Autonomous Agents and Multi-Agent Systems (2023) 37:47
https://doi.org/10.1007/s10458-023-09629-2

RGS⊕: RDF graph synchronization for collaborative robotics

Cyrille Berger[1] · Patrick Doherty[1] · Piotr Rudol[1] · Mariusz Wzorek[1]

Accepted: 24 October 2023 / Published online: 24 November 2023

Abstract
In the context of collaborative robotics, distributed situation awareness is essential for supporting collective intelligence in teams of robots and human agents where it can be used for both individual and collective decision support. This is particularly important in applications pertaining to emergency rescue and crisis management. During operational missions, data and knowledge is gathered incrementally and in different ways by heterogeneous robots and humans. The purpose of this paper is to describe an RDF Graph Synchronization System called RGS⊕. It is assumed that a dynamic set of agents provide or retrieve knowledge stored in their local RDF Graphs which are continuously synchronized between agents. The RGS⊕ System was designed to handle unreliable communication and does not rely on a static centralized infrastructure. It is capable of synchronizing knowledge as timely as possible and allows agents to access knowledge while it is incrementally acquired. A deeper empirical analysis of the RGS⊕ System is provided that shows both its efficiency and efficacy.

Keywords Multi-robot collaboration · Unmanned aerial vehicles · Distributed knowledge representation · Distributed situation awareness · Semantic web technology · RDF graph synchronization · Multi-agent human/robot interaction

This work has been supported by the ELLIIT Network Organization for Information and Communication Technology, Sweden (Project B09), the Swedish Foundation for Strategic Research SSF (Smart Systems Project RIT15-0097), and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

✉ Cyrille Berger
cyrille.berger@liu.se

Patrick Doherty
patrick.doherty@liu.se

Piotr Rudol
piotr.rudol@liu.se

Mariusz Wzorek
mariusz.wzorek@liu.se

[1] Department of Computer and Information Science, Linköping University, 581 83 Linköping, Sweden

1 Introduction

This paper addresses the issue of efficient data and knowledge sharing between multiple autonomous agents operating in the field. An example application is providing support in the coordination of rescue operations after natural or human-made disasters. In such scenarios, the use of autonomous agents, such as Unmanned Aerial or Ground Vehicles (UAVs or UGVs), provides a means to quickly acquire situation awareness in the affected regions in the form of maps, images, potential victim locations etc. Such information is essential in order to increase the efficiency of rescue teams in time-critical life-saving activities [56].

During the initial phases of any rescue operation, one of the main activities is to explore the environment, where human and robotic agents collect information such as the state of buildings, changes in terrain, locations, and the state of potential victims. Based on the collected information, a rescue operation can be planned to provide medical assistance by professionals or to deliver supplies using robotic platforms to where it is needed most. Because such operations rely on large amounts of data, agents normally do not have access to a full view of the collective situational awareness.

Generally, each team agent collects information contributing to its local knowledge of the surrounding environment. In order to provide a globally consistent view of the entire environment, agents need to share their information with other team members. For instance, a UAV agent, while collecting images of part of the operational environment, may need to send them to a cloud service for additional computation-intensive processing in order to extract the locations of potential victims. These victim locations can then be sent to Command and Control (C2) centers and used in the planning of rescue relief efforts.

The data collected during a rescue operation ranges from low-level raw sensor data, such as images from camera sensors, to high-level semantic data, such as the location and status of victims. When considering sharing of information, these different types of data have different properties and requirements. The low-level raw sensor data usually has a very large volume and is acquired at high frequency, while the volume of symbolic data is relatively small and has a lower update rate. In this paper, the focus is on information with a symbolic representation that can be stored as Resource Description Framework (RDF) Graphs. This is one of the components of a larger infrastructural multi-agent-based knowledge framework being developed to leverage the use of heterogeneous teams of human agents and autonomous robotic platforms [29], which also covers bandwidth-intensive low-level data exchange.

The purpose of the RGS$^{\oplus}$ System presented in this paper is to ensure that all agents in a team have access to identical RDF Graphs for selected knowledge published by each individual agent. Consequently, after the synchronization process, all team agents have the same view of collective high-level knowledge. Each agent can then use this knowledge for reasoning and decision-making processes, allowing them to accomplish their goals. Examples of shared common knowledge in collaborative robotics within the emergency rescue domain include locations of potential victims or specifications of agents' capabilities.

The design of the proposed RGS$^{\oplus}$ System has been driven by constraints and practical requirements of in-the-field search and rescue operations. In this context, teams of heterogeneous agents are deployed in challenging environments where one of the critical problems is providing a reliable means of communication for data exchange and control over mission execution. In many other applications, deployed robotic platforms can rely on pre-existing communication infrastructures, which is not the case in search and rescue operations. In typical after-disaster circumstances, many critical infrastructures, such as communication [24], are not fully operational or not reliable and may have limited bandwidth. Thus, the RDF

Graph synchronization approach requires two fundamental properties to account for communication constraints. First, the RGS process needs to be fully automatic and cannot rely on any human intervention. Second, the system has to handle situations where the set of agents participating in the synchronization process can change over time due to communication interruptions or as agents join or leave a mission.

To account for these two properties, the RGS$^{\oplus}$ System has been developed. It is decentralized and builds and retains a history of changes in RDF graphs with cryptographic authentication of authorship. The decentralized approach relies on the dynamic selection of a master node among a set of available agents. The approach allows for efficient synchronization among a team of agents while propagating knowledge promptly, allowing for frequent updates to RDF Graphs. It also provides a general platform for future extensions, for instance, reasoning about the source of information to address issues of trust as well as providing a means to allow for selective access control to knowledge.

1.1 SymbiCloud HFKN framework

The work presented in this paper is part of a broader framework: the *SymbiCloud HFKN Framework* [29], which includes a data/knowledge management infrastructure that is intended to be used to support distributed, collaborative collection of data and knowledge and its shared use in multi-agent systems. In this framework, each agent is assumed to have a *SymbiCloud module* (SCModule) containing its local or contextual perspective of its operational environment. Additionally, it is assumed that each robotic agent uses a communication framework such as the Robot Operating System (ROS) [46].[1] Agents are also capable of generating, allocating, and executing tasks using a delegation-based framework for collaborative robotics [26–28]. The delegation framework is used to specify and generate complex composite tasks among teams of robotic and human agents, such as exploration and mapping. In order to acquire situational awareness within the operational environment, individual results of exploration missions need to be synchronized among all agents. This process involves sharing both low-level sensor data as well as high-level symbolic knowledge represented as a collection of RDF Graphs.

In order to represent high-level semantic information, *SCModules* use the Resource Description Framework (RDF) [34]. RDF provides a representational foundation for the Semantic Web and was developed within the World Wide Web Consortium (W3C). The RDF framework has a very simple, but powerful set of representational primitives. Most important are RDF Triples consisting of a $subject$, $predicate$ and $object$. $Subject$ and $object$ are nodes in an RDF Triple and $predicate$ is a label for the edge connecting them that expresses a relationship between the $subject$ and the $object$. $Subject$ and $predicate$ are specified using Internationalized Resource Identifiers [30] (IRIs), that provide a unique identity to any subject or predicate, while $object$ can have any value, including an IRI.

It is useful to view RDF Triples graphically (see Fig. 1), since collections of such interconnected triples can be conceptually viewed as RDF (knowledge) Graphs. Logically, an RDF Triple ($subject$, $predicate$, $object$) corresponds to an atomic formula, $predicate(subject, object)$. This correspondence is very powerful since it provides a formal semantic interpretation of collections of RDF Triples as logical theories. There are many extensions to the RDF specification such as RDF Schema (RDFS) [15] and the Web Ontology Language OWL [39] which extends RDFS and is used for specification and reasoning about ontologies. These extensions are leveraged within the HFKN Framework.

[1] https://www.ros.org.

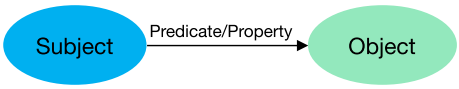


Fig. 1 Components in an RDF triple

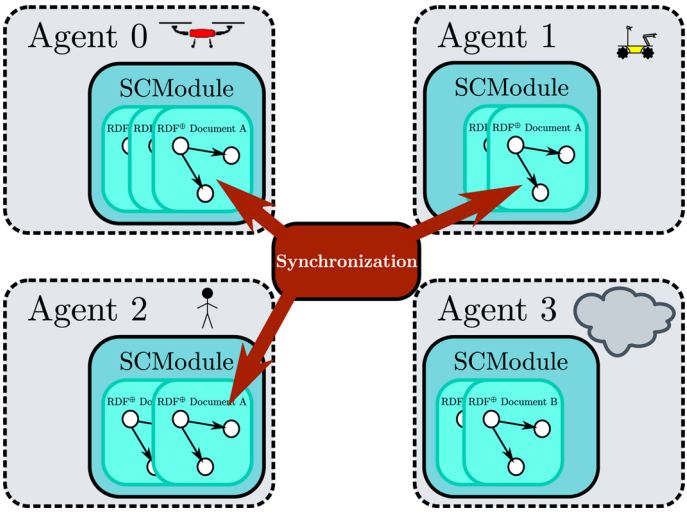


Fig. 2 RDF document synchronization between four agents

Although a collection of RDF Triples can be viewed as an RDF Graph, such graphs are encoded as RDF Documents. Consequently, we use the terms RDF Document and RDF Graph more or less interchangeably in the paper. Later, we will define an extended version of an RDF Document, denoted RDF$^{\oplus}$ Document, that is given as input to the RDF Graph synchronization algorithms used to synchronize shared RDF Documents between agents. A common way to store and access content in an RDF Graph is to store the triples in a SQL database, create appropriate RDF Views for the content, and then use SPARQL (SPARQL Protocol and RDF Query Language) [50] together with these RDF Views to query RDF Graphs.

In the HFKN Framework, the RGS$^{\oplus}$ System is used for automatic knowledge synchronization among agents, as shown in Fig. 2. Agents advertise the availability of the knowledge stored locally as RDF Graphs to the RGS$^{\oplus}$ System, which synchronizes the advertised graphs automatically among the interested agents. Agents have to explicitly decide to subscribe to advertised knowledge graphs that they are interested in using or contributing to.

While the RGS$^{\oplus}$ System has been integrated with the HFKN Framework and uses the framework in the presented field-test experimentation, the proposed synchronization algorithms are not specific to the HFKN Framework. They can be used independently in different contexts with different frameworks.

1.2 Contributions and content

This paper aims to address the problem of automatic synchronization of semantic knowledge between a group of autonomous agents, under the following realistic operational constraints:

– The synchronization process should be running without any need for intervention by a human operator.

– The system should be to handle a dynamic team of agents and should be robust with respect to unreliable communication.

To address those issues, the paper includes the following contributions:

– Improvement to state-of-the-art algorithms for handling revisions of RDF documents.
– A protocol that allows for the automatic distribution and sharing of information and knowledge between agents through the synchronization of RDF Documents/Graphs. A key feature of the protocol is that it does not rely on known centralized authorities.
– A synchronization mechanism that handles unreliable communication and a dynamic population of agents, with agents joining and leaving the system.
– An empirical evaluation of the graph synchronization algorithms, which includes (1) a complexity analysis, (2) an empirical validation of that analysis, and (3) a simulation of the behavior of the system in realistic circumstances.
– Integration of the work presented in the paper on actual systems, including a demonstration in field robotics under realistic conditions.
– The implementation of the proposed algorithms and execution of the presented benchmarks has been released as an open-source project [21].

The paper is structured as follows. In Sect. 3, the main algorithm and protocol for distributed data and knowledge synchronization used by the HFKN Framework is presented. In Sect. 4, a number of validation experiments related to the synchronization are provided. Sections 5 and 6 provide a description of related work and conclusions, respectively.

2 Delimitations

Consistency The RGS$^{\oplus}$ System presented in this paper is concerned with guaranteeing that the RDF Documents shared between agents converge towards containing the same set of RDF Triples. The RGS$^{\oplus}$ System does not consider the issue of the logical (i.e. semantic) consistency of RDF Graphs. Thus, the knowledge represented by the RDF Graphs can contain inconsistent and contradictory information, even though the RDF Graphs are guaranteed to be identical when the synchronization process is finished.

Generally, graph inconsistencies can be caused by the generation of statements that are contradictory or in violation of given RDF constraints. For example, two agents observing the same physical object in the environment (*object A*) may classify it differently based on their point of view or the sophistication of their sensors and perception functionalities. Thus the graphs may contain two contradictory statements (i.e. *object A* is a *table* and a *chair*).

