Map API - Scalable Decentralized Map Building for Robots

Titus Cieslewski, Simon Lynen, Marcin Dymczyk, Stéphane Magnenat and Roland Siegwart
Autonomous Systems Lab, ETH Zurich

***Abstract*— Large scale, long-term, distributed mapping is a core challenge to modern field robotics. Using the sensory output of multiple robots and fusing it in an efficient way enables the creation of globally accurate and consistent metric maps. To combine data from multiple agents into a *global* map, most existing approaches use a central entity that collects and manages the information from all agents. Often, the raw sensor data of one robot needs to be made available to processing algorithms on other agents due to the lack of computational resources on that robot. Unfortunately, network latency and low bandwidth in the field limit the generality of such an approach and make multi-robot map building a tedious task. In this paper, we present a distributed and decentralized back-end for concurrent and consistent robotic mapping. We propose a set of novel approaches that reduce the bandwidth usage and increase the effectiveness of inter-robot communication for distributed mapping. Instead of locking access to the map during operations, we define a version control system which allows concurrent and consistent access to the map data. Updates to the map are then shared asynchronously with agents which previously registered notifications. A technique for data lookup is provided by state-of-the-art algorithms from distributed computing. We validate our approach on real-world datasets and demonstrate the effectiveness of the proposed algorithms.**

I. Introduction

In collaborative robotics tasks, and in particular in collaborative Simultaneous Localization and Mapping (SLAM), data sharing between agents is a central issue. Imagine a search and rescue scenario where we demand continuous, real-time and in-field mapping. Because the chance of finding survivors decreases over time, speed of mapping is of paramount importance and obviously, using multiple robots can heavily parallelize the mapping process. To avoid mapping redundant areas and to make the most out of the information captured by the agents, it is essential to have efficient, low overhead communication between them. The ultimate goal of distributed robotic mapping is to build a *global* map that contains the joint estimate of both the state of the environment and the robots.

A wide body of work in the area of multi-robot SLAM focuses on distributed mapping and information sharing from an estimation standpoint. Approaches such as the work of Fox [1], Martinelli [2], Trawny [3] or more recently Carillo-Arce [4] to name a few, demonstrated impressive results on how robots can estimate their position in a common frame of reference using a minimum amount of data exchange.

C2TAM [5] and CVSLAM [6] use an approach more independent of the estimation algorithm by building a local map of the environment on board the robots, then sending a subset of it to a central server where it gets fused into a global map in which all robots can co-localize.

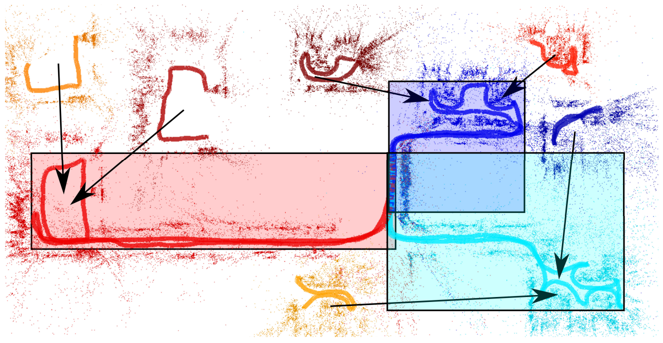
Fig. 1: Map API provides a version control system for mapping data, a novel concept of update notification, and spatial queries. It thus allows large scale collaborative map-building for robots both in the cloud and in the field.

In contrast, RoboEarth [7], Rapyuta [8] and *Map API*, which we propose in this paper, offer solutions to the question of distributing data and computational work between agents which we see as a foundation for any distributed estimation algorithm. Instead of focusing on estimator consistency we concentrate on the consistency, integrity and availability of mapping *data* within the robotic swarm.

There is however a major challenge in the task of *collaboratively* building a metric map of the environment in a distributed system: How can we fuse data asynchronously into a *global, but distributed* map without having a central authority that performs or approves changes to the common mapping data? Despite their decentralized architecture, frameworks such as C2TAM [5] and CVSLAM [6] fall back to a central entity for map assembly.

However, there are situations in which a map is best built and maintained *decentrally*: Cunningham [9] as well as Vidal-Calleja et al [10] follow a decentralized approach in which data is owned by the agents that created it, assuming that these agents are available during the scope of the distributed mapping application. Both frameworks however do not allow the integration of collected data into a common, global map during the mapping-operation: The collaborating entities in DDF-SAM [9] each have a well-maintained local map and only share summarized maps with neighboring agents. In the work of Vidal-Calleja et al [10], only a high-level global map is shared and detailed maps can be requested from individual agents. Instead, and this is the main contribution of Map API, our framework allows agents to *collaboratively* build a *single global map* without requiring a central authority.

