Robustness to Connectivity Loss for Collaborative Mapping

Anwar Quraishi*, Titus Cieslewski*, Simon Lynen*† and Roland Siegwart*
*Autonomous Systems Lab ETH Zurich, †Google Inc.

***Abstract*— Having a team of robots to perform a task such as mapping is faster and more reliable than doing the same with a single robot, which can be crucial in scenarios such as search and rescue. We are developing a fully distributed framework for collaborative mapping with large robot swarms that is robust to abrupt departure of robots due to malfunctions or network problems. While several approaches to multi-robot mapping have been proposed, most of them either build a collection of local sub-maps, or rely on a central authority to merge maps built by individual robots. Our framework is unique in that it requires no central authority, yet allows robots to simultaneously contribute to a single global map, which is stored in a decentralized fashion. This greatly improves the scalability of our system with respect to number of robots. However, our approach requires systematic coordination among robots in order to make modifications to the map. Unannounced departure of the robots makes coordination challenging, and can potentially make the map inconsistent or result in loss of data. We borrow ideas from the domain of distributed computing to address those challenges. Further, we demonstrate the robustness of the proposed system by subjecting it to various conditions in which participating robots fail.**

I. Introduction

In many situations, the use of multiple robots instead of a single one can accelerate task completion. In search and rescue scenarios [1], [2] for instance, multiple robots can be dispatched to perform mapping in different parts of the scene, such that an overall map of this scene can be obtained more rapidly. Given, then, that our goal is that all robots can operate on and contribute to the same global map, as illustrated in Fig. 1, a central challenge is defining how the mapping data that is created by individual robots is shared in a manner that the resulting global map structure is consistent. A simple way to achieve this is through a centralized system, where a central entity serves as a repository for the data collected by all robots [3], [4]. The disadvantage of a centralized system is that it cannot scale arbitrarily with respect to amount of data due to limited computational power and communication bandwidth. Furthermore, it introduces a central point of failure into the system.

To combat the disadvantages of centralized systems, several decentralized mapping systems have been proposed recently [5], [6], [7]. In these systems, each robot typically keeps its own data locally, and parts of that local data are shared with other robots as needed. As a consequence, the map structure is inherently fragmented, access to a global map is often available only offline in post-processing, and there is no or a very application-specific notion of consensus among robots on the state of the common data.

In our previous work, we have developed a decentralized mapping framework, Map API [8], which does provide implicit consensus on shared data by having robots exchange messages in a specific order to reach agreement on modifications to it. The framework also provides a mechanism for efficient lookup of said data in a multi-agent system. However, a major limitation in the protocols that we proposed to achieve this is that they are not robust to loss of connectivity. Such loss of connectivity is common when robots operate in the field, and can happen when they shut down due to a malfunction, temporarily drive outside of the range of the wireless network used for communication, or due to network failures. When this happens, some robots might not receive messages related to modifications of the shared map data. The robots then possess an older version of data when they rejoin the network, thereby leaving the map in an inconsistent state. Additionally, data stored on those robots necessary for the lookup mechanism to function may become inaccessible.

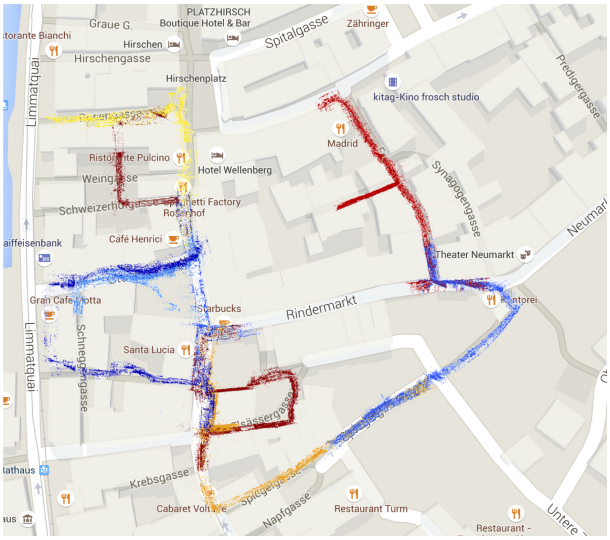

Fig. 1: An example scenario of operation of Map API, where trajectories of individual agents, shown in different colors, are merged into a single map. The generated map is superimposed on a street map from Google Maps.