While the RDF constraints can be checked using existing tools (e.g. SHACL [33]), solving the semantic consistency problem, in general, is more complex and will be part of future work where we envision two alternative solutions. The first approach is to use an underlying RDF model where inconsistencies cannot happen. For example, instead of stating that an object is of a certain type or has a certain property, we record the information as an observation (i.e. *agent A* observes that *object A* is a *table* and *agent B* observes that *object A* is a *chair*). The observations are contradictory, and the decision on which one is correct is left to be resolved by other means than the graph synchronization process (e.g. with the help of a human operator or by using a probabilistic approach where each observation is annotated with an uncertainty measure).

An alternative approach is to correct the inconsistencies during the synchronization process. This could be achieved by integrating repair algorithms into the synchronization process

(e.g. using a combination of SHACL and Answer Set Programming [1]). A short discussion of the necessary changes is presented in Sect. 3.5.

Communication network Communication links established between agents are assumed to be transitive in the following manner. Given agents a, b, and c, it is assumed that if agent b can communicate with agents a and c, both agents a and c can also communicate with each other. This is a fair assumption if we consider networks based on the TCP/IP protocol, as commonly used with 802.11x wireless networks, where all the agents are connected to the networking infrastructure (i.e. wireless routers) with common routing tables. In the case of other types of network infrastructures (e.g. Zigbee), additional care has to be taken to assure transitiveness by choosing an appropriate communication protocol.

3 RDF$^{\oplus}$ graph synchronization (RGS$^{\oplus}$)

Agents use RDF Graphs to store various types of information, such as agent capabilities, list of salient objects, metadata about datasets, general knowledge useful for decision making and planning, etc. Throughout a mission, agents make changes to these RDF Graphs. For instance, while collecting sensor data and processing it, an agent will need to update the list of salient objects. Achieving a common view of a shared RDF Graph between agents by simply exchanging it in its entirety is infeasible and impractical for several reasons. First, it is inefficient when exchanging all of its content at all times. Moreover, agents may join and leave throughout the execution of a task or a mission. Communication between agents is also unreliable by nature since we deal with mobile robots and wireless communication links. This may result in partial transmissions or receiving incomplete information. Most importantly, a simple exchange of an entire set of RDF Graphs does not solve the consistency problem when multiple agents sharing the same RDF Graph make simultaneous updates. Instead of communicating the whole RDF Graph content after every change, the RGS$^{\oplus}$ System encodes changes as differences representing incremental modifications of an RDF Graph. These differences are then used more efficiently in the synchronization processes.

It is assumed that each agent has a dedicated RDF Document containing information about those RDF Graphs it will make publicly accessible. This information can of course be updated throughout an operational mission. As mentioned previously, a publish/subscribe mechanism is built into the system that allows agents to both publish (share) and subscribe to public documents. If agent A subscribes to an RDF Document from agent B, then that document becomes shared between them and can then be updated by both. Prior to this, the document can only be updated by Agent A (if there are no other agents that have already subscribed to this document). This generalizes to teams of agents collectively sharing and thus being able to update collections of RDF Graphs.

The main problem then becomes how to guarantee that this collective sharing of RDF Graphs stays synchronized (consistent) across all agents involved, while updates are made locally and concurrently by these agents. As mentioned, the added difficulty is due to the fact that there can be communication breakdowns among agents and that agents both enter and leave the operational environment.

The approach used here is inspired by software systems used for code-versioning where new data is incremental and forms a graph in which new versions are nodes (called revisions). Incremental changes are differences between subsequent revisions (called deltas) and are the edges between revisions. This approach allows for recreating complete instances of the versioned information by starting at a specific revision, backtracking recursively through

the parents to the initial revision, and finally applying all the deltas sequentially in a forward fashion. While in the traditional application, versioning is performed when needed as determined by the programmer, in the RGS$^{\oplus}$ System, we are interested in continuous synchronization as soon as new information is available.

The RGS$^{\oplus}$ System is directly inspired by traditional code-versioning systems, such as Git. Instead of representing deltas between two revisions as a text difference, in the RGS$^{\oplus}$ System, deltas representing differences between two RDF Graph revisions are computed at the syntactic structure level rather than at the encoding level. Deltas describe the evolution of an RDF graph in the form of added and/or deleted RDF Triples that are to be shared among all participating and interested agents. Using such an approach, the system can recognize the difference between two versions of an RDF Graph independently of its encoding. The basic concepts of revision management, such as merge and rebase used in the RGS$^{\oplus}$ System, are directly inspired by those available in code-versioning systems. Both concepts have been adapted to handle computing of deltas at a syntactic structure (RDF Triple) level.

Synchronization of RDF Graphs among agents in the RGS$^{\oplus}$ System is achieved by exchanging messages with the goal of having a common and consistent view of the respective shared RDF Graphs. There are several assumptions and properties concerning participating agents, their roles, communication capabilities, and exchanged messages. The following assumptions exist regarding the operation of agents:

- Agents operate asynchronously.
- Agents may experience failures to operate or communicate.
- Agents store data locally in permanent storages.
- Agents operate in good faith and are truthful.
- Agents use synchronized clocks.
- Agents are aware of other agents' existence based on periodically exchanged status messages.

Agents can take one or two roles in the RDF Graph synchronization mechanism:

- Each agent has a role of a standard agent that is interested in providing or accessing knowledge.
- One agent has an additional role of a merge master awarded in an election process if two or more agents. The merge master is part of the RDF Graph synchronization mechanism and is responsible for integrating all the changes from the other agents into the common view. There can be different merge masters for different documents.

Synchronizing RDF Graphs among a team of agents assuming the above-listed properties requires that each agent interested in a team's shared common knowledge performs several tasks described in Sect. 3.2. The different algorithms allow for synchronizing different revisions of RDF Documents and handling the exchange of necessary information required for synchronization. The responsibilities of a merge master and the process of electing the master among the agents are detailed in Sect. 3.2.4.

Agents cooperate by exchanging messages with the following properties:

- Messages are sent asynchronously and without guarantees on arrival time bounds.
- Messages can be lost, arrive out of order, or be duplicated.
- Received messages are not corrupted.
- Agents exchange four types of messages: `Status`, `Revisions-request`, `Revision`, `Vote`.

These assumptions stem mainly from the fact that the communication between agents is achieved using wireless links, which are inherently unreliable. Moreover, partially received

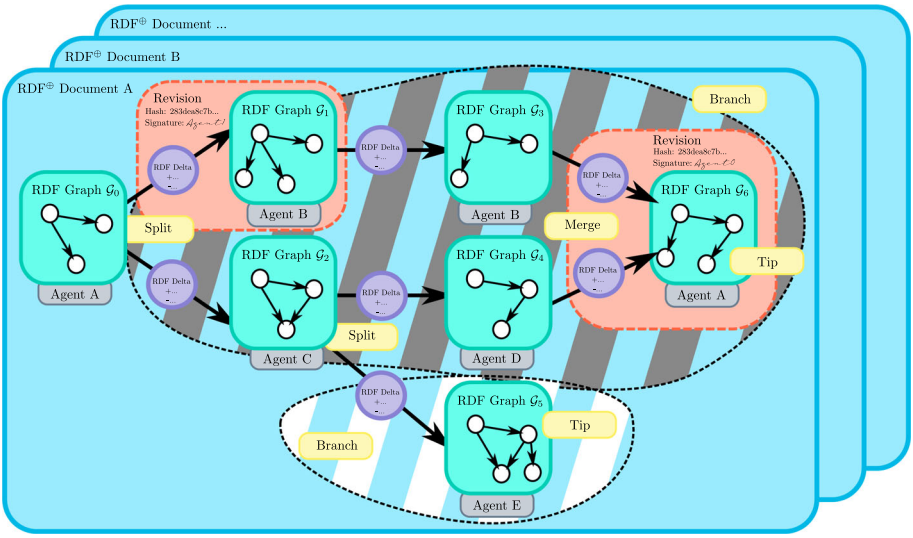


Fig. 3 Terminology and concepts in the RGS$^{\oplus}$ System. An agent can store multiple RDF$^{\oplus}$ Documents, for instance, to store capabilities, information about situation awareness, etc.

or corrupted messages are ignored upon detection. A detailed description of the four types of exchanged messages is presented in Sect. 3.1.5.

In the remainder of this section, necessary terminology is defined, followed by a description of the method used for RDF Document synchronization among agents in Sect. 3.2. Detailed descriptions of two major algorithmic components of the proposed solution, namely *merge* and *rebase* algorithms, are then presented in Sects. 3.3 and 3.4, respectively.

3.1 Terminology and definitions

In this section, we consider both terminology and definitions used in the RDF Graph Synchronization processes.

3.1.1 RDF$^{\oplus}$ document

In [20], an RDF Document is defined as an encoding of an RDF Graph. To facilitate RDF Graph synchronization, the concept of an RDF$^{\oplus}$Document is introduced. It is defined as the current instance of an RDF Graph and its history of changes. This history is represented using a *Graph of Revisions*, where vertices represent particular *revisions* of an RDF Graph and edges represent incremental differences (i.e. *deltas*) between revisions. Intuitively an RDF$^{\oplus}$Document can be viewed as a checked-out Git repository, where both the latest version is available as well as the history of revisions, in the `.git` folder. This allows one to use the RDF infrastructure to reason about RDF$^{\oplus}$Documents (e.g. SPARQL, OWL...) and to build a synchronization mechanism.

Figure 3 presents a schematic of an example RDF$^{\oplus}$Document. It depicts the main terms and concepts, as well as relations between them. At any given time, there exist several RDF$^{\oplus}$Documents (here: A and B) which need to stay synchronized between interested agents. A specific RDF$^{\oplus}$Document encodes an RDF Graph and its history in the form of incremental

changes expressed using RDF Deltas, which are parts of Revisions (highlighted by red dashed lines in Fig. 3), a formal definition of the deltas is given in Sect. 3.1.2.

The following content defines a *revision*:

- A list of RDF Deltas associated with parent revisions - $\Delta_{i,j}$, where i and j are hashes (see below) of the parent and this revision, respectively. In the RGS$^{\oplus}$System, each revision has one or two parents (see Fig. 3). Root revision is an exception and has no parents.
- The Universally Unique Identifier (UUID) of the author of the revision.
- The timestamp corresponding to the time when the revision was created.[2]
- A hash used to uniquely identify the revision, and computed over all its content using the SHA-512 algorithm [25] as follows[3]:

$$SHA_{512}(uuid(author), timestamp, \bigcup SHA_{512}(\Delta_{parent,revision}, uuid(parent))) \tag{3.1}$$

- A cryptographic signature computed over the hash using the RSA algorithm [8].

The combined set of revisions forms a Graph of Revisions (GoR). The cryptographic signature will be used in a future version of the RGS$^{\oplus}$System, currently it is assumed that all agents are trustworthy. Nevertheless, it is included in preparation for extending the framework to handle the trust level between agents. The root revision node in any GoR is defined as a *null* revision. Specifically, the list of RDF Deltas is empty (i.e. it has no parents), and the values of the author's UUID, timestamp, and signature are null.

Figure 3 shows an example of GoR, there are seven RDF Graph versions created by five agents (Agent A...E) at different times during a mission execution. The history of changes of an RDF$^{\oplus}$Document represented as Nodes in a GoR represent revisions, while edges represent deltas. The leaf nodes in the graph, that is the latest versions from the agents' perspectives, are called *tips*. The set of tips is denoted as $\mathcal{H}$, in the figure, $\mathcal{H} = (\mathcal{G}_5, \mathcal{G}_6)$. There can be multiple tips because agents can make changes in parallel to an RDF$^{\oplus}$ document. This also causes splits ($\mathcal{G}_0$ and $\mathcal{G}_2$) in the graph and can lead to the existence of *branches*. Two example branches $\{\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3, \mathcal{G}_4, \mathcal{G}_6\}$ and $\{\mathcal{G}_5\}$ are marked with striped areas in the figure. To achieve a common view of the RDF Graph, it is necessary to combine the tips, which can be done by two operations *merge* or *rebase*, which are considered in Sect. 3.1.3. In Fig. 3 a result of a merge between two revisions $\mathcal{G}_3$ and $\mathcal{G}_4$ results in a new tip revision $\mathcal{G}_6$.

Using an RDF$^{\oplus}$Document, it is possible to instantiate all past versions of the RDF Graph associated with it, including the latest version (i.e. tip), which should be in sync between all collaborating agents that share the document. This structure allows for implementing an efficient mechanism for creating, updating, and synchronizing RDF Graphs shared among a team of agents. Multiple agents can work in parallel on the same RDF Graph, and new agents can obtain the newest revision of an RDF Graph created by other agents.

Revisions can be "shared" between agents or they can be "local" in which case the revision is only available on a single agent and has not been exchanged yet with other agents. Shared revisions are immutable, meaning that the deltas, parents, author, and timestamp cannot change after publication. Before publication, local revisions can be moved to a different branch, using the rebase operation, which means changing the hash and parent of a revision.