Furthermore data is not tied to any owner authority; in contrast, agents using Map API can request and release data co-ownership dynamically. Data persistence in Map API is thus no longer limited by the availability of a single owner, but rather by its perceived relevance among peers. Data in the swarm can be localized using the concept of Distributed Hash Tables (DHTs) [11].

In such a distributed system, modifications to data need to be synchronized in a more complex way than is necessary in a centralized system. Commonly, the shared map data is locked for the time of a given operation to guarantee consistency by serializing data-access by multiple peers [6]. Naturally, such systems rapidly degrade as operations take a longer time or the number of peers increases. We therefore propose a *map version control* system which allows access, modification and additions to the map data by multiple agents *concurrently*. In particular, we introduce the concept of views, which allow access to the mapping data in the *state it has been at a given time* without locking the map data for the entire time of an operation. Views can be filled selectively with data which is looked up using DHTs. After changes are made, updates can be *committed* to the database.

Allowing concurrent access and modifications to the map data means that conflicts can arise which need to be dynamically resolved, typically using distributed consensus. The simplest, yet limited approach to consensus is to average conflicting values [12, 13]. Often however averaging multiple solutions is not possible, for instance when a conflict exists between discrete values such as update and deletion. In Map API, we detect conflicts at commit time and let the user implement a resolution algorithm of her choice to resolve them. Doing this in a decentralized system is difficult and has led us to implement a first version which is not robust to unannounced peer departure. However, previous work in Distributed Computing [14, 15] shows that this problem, presumably very relevant for field robotics, can be solved.

Once consensus has been found, other agents need to be notified about changes in the data, where *pushing* all changes to all agents is infeasible. We therefore propose a system in which agents can *register* a distributed notification request, dubbed "Trigger", about changes to the data in which they are interested. The database literature knows triggers since a long time [16] as a means of running a block of code when a certain condition is met such as a change to data.

In summary we contribute a back-end for distributed, decentralized mapping, which:

- Provides global data lookup without a central index.
- Allows agents to concurrently contribute to a common global map through dynamic ownership of data, distributed consensus and a history on the data.
- Distributes changes asynchronously based on notifications that peers can register.

The proposed algorithms allow us to scale the system to large scale mapping problems while keeping communication between peers at a minimum.

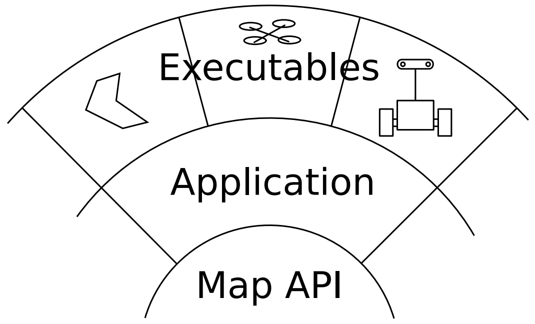

Fig. 2: The three-layer architecture of Map API projects: At the bottom, Map API provides the generic map data management discussed in this paper. Different projects using Map API can define libraries on top of it, in the application layer. Finally, the agents involved in a mapping project use different executables on top of the application layer.

II. Methodology

A. System architecture

The Map API back-end forms the base of a three-layer architecture (see Fig. 2): At the bottom, the system provides the general distributed functionality described in this paper. In an intermediate layer, which we call the application layer, a set of mapping methods provide several data structures and libraries based on Map API. Finally, the top layer is comprised of executables for the different roles in a mapping application. The intermediate and top layer can be fully customized by the users. In particular, the application layer defines tables, which is how data types are organized in Map API. As a proof of concept we have implemented a visual-inertial mapping framework on top of Map API [17].

B. Data distribution

The optimal granularity in which to share data addition and updates between peers lies somewhere between the extremes of notifying all peers about all changes or individual peers on a single item basis. Finding the right trade-off is crucial: Sharing a monolithic map with everyone is infeasible from a network bandwidth point. A too fine granularity, for instance by packaging every item individually, leads to the situation in which the memory use is dominated by metadata. We therefore suggest to split data from each table into application-defined sub-portions, so called Map API data *chunks*. The size of chunks should be chosen in a way that in a typical use case, only a handful of chunks need to be loaded. Each peer that is interested in data from some chunk must *participate* in it. Participating in a chunk implies that the peer must replicate the current state of the chunk and provide its data to new peers that are interested in the chunk. On the other hand, it implies that the peer gets informed about updates in the chunk immediately and incrementally. Thus, the latest state of the chunk is updated in the network without polling and therefore implies a small networking footprint.