In this paper, we show how two protocols developed by the computer science community, Raft [9] and Chord [10] can be incorporated into Map API to provide robustness to loss of connectivity. Raft allows a group of robots to reach consensus, even if there are robot malfunctions or network failures. Chord distributed lookup along with our proposed features provides a robust mechanism for lookup of mapping data in the multi-agent system. In particular, our contributions are:

- Incorporating Raft in Map API for robust consensus on shared map data.

- Developing an additional protocol running alongside Raft that ensures consistency of the global map when an operation is performed on it.
- Proposing a data replication scheme that minimizes the risk of loss of information required by Chord lookup when robots depart abruptly.

While the discussion and experiments are focused on Map API, we believe that the presented adaptations of Raft and Chord can be applied to provide robustness to loss of connectivity in any other decentralized mapping system.

II. Methodology

A. Relevant Map API components

We identify three main components of Map API for which we will discuss application of Raft and Chord in the following sections:

Chunk consensus Map API introduces a subdivision of the map data into so-called *chunks*. For instance, a particular chunk might contain a set of sensor measurements, or robot poses. Agents either fully replicate all data in a chunk (*participate* in a chunk) or don't have access to it at all. A consensus protocol is then executed individually for each chunk and its participants, as different chunks can have different sets of participating agents.

Inter-chunk consistency Since operations on map data, such as bundle adjustment, commonly affect data in multiple chunks, an additional protocol on top of the chunk consensus protocol ensures that such operations are applied consistently, i.e. that any agent either perceives all of the changes of a given operation, or none of them. This prevents that the effects of an operation are only perceived partially. To prevent that time taking operations performed by one agent prevent other agents from accessing data, this consistency protocol employs optimistic concurrency control: An agent reads the map data at a particular logical time [11], the *begin time*, and computes the set of changes to be applied to the map, which are stored in a *transaction*. Similarly, each consistent set of changes in a transaction proposed by the agent is assigned a unique logical time, the *commit time*. To ensure consistency and consensus, the change set can only be applied to the map if all changed items haven't been modified by another agent between begin and commit time. Otherwise, the transaction is aborted.

Lookup Finally, multiple ways of looking up chunks that an agent does not yet participate in are provided by a simplified version of the Chord protocol [10] which is not inherently robust to loss of connectivity.

B. Chunk consensus with Raft

At a fundamental level, ensuring consistency of map data requires guaranteeing that all robots participating in a particular chunk view the exact same state of the data contained in it. This can be achieved if all active members of the chunk have consensus on each modification to the data. If the communication channel and all agents are guaranteed to be failure proof, consensus can be achieved by exchanging a set of messages among chunk participants in a specific order, as discussed in [8]. Realistically, reaching consensus is difficult if those messages are not acknowledged due to agent or network failures. In particular, two theoretical results from the field of distributed systems of relevance here are the *FLP result* [12], and the *CAP conjecture* [13]. The FLP result proves that consensus between a set of agents is impossible in an asynchronous (no upper limit on time to process a request) system subject to failures, if even one agent can possibly fail. The CAP theorem formalizes the tradeoff between Consistency, Availability and Partition tolerance – it is not possible to have more than two of these properties in an unreliable distributed system.