[2] Expressed as Unix time.

[3] We use the SHA-512 of the delta for convenience, as deltas and revisions are stored separately in the database and associating a hash for the deltas allows more robust indexing.

3.1.2 RDF delta

There have been several proposals for efficiently computing the differences between two RDF/Schema Knowledge Bases [61] or RDF Graphs [10, 60]. These proposals are based on syntactic, semantic, or combinations of syntactic and semantic characteristics of the RDF Graphs in question. The approach used in this paper is based on one of the more straightforward definitions of deltas defined in terms of a plain set-theoretic semantics based on two operations of *insertion* and *deletion* of RDF Triples. Other approaches in the literature could be used that provide more efficient deltas without changing the basic operations of the RGS$^{\oplus}$ System, but this is saved for future investigation.

An RDF Delta (Δ) is an expression of the difference between two individual RDF Graphs (or two versions of the same RDF Graph). Given two RDF Graphs $\mathcal{G}_i$ and $\mathcal{G}_j$, the difference can be summarized with the list of *inserted* triples $\mathcal{I}_{i,j} = \{...(s_k^I, p_k^I, o_k^I)...\}$ and the list of *removed* triples $\mathcal{R}_{i,j} = \{...(s_k^R, p_k^R, o_k^R)...\}$, such that:

$$\mathcal{G}_j = (\mathcal{G}_i \setminus \mathcal{R}_{i,j}) \cup \mathcal{I}_{i,j} \tag{3.2}$$

$$\mathcal{R}_{i,j} = \mathcal{G}_i \setminus \mathcal{G}_j \tag{3.3}$$

$$\mathcal{I}_{i,j} = \mathcal{G}_j \setminus \mathcal{G}_i \tag{3.4}$$

The *removed* and *inserted* lists are swapped when switching the order of graph indices i and j such that:

$$\mathcal{R}_{j,i} = \mathcal{I}_{i,j} \tag{3.5}$$

$$\mathcal{I}_{j,i} = \mathcal{R}_{i,j} \tag{3.6}$$

A *delta* between $\mathcal{G}_i$ and $\mathcal{G}_j$ is defined as:

$$\Delta_{i,j} = (\mathcal{I}_{i,j}, \mathcal{R}_{i,j}) \tag{3.7}$$

RDF Deltas are encoded using a subset of SPARQL Update query [50]. Only `INSERT DATA` and `DELETE DATA` queries are allowed, because they are the only two types of queries to unambiguously identify the inserted and removed triples. RDF Blank Nodes[4] are encoded using the *skolemization* process [38], i.e. blank nodes are replaced by a unique IRI assigned by the KDB Manager which is part of the HFKN framwork.

An example of an RDF Delta with one triple being added and one removed is encoded as a SPARQL Update query in the following way:

```
PREFIX ex: <http://example.org/>

INSERT DATA {
 ex:a ex:b ex:c
}
DELETE DATA {
 ex:d ex:e ex:f
}
```

[4] Blank nodes can be used to identify an element of an RDF Triple without providing an explicit IRI.

where `ex:a`,..., `ex:f` are shortened IRIs corresponding to `http://example.org/a`,..., `http://example.org/f`. In this delta, the triple `ex:a ex:b ex:c` is added and the triple `ex:d ex:e ex:f` is removed.

3.1.3 Combining branches: merge/rebase overview

Agents may create changes to one RDF$^{\oplus}$ Document concurrently, either because they make a change simultaneously or because they are out of communication range. When this happens, the document will have several branches (i.e. tips) that need to be combined into a single one. This is depicted in Fig. 3 as a split and merge.

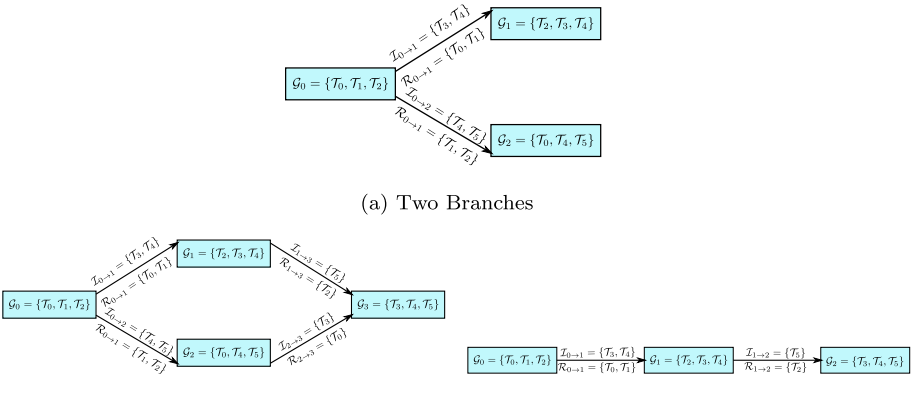
In this work, two approaches for combining RDF Graph branches are proposed: *merge*, which creates a new combined revision based on deltas of two other revisions, and *rebase*, which moves the revisions from one branch to another. Merge is more general and is always applicable, while rebase can only be applied if the revisions are still *local*, which means they have not been published yet.

Illustrative examples of combining revisions using the two proposed techniques are presented in Fig. 4, where three agents: A, B, and C, create new revisions at different time points. An example scenario considering a merge procedure is depicted in Fig. 4a, b. Agent A is responsible for performing the merge operation. At first (i.e. time $t-1$), the document contains the triples $\mathcal{T}_0$, $\mathcal{T}_1$, $\mathcal{T}_2$, which correspond to revision $\mathcal{G}_0 = \{\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2\}$. Next, at time t, Agent B adds triples $\mathcal{T}_3$ and $\mathcal{T}_4$ and removes $\mathcal{T}_0$ and $\mathcal{T}_1$. This constitutes revision $\mathcal{G}_1$, $\mathcal{I}_{0\to1} = \{\mathcal{T}_3, \mathcal{T}_4\}$ and $\mathcal{R}_{0\to1} = \{\mathcal{T}_0, \mathcal{T}_1\}$, which corresponds to revision $\mathcal{G}_1 = \{\mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_4\}$.

Concurrently, at time t, Agent C adds triples $\mathcal{T}_4$ and $\mathcal{T}_5$ and removes $\mathcal{T}_1$ and $\mathcal{T}_2$. This constitutes revision $\mathcal{G}_2$, $\mathcal{I}_{0\to2} = \{\mathcal{T}_4, \mathcal{T}_5\}$ and $\mathcal{R}_{0\to2} = \{\mathcal{T}_1, \mathcal{T}_2\}$, which correspond to revision $\mathcal{G}_2 = \{\mathcal{T}_0, \mathcal{T}_4, \mathcal{T}_5\}$. The resulting graph of revisions is shown in Fig. 4a.

Finally, at time $t+1$, Agent A receives the deltas $(\mathcal{I}_{0\to1}, \mathcal{R}_{0\to1})$ and $(\mathcal{I}_{0\to2}, \mathcal{R}_{0\to2})$ from Agent B and Agent C, respectively. It then creates a single revision $\mathcal{G}_3 = \{\mathcal{T}_3, \mathcal{T}_4, \mathcal{T}_5\}$, with two deltas by applying the merge operation, which will be described in Sect. 3.3:

- $\mathcal{I}_{1\to3} = \{\mathcal{T}_5\}$ $\mathcal{R}_{1\to3} = \{\mathcal{T}_2\}$
- $\mathcal{I}_{2\to3} = \{\mathcal{T}_3\}$ $\mathcal{R}_{1\to3} = \{\mathcal{T}_0\}$



(a) Two Branches

(b) After Merge

(c) After Rebase

Fig. 4 Example graphs of revisions before and after applying merge and rebase procedures. $\mathcal{G}$, $\mathcal{I}$ and $\mathcal{R}$ denote a revision, inserted and removed triples, respectively

The structure of the resulting revision $\mathcal{G}_3$ is shown in Fig. 4b. The delta generated by the merge operations are minimal. Triples, that are inserted in both branches, are not part of the merge deltas.

The alternative *rebase* process is illustrated in Fig. 4c. At time t, Agent C creates revision $\mathcal{G}_2$ locally (i.e. does not publish it). Next, at time $t + 1$, the agent receives a new revision $\mathcal{G}_1$ from Agent B. Agent C then can rebase $\mathcal{G}_2$ on top of $\mathcal{G}_1$ and create a revision such that $\mathcal{I}_{1\to 2} = \{\mathcal{T}_5\}$ $\mathcal{R}_{1\to 2} = \{\mathcal{T}_2\}$. The structure of the revisions is shown in Fig. 4c.

In summary, the RGS$^{\oplus}$ System uses two techniques for combining changes in RDF$^{\oplus}$ Documents:

- *merge* combines two branches by creating a new revision with two RDF Deltas. The advantage of *merge* is that it can be applied to any branch, and the new revision can be synchronized with other agents. The drawback is that it creates a new revision, which makes the revision graph more complex. Additionally, if only merge is used, and it is not performed in a timely fashion, the global synchronization between all agents may not be guaranteed in certain scenarios due to practical limitations (discussed in Sect. 3.1.4).
- *rebase* moves a branch on top of another branch, resulting in a single branch with all the revisions. The advantage is that it reduces the complexity of the revision graph and minimizes the number of revisions. In the RGS$^{\oplus}$ System, we limit the use of rebase only to local revisions to avoid the need for additional functionalities to guarantee that identical rebase operations are performed by all agents.

Further details of merge and rebase algorithms are presented in Sects. 3.3 and 3.4, respectively. The RGS$^{\oplus}$ System uses a combination of merge and rebase operations in order to guarantee a global synchronization of RDF$^{\oplus}$ Documents among all agents. A detailed discussion of the interactions between the two algorithms and their use is provided in the following subsection.

3.1.4 Synchronization protocol

Merge and rebase are two possible approaches for combining the changes in the RDF Graph made by different agents. The Synchronization Protocol describes how these two approaches are used to achieve a common view of an RDF$^{\oplus}$ Document.

A naive approach in which all agents individually apply only merge operations as soon as they have more than one tip in their GoRs would result in a never-ending merge loop creating an infinite number of new merge revisions. To solve this problem, distributed code-versioning systems rely on one or several central authorities. Whenever a participant has a new version of their code that should be shared with other participants, the participant makes sure they are synchronized and consistent with at least one of the central authorities.

This approach alone only works if the rate of changes of the RDF$^{\oplus}$ Document is low, as each agent has to have enough time to ensure its GoR is consistent with the central authority.[5] In the RGS$^{\oplus}$ System, agents can make changes at a much higher rate, up to several times per second. Agents may not have time to ensure consistency of their GoR with the central authority before publishing their changes. Thus, in the RGS$^{\oplus}$ System, agents publish their changes without ensuring consistency, and merging the changes is the responsibility of a central authority. This central authority is called the *Merge Master*. Although, the concept of central authority is directly inspired by distributed code-versioning systems, its use in the

[5] For reference, an active open source project such as Linux kernel has an average of less than ten commits per hour.

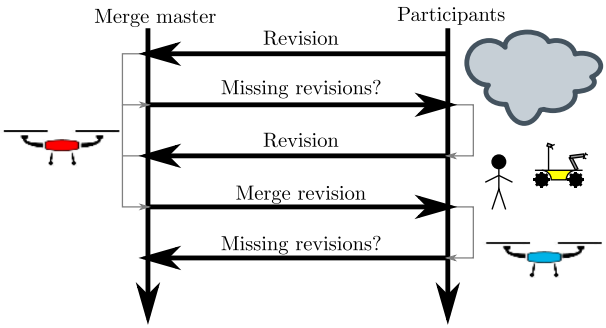


Fig. 5 Protocol for graph synchronization

RGS$^{\oplus}$ framework is novel. In the code-versioning systems, the central authority is static, while in the RGS$^{\oplus}$ System, the role of the merge master is assigned dynamically in the process of election (Sect. 3.2.4). Additionally, the merge master integrates changes introduced by other agents, unlike in traditional version control systems. Each agent can be selected as the merge master at any time, and the election is performed among all agents within the communication range. This is done for redundancy and reliability reasons to deal with dynamic mission scenarios where agents join and leave missions, as well as to deal with limitations of wireless communication links, such as disconnections. In case the set of agents is divided into several partitions that cannot communicate with each other, one merge master gets elected per partition. This is discussed in more details in Sect. 3.2.4.