C. Concurrent data access

A main goal of Map API is the abstraction of the complexity of distributed concurrency. The primary challenge in this area is to provide consistent and concurrent data access. At the core, this is about handling the situation in which several peers concurrently try to modify data while others try to read it. For such problems mutual exclusion locks are the solution of choice for single-process-multiple-threads applications. These locks ensure that only one thread at a time can execute critical code sections and thus ensure consistency by protecting data from corruption. The naive approach to handling concurrency in a distributed system like Map API could thus consist of acquiring a *global* lock on the map before data is modified or read. However, mutual exclusion on the map is not practical for collaborative mapping for several reasons: Many operations in large scale mapping, such as map optimization by bundle-adjustment [18], can last several minutes. If the system uses mutual exclusion, even with fine lock granularities, other peers are excluded from data access while a peer is working on the data. This includes adding completely new data to the map or reading it for visualization purposes, both being actions which commonly require access to the map for a short time span only.

Instead, a much better approach than mutual exclusion is provided by Optimistic Concurrency Control with transactions (see Fig. 3): A peer that runs an algorithm that modifies data items does not apply these changes to the map right away. Instead, it keeps a log of changes that it wants to apply in a transaction. Only once a consistent set of changes is complete, the attempt to *commit* the transaction is made, which then writes the change set to the map. This means that only for the *short* time during transaction commit a *distributed* lock needs to be acquired. Because we lock the map only when we want to write the changes from a transaction, we have to make sure the state of the map which is read is consistent. Each transaction therefore defines a begin time, ensuring that all data that is accessed through the transaction comes from the state of the map *as it was at that time*, independently of what happens to the map in the meantime. This model allows us to acquire exclusive locks only for committing the transaction and synchronizing the new state of the map with other peers. After the synchronization, the updated map then forms the basis for transactions at a later time. During the commit, Map API verifies that no updates have happened on the same data since the transaction begin time. Only if that is the case the changes are applied to the database. Otherwise, a conflict exists for which Map API provides an interface similar to conflict resolution in Git: If a commit fails, the application can retrieve both new versions of all conflicting items, as well as a transaction containing the log of the non-conflicting items. With those, the user of Map API is free to implement conflict resolution tailored to the given application.

D. Distributed concurrency

Given the above requirements it is pretty straightforward to implement Optimistic Concurrency Control on a centralized unit. The implementation in a decentralized system however poses additional challenges. In a distributed system all communication between peers, including the acquisition of locks, takes place over the network. Therefore, no locking command is atomic as it would be commonly the case

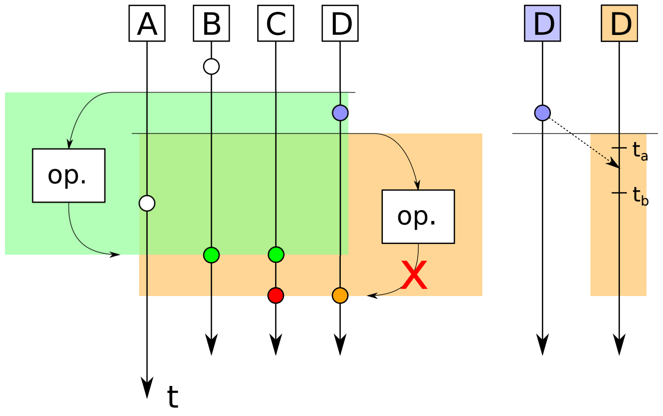

Fig. 3: On the left, an illustration of conflict detection in Map API. The semi-transparent overlapping boxes depict views, which are used by one agent to operate on B and C and another agent to operate on C and D. Time evolves from top to bottom. The time lines of data items A – D are shown, circles indicate updates of the respective item. The views begin at their begin time, indicated by horizontal lines, and end with the simultaneous update of the modified data items. In the shown situation, the commit on the right side fails because C has been updated since the view begin time. The right side of the figure highlights the problems caused by time delays in a naive implementation of a message passing algorithm. Reading D would have different results at t_a and t_b.

