\}$ cm/s
Load factor	$\{0.6, 0.7, 0.8, 0.9, 1\}$
Event sensing range	1 m
Event types	12
Events generation rate	5 events/s
Query generation rate	1 query/s

TABLE II: Tuple retention for $N = 50$ across load factors.

Load Factor			.6	.7	.8	.9	1
category hashing	static topology	min	1	1	1	.996	.95
		mean	1	1	1	.999	.983
		max	1	1	1	1	1
	dynamic topology	min	1	1	1	.996	.986
		mean	1	1	1	.999	.992
		max	1	1	1	1	1
spatial hashing	static topology	min	.993	.932	.973	.909	.758
		mean	.999	.99	.987	.948	.899
		max	1	1	1	.98	.954
	dynamic topology	min	.997	.994	.985	.984	.94
		mean	.999	.999	.996	.993	.985
		max	1	1	1	1	.998

forward speed of 5 cm/s. For the purpose of testing all available features, robots are equipped with a range and bearing sensor ($C = 2$ m), a GPS and a sensor detecting colored spheres. We disabled line-of-sight obstructions. To materialize events, we put colored spheres in the environment with each color representing a category of event. Events were generated in time according to a Poisson distribution and placed in space according to a uniform distribution.

1) Memory-Related Performance: In our simulations, we allocate limited memory and bandwidth to the data sharing process. Upon receiving tuples that it can store, a robot progressively fills its storage memory. The cap on bandwidth combined with the decision to route one tuple per time step results in some tuples being temporarily placed in a routing memory. The goal is to keep the storage memory under a value S and the routing memory under a value R. However, we allow either memory to temporarily cross that threshold provided that the combined memory usage stays under the memory capacity M_i. Any memory overflow leads to robots discarding tuples of lowest rank. To assess the ability to retain large amounts of information in the data structure, we generate different numbers of inputs corresponding to load factors between 0.6 and 1.0. We repeated simulations with and without robot motion and using either the h_C or h_{SP} hash function. Tables II and III show that the fraction of retained tuples is almost always equal to 1, even with high load factors. This indicates that the collective memory is properly utilized with robots sharing the data load. In comparison, an approach that uses full duplication and the same individual memory constraints would retain N times less tuples (excluding the routing memory).

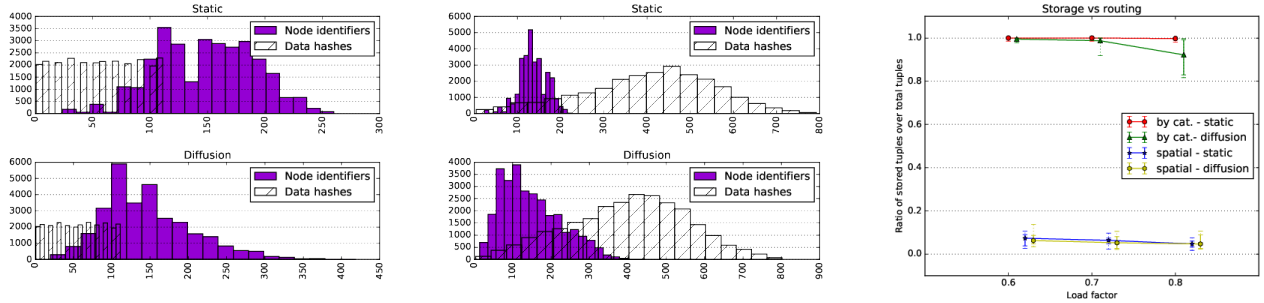
In order to evaluate key partitioning, we show histograms of node identifiers and data hashes across all simulations with

TABLE III: Tuple retention for $N = 100$ across load factors.

Load Factor			.6	.7	.8	.9
category hashing	static topology	min	1	1	1	.997
		mean	1	1	1	.999
		max	1	1	1	1
	dynamic topology	min	1	1	.991	.977
		mean	1	1	.999	.996
		max	1	1	1	1
spatial hashing	static topology	min	.955	.912	.841	.72
		mean	.982	.97	.927	.866
		max	1	.999	.989	.977
	dynamic topology	min	.995	.993	.994	.972
		mean	.999	.999	.998	.997
		max	1	1	1	1