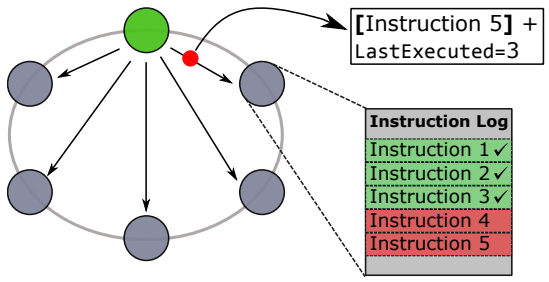

Fig. 2: Along with periodic heartbeat messages, the leader (green) of a Raft group sends new instructions to be replicated, and the index of the latest instruction allowed to be executed. All members (including the leader) store these instructions in a log, and execute them if their index is $\leq$ `LastExecuted`.

In our system, we can make some assumptions at this point. Firstly, we can assume that an agent takes a finite time to process a request, and that an agent has departed if it doesn't respond within a stipulated time. Secondly, we can partially relax the consistency requirement. We instead ask for 'eventual consistency' – consistent given sufficient time – and a way to check if consistency has been achieved yet. These assumptions allow us to use the Raft consensus protocol [9] for consensus among chunk members. The protocol guarantees consensus in a group of machines even if up to half of the members in the group fail. We have an instance of the Raft protocol running for each chunk.

Raft works by electing a leader from among the members in a chunk, a new one being re-elected automatically if the previous one fails. The leader periodically sends so-called *heartbeat* messages to each member, and detects failure of a member when an acknowledgment is not received. Similarly, chunk members detect failure of leader if no heartbeat is received for a certain amount of time, triggering a new leader election. On each member in the consensus group, the leader manages a replicated log of instructions. The types of log instructions used in Map API are listed in table I. Instructions are first replicated, and only if replication is confirmed on the majority of the members does the leader allow the members to execute an instruction (see Fig. 2). Given the safety properties of Raft, which are proved in [14], once an instruction is executed by a majority of the members,

1) it is guaranteed to be eventually executed by the remaining members, and,

Log entry	Action on execution
`ReadLockReq`/ `WriteLockReq`	Grant lock or add sender to lock queue.
`ModifyData`	If sent by the write lock holder, store the data changes (additions or modifications) to be applied to the chunk data later.
`UnlockRequest`	If sent by the write lock holder, apply data changes contained in the `ModifyData` entries sent after the corresponding `LockRequest` entry, and release lock. If there is another agent in the lock queue, grant the lock to that agent.
`AddMember`	Add the new agent to member list.
`RemoveMember`	Remove an agent from member list. If the agent had acquired the chunk lock, release it. Discard all the `ModifyData` entries sent after the corresponding `WriteLockReq`.

TABLE I: Types of Raft log instructions for various actions in a chunk.

2) the replicated Raft logs on all members are guaranteed to be consistent up till this point.

Raft guarantees presence of a single leader, and that the order of execution of the instructions on each member will be the same even if there is a leader change. While the time of execution of those instructions on each member may be different, they would eventually execute the same sequence of instructions, and hence compute the same output. This guarantees consistency of data at the level of chunk. Note that Raft is not a centralized system because the leader is not a special entity; any member of the consensus group can become the leader.

C. Chunk membership

While Raft has a mechanism that anticipates preferentially occasional membership changes, we modified it to suit more dynamic changes in chunk membership as could be envisioned in robotic swarms. Changing membership is a delicate process because instruction processing rules and leader election in Raft are based on majority voting, for which robots must be aware of all the other members of the group. To join or leave a chunk, robots first request the leader. The leader then appends `AddMember` or `RemoveMember` instructions to the log so as to have consensus on chunk membership. The robots are allowed to join or leave only after a majority of the existing members process the aforementioned instructions from their log.

On detecting failure of a member, the leader appends the `RemoveMember` instruction to the log to notify the other members. Between the time of failure of the member and the time when the relevant instruction is processed by the majority (*i.e.* consensus is achieved), the failed member is considered in the majority, with its response always being negative, since it does not respond. This effectively makes the voting process stricter, by requiring agreement from more than the majority of active members. Therefore, if half or more members fail simultaneously, then no new instructions are processed because the majority response is always negative. This effectively halts the operation of the chunk until sufficient number of agents resume responding to outstanding requests.