on a single machine. This implies that first and foremost consensus needs to be achieved among peers w.r.t. acquiring mutual exclusion locks over the network. State-of-the-art distributed applications, such as Google Chubby [19] achieve this using distributed consensus protocols like Paxos [14] or Raft [15]. Because these protocols are notoriously hard to integrate [19], Map API currently uses a simpler approach based on the assumption of no unannounced loss of connectivity. We implemented a distributed reader-writer lock to control access *at the level of chunks*: Reader-writer locks are based on the observation that while conflicts may arise in write-write and read-write contention, they do not arise when two peers try to read the same data. Because of this, it is possible to implement distributed reader-writer locks such that network communication is only necessary when the lock is acquired in *write-mode*: Peers that want to perform read operations acquire a read-lock locally. Only for write operations all other peers of a chunk are asked for permission. If a peer is currently reading from the affected chunk, write lock acquisition is deferred until the read-lock is released. Similarly, read locks can only be acquired if no write-lock has been granted. Given that an algorithm operates on multiple chunks simultaneously, care must be taken to avoid deadlocks. We solve this by using the well-known approach of imposing a lock ordering.

A second challenge is posed by the fact that distributed systems exhibit message delay; messages do not arrive immediately after they are sent, as shown on the right side of Fig. 3. It could therefore be possible that data-updates from a commit that has happened a long time ago arrive at a peer only after it has started another transaction, leading to an inconsistent view. The solution to this is to synchronize time between peers both before and after defining the commit time. Time synchronization can be accomplished at the same time as the write-lock is acquired and released, respectively.

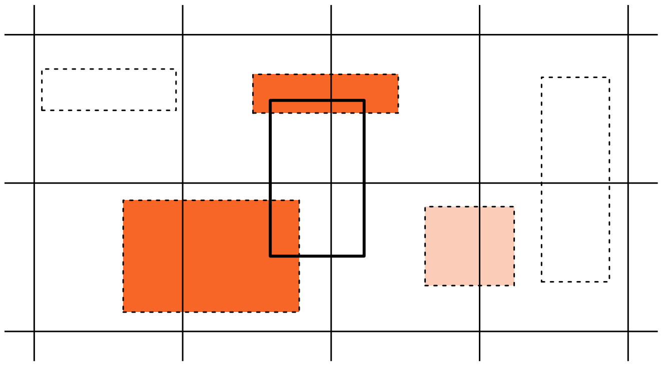

Fig. 4: A 2-D spatial index. A bounded space is subdivided into regular cells, which are deterministically assigned DHT keys. Data is registered using bounding boxes (dashed outlines) and assigned to the overlapping cells. Similarly, a query (solid bold outline) returns all data references from the cells that it itself overlaps (shaded boxes). The data in the lightly shaded box is returned by the query, although not overlapping the query.

We adopt Logical Clocks [20], which are monotonically increasing counters that capture chronological and causal relationships between data. These counters dictate clock synchronization inherently with message exchange.

E. Distributed lookup

As mentioned earlier, all data from tables is divided into chunks to control the granularity of data-sharing and data-locking. Access to a chunk can be obtained trivially by knowing both its identity and any peer that participates in it. Unfortunately, a brief reflection upon mapping use cases shows that the condition when both are known is rarely met. We therefore need different, more abstract ways of accessing chunks to start participation.

We distinguish the following ordinary access types:

- **By peer:** The identity of a peer is known, but not the identity of the requested data. This can for instance be the case for entities overseeing a field operation where new data is gathered.
- **By reference:** The identity of a piece of data is known, but not that of any peer that holds it. This can happen when a peer stumbles upon an unknown data reference, e.g. during map-structure traversal.
- **By extrinsics:** Neither the identity of the piece of data nor of any peer holding it is known, but the requested data can be defined in terms of values extrinsic to Map API. For instance, new users of a Map API application might be interested in obtaining data from a certain GPS bounding box.

Providing access by peer is trivial: One only needs to request from that peer to send all the data it generates.

Access by reference and access by extrinsics is more interesting: We reduce the problem of access by reference to mapping of references to peers and the problem of access by extrinsics to mapping of extrinsics to data reference lists. In particular, access by extrinsics is achieved by subdividing the lookup space into regular cells and assigning each cell a space-reference deterministically (see Fig. 4). With that reduction, we can apply Distributed Hash Tables (DHTs) to both problems. DHTs are state of the art protocols in