100 robots in Figure 2. In Figure 2a, we use the category-based hash function h_C with a mapping of 12 types of events to values in $\{1, 11, 21, \ldots, 121\}$. We generated the types of events uniformly in the simulations which naturally leads to the white dashed bar graphs in Figure 2a. Node identifiers are the product of the current node degree in the communication graph and the node's remaining storage memory. Therefore, the node identifier distribution depends on a combination of communication graph topology and load allocation. Both situations represented for the category-based have the node identifier distribution to the right of the data hashes. In the upper graph, robots are static and their spatial coordinates are sampled in uniform distributions. In the bottom graph, robots diffuse in an arena sized to impose certain robot densities (see Table I). In Figure 2b, we use the spatial hash function h_{SP} and we show a situation where events are uniformly generated up to 8 m from the origin of the global reference frame. The function h_{SP} maps the distance in cm to ρ_τ which yields the Gaussian distribution represented by the white bar graphs. In both the static and moving case, the node identifier distribution is to the left of the data keys distribution. This means that, given the key partitioning condition (H), suitable nodes for storing tuples are scarce or non-existent. As evidenced by Table III, we were still able to retain tuples with high load factors even in this situation. The reason is that robots shift the load from their storage memory to their routing memory. This is apparent in Figure 2c in which the number of tuples in storage memory normalized by the total number of tuples shows the difference between the use of h_C and h_{SP}. With the latter, most tuples remain in routing and bounce between robots with more free memory. This is not a desirable solution as it increases the communication overhead. However, it demonstrates a certain tolerance and seamless adaptation to inappropriate node partitioning. In practice, with a guess of the environment scale and typical distances, h_{SP} can be scaled so as to provide mapping to a range matching the node identifiers.

2) Routing Performance: In our approach, routing mechanisms depend on the type of query. A write operation `put()` triggers a flooded `erase()` operation and a `store()` propagated through key-based routing (see Section IV-D.3). The timing for storing a tuple depends on how difficult it is to reach a suitable node given the tuple key. Figure 3 shows the median routing time with 10 and 100 robots for

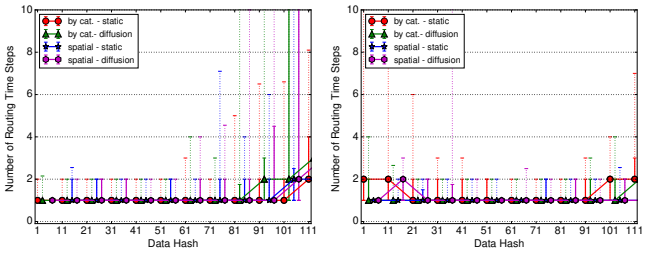


(a) Node identifiers δ_i distribution in category-based hashing.

(b) Node identifiers δ_i distribution in spatial hashing.

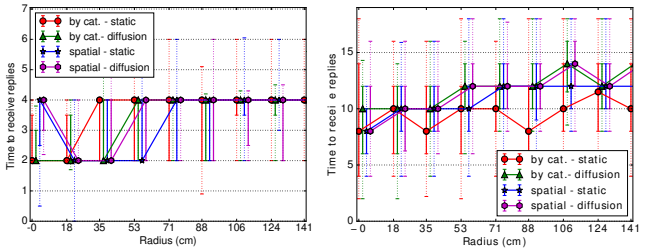
(c) Ratio of stored to routed tuples.

Fig. 2: Performance of key partitioning with $N = 100$ robots.



(a) $N = 10$.

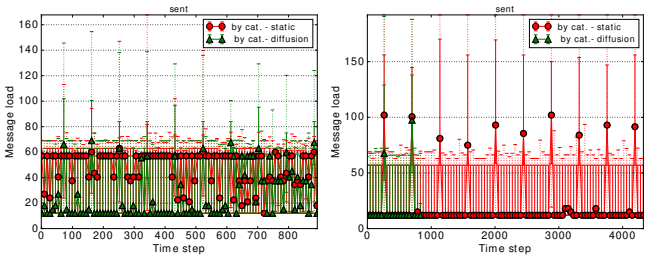
(b) $N = 100$.



(c) $N = 10$.

(d) $N = 100$.

Fig. 3: Number of time steps (100 ms each) for the completion of `store(k, v)` operations (a), (b) and `get(x, y, r)` queries (c), (d).



(a) $N = 10$

(b) $N = 50$

Fig. 4: Bandwidth usage in bytes over time. Time is measured in time steps (100 ms).

the `store()` operation. The time tends to increase with the key indicating that lower range keys find a match faster.