D. Distributed concurrency control

Concurrent access to shared data may result in situations where two or more agents concurrently attempt to make conflicting changes to the same data. To avoid this, we provide exclusive data access at the level of chunks by way of reader-writer locks, which a member can acquire by sending a request to the leader. Granting such locks to an agent requires consensus among chunk members, which the leader achieves with the help of `ReadLockReq` or `WriteLockReq` Raft log entries. The lock requests are queued and granted on a first-come-first-serve basis. We differentiate between read and write locks because a conflict does not arise when several agents read the same data. While several members can acquire a read lock on the same chunk simultaneously, a write lock is granted only if no other member has read or a write lock. Together with optimistic concurrency control (see Section II-A), write locks guarantee that there are no conflicts in map data.

E. Multi-chunk transactions

While the Raft protocol helps ensure consistency of data within a chunk, a transaction may affect data in multiple chunks, as explained earlier in this section. The central contribution of this work is a protocol that runs alongside Raft and guarantees consistency of data across chunks during a transaction. Failure of agents other than the one performing the transaction can be handled by individual chunks of which the agent was a member, and does not pose a risk to inter-chunk consistency. Problems arise when a robot performing such an operation fails after committing the modifications in the transaction only to a subset of the chunks involved. In such a case, while data within all the chunks remains consistent, some chunks may become inconsistent with others. The key to addressing this is avoiding partial commits of transactions.

Our protocol requires members of each chunk involved to collectively maintain the state of the ongoing transaction, as shown in Fig. 3. An agent begins a transaction by acquiring a read lock on relevant chunks, reading required data, and releasing the read lock. It then computes the changes to be made. No actual changes are made until this point.

The agent finally proceeds to *commit* the transaction, which involves the following steps, in order:

1) Acquire write locks on all involved chunks. Abort if any relevant data items in an involved chunk have been modified by another agent since the beginning of this transaction
2) Send description of the transaction to each affected chunk.
3) Send the change items to be applied to respective chunks.
4) Send apply-and-unlock instruction to all chunks.

A specific ordering is imposed for acquiring locks on multiple chunks to avoid deadlocks if there are several active

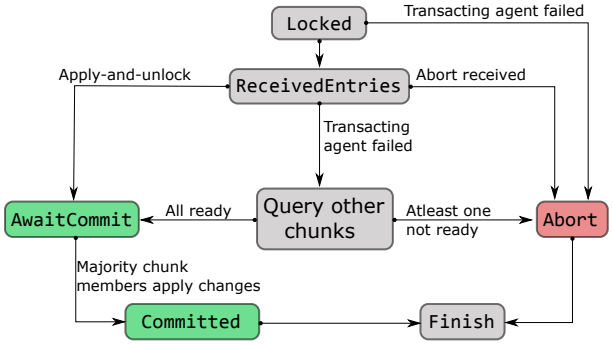

Fig. 3: The state of a multi-chunk transaction maintained by each of the involved chunks, and transitions between them. If the chunks detect failure of the agent performing the transaction after having received all the related modifications but before receiving the apply instruction, they query the other chunks before deciding to either commit or abort the transaction.

transactions with an overlapping set of affected chunks. The description of the transaction sent to each chunk contains the number of change items for that chunk and a list of all chunks involved in the transaction. Once a chunk receives all changes to be applied in it, the transaction state with respect to this chunk changes to `ReceivedEntries` (see Fig. 3). The crucial step here is the 'send apply-and-unlock', which is sent only ofter all change items have been sent to their respective chunks. Therefore, if apply-and-unlock is received by a certain chunk, it is safe to assume that all the other chunks have at least received their respective changes, and the changes for that chunk can be committed.