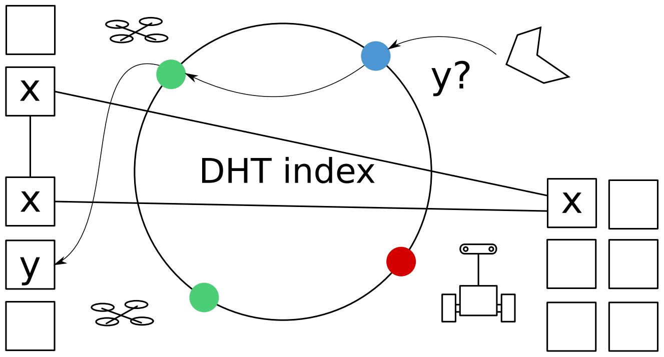

Fig. 5: Data in Map API. Using Map API only obligates providing a lightweight portion of the Distributed Hash Table index (denoted by circles). Heavy-weight mapping data is held only by peers interested in it. Here, the two quadrotors and the ground robot share data x. Consensus on modifications of x is established among them, as indicated by solid lines. A data-reference based or spatial DHT index is used to locate the data y.

distributed computing that allow $\mathcal{O}(\log n)$ key-value mapping operations while evenly distributing the work and data load among the participants. They are good for lookup of constant values or values whose consistency is not of utmost importance, but not for maintaining consistent state. However this information is sufficient for lookup, since given a reference a single peer is sufficient to provide data access. For Map API, we have chosen to use the Chord DHT protocol [11], which at the time of writing seems to be the most widespread. As with consensus, we envision rendering the system robust to connection loss in future iterations using more complicated variants of Chord that attempt to approach perfect availability [21]. To summarize, the interplay between distribution, consensus and lookup is shown in Fig. 5.

F. Update notification and Triggers

If a peer participates in a chunk, updates are automatically sent to the peer as described in Section II-B. However, what about situations in which a peer is interested in new insertions or updates to data that it does not (yet) participate in or cannot hold permanently? Providing notifications in such situations would be very useful if a robot is interested in updates to map data within a particular region, for instance to update path-planning results. Updates to this area should be filtered to extract only the relevant information an agent needs which is then sent to that agent.

We formalize this concept using a mechanism called "Trigger". A trigger is an update notification request that an agent can attach to a piece of information. On changes to this data the trigger function is executed on the updated information such that the relevant parts can be identified before sending data to the peer that registered the trigger. In this way triggers represent a concept of *information flow control* in the distributed system.

The most trivial such function takes the entire data that lies within a spatial region and pushes all changes to the peer. We however see the main strength of this concept in more complex triggers, which for instance apply sophisticated filters on the data or even pre-process it before sending. The possibility to remotely filter and pre-process data is key to keeping the bandwidth usage low, especially as the size of

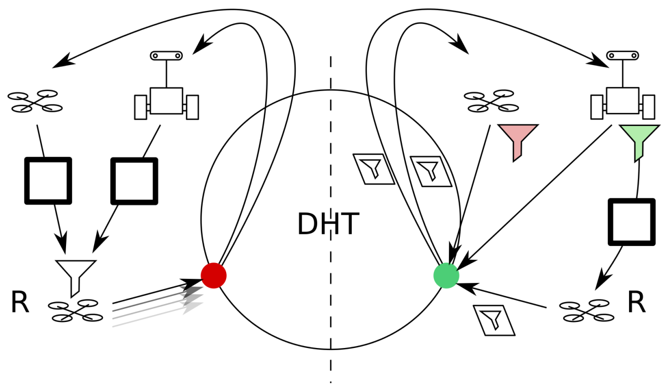

Fig. 6: This figure compares the trigger system suggested for Map API (right) with a polling based approch (left). On the left side, R continuously polls a spatial index to stay updated about a region. It must then fetch all the data from previously unseen references and filter it according to its needs. On the right side, we depict the advantages of triggers. Here R sends the blueprint of its filter once to the spatial index. Then, if agents register new data in the spatial index, the spatial index sends them all its filter blueprints. The agents then apply the filters locally and only send the data to R if it passes the filter. Thus, less data is sent over the network (here, one chunk instead of two).

maps and the number of agents grow (see Fig. 6). In the current version of Map API we have only implemented an interface to attach callbacks to updates in chunks currently co-owned. Triggers that are run entirely remotely and support complex spatial and other filters are subject to future work.