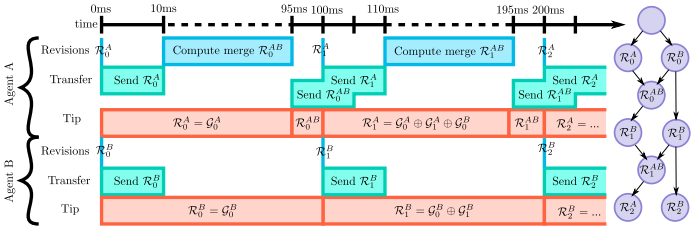
An overview of the synchronization protocol is shown in Fig. 5. When an agent makes a change, it broadcasts its new revision. On reception of the new revision, the merge master checks that the merge operation can be performed. This operation can be performed if the GoR of the merge master has all the parent revisions of the newly received revision. If this is not the case, the merge master sends a request to the other merge agents for any missing revisions. Once the merge master has received all the revisions, it can then compute a merge revision by applying the merge algorithm. The resulting new merged revision is then broadcasted to other agents. When an agent receives the merge revision, it checks if its GoR has all the parent revisions so that the merge revision can be directly incorporated. In case any revisions are missing from its GoR, the agent will send a request for the missing revisions.

If agents create new revisions too quickly in parallel, there is a risk that the merge master cannot cope with the load. If that happens, the graph of revisions for each agent would include several branches which are never combined. Agents would only be able to make changes in their respective branches and a common view of the RDF Graph with all information from other agents would never be achieved. We call this the *Never Synchronized Problem*.

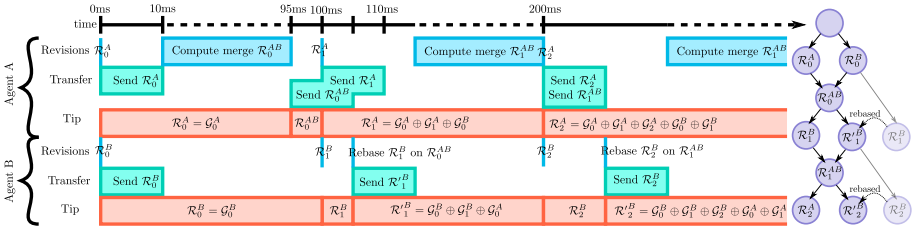
The Never Synchronized Problem Consider the following hypothetical scenario which includes two agents: A and B. Agent A is the merge master. Both agents create a new revision every 100 ms. It takes 10 ms to transmit revisions between agents and 85 ms for A to complete a merge.

The timeline of operations shown in Fig. 6a is as follows:

- At $t = 0$ ms, Agent A creates revision $\mathcal{R}_0^A$ and Agent B creates $\mathcal{R}_0^B$. The revisions are published.
- At $t = 10$ ms, Agent A receives $\mathcal{R}_0^B$ and initiates a merge.
- At $t = 95$ ms, Agent A completes the merge revision $\mathcal{R}_0^{AB}$ between $\mathcal{R}_0^A$ and $\mathcal{R}_0^B$.
- At $t = 100$ ms, Agent A creates revision $\mathcal{R}_1^A$ and Agent B creates $\mathcal{R}_1^B$.



(a) Only merge



(b) Merge and rebase

Fig. 6 Agent A is the merge master. The first line (Revisions) for each agent shows the revisions that are created. $\mathcal{R}_i^A$ and $\mathcal{R}_i^B$ are created when the agents change the document, while $\mathcal{R}_i^{AB}$ represents a merge. The second line (Communication) represents the revisions that are sent by each agent and the delay in communication. The last line (tip) represents the current tip for each agent and the content of that tip. The final GoR is shown on the right side of the figure. As shown in (**a**), Agent B does not receive the changes created by Agent A before creating its new local revision

- At $t = 105$ ms, Agent B receives $\mathcal{R}_0^{AB}$.
- At $t = 110$ ms, Agent A receives $\mathcal{R}_1^B$ and initiates a merge.
- ...

As can be seen in this example, Agent A gets access to the changes of Agent B, but Agent B does not receive the merge information in time before creating its new revision. This results in Agent B not being able to incorporate the RDF knowledge graph changes created by Agent A.

Rebase local changes and Merge public revisions To avoid the Never Synchronized Problem, agents store local revisions for all the changes made until their GoRs are synchronized with the merge master. The following procedure, which interleaves the use of merge and rebase algorithms, is used.

- When an agent makes a change in an RDF Graph:
  - The agent checks if its current revision inherits from the current revision of the merge master (it is known based on the status message). If that is the case, the agent creates a new public revision and publishes it.
  - Otherwise, the agent creates a new local revision (i.e. the revision is not published).

  The details of this behavior are described in Sect. 3.2.1.
- When an agent receives a merge revision and if it has local revisions, the agent will apply the rebase algorithm using the local revisions and the latest available merge revision. This will result in a new revision that combines all changes that will be published. This is described in more detail in Sect. 3.2.2.

The timeline for the new behavior of the two agents is shown in Fig. 6b. As can be observed, both Agents A and B can make their local changes and access the other agent's changes. The timeline of operations now becomes as follows:

- At $t = 0$ ms, Agent A creates revision $\mathcal{R}_0^A$ and Agent B creates $\mathcal{R}_0^B$. They believe they are synchronized and publish their revisions.
- At $t = 10$ ms, Agent A receives $\mathcal{R}_0^B$ and initiates a merge. At that point, Agent A and B believe they are not synchronized anymore.
- At $t = 95$ ms, Agent A complete the merge revision $\mathcal{R}_0^{AB}$ between $\mathcal{R}_0^A$ and $\mathcal{R}_0^B$. At that point, Agent A has access to $\mathcal{R}_0^A$ and $\mathcal{R}_0^B$.
- At $t = 100$ ms, Agent A creates revision $\mathcal{R}_1^A$ and Agent B creates a local revision $\mathcal{R}_1^B$.
- At $t = 105$ ms, Agent B receives $\mathcal{R}_0^{AB}$. Agent B rebases $\mathcal{R}_1^B$ on top of $\mathcal{R}_0^{AB}$. Agent B publishes $\mathcal{R}_1^B$. At that point Agent B has access to $\mathcal{R}_0^A$, $\mathcal{R}_0^B$ and $\mathcal{R}_1^B$.
- At $t = 115$ ms, Agent A receives $\mathcal{R}_1^B$ and initiates a merge.
- ...

By combining the merge and rebase methods for managing branches in GoRs, it is possible to maintain the RDF Graph synchronization between agents. Further details of the two methods and their relations are described in Sect. 3.2.

3.1.5 Messages

All communication between agents allowing for maintaining a synchronized view of the RDF Graph is realized using the following messages:

- The `Status` message is used to discover agents subscribed to a document and that are within the communication range. It contains meta-information about the agent: its name, UUID, the public cryptographic key (RSA), and the state of the document. It includes the tip revision hash and whether the agent believes and acts as a merge master.
- The `Revision` message is used to exchange a revision between agents. It contains all the information needed to insert a new revision in the Graph of Revisions: author's agent UUID, revision delta(s), timestamp, revision hash (Eq. (3.1)), and a cryptographic signature.
- The `Revision-request` message is used to request a revision that is not available in an agent's Graph of Revisions. The message contains the requester's UUID and a list of the required revision hashes.
- The `Vote` message is used during the election of the merge master. It contains the voter's UUID, candidate's UUID, the election round number, the vote timestamp, and the election timestamp.

A *status* message is sent periodically by all agents and has three purposes. First, it allows for detecting the availability of other agents. Second, it allows an agent to know other agents' tip revisions. This information is used to detect if all the revisions have been received. Third, it allows for knowing which agent has the role of the merge master and if an election is necessary (i.e. no or multiple masters are present).

A *revision* message is sent when a change to an RDF$^{\oplus}$ Document is to be communicated to other agents or upon an explicit request. Such request would happen if an agent detect missing revisions with the help of the *status* message.

When an agent is aware of the existence of a revision through a status message or a parent of a known revision, it can use the *revision-request* message to obtain the missing information.

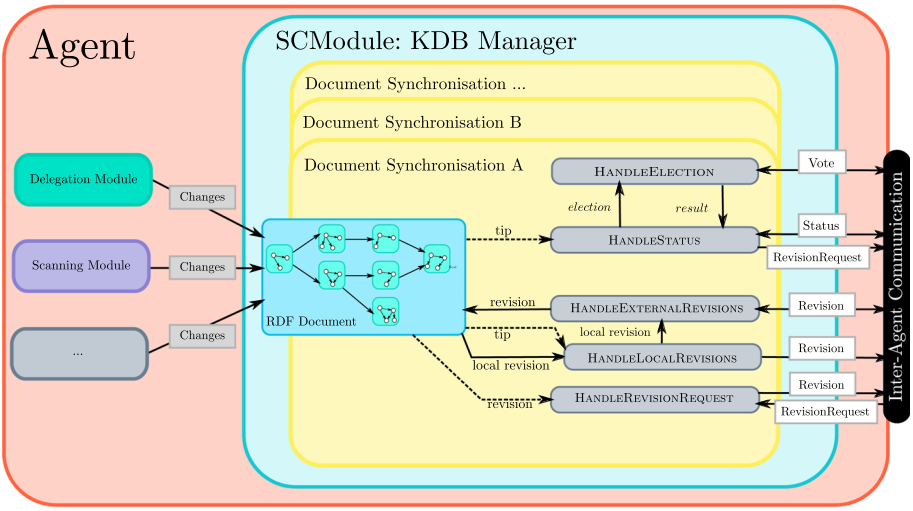


Fig. 7 Schematic view of the RDF$^{\oplus}$ Document synchronization mechanism and processes involved

A *vote* message allows for initiating a merge master election process and casting votes. More information is presented in Sect. 3.2.4.

3.2 RDF$^{\oplus}$ document synchronization

As outlined in previous sections, agents acquire new knowledge throughout a mission execution, and its high-level description is represented as RDF Graphs encoded with their history in RDF$^{\oplus}$ Documents. To achieve a common view of the continuously updated RDF$^{\oplus}$ Documents among all participating agents, the RGS$^{\oplus}$ System uses a number of algorithms and protocols that allow for efficient and robust RDF$^{\oplus}$ Document synchronization.

A schematic view of the processes, functionalities, and the exchanged messages is presented in Fig. 7. The view focuses on the perspective of one agent, which hosts several processes such as Delegation or Scanning Modules and, most importantly, from the perspective of this work, the KDB Manager. The manager receives data from these modules and updates its RDF$^{\oplus}$ Documents. For each document, the agent uses several processes (in gray) to handle RDF$^{\oplus}$ Document synchronization as well as communication with other agents by exchanging messages (in white) to maintain the common view of the RDF$^{\oplus}$ Document.

The following processes are used to achieve knowledge synchronization. Updates reflect the new information obtained by the agent itself to the RDF$^{\oplus}$ Document by creating a new *local* revision. The revision is either directly broadcasted to other agents or kept local. This decision is made by the *HandleLocalRevisions* process. Upon receiving revisions from other agents, the *HandleExternalRevisions* process is responsible for incorporating the new information into its version of the RDF$^{\oplus}$ Document. The *HandleRevisionRequest* process is responsible for providing revisions when other agents request them. The periodic sending of the status information as well as receiving the information from others is the responsibility of *HandleStatus* process. Finally, initiating and participating in the merge master election is handled by the *HandleElection* process. The processes and the algorithms they employ are described in the following subsections.

3.2.1 Handling local revisions

When an agent performs its tasks, the operation results often require updating the information contained in RDF$^{\oplus}$ Documents. Examples of this include obtaining new sensor data such as images or LIDAR data, specifying the agent's capabilities, battery level, to name a few. When this need arises, a new revision is created. The decision whether it should be directly published to other agents is taken according to Algorithm 1.

Algorithm 1: HANDLELOCALREVISIONS

```
   Input: G_new
1  foreach G_i ∈ G_new do
2      if G_mergemaster ∈ Ancestors(G_i) then
3          Publish(G_i)
4      else
5          AddToLocalRevisions(G_i)
```

The algorithm checks if the latest merge master's revision ($\mathcal{G}_{mergemaster}$), received from the merge master's status message, is one of the ancestors of the new revision ($\mathcal{G}_i$). The list of ancestors is computed by recursively listing all the parents of a given revision. If that is the case, it means the current changes are ahead of the merge master, and the new revision can be automatically published to other agents. Otherwise, $\mathcal{G}_i$ is added to a *list of local revisions* and will ultimately be published when the latest revision is received from the merge master, that is when they become synchronized.

3.2.2 Handling external revisions

When an external revision is received from another agent, it is incorporated into the appropriate RDF$^{\oplus}$ Document through an update to its Graph of Revisions (GoR). The process is performed according to Algorithm 2.

Upon receiving a new revision, $\mathcal{G}_{new}$, the agent verifies that its GoR contains the parent revision(s) of $\mathcal{G}_{new}$. If it does not, it requests that revision. If the revision is not already in the GoR, it is then inserted.