If the agent performing the transaction fails, this is detected by the chunks independently. A chunk would immediately abort the transaction if the failure is detected before the chunk received all of its respective changes. In contrast, the chunk proceeds to commit the changes if it already received apply-and-unlock instruction. The non-trival scenario is where a chunk detects failure after all its change items are received but before the apply-and-unlock instruction is received. In this case, the chunk leader queries other chunks and decides to either commit or abort the transaction accordingly.

F. Robust lookup

For looking up data, we have a number of indices, each of which which associate certain keys to one or more values. For example, in our system, each chunk has a unique id, and a chunk lookup index associates chunk ids to identities of all members of the chunk. If an agent needs access to data in a certain chunk, it can use this lookup index to obtain the identity of a member of the chunk, and send a request for membership and data access. We employ a simplified version of the Chord Distributed hash table [10] to distribute entries in these indices among active agents, in the interest of scalability. Chord logically arranges agents in a circular ring as a deterministic function of their identifiers, where each agent is linked to its predecessor, successor and additional agents on the ring, as shown in Fig. 4. The index entries are also deterministically assigned a position on the ring and distributed among the agents depending on

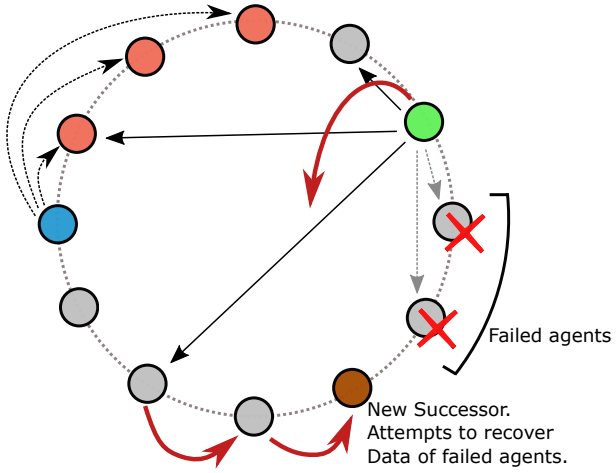

Fig. 4: Chord logically arranges agents (shown by shaded circles) in a ring, as a function of their id. The agent shaded in blue shows an example of replication on 3 immediate successors, *i.e.* 4 coipes of the same data (replication degree, $d = 4$). The agent shaded in green shows an example of links to predecessor, successor and additional agents. On failure of its two consecutive successors, it uses its links to other agents in the ring to find the new successor. Data is recovered since the new successor would have replicated data on the failed agents.

that position. A lookup query is passed around using the aforementioned links until a result is found. The inter-agent connections and the lookup protocol is implemented in a way that allows $\mathcal{O}(\log n)$ time lookup with respect to the number of participating agents. Chord is not robust in that unannounced departure of agents causes loss of data, and also breaks the neighbor relationships, which may result in failure of lookup queries. A number of approaches have been proposed to make Chord robust to agent failure [15], [16], [17], [18]. The common idea behind most approaches is to maintain replicas of Chord data. While data loss on agent failure can't be completely eliminated, replication can reduce the probability of the same. We implement a replication scheme along with a method to restore neighbor relationships with functioning agents (see Fig. 4).

Each agent periodically monitors its successor (clockwise) on the Chord ring by sending it a message and expecting an acknowledgment. Further, the data for which an agent is responsible is replicated on the agent's next $d-1$ successors on the ring, where d is called the 'replication degree'. On detecting failure of one or more of its successors, an agent first attempts to find a new functioning successor using the additional links to other agents on the ring, as schematically shown in Fig. 4. The new successor attempts to recover the data of the failed agents. In such a setup, data is lost only if the set of failing agents between recoveries contains at least one set of d consecutive agents from the chord ring. A numerical upper-bound on the probability of loss of any data when a given number of agents fail is derived in [17]. If k agents fail in a Chord system with replication degree d ($k > d$ and $d > 1$), we have

$$\mathrm{P}\left(\text{data is lost}\right) \leq \frac{nk^d}{(n-(d-1))^d}. \tag{1}$$

Note that this represents a *numerical* upper-bound on probability, and not actual probability.