III. Experiments

To demonstrate and validate Map API, we have developed a visual-inertial mapping application, which takes mapping data from Google Tango devices [22] and represents it using a pose-graph and sparse landmarks. Everything has been implemented in C++11, using libraries such as Eigen, Protocol Buffers and ZeroMQ. One of the main contributions of Map API is providing a decentralized way of collaborating on a global map. We therefore first present a simple experiment that validates the claim that providing this functionality is worthwhile. Because most of Map API deals with concurrency, it is vital to verify and validate its implementation. Therefore, we have put important features in a carefully designed testing framework which allows continuous integration and monitoring throughout Map API development. A simple example of such a test is shown in algorithm 1. Alas, most of such tests concerning conflicts in concurrency, consistent distributed consensus and version control work but do not provide much interesting data for analysis. One of the earlier experiments we have performed on the distributed locking protocol investigated the effect of low-level design decisions on the performance in situations with high write-contention. However, we have found that such high write-contentions do not fit the usual Map API use case and are thus not compelled to discuss the results here. In contrast, methods for lookup provide a more fruitful base for further analysis. Thus, we present an experiment on the spatial lookup method and use the results to discuss how it can be best used in different scenarios. Independently of the experiments presented here, we have developed a project on lifelong mapping on top of Map API [17].

Algorithm 1 This procedure is executed on P peers in parallel to test decentralized conflict detection. If decentralized conflict detection works correctly, the final value of a is consistently $a_0 + PN$.

```
procedure ViewTest(Table $T$)
    for $i \in \{1; N\}$ do
        while Commit fails do
            $v \leftarrow$ new view($t$)
            $n \leftarrow v$.get($T$, $a$)
            $v$.update($T$, $a$, $n+1$)
            $v$.commit()
        end while
    end for
end procedure
```

A. Experimental setup

For the first experiment, where we show that having decentralized processing units is beneficial, we choose the scenario presented in Fig. 1. There, we have a robotic operation that is carried out over three rooms spanning a full floor of an office building. We emulate the situation where the big, merged datasets (red, blue, cyan) are created by ground robots and the smaller datasets are created by aerial robots. To benchmark centralized serial processing against decentralized parallel processing, we use the operation of registering the data from the aerial robots with the data from the ground robots, for which we use loop closure [23] and bundle adjustment optimization [18]. For the experiment, we compare the run time of running the registration operation on the three emulated ground robots versus a central instance with serialized map processing. To represent the difference of computational power between ground robots and central unit, and to involve some actual networking in the experiment, the ground robots have been emulated on three separate Amazon EC2 m3.medium instances. These instances each provide a single hyperthread on a 2.5 GHz Intel Xeon, while the central instance provides eight hyperthreads on a 3.3 GHz Intel Xeon. However, in addition to performing the map registration computation, we also simulate the bandwidth between the peers. Assuming that a good direct link can be established between aerial robots and nearby ground robots, but that the link between ground robots and the central is of worse quality, we set the former bandwidth to 1 MBps and the latter to 100 kBps.

For the experiment characterizing spatial lookup, we take five datasets collected on four floors of a building. Each dataset has been individually optimized using bundle-adjustment and loop closure optimization, and the datasets have then been co-registered with each other. After registering the total bounding box has been calculated and used as a base for a spatial index, while the bounding boxes of the individual datasets have been used to register them in the spatial index. Using the spatial index, 30 queries of randomly placed cubes of 3 m have been performed. For each query, a separate peer starts up, performs the query,

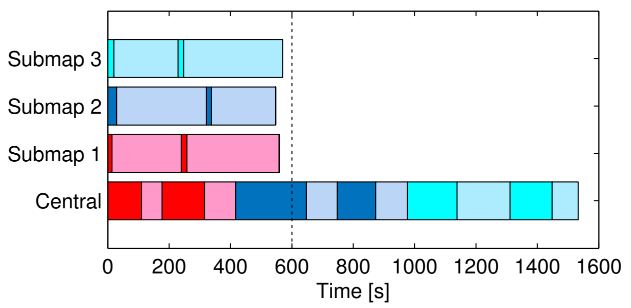

Fig. 7: Time, in seconds, taken for processing the same data with a decentralized setup versus a centralized one, where neighboring aerial and ground robots have a 1 MBps link versus a 100 kBps link between aerial robots and centralized unit. The colors correspond to the ground robots from Fig. 1. Darker colors denote data transfer while lighter colors denote data processing.