Read operations of any type generate a message flooded to all robots. Replies come back through gradient routing (see Section IV-C.3). Figure 3 reports the median duration between a robot emitting a spatial `get()` query and receiving the last reply to the query. This duration tends to increase with the network size and with the radius of the query.

3) Message Load: The outgoing bandwidth was set to 570 bytes per time step for each robot, with a time step covering 100 ms. However, this allowance was rarely needed. Figure 4 shows the median bandwidth usage across simulations over time, which remains well below the limit.

VI. Conclusions

We presented SwarmMesh, a distributed data structure for low-memory, low-bandwidth, highly mobile multi-robot systems. The main insight in the design of SwarmMesh is that the features that characterize the data items are key in deciding where to store the items. Our design is modular, in that the logic that governs data distribution can be chosen by the user. We proposed two methods of distributing storage responsibility, one based on the category of the data items (for applications in which certain data types are more important than others), and another based on the position of each data item. Our evaluation shows that SwarmMesh displays near-perfect levels of data retention even for extremely high load factors, adaptively switching from static storage in the robot memory when load factors are low, to dynamic storage through frequent data exchange when load factors are severely high. Future work involves applying our work to scenarios such as task allocation in dynamic environments and collaborative mapping. For the latter, we will investigate how to incorporate the size of the data items as a factor in the data redistribution logic. Finally, our approach lends itself to privacy and security considerations, whereby the decision on where to store certain data depends on the reputation of the robots. [15]

Acknowledgments

This work was funded by a grant from mRobot Technology Co, Shanghai, China.

References

[1] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg, "A survey of research on cloud robotics and automation," *IEEE Transactions on automation science and engineering*, vol. 12, no. 2, pp. 398–409, 2015.

[2] G. Urdaneta, G. Pierre, and M. V. Steen, "A survey of DHT security techniques," *ACM Computing Surveys*, vol. 43, no. 2, pp. 1–49, 2011. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1883612.1883615

[3] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys & Tutorials*, vol. 7, no. 2, pp. 72–93, 2005.

[4] B. Cohen, "Incentives build robustness in bittorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6, 2003, pp. 68–72.

[5] J. P. Ahulló, P. G. López, M. S. Artigas, and A. F. Gómez Skarmeta, "Supporting geographical queries onto DHTs," *Proceedings - Conference on Local Computer Networks, LCN*, pp. 435–442, 2008.

[6] W. Wu, Y. Chen, X. Zhang, X. Shi, L. Cong, B. Deng, and X. Li, "Ldht: Locality-aware distributed hash tables," in *2008 International Conference on Information Networking*. IEEE, 2008, pp. 1–5.

[7] A. Pethalakshmi and C. Jeyabharathi, "Geo-chord: Geographical location based chord protocol in grid computing," *International Journal of Computer Applications*, vol. 94, no. 3, 2014.

[8] S. Matsuura, K. Fujikawa, and H. Sunahara, "Mill: A geographical location oriented overlay network managing data of ubiquitous sensors," *IEICE transactions on communications*, vol. 90, no. 10, pp. 2720–2728, 2007.

[9] F. Araujo, L. Rodrigues, J. Kaiser, C. Liu, and C. Mitidieri, "Chr: a distributed hash table for wireless ad hoc networks," in *25th IEEE international conference on distributed computing systems workshops*. IEEE, 2005, pp. 407–413.

[10] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron, "Exploiting network proximity in distributed hash tables," in *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, 2002, pp. 52–55.

[11] C. Pinciroli, A. Lee-Brown, and G. Beltrame, "A tuple space for data sharing in robot swarms," in *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*. ICST (Institute for Computer Sciences, Social-Informatics and , 2016, pp. 287–294.

[12] V. S. Varadharajan, D. St-Onge, B. Adams, and G. Beltrame, "Soul: data sharing for robot swarms," *Autonomous Robots*, pp. 1–18, 2019.

[13] P. S. Almeida, C. Baquero, and V. Fonte, "Interval tree clocks," in *International Conference On Principles Of Distributed Systems*. Springer, 2008, pp. 259–274.

[14] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo, "ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems," *Swarm Intelligence*, vol. 6, no. 4, pp. 271–295, 2012.

[15] G. Frazier, Q. Duong, M. P. Wellman, and E. Petersen, "Incentivizing responsible networking via introduction-based routing," in *International Conference on Trust and Trustworthy Computing*. Springer, 2011, pp. 277–293.