III. Experiments

We tested our system with both mapping and non-mapping data, by artificially provoking agent failures. Essentially, our goal is to demonstrate the robustness of the system under unannounced agent departures. Note that both network failures and agent malfunction are treated in the same manner, since it is impossible to distinguish between them.

To begin with, we consider individual chunks and consistency of data within them. We perform an experiment where a chunk has 10 members, and a number of them fail. We demonstrate that consistency is preserved when up to half of the members fail simultaneously.

Further, we show that inter-chunk consistency is preserved when an agent performing a transaction affecting multiple chunks fails while the transaction is underway. We deliberately shut down the agent performing such a transaction involving a number of chunks, just after it communicates the data changes to all chunks but before it commits the transaction. We demonstrate that all chunks reach a consistent decision in finite time.

Finally, we experimentally demonstrate that by implementing a replication strategy for the Chord distributed lookup, we reduce the probability of data loss when some agents abruptly depart. We set up a system with our implementation of robust lookup, with 20 agents collectively storing 20000 key-value entries. We then shut down different numbers of agents and measure data loss for various levels of replication, showing that the probability of data loss is significantly reduced for each additional replication.

A. Experimental setup

We perform all tests locally on a single computer, where each agent is simulated as a separate process. The communication between agents has a simulated latency of 4ms. The agents have no knowledge of each others state, and the only way an agent can detect failure of another agent is when the failed agent does not respond to network messages. The leader sends heartbeat messages every 200ms, while the heartbeat timeout on followers is set to 500ms.

IV. Results

Fig. 5 shows time taken for surviving chunk members to reach consensus on removal of agents from the chunk after their failure is detected by the leader. While this time increases with number of agents failing simultaneously, consensus is reached in finite time if up to half of the members fail simultaneously. On detecting failure of members, the leader attempts to achieve consensus on their removal by way of the `RemoveMember` log entries. If the leader itself is one of the failing agents, a new leader is elected and attempts to do the same for the old leader and other failing members. Consensus is said to have been reached when the leader detects that a majority of the members have processed the `RemoveMember` log entries. Given the safety properties of Raft summarized in section II-B these entries being processed by a majority of the members implies that the map data in the chunk is consistent.

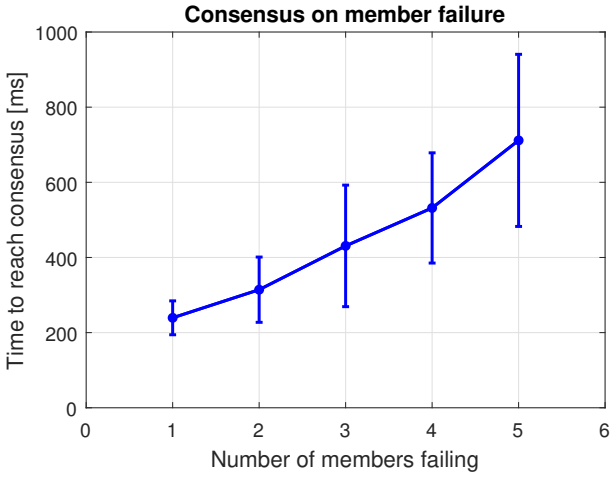

Fig. 5: Time taken by surviving members in a chunk to reach consensus after some members fail simultaneously and their failure is detected by the leader. There are originally ten members. The data is averaged over 5 trials.