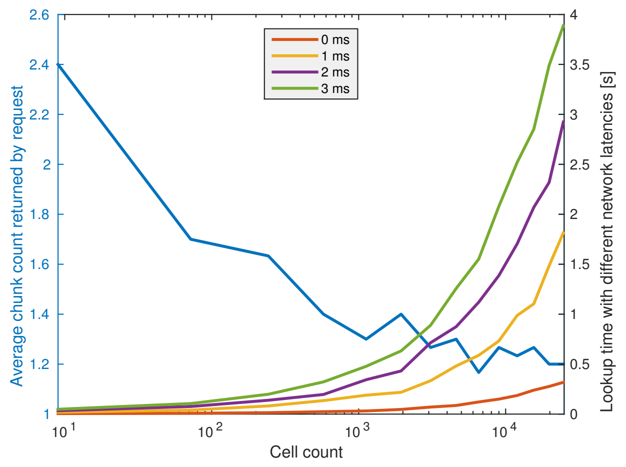

Fig. 8: Effect of granularity on the spatial index performance. A finer granularity generally decreases the amount of returned data while increasing the duration of a request. The latter also increases with network latency.

and loads the resulting chunks. We are most interested in seeing the effect of choosing different cell granularities. As with locking, we expect that a lower granularity results in more concise responses while increasing the query overhead. Other parameters that we have varied are the amount of by-standing peers (affects chord index performance) and simulated network latency.

IV. Results

Fig. 7 shows the results for the experiment of parallel decentralized versus serial centralized map processing. We can see that even using less computational power, the decentralized entities have finished registering the data within about ten minutes (vertical line), while the central entity takes about 25 minutes to process all the data. Since in our example data transfers dominate the time budget, even parallelizing the processing on the central unit would still be much slower than the distributed setup. Apart from proving the point about decentralization in such network conditions, this experiment also shows that Map API can be successfully executed among peers in the cloud, in this case linking a peer from Switzerland to peers located in Ireland.

Fig. 8 shows the most relevant results of the experiments regarding spatial lookup. As shown, the amount of missions a query returns generally decreases with a finer granularity. In contrast, the time it takes to process a query increases linearly with the cell count. This meets our expectations, since the amount of cells in the same bounding box increases linearly as the total cell density increases. Because each cell is mapped to a Chord index datum, the amount of Chord requests increases only linearly. Finally, the query time increases linearly with network latency and inversely linear with bandwidth as well. An optimal cell count is one that minimizes the total time of a query, which consists of requesting references to data that could potentially overlap with the query, and retrieving those data by reference. Thus, given the above results, we can conclude that there is no general rule for an optimal cell count. Instead, one needs to take into account system properties like expected dataset size, network bandwidth and latency and values that are harder to model, such as the density of registered data bounding boxes in areas typical for requests. On one hand, for instance, with big datasets and low bandwidth, yet small latency, it is worth having more cells, such as to only obtain the required data. On the other hand, with high latency but high bandwidth and smaller datasets, coarser cells could be preferable because the cost of retrieving irrelevant data is relatively low. Indeed, one could also think of attaching the actual bounding boxes as metadata to the data references in the spatial cell and to the Chord queries sent to the peers responsible for each cell, as discussed in Section II-F. Hence, the filtering of irrelevant data could be delegated to the remote peer, trading computing power for less network load. This, however, would also lead to increased memory overhead for the spatial index data itself.

V. Conclusion

In this paper, we have described a back-end for distributed collaborative mapping, which:

- Distributes a *global* map that agents can lookup and work on concurrently.
- Enables dynamic data ownership using distributed consensus and optimistic concurrency control.
- Optimizes network usage using Triggers which implement distributed filters and update notifications.

A first version of this back-end exhibiting most features has been implemented and successfully tested, locally and on the cloud. Next steps include the full implementation of the mentioned features and rendering Map API robust to unannounced peer loss by more fully embracing work from the domain of Distributed Computing. Although Map API has already been used for a project showcasing long-term mapping, we are eager to integrate it with more collaborative mapping schemes, in particular ones doing mapping on a large scale. While this article has not treated the algorithms performing collaborative state estimation itself, we are confident that providing the flexible way of collaborating on mapping data contributed by Map API has the potential to simplify the integration of existing mapping schemes or even unlock new ones.

VI. Acknowledgments

The research leading to these results has received funding from Google's project Tango and Willow Garage.

The authors are grateful to the help of Christian Decker from ETHZ's Distributed Computing Group for the fruitful discussions about Distributed Computing and to Professor Auke Ijspeert from EPFL's Biorobotics Laboratory for co-supervising the associated Master Thesis.

Furthermore, we thank Francis Colas and Paul Furgale for feedback on early design drafts.