The algorithm continues in line 6 where the current tip revision $\mathcal{G}$ is obtained followed by handling any local revisions in line 7. If $\mathcal{G}_{new}$ is a merge revision (i.e. it has two RDF Deltas related to two parents) and a rebase operation can be performed (i.e. all revisions in $\mathcal{G}$ are local and $\mathcal{G}$ and $\mathcal{G}_{new}$ are connected in the agent's GoR) the following steps are performed. First, revisions $\mathcal{G}$ and $\mathcal{G}_{new}$ are combined by the rebase algorithm (Algorithm 8, Sect. 3.4). The resulting revision is then published to the other agents in line 8.

The algorithm continues in 9 with the tasks of the merge master. First, the agent waits until there is a need for merging of revisions, which means that there is more than one tip in the graph of revisions. The *MergeRevision* Algorithm 6 is applied to the first two tips in $\mathcal{H}$ and the result is then inserted in the graph of revisions and published using the *InsertRevision* and *PublishRevision* functions, respectively. The algorithm continues until all revisions are merged, that is the number of tips is one.

Algorithm 2: HANDLEEXTERNALREVISIONS

```
1  while true do
2      G_new ← receiveNewRevision()
3      unless HasRevision(parent(G_new)) then
4          RequestRevision(parent(G_new))
5      InsertRevision(G_new)
6      G ← CurrentTipRevision()
7      if IsMergeRevision(G_new) and CanRebase(G, G_new) then
8          PublishRevisions(RebaseRevisions(G, G_new))
9      if agentIsMergeMaster() then
10         H ← RetrieveTips()
11         while |H| ≠ 1 do
12             G_i ← H[0]
13             G_j ← H[1]
14             G_merge ← MergeRevision(G_i, G_j)
15             InsertRevision(G_merge)
16             PublishRevision(G_merge)
17             H ← RetrieveTips()
```

3.2.3 Handling status

All agents participating in the RDF$^{\oplus}$ Document synchronization process send their status information as well as receive the statuses from all other agents. The purpose of the status message as defined in Sect. 3.1.5 is threefold. First, it allows for detecting the existence and availability of other agents. Second, it enables the merge master to determine other agents' tip revisions and thus conclude whether it has all the latest revisions of an agent before performing the merge operation. Third, it is used to establish which agent has the role of the merge master and if an election is required (i.e. none or multiple masters present). The procedure for handling the status messages is presented in Algorithm 3.

Algorithm 3: HANDLESTATUS

```
1  while true do
2      foreach σ ← ReceiveStatus() do
3          Γ ← Update(Γ, σ)
4          unless HasRevision(RevisionOf(σ)) then
5              RequestRevision(RevisionOf(σ))
6      if MinUUID(Γ) = AgentUUID then
7          MM ← GetMergeMasters(Γ)
8          if |MM| ≠ 1 then
9              election ← true
10     PublishStatus(Γ)
```

When an agent receives a status message from another agent (σ in line 2) it first updates its own status Γ and checks in line 4 if its GoR contains the revision designated by σ. If it does not, the agent requests the missing revision in line 5.

Subsequently, after incorporating all the received status messages, the agent checks whether there is a need for a new merge master election. The check happens after updating the status to ensure the system reacts quickly to an incorrect number of merge masters. First, the agent checks if its UUID is the lowest among all available agents. This is done to minimize the number of initiated elections as only the agent with the lowest UUID can start the election process. Second, in line 8 the agent checks the number of agents which consider themselves merge masters ($\mathcal{MM}$) and if that number is different than one it initiates a new election in line 9. This only happens when none or more than one agent consider themselves and act as merge masters.

Finally, in line 10 the updated version of the agent's status Γ is used to publish the agent status message as defined in Sect. 3.1.5.

3.2.4 Merge master election

When multiple agents are connected, they select a single merge master as discussed in Sect. 3.1.3. An overview of the procedure each agent follows to handle an election is presented in Algorithm 4 and graphically depicted in Fig. 8.

Algorithm 4: HANDLEELECTION

```
1  while true do
2      V ← ReceiveVotes()
3      if |V| > 0 then
4          election ← true;
5      if election = true then
6          SendVote()
7          V ← V ∪ ReceiveVotesWithTimeout()
8          MM ← GetMergeMasters(V)
9          while |MM| ≠ 1 do
10             SendVote()
11             V ← ReceiveVotesWithTimeout()
12             MM ← GetMergeMasters(V);
13         Γ ← UpdateMergeMaster(Γ, MM)
14         election ← false
```

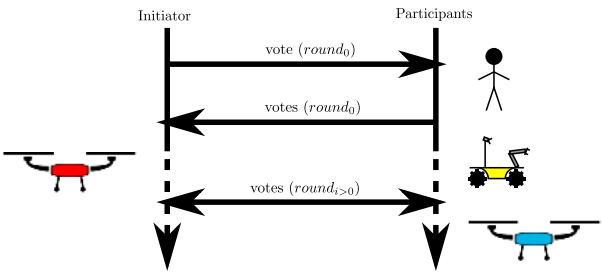


Fig. 8 Overview of a *merge master* election protocol. All agents elect a single master by casting votes. In case of ties, subsequent rounds of the election are performed

When an agent starts an election process as specified in line 9 of Algorithm 3, it publishes a vote message (Algorithm 4 line 6). Similarly, when an agent receives a vote, and is not in the election mode ($election = false$), then the agent knows that a new election has started (line 4) and its vote is published in line 6. The structure of the vote message is defined in Sect. 3.1.5 and the vote value is determined according to the following rules:

- pick the last agent for which the elector voted for to be a merge master if that agent is still connected, or
- pick the agent to which the elector has been connected to for the longest time.

Each agent receives the voting information (line 7), which can then be used to determine the unique merge master in a decentralized manner. To handle unreliability in communication, agents wait for a specific duration after the start of the election before they decide on the winner (line 7).

In the case of ties, a second stage of the election is necessary (i.e. resolution), in which only the agents with the most votes are considered. In the resolution stage, the agents vote (line 10) for a random candidate, and the procedure is repeated until a unique *merge master* is selected. With each new round of the election, the election round number ($round_i$) is incremented in the `Vote` message.

3.2.5 Requesting revisions

An agent can become aware of the existence of revisions through a status message from another agent, or by receiving a revision with parent revisions which does not exist in its GoR. In such a case, an agent can request the missing revision and complete its graph of revisions. This is performed in Algorithm 2 (line 4) or Algorithm 3 (line 5). Notice that this also happens when a new agent joins the mission or becomes reconnected after a temporary communication loss.

A simple strategy for handling responses to requests which would require an agent to respond to all requests individually (assuming it has the requested revision) would be highly inefficient. Since the requested information in many cases can be already stored by multiple agents, this would lead to sending an unnecessary amount of duplicated messages. Instead, to minimize the number of responses and exchanged messages, we propose to use the strategy presented in Algorithm 5.

Algorithm 5: HANDLEREVISIONREQUEST

```
1 while true do
2     foreach ρ ← ReceiveRevisonRequest() do
3         if Requester(ρ) = UUID(MM) then
4             if RevisionCreator(ρ) = UUID(self) or not
               ConnectionBetween(RevisionCreator(ρ), UUID(self)) then
5                 Publish(Revision(ρ))
6         else if UUID(self) = UUID(MM) then
7             Publish(Revision(ρ))
```

After an agent receives a revision request (line 2), it responds by publishing the requested revision if the following conditions hold. First, the request has come from the merge master

(line 3). Second, the agent is the creator of the requested revision, or the agent is not connected with the revision creator (line 4).

In case the request has come from an agent other than the merge master (line 6), the agent responds only if itself is the merge master. By using Algorithm 5, the RGS$^{\oplus}$ System reduces the number of messages exchanged while guaranteeing that the requested revisions are sent.

3.2.6 RGS$^{\oplus}$ synchronization resiliency analysis

To illustrate the resiliency of our approach, we present the behavior of the system when a deviation from the optimal operation conditions occur. Optimal conditions assume a fixed set of agents operating with reliable communication. The deviations can occur when new or existing agents join or leave the group, which can be caused by communication issues.

A new agent joins a mission and has no previous knowledge of a particular RDF$^{\oplus}$ Document A common scenario during any mission execution may involve a team of agents collectively acquiring sensor data and updating shared RDF$^{\oplus}$ Documents. When a new agent joins the team and has no previous knowledge of the particular shared RDF$^{\oplus}$ Document, its graph of revisions is empty. The new agent will receive the status message from the merge master (Algorithm 3 line 2) with the hash value of the latest revision. The new agent will then first request from the merge master the latest revision (Algorithm 3 line 5), and then recursively all the past revisions (Algorithm 2 line 4). The recursive request process will continue until the new agent has access to the complete GoR and therefore has the synchronized view of the shared RDF$^{\oplus}$ Document. Consequently, the agent can create an instance of the latest RDF Graph.

The initial synchronization process can be further sped up by introducing a new message where an agent can request from the merge master the entire GoR at once. This requires extending the synchronization protocol (Sect. 3.1.4), and will need to take into consideration unreliable communication. This will be considered in future work.

The merge master disappears without correctly publishing the last merged revision When that happens, the other agents will keep updating their RDF$^{\oplus}$ Documents locally by creating new local revisions (Algorithm 1 line 5). Since the merge master is no longer available (e.g. due to communication failure or hardware malfunction) its status message is not broadcasted anymore. And we end up in the next case when there is no merge master.

There is no merge master In this case, the synchronization process is suspended, which means that agents will not get the information from other agents. However, agents are still able to update their RDF$^{\oplus}$ Documents locally by creating new local revisions (Algorithm 1 line 5). In parallel, a new election is started by the agent with the lowest UUID (Algorithm 3 line 9). And after it is completed, the newly selected merge master will resume the process of merging revisions (Algorithm 2 line 9).

There are multiple merge masters The team of agents may become split into two or more groups due to communication range limitations. Each group, in that case, will have its own merge master elected. When communication between agents is re-established, all the agents will receive a status message from two or more merge masters (Algorithm 3 line 2), and the agent with the lowest UUID will trigger an election (Algorithm 3 line 9), the merge masters will complete their current merge but will not start new merge operations. After the election process is completed, a single merge master is selected and the synchronization process continues normally.

Intermittent communication interruptions Consider the following scenario involving four agents: A, B, C, and D. Agents A and B form *Group 0*, and agents C and D form *Group 1*. Agents within each group are assumed to have reliable communication. In this case, agents

A and B will always receive each other's status messages, and the same is true for agents C and D (Algorithm 3 line 2). Agents A and C, due to their UUID numbers, are always elected as merge masters in their respective groups.

Consider a case in which the communication between *Group 0* and *Group 1* is being constantly interrupted, i.e. very unstable, groups are disconnected and re-connected. When communication between groups is re-established, the agents will have two merge masters in the communication range, which will trigger an election (Algorithm 3 line 9), where either agent A or C is elected. When communication between groups breaks down, one group will still have a merge master, while the other will require a new election (Algorithm 3 line 9). If the communication links get constantly interrupted, there will be a situation where the merge master election process will be triggered repeatedly. This will slow down the synchronization process as merging of new changes is not performed during an election. However, the process of merging will resume immediately after the election process is completed (Algorithm 2 line 9).

Proof that merge operation occurs In this paragraph, we provide proof that under certain conditions, the knowledge shared among agents is eventually synchronized, that is, the merge operation is completed. For simplicity, let us consider a scenario in which two agents create new revisions in parallel. Assuming these agents remain in communication range, eventually, there is a time when the newly created revisions are merged. The scenario generalizes to any number of agents and can be described by the following theorem:

Theorem 3.1 *Two agents a_i and a_j have created parallel branches R_i and R_j, respectively. There exists a time t, at which at least one agent creates a new revision R_{ij} which has R_i and R_j as an ancestor, given that agents stay connected for at least $t \geq \delta_{enough}$.*

According to this theorem, as long as the two agents stay connected for δ_{enough} time, there is a guarantee that, eventually, a merge master will combine the two revisions. A lower bound of δ_{enough} is given by the number of merges and elections that may need to occur before merging R_i and R_j:

$$\delta_{enough} \geq M(\delta_{merge} + \delta_{election}) \tag{3.8}$$

where δ_{merge} and $\delta_{election}$ are the maximum times required for exchanging/merging the RDF deltas and the election process, respectively. M is the number of revisions that need to be merged before R_i and R_j. The time span δ_{enough} is an upper bound. In practice, however, this time can be shorter because the agents may have already exchanged their deltas, and the merge master can complete the merge even though the communication has been interrupted.