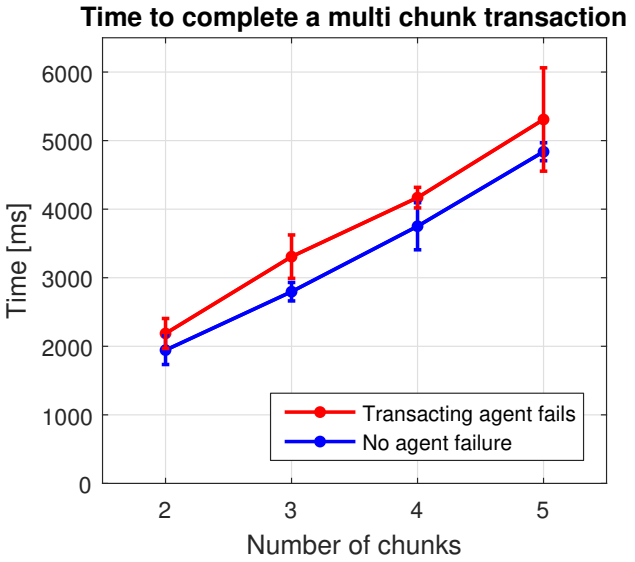

Fig. 6: Time taken to arrive at a consistent decision on a multi-chunk transaction involving various numbers of chunks, when the agent performing the transaction fails. The no-fail case is shown for reference. The data is averaged over 5 trials.

Fig. 6 shows the time taken for a multi-chunk transaction to complete in the case where the agent performing the transaction crashes just before committing the changes to be made. The transaction is said to be completed when all the chunks involved either apply or discard their relevant data changes. The chunks, each of which is an individual consensus group, reach a consistent decision on the transaction in a finite amount of time.

Fig. 7 shows the the fraction of trials from among 100 trials in which lookup data is lost, for various numbers of agents failing and various degrees of replication. Notice that for each additional replication, the probability of loss of data decreases significantly. This is because with additional

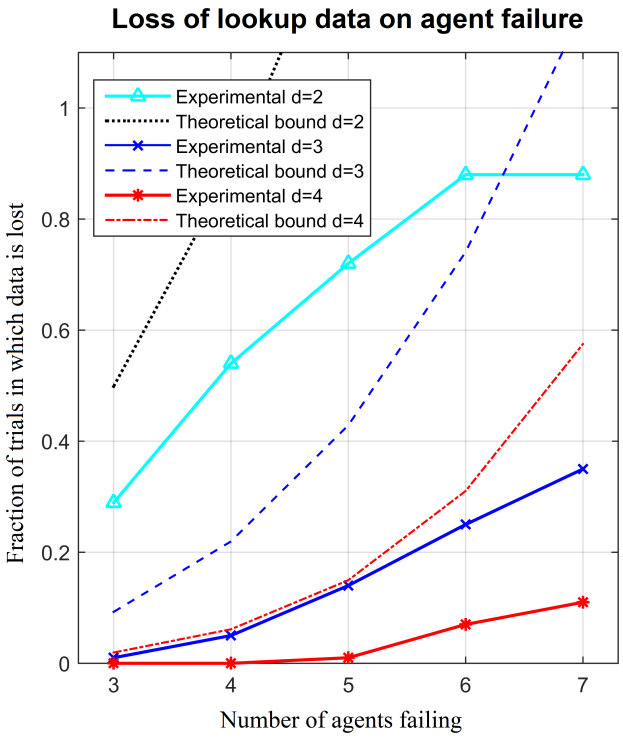

Fig. 7: Loss of look up data when agents drop, for various replication degrees d. The dashed lines show numerical upper-bounds on probability of data loss (note that these are not actual probability but just numerical upper-bounds) for various d. The system with 20 agents was loaded with experimental data containing 20000 key-value entries. In each trial, a given number of agents were crashed. Each solid line represents the fraction of trials in which data was lost, from among 100 trials, for a particular d.

replication the probability that a set of failing agents contains an agent and all its replicators decreases. The dashed lines show the numerical upper-bound on the probability of data loss computed using (1).