VII. API

The following is a simplified example showcasing Map API in use. The first half of the code shows what is typically supposed to happen in the application layer - table definition. The second half shows the implementation of the inner loop of algorithm 1, applied to the subvalue of a data structure.

```
MAP_API_REVISION_PROTOBUF(proto::DataType);
enum Fields { kField };

map_api::TableDescriptor descriptor;
descriptor.setName("my_table");
descriptor.addField<proto::DataType>(kField);
map_api::NetTable* my_table = map_api::↩
    NetTableManager::addTable(descriptor);

my_table->getChunk(chunk_id);  // Assumed given.
map_api::Transaction transaction;
map_api::Revision revision = transaction.getById(↩
    my_table, item_id);  // Assumed given.
proto::DataType data;
revision.get(kField, &data);
data.set_some_subvalue(data.some_subvalue() + 1);
revision.set(kField, data);
transaction.update(my_table, revision);
if (transaction.commit()) {
  LOG(INFO) << "Commit succeeded!";
}
```

References

[1] D. Fox, W. Burgard, H. Kruppa, and S. Thrun, "A probabilistic approach to collaborative multi-robot localization," *Autonomous robots*, 2000.

[2] A. Martinelli, "Improving the precision on multi robot localization by using a series of filters hierarchically distributed," in *Intelligent Robots and Systems, IEEE/RSJ International Conference on*. IEEE, 2007.

[3] N. Trawny, S. I. Roumeliotis, and G. B. Giannakis, "Cooperative multi-robot localization under communication constraints," in *Robotics and Automation, IEEE International Conference on*. IEEE, 2009.

[4] L. C. Carrillo-Arce, E. D. Nerurkar, J. L. Gordillo, and S. I. Roumeliotis, "Decentralized multi-robot cooperative localization using covariance intersection," in *Intelligent Robots and Systems (IROS), IEEE/RSJ International Conference on*, 2013.

[5] L. Riazuelo, J. Civera, and J. Montiel, "C2tam: A cloud framework for cooperative tracking and mapping," *Robotics and Autonomous Systems*, 2014.

[6] C. Forster, S. Lynen, L. Kneip, and D. Scaramuzza, "Collaborative monocular slam with multiple micro aerial vehicles," in *Intelligent Robots and Systems (IROS), IEEE/RSJ International Conference on*, 2013.

[7] M. Waibel, M. Beetz, *et al.*, "Roboearth," *Robotics Automation Magazine, IEEE*, 2011.

[8] G. Mohanarajah, V. Usenko, M. Singh, M. Waibel, and R. DAndrea, "Cloud-based collaborative 3d mapping in real-time with low-cost robots," *IEEE Transactions on Automation Science and Engineering*, 2014.

[9] A. Cunningham, V. Indelman, and F. Dellaert, "Ddf-sam 2.0: Consistent distributed smoothing and mapping," in *Robotics and Automation (ICRA), IEEE International Conference on*, 2013.

[10] T. A. Vidal-Calleja, C. Berger, J. Solà, and S. Lacroix, "Large scale multiple robot visual mapping with heterogeneous landmarks in semi-structured terrain," *Robotics and Autonomous Systems*, 2011.

[11] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *SIGCOMM Comput. Commun. Rev.*, 2001.

[12] W. Ren and R. Beard, *Distributed consensus in multi-vehicle cooperative control: theory and applications*. Springer, 2007.

[13] E. Montijano and C. Sagues, "Distributed multi-camera visual mapping using topological maps of planar regions," *Pattern Recognition*, 2011.

[14] L. Lamport, "Paxos made simple," *ACM Sigact News*, 2001.

[15] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc ATC, USENIX Annual Technical Conference*, 2014.

[16] K. P. Eswaran, "Aspects of a trigger subsystem in an integrated database system," in *Proceedings of the 2nd international conference on Software engineering*, 1976.

[17] M. Dymczyk, S. Lynen, T. Cieslewski, M. Bosse, R. Siegwart, and P. Furgale, "The gist of maps – summarizing experience for lifelong localization," in *Robotics and Automation (ICRA), IEEE International Conference on*, 2015.

[18] B. Triggs, P. McLauchlan, R. Hartley, and A. Fitzgibbon, "Bundle adjustment — a modern synthesis," in *Vision Algorithms: Theory and Practice*, 2000.

[19] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006.

[20] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, 1978.

[21] "How to improve the reliability of chord?" in *On the Move to Meaningful Internet Systems: OTM Workshops*, 2008.

[22] J. C. Lee, R. Dugan, *et al.*, "Google project tango," https://www.google.com/atap/projecttango/#project.

[23] S. Lynen, M. Bosse, P. Furgale, and R. Siegwart, "Placeless place-recognition," in *3D Vision (3DV), 2nd International Conference on*, 2014.