To prove Theorem 3.1, it is necessary to introduce a connectivity function c. Let us consider a set of n agents $A = \{a_1, \ldots, a_n\}$ and define a binary and symmetric connectivity function c as follows:

$$\begin{aligned} c(t, i, j) = 1 &\iff a_i \text{ is connected with } a_j \text{ at time } t \\ c(t, i, j) = 0 &\iff a_i \text{ is not connected with } a_j \text{ at time } t \\ c(t, i, j) &= c(t, j, i) \end{aligned} \tag{3.9}$$

The network transitivity assumption described in Sect. 2 implies that at a time t the function c is transitive. Given agents a_i, a_j, and a_k, if agents a_i and a_j are in the communication range (i.e. $c(t, i, j) = 1$) and agents a_j and a_k are also connected (i.e. $c(t, j, k) = 1$) then agents a_i and a_k can also communicate with each other (i.e. $c(t, i, k) = 1$). In function $c(t, i, j)$, the parameters i and j refer to agent a_i and a_j, respectively.

At any given time t, the set of agents A can be divided into subsets of agents A_l^t based on the connectivity function c values, where each subset contains agents that are in communication range, i.e. $A_l^t = \{a_i / \forall a_j \in A_l^t, c(t, i, j) = 1\}$. For the merge process to complete successfully, it is necessary to have one merge master agent elected in each subset of connected agents. We define a binary function mm, which denotes if an agent a_i is a merge master, where $mm(a_i) = 1$ if a_i is a merge master, and $mm(a_i) = 0$ otherwise. We denote $|mm(A_l^t)|$ as the number of merge masters in the subset A_l^t of connected agents. According to the merge master election protocol presented in Sect. 3.2.4, if there is no merge master or many merge masters are present (i.e. $|mm(A_l^t)| \neq 1$) an election is eventually triggered at a time t'. After a time $\delta_{election}$, a new merge master is elected so that for $t \geq t' + \delta_{election}$ $|mm(A_l^t)| = 1$.

To prove Theorem 3.1, let us consider two revisions R_i and R_j created concurrently by agents a_i and a_j, respectively. R_i and R_j can be merged if and only if: 1) the agents are in communication range and can exchange a delta calculated as the difference between R_i and R_j; 2) there is at least one merge master. This can be formally described by the following theorem:

Lemma 3.1 *Considering two revisions R_i and R_j, created by agents a_i and a_j, respectively, there exists a time t at which $c(t, i, j) = 1$ and there exist an l subset such that $a_i, a_j \in A_l^t$, $|m(A_l^t)| \geq 1$.*

This lemma states that there is a time t at which it is possible to merge the two revisions R_i and R_j. The lemma does not require that there is only one single merge master, the existence of multiple merge masters allows for the merging of the revisions, even though this may trigger unnecessary merges.

To prove the lemma, we consider that $\forall i, j \ c(t, i, j)$ is constant for $t \in [t_0, t_1[$ and $t_1 - t_0 > \delta_{election}$, there is a single merge master for each A_l^t.

Assuming that for $t > t_1$, $\exists i, j \ c(t, i, j) \neq c(t_0, i, j)$, it is possible that there is at least one set A_l^t such as $|mm(A_l^{t_1})| \neq 1$ (this is equivalent to scenarios covered in *There are multiple merge masters* or *There is no merge master*). Given that $\forall i, j \ c(t, i, j)$ is constant for $t \in [t_1, t_2[$, if $t_2 - t_1 \geq \delta_{election}$, then the election will conclude and determine a merge master.

If $t_2 - t_1 < \delta_{election}$, it may happen that A_l^t gets divided at t_2. In this case, one subset of A_l^t will finish the election and have a merge master, while the second subset will fail and have to trigger a new election.

In case c changes again at $t_3 > t_2$, two cases can occur depending on whether the election has been completed or not. If $t_3 - t_1 < \delta_{election}$, the election is in progress and will continue to its completion. If $t_3 - t_1 \geq \delta_{election}$, the last election was completed, and depending on the number of merge masters, this might trigger a new election cycle. These cases relate to the scenario with *Intermittent communication interruptions*. This demonstrates that there is always a time t when there is at least one merge master that will attempt to merge revisions.

Once an election is completed, the newly elected merge master will start merging revisions, starting with the oldest one. Thus, by proving there is a time when the election is completed (i.e. proving Lemma 3.1), we effectively prove Theorem 3.1. Consequently, for as long as the agents stay in communication range for a δ_{enough} time, there is a time when the merger master will combine their changes.

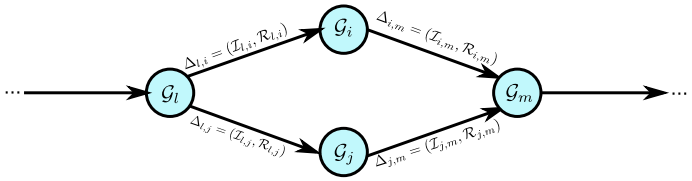


Fig. 9 Example simple merge problem with single revision in the concurrent paths

3.3 Merge algorithm

The merge algorithm is one of the two approaches used in the RGS$^{\oplus}$ System that can be used to combine two branches of an RDF$^{\oplus}$ Document.

Consider a simple example of a GoR with two branches that contain a single change between revisions as shown in Fig. 9. Given two concurrent versions: $\mathcal{G}_i$ with the delta $\Delta_{l,i} = (\mathcal{I}_{l,i}, \mathcal{R}_{l,i})$, and $\mathcal{G}_j$ with delta $\Delta_{l,j} = (\mathcal{I}_{l,j}, \mathcal{R}_{l,j})$ split from the common ancestor $\mathcal{G}_l$, the goal is to compute a new version $\mathcal{G}_m$ with two deltas: $\Delta_{i,m} = (\mathcal{I}_{i,m}, \mathcal{R}_{i,m})$ and $\Delta_{j,m} = (\mathcal{I}_{j,m}, \mathcal{R}_{j,m})$. The indices i, j, and l refer to revisions and not to specific agents.

In a general case $\mathcal{G}_m \subseteq \mathcal{G}_i \cup \mathcal{G}_j$. However, if a triple has been removed in the branch leading to $\mathcal{G}_i$, that triple should not appear in $\mathcal{G}_m$ even though it is available in $\mathcal{G}_j$. According to [4] the new revision is defined as:

$$\mathcal{G}_m = (\mathcal{G}_l \setminus (\mathcal{R}_{l,i} \cup \mathcal{R}_{l,j})) \cup \mathcal{I}_{l,i} \cup \mathcal{I}_{l,j} \tag{3.10}$$

Finally, the wanted deltas: $\Delta_{i,m}$ and $\Delta_{j,m}$ can be computed using Eq. (3.10) taking into account the properties of Eq. (3.7):

$$\mathcal{I}_{i,m} = \mathcal{I}_{l,j} \setminus \mathcal{I}_{l,i} \tag{3.11}$$

$$\mathcal{R}_{i,m} = \mathcal{R}_{l,j} \setminus \mathcal{R}_{l,i} \tag{3.12}$$

where, indices i and j can be interchanged in Eqs. (3.11) and (3.12).

In the general case, there may be multiple revisions leading to $\mathcal{G}_i$ and $\mathcal{G}_j$, and the idea is to reduce those multiple revisions into a single one and apply Eqs. (3.10) to (3.12). To reduce multiple revisions, we use the following *combine* function:

$$combine(\mathcal{I}_{i,i+1}, \mathcal{R}_{i,i+1}, \mathcal{I}_{i+1,i+2}, \mathcal{R}_{i+1,i+2}) \rightarrow \mathcal{I}_{i,i+2}, \mathcal{R}_{i,i+2} \tag{3.13}$$

$$\mathcal{I}_{i,i+2} = (\mathcal{I}_{i,i+1} \setminus \mathcal{R}_{i+1,i+2}) \cup \mathcal{I}_{i+1,i+2} \tag{3.14}$$

$$\mathcal{R}_{i,i+2} = \mathcal{R}_{i,i+1} \cup \mathcal{R}_{i+1,i+2} \tag{3.15}$$

The *combine* function is derived from:

$$\mathcal{G}_{i+2} = (\mathcal{G}_{i+1} \setminus \mathcal{R}_{i+1,i+2}) \cup \mathcal{I}_{i+1,i+2} \tag{3.16}$$

$$= (((\mathcal{G}_i \setminus \mathcal{R}_{i,i+1}) \cup \mathcal{I}_{i,i+1}) \setminus \mathcal{R}_{i+1,i+2}) \cup \mathcal{I}_{i+1,i+2} \tag{3.17}$$

$$= (\mathcal{G}_i \setminus (\mathcal{R}_{i,i+1} \cup \mathcal{R}_{i+1,i+2})) \cup (\mathcal{I}_{i,i+1} \setminus \mathcal{R}_{i+1,i+2} \cup \mathcal{I}_{i+1,i+2}) \tag{3.18}$$

$$= (\mathcal{G}_i \setminus \mathcal{R}_{i,i+2}) \cup \mathcal{I}_{i,i+2} \tag{3.19}$$

The *MergeRevision* algorithm is shown in Algorithm 6. The algorithm uses the *CombineMany* operation (Algorithm 7) which applies recursively the combine function (Eqs. (3.13) to (3.15)) to reduce a path of revisions into a single RDF Delta. The *CommonAncestor* function finds the common ancestor $\mathcal{G}_l$ using a *Bidirectional-Dijkstra* algorithm [22].

Algorithm 6: MERGEREVISION

Input: $\mathcal{G}_i, \mathcal{G}_j$
Output: $\mathcal{G}_m, \Delta_{i,m}, \Delta_{j,m}$
$\mathcal{G}_l \leftarrow CommonAncestor(\mathcal{G}_i, \mathcal{G}_j)$
$\mathcal{I}_{l,i}, \mathcal{R}_{l,i} \leftarrow CombineMany(RevisionsBetween(\mathcal{G}_l, \mathcal{G}_i))$
$\mathcal{I}_{l,j}, \mathcal{R}_{l,j} \leftarrow CombineMany(RevisionsBetween(\mathcal{G}_l, \mathcal{G}_j))$
$\mathcal{G}_m \leftarrow (\mathcal{G}_l \setminus (\mathcal{R}_{l,i} \cup \mathcal{R}_{l,j})) \cup \mathcal{I}_{l,i} \cup \mathcal{I}_{l,j}$
return $\mathcal{G}_m$, $(\mathcal{I}_{l,j}, \mathcal{R}_{l,j})$, $(\mathcal{I}_{l,i}, \mathcal{R}_{l,i})$

Algorithm 7: COMBINEMANY

Input: $revisions = \Delta_{i,i+1}, \Delta_{i+1,i+2}, \Delta..., \Delta_{j-1,j}$
Output: $\Delta_{i,j}$
if $|revisions| = 1$ **then return** $\Delta_{i,i+1}$ ▷ end of recursion
$\Delta_{i,i+2} \leftarrow combine(\Delta_{i,i+1}, \Delta_{i+1,i+2})$
return $CombineMany(\Delta_{i,i+2}, \Delta_{i+2,i+3}, \Delta..., \Delta_{j-1,j})$

3.3.1 Complexity

The time complexity of Algorithms 6 and 7 is considered below. Note that when integrated with the full HFKN Framework, there is additional communication and other types of overhead involved. This is considered in Sect. 4.

Complexity of a Single Merge Assume that each revision has a maximum of n_t^{max} triple removals or additions. The distance in the graph of revisions between two RDF Graph revisions, e.g. $\mathcal{G}_i$ and $\mathcal{G}_j$ is denoted as $d(\mathcal{G}_i, \mathcal{G}_j)$. The maximum value of d is the total number of revisions, i.e. $|\mathcal{G}|$. The $combine(\mathcal{G}_i, \mathcal{G}_{i+1})$ function in (3.13) needs to be executed in a loop over the n_t^{max} changes in $\mathcal{G}_{i+1}$ resulting in the worst-case time complexity of:

$$O((n_t^{max})^2) \tag{3.20}$$

It is possible to avoid iterating over all the triples by using a hash table representation to index the triples. A hash table has a worst-case time complexity of $O(n)$ and an average-case time complexity of $O(1)$ [19].

The complexity of the *combine* function can be reduced to the following average-case time complexity (over all the possible inputs to the *combine* function):

$$O(n_t^{max}) \tag{3.21}$$

$CombineMany(\mathcal{G}_l, \mathcal{G}_i)$ (Algorithm 7) applies the *combine* function $d(\mathcal{G}_l, \mathcal{G}_i)$ times and has therefore a worst-case time complexity of:

$$O((n_t^{max})^2 d(\mathcal{G}_l, \mathcal{G}_i)) \sim O((n_t^{max})^2 |\mathcal{G}|) \tag{3.22}$$

with an average-case time complexity (i.e. using hash table representation) of:

$$O((n_t^{max})^2 d(\mathcal{G}_l, \mathcal{G}_i)) \sim O(n_t^{max} |\mathcal{G}|) \tag{3.23}$$

$MergeRevision(\mathcal{G}_i, \mathcal{G}_j)$(Algorithm 6) applies the $CombineMany$ algorithm two times. Therefore, it has the same time complexity.