V. Conclusion

In this paper, we described our approach to making Map API robust to unannounced departure of participating robots. To begin with, we integrated the Raft consensus protocol with our framework to guarantee consensus at the level of chunks, the subdivision of data defined by Map API. With raft, each chunk can tolerate simultaneous unannounced departure of up to half of its members, beyond which the activity of the chunk is suspended since consensus can not be guaranteed. This prevents data in the chunk from becoming inconsistent. The implication of this is that if more than half of the members in a chunk depart abruptly, the data in that chunk is not accessible, until some of those members rejoin the network and a majority of the chunk members start functioning again. Therefore, while Map API guarantees consistency of all available map data, availability of data is not guaranteed. Further, while it has been shown that it is impossible to completely eliminate possibility of data loss in distributed lookup, our replication strategy significantly reduces its probability. We individually tested the robustness of consensus and that of lookup by subjecting the system to a variety of conditions where peers fail.

Next steps include investigating if Map API can function over ad-hoc networks, so that it can be deployed in places without existing network infrastructure. Since ad-hoc networks are prone to partitions, this requires studying to what extent network partitions can be tolerated and how map data can be merged when partitions join. Further, to make Map API suitable for applications where bandwidth is limited, we would like to explore various approaches, such as simplifying Raft leader election and relaxing the complete consistency requirement on the application side.

VI. Acknowledgments

The research leading to these results has received funding from Google's project Tango. The authors are thankful to Professor Auke Ijspeert from EPFL's Biorobotics Laboratory for cosupervising the associated Master Thesis.

References

[1] L. Marconi, C. Melchiorri, M. Beetz, D. Pangercic, R. Siegwart, S. Leutenegger, R. Carloni, S. Stramigioli, H. Bruyninckx, P. Doherty, *et al.*, "The sherpa project: Smart collaboration between humans and ground-aerial robots for improving rescuing activities in alpine environments," in *SSRR*, 2012.

[2] E. F. Flushing, M. Kudelski, L. M. Gambardella, and G. A. Di Caro, "Connectivity-aware planning of search and rescue missions," in *SSRR*, 2013.

[3] C. Forster, S. Lynen, L. Kneip, and D. Scaramuzza, "Collaborative monocular slam with multiple micro aerial vehicles," in *IROS*, 2013.

[4] L. Riazuelo, J. Civera, and J. Montiel, "C^{2}tam: A cloud framework for cooperative tracking and mapping," *RAS*, 2014.

[5] A. Cunningham, V. Indelman, and F. Dellaert, "Ddf-sam 2.0: Consistent distributed smoothing and mapping," in *ICRA*, 2013.

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

[7] A. Franchi, L. Freda, G. Oriolo, and M. Vendittelli, "The sensor-based random graph method for cooperative robot exploration," *Mechatronics, IEEE/ASME Transactions on*, vol. 14, no. 2, pp. 163–175, 2009.

[8] T. Cieslewski, S. Lynen, M. Dymczyk, S. Magnenat, and R. Siegwart, "Map API - scalable decentralized map building for robots," in *ICRA*, 2015.

[9] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX ATC*, 2014.

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

[11] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, 1978.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[13] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[14] D. Ongaro, "Consensus: Bridging theory and practice," Ph.D. dissertation, Stanford University, 2014.

[15] M. Naor and U. Wieder, "A simple fault tolerant distributed hash table," in *Peer-to-Peer Systems II*. Springer, 2003, pp. 88–97.

[16] N. Antonopoulos and J. Salter, "Efficient resource discovery in grids and p2p networks," *Internet Research*, vol. 14, no. 5, 2004.

[17] R. Kapelko, "Towards fault-tolerant chord p2p system: Analysis of some replication strategies." in *APWeb*. Springer, 2013.

[18] S. Ktari, M. Zoubert, A. Hecker, and H. Labiod, "Performance evaluation of replication strategies in dhts under churn," in *Proceedings of the 6th international conference on Mobile and ubiquitous multimedia*. ACM, 2007, pp. 90–97.