Merge K Revisions with the Same Ancestor Assume K agents make a concurrent change to the same revision $\mathcal{G}_0$ (see Sect. 4.2). At the k-th merge, the number of revisions is equal to

$|\mathcal{G}| = k + 1$. The total worst-case time complexity of the merge of the K revisions is given by:

$$O((n_t^{max})^2 \sum_{k=1}^{K} k + 1) \sim O((n_t^{max})^2 K^2) \tag{3.24}$$

with an average-case time complexity (i.e. using hash table representation) of:

$$O(n_t^{max} K^2) \tag{3.25}$$

General case In the general case, when merging two RDF Graph revisions, for example $\mathcal{G}_i$ with $\mathcal{G}_j$ in Fig. 9, the merge algorithm has to first find a path between the two revisions and then apply the merge operation using the Dijkstra algorithm. In a sparse graph, the complexity of the Dijkstra algorithm is bounded by:

$$O((|E| + |\mathcal{G}|)log(|\mathcal{G}|)) \tag{3.26}$$

where $|\mathcal{G}|$ is the number of revisions and $|E|$ is the number of edges between each revision. In a GoR, each revision has a maximum of 2 parents, therefore $|E| \leq 2|\mathcal{G}|$. The complexity of finding the shortest path between two RDF Graph revisions, for example $\mathcal{G}_i$ and $\mathcal{G}_j$ becomes:

$$O(3|\mathcal{G}|log(|\mathcal{G}|)) = O(|\mathcal{G}|log(|\mathcal{G}|)) \tag{3.27}$$

The complexity of computing the delta is given by:

$$O((n_t^{max})^2 d(\mathcal{G}_i, \mathcal{G}_j)) = O((n_t^{max})^2 |\mathcal{G}|) \tag{3.28}$$

The worst-case time complexity of a single merge is therefore given by:

$$O(|\mathcal{G}|log(|\mathcal{G}|) + (n_t^{max})^2 |\mathcal{G}|) \tag{3.29}$$

As the graph grows, it is likely that $log(|\mathcal{G}|) \gg (n_t^{max})^2$ and therefore the worst-case complexity is given by Eq. 3.27.

3.4 Rebase algorithm

An alternative to merging revisions is the *rebase* operation. Rebase changes a parent of a revision, essentially moving a branch in the Graph of Revisions. In the RGS$^{\oplus}$ System, the rebase is only applied to linear branches. A linear branch is defined as a branch that does not contain any split or merge revisions. This is not a limiting factor as changes made locally by an agent to an RDF$^{\oplus}$ Document are always linear in nature. The rebase process applied in a simple example GoR with two branches is presented in Fig. 10. In this scenario, two agents (A and B) created two branches in an RDF$^{\oplus}$ Document. The branch created by Agent B is done locally (i.e. not broadcasted to other agents). The rebase operation applied by Agent B moves the source branch $\mathcal{G}_l$ (created by Agent B) to the destination branch $\mathcal{G}_k$ (created by Agent A).

The rebase process follows Algorithm 8. In line 1, the algorithm looks for the common ancestor, using the Dijkstra search. In the case of the example shown in Fig. 10, the common ancestor is $\mathcal{G}_i$. $\mathcal{G}_{next}$ holds the next parent revision when moving, it is initialized in line 4 to $\mathcal{G}_k$, so that the revision $\mathcal{G}_l$ is moved on $\mathcal{G}_k$ in line 6. $\mathcal{G}_{next}$ is updated to the latest created revision in line 7 (e.g. to $\mathcal{G}'_l$ for the first iteration). Finally, the local revisions that have been rebased are removed in line 9.

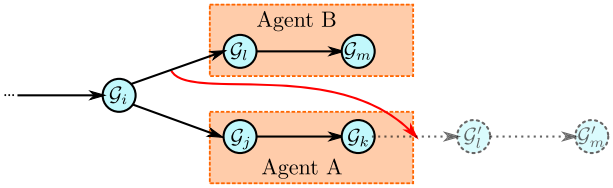


Fig. 10 An illustrative example of applying the rebase operation to Graph of Revisions with two branches created by two agents

Algorithm 8: REBASEREVISIONS

Input: $\mathcal{G}_m, \mathcal{G}_k$
Output: $\mathcal{G}'_{l\to m}$

$\mathcal{G}_i \leftarrow CommonAncestor(\mathcal{G}_m, \mathcal{G}_k)$
$\mathcal{G}_{i\to m} \leftarrow RevisionsBetween(\mathcal{G}_i, \mathcal{G}_m)$
$\mathcal{G}'_{i\to m} = \emptyset$
$\mathcal{G}_{next} \leftarrow \mathcal{G}_k$
foreach $\mathcal{G} \in \mathcal{G}_{i\to m}$ **do**
  $\mathcal{G}' \leftarrow InsertRevisionCopyAfter(\mathcal{G}, \mathcal{G}_{next})$
  $\mathcal{G}_{next} \leftarrow \mathcal{G}'$
  $\mathcal{G}'_{i\to m} = \mathcal{G}'_{i\to m} \bigcup \{\mathcal{G}'\}$
$RemoveRevisions(\mathcal{G}_{i\to m})$
return $\mathcal{G}'_{i\to m}$

After applying the rebase algorithm presented above, all local revisions are combined into the final GoR, which means the history of changes is preserved. This will directly influence the time required to synchronize changes among all agents as the number of messages used in the overall synchronization mechanism is proportional to the number of newly rebased revisions. If preserving the history of all revisions is less important than how fast the shared information is synchronized among all agents, an alternative approach to the default rebase algorithm implemented in the RGS$^{\oplus}$ System can be used. The alternative adds an operation, called *squashing* before the default rebase algorithm is applied. The *squashing* procedure combines all of the revisions in the source branch (i.e. $\mathcal{G}_l$ to $\mathcal{G}_m$ created by Agent B) into a single revision, thus reducing the number of revisions in the final GoR. *Squashing* is performed using the *CombineMany* operation (Algorithm 7). The performance of both variants of the rebase algorithm is evaluated in Sect. 4.3.

3.4.1 Complexity

Following the notation from Fig. 10, when rebasing revision $\mathcal{G}_m$ on $\mathcal{G}_k$ with a common ancestor $\mathcal{G}_i$, the complexity of performing the rebase operation is given by:

$$O(n_t^{max} d(\mathcal{G}_i, \mathcal{G}_m)) \tag{3.30}$$

However, it is necessary to first compute the path between $\mathcal{G}_m$ and $\mathcal{G}_k$, which means the complexity of rebase is also bounded by the complexity of the shortest path algorithm given by Eq. (3.27).

3.5 Consistency and synchronization timing

Consistency As discussed in Sect. 2, the RGS$^{\oplus}$ System does not guarantee logical (i.e. semantic) consistency of the RDF Documents. In future work, however, the system can be extended to include an automatic correction step, by adding it to the Algorithm 6 executed during the synchronization process by the merge master. A function call to the correction step would be added, (i.e. $\mathcal{G}_m \leftarrow correct(\mathcal{G}_m)$) after line 5 of the algorithm. The correction algorithm could be based, for example, on a combination of SHACL and Answer Set Programming [1].

The RGS$^{\oplus}$ System does guarantee syntactic consistency, i.e. RDF Documents shared between agents converge towards containing the same set of RDF Triples. When dealing with syntactic consistency two types of conflicts may occur and need to be handled by the system.

The first type of conflict can occur between two branches during the merge or rebase operation. For example, when a triple τ is added in one branch i and removed in the other branch j, i.e. $\tau \in \mathcal{I}_{i,m}, \tau \in \mathcal{R}_{j,m}$. However, such a situation is not possible in the RGS$^{\oplus}$ System, due to how revisions are constructed. Such a conflict would mean that the triple τ is both included and not included in the common ancestor $\mathcal{G}_l$:

$$\tau \in \mathcal{R}_{j,m} \Rightarrow \tau \in \mathcal{R}_{l,i} \Rightarrow \tau \in \mathcal{G}_l \tag{3.31}$$

$$\tau \in \mathcal{I}_{i,m} \Rightarrow \tau \in \mathcal{I}_{l,j} \Rightarrow \tau \notin \mathcal{G}_l \tag{3.32}$$

The second type of conflict can occur when a triple τ is added in one branch i and then added and removed in the other j. This type of conflict is not a problem, since during the merge operation adding and removing of the triple τ is treated in branch j as if it was never there.

Synchronization Timing For each subset of RDF Graphs shared by an agent, the union of those subsets is guaranteed to be weakly consistent after termination of synchronization processes associated with the synchronization algorithms subject to certain timing and computational assumptions. These assumptions are clarified in the following scenario. Consider the following scenario in which m RDF$^{\oplus}$ Documents are shared among a team of agents. The team has made n asynchronous and/or concurrent changes to each of the shared RDF Graphs and no other changes occur. Under these assumptions, the RDF$^{\oplus}$ Document synchronization mechanism will ensure that the changes are synchronized among all team members within a certain time T_{Total} dependent on the computational capabilities of each agent and the performance of communication links. The time is defined as:

$$T_{Total} = m \times T_S + m \times T_M(n) + m \times T_C(n) + m \times T_U(n) \tag{3.33}$$

where T_S is the maximum time to select a merge master for each RDF$^{\oplus}$ Document. $T_M(n)$ is the maximum time needed to merge all changes introduced by each agent to each RDF$^{\oplus}$ Document to produce a consistent version of common graphs. Time $T_C(n)$ is the maximum time required to transfer necessary revisions to all agents. Time $T_U(n)$ is the maximum time required for each agent to apply the updates received from the merge master to its copy of the RDF$^{\oplus}$ Document.

In a general case, for all scenarios where the asynchronous and concurrent changes to the shared RDF$^{\oplus}$ Documents finish at time t, the RGS$^{\oplus}$ System will ensure that all changes are synchronized among all participating agents at time $t + T_{Total}$.

4 Empirical evaluation of RDF$^{\oplus}$ graph synchronization

The RGS$^{\oplus}$ System, described in Sect. 3, provides the backbone for keeping distributed knowledge collected by a team of agents synchronized, up-to-date, and accessible for use by the team. In this section, simulation, and field robotics experiments designed to evaluate the performance of this backbone are considered.

Several aspects of the synchronization mechanism are evaluated. The first set of experiments focuses on the performance of the merge algorithm used by the merge master to synchronize RDF Graphs. The algorithm's efficiency is evaluated in a simulated scenario where a large number of new revisions are created, with varying size of RDF Triple changes (additions or deletions). The experiments solely focus on measuring the timing of the algorithm excluding any extraneous factors related to communication links. The results are presented in Sect. 4.2. Similar evaluation was performed for the rebase algorithm with the results presented in Sect. 4.3.

Additionally, a more practical scenario, in which all functionalities of the RDF Graph synchronization mechanism are used, was created. It includes multiple agents creating, changing and synchronizing multiple RDF Graphs. During the experiment network disconnections were introduced to showcase the resilience of the RGS$^{\oplus}$ System. The evaluation results presented in Sect. 4.4 include an in-depth analysis of the RGS$^{\oplus}$ System behaviour in this complex scenario. The results include message exchange statistics, timeline for synchronization events and the history of all revisions over the course of the experiment.

Moreover, a set of experiments was conducted to evaluate the scalability of the RDF Graph synchronization mechanism. In particular, the focus was to find the maximum rate at which new revisions can be created and still synchronized in practice among all agents. This rate depends on multiple factors (e.g. number of agents, number of RDF$^{\oplus}$ Documents etc.) which were taken into account during the experiment design. The results of the evaluation in scenarios including missions with up to 20 agents are presented in Sect. 4.5.

Finally, the RDF Graph synchronization mechanism was used during several field robotics experiments. For example, in [29], an experiment using the full HFKN Framework, with a number of deployed UAVs and human operators, is described. In Sect. 4.6 we present another field robotics experiment where two UAVs are used to execute an exploration mission with the goal of gathering LIDAR sensor data over specified geographical regions.

For the validation of the RGS$^{\oplus}$ System using the listed experiments, an RDF Document is stored using three SQL tables in a PostgreSQL database. One table is used as an RDF triplestore, where each row corresponds to a single triple. The other two tables are used to store the revisions and deltas from the GoR.

Changes to the RDF Document occur within a database transaction, during which the database reports to the RGS$^{\oplus}$ System a list of added and removed rows in the table. Effectively, the list corresponds to RDF triples that are added or removed from the RDF Document. When the database transaction is completed, it is straightforward to generate an RDF Delta using the list of added and removed triples. The RDF Delta and its corresponding revisions are then inserted in the two tables.

The RDF$^{\oplus}$ System was integrated with the HFKN Framework presented in [29], and more details on the use of the database are included in that paper.

It is worth noting that the work presented in this paper is agnostic to the type of storage (e.g. database) used for RDF Documents. However, to benefit from the improvements presented in our work, namely the linearity in changes, it is required that the storage subsystem can provide the list of triples added and removed during a transaction.

Table 1 Benchmark results for the `INSERT` and `DELETE` queries and the versioning overhead for the RGS$^{\oplus}$ system compared to the Quit Store, where Qps stands for Queries per second

Queries	No versioning	Versioning	Overhead	Quit store overhead
`INSERT`	11.04 Qps	7.00 Qps	1.57×	≈ 2×
`DELETE`	140.43 Qps	70.15 Qps	2.00×	≈ 200×

4.1 Evaluation of the versioning overhead

To evaluate the overhead of versioning for our triplestore, we use the same standard *Berlin SPARQL Benchmark (BSBM)* [13], as in the *Quit Store* approach [4]. The *update and explore* variant of the benchmark was used, which executes `INSERT`, `DELETE`, and a number of `SELECT` queries. The detailed results for the *update* part are shown in Table 1. In comparison, the Quit Store approach has an overhead of approximately 2× and 200× slower for `INSERT` and `DELETE` queries, respectively. In our approach, the overhead for `INSERT` query is slightly smaller at 1.57× slower, while the overhead for `DELETE` queries is significantly smaller at 2.00× slower. The versioning system is not used when *exploring*, and therefore the query throughput is similar with or without versioning.

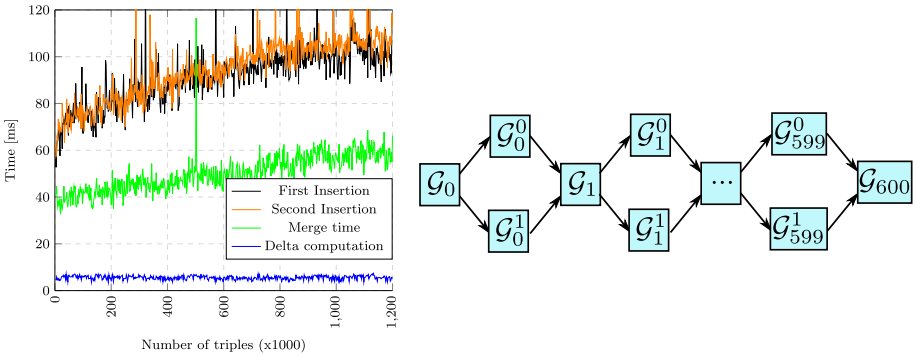
4.2 Evaluation of the merge algorithm

The empirical evaluation of the merge algorithm is carried out with two experiments. In the first one, we demonstrate that the computation of the delta for merging is only dependent on the number of inserted and removed triples in the deltas, and independent of the total number of triples in the RDF Graph. The second experiment demonstrates the performance of the merge algorithm when merging many concurrent revisions.

Many repeated merges In this experiment, at every step, two RDF Graph revisions are created concurrently and merged. Each new revision is created by inserting 1000 new unique triples. Effectively, if the tip revision is $\mathcal{G}_i$, two revisions $\mathcal{G}_i^0$ (1000 new triples) and $\mathcal{G}_i^1$ (1000 new triples) are created with a common ancestor $\mathcal{G}_i$. $\mathcal{G}_i^0$ and $\mathcal{G}_i^1$ are merged into $\mathcal{G}_j$, which contains 2000 new unique triples in addition to the original triples from G_i. The process is repeated 600 times and the results are shown in Fig. 11. The results show that the cost of computing the delta is constant and independent of the number of triples in the graph. This is consistent with the complexity analysis presented in Sect. 3.3.1. Figure 11 shows an increase in the cost of inserting data and inserting the merge revision. This is caused by the added cost of inserting new triples in the database and, in particular, the maintenance of the indexes.

Merge of many concurrent revisions The performance of the merge algorithm presented in Sect. 3.3 was evaluated in an experiment focused on the timing requirements of the merge operation itself. This means that the overhead of communication required for the RDF Graph synchronization is not included. The experimental setup can be described by the following scenario. Consider 1000 agents making concurrent changes (in form of adding or removing RDF Triples) to a single RDF Graph. The graph is to be synchronized among all agents. Each change results in a new revision of the RDF$^{\oplus}$ Document, such that its GoR will contain multiple branches. The merge algorithm is used to combine newly created revisions into one merged revision which includes all the changes introduced by all agents.

Figure 12 depicts the GoR after the experiment is finished. All generated revisions ($\mathcal{G}_1 \ldots \mathcal{G}_{1000}$) share the same parent revision $\mathcal{G}_0$. Since the merge operation is applied to



(a) Timing Results (b) Resulting GoR

Fig. 11 Results of repeatedly creating two branches with a 1000 new triples in each branch, and merging them, for 600 iterations. The right figure shows the resulting GoR, while the left figure shows the time of inserting the triples (in orange and black), the total time of merging (in green) and the time to compute the deltat between two revisions to merge (in blue) (Color figure online)

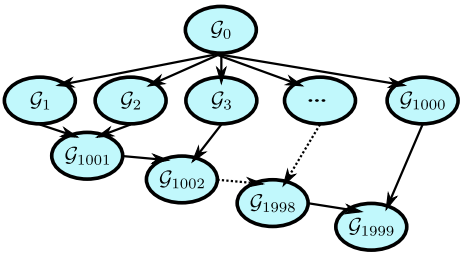
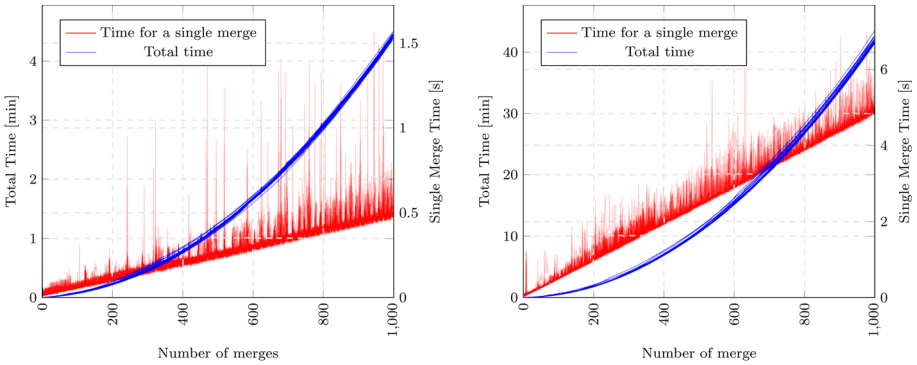


Fig. 12 The final graph of revisions in the experimental scenario used for the performance evaluation of the merge algorithm

two branches at a time, it has to be applied 999 times in this scenario in order to create the final merge revision which includes all the changes (i.e. $\mathcal{G}_{1999}$). In order to quantify the influence of the size of the changes applied to the RDF Graph on the performance of merge algorithm, two cases of the experiment were performed, each containing changes consisting of 10 or 100 RDF Triples. The experiment was performed execution all agents' software on a computer with an `Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz` with six cores and 16GB of RAM.[6]

The results of the experiment are presented in Fig. 13. They confirm that the merge operation time complexity is linear in the number of revisions that need to be combined when merging (see the red curve depicting the time required for a single merge as a function of the number of merge operations). This observation follows the complexity analysis presented in Sect. 3.3.1. Additionally, the update size for each revision (i.e. 10 vs 100 tuples) increases the time required by a constant factor. For instance, the average time in the case of the last merge operation (i.e. 999) increases from 0.5s to 5s for 10 and 100 tuples per revision, respectively. Inconsistencies in timing when considering single merge operations (spikes present in the red plot) are most likely caused by the overhead of accessing the database and variations of the computer system performance. The total complexity of merging is quadratic in the number of revisions to merge as can be seen in Fig. 13 (blue curves).

[6] All algorithmic code is implemented in C++.



(a) 10 tuples per revisions (b) 100 tuples per revisions

Fig. 13 Results of the performance evaluation of the merge algorithm. Total times required for merging 1000 revisions into the final merge revision is shown in blue. Individual merge operation times are depicted in red. Results for two variations of the experiment consisting of 10 and, 100 changes (i.e. adding or removing triples) are presented on the left and right side, respectively. The experiment was performed 20 times and all the individual results are shown in the figures (Color figure online)

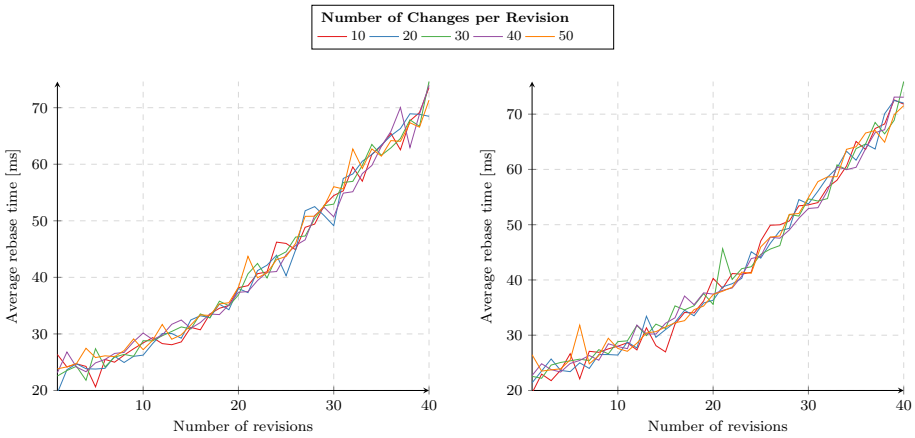
4.3 Evaluation of the rebase algorithm

The purpose of this experiment is to evaluate the performance of the rebase operation described in Sect. 3.4. As in Fig. 10, we consider two agents A and B creating new revisions in parallel that correspond to two branches in the graph of revisions of the RDF$^{\oplus}$ Document. The rebase source branch (top of Fig. 10) created by agent B has a varying number of revisions with varying number of changes in each revision. The content of the rebase destination branch created by agent A does not influence the performance of the rebase algorithm, therefore the destination branch contains only a single revision in this experiment. Two variations were evaluated:

- move all the individual revisions from the source to the destination branch (see Fig. 10, revisions G_l and G_m are kept separate). This is currently the default behavior of the rebase operation as used in the RGS$^{\oplus}$ System, see Algorithm 8.
- combine all revisions from the source branch into a single one before applying the rebase algorithm, using the squashing operation.

The number of revisions in the source branch varied from 1 to 40, and the number of changes (additions and removals) in each revision was from 10 to 50. The experiment was performed on a single computer omitting the communication links between agents. Other aspects, such as access to the database, were considered. The timing was measured for each variant of the rebase and each variation of the number of revisions and changes.

Figure 14 shows the average time it takes for the rebase algorithm for the different configurations. The rebase operation has a constant time cost of about 23 ms when only one revision is rebased. Each additional revision takes 1.3 ms on average, which results in 72 ms for the case of rebasing 40 revisions as shown in Fig. 14. Additionally, the rebase with squashing (Fig. 14b) or without (Fig. 14a) has very little influence on the overall performance of the rebase operation. Squashing reduces the number of revisions in the graph, which means that the number of messages exchanged between agents is reduced. This in turn minimizes the number of messages which have to be transmitted in the overall process of the RDF Graph synchronization mechanism. Therefore performing squashing should be preferred if



(a) Without squashing (b) With squashing

Fig. 14 Results of the performance evaluation of the rebase algorithm. The time is averaged over 20 rebase operations

the quickest synchronization between all agents is the main priority. It is important to note that in the evaluation the dominating factor is the overhead caused by accessing the database to fetch the revisions. For this reason, the number of changes and whether squashing is performed has minimal influence on the overall performance of the rebase operation.

4.4 Evaluation of the RDF graph synchronization

Unlike the evaluations described thus far, in this subsection the evaluation of the complete RDF Graph synchronization mechanism is presented. This experiment was designed to demonstrate the ability of the RGS$^{\oplus}$ System to handle concurrent changes in an RDF Graph shared among agents and robustness against communication interruptions. A typical data-gathering mission is simulated where twelve agents explore an environment and new datasets metadata are created and exchanged. Note that the focus is put only on how this affects modifications to the shared RDF Graph. Thus for this experiment, sensor readings were not simulated, and no sensor data was transferred between agents. Only the RDF Graph synchronization protocol as presented in Sect. 3 was used. For the purpose of this experiment, the systems are simulated, as this allows us to include a larger number of agents and to control the communication network disruptions. However, the simulated systems are using the same software architecture as what is used on real robots during the field experiments. The communication between agents is handled using the ROS [46] middleware and the systems run a full SymbiCloud architecture, as presented in [29].

Twelve agents are split into three groups of four as shown in Fig. 15. Each group is simulated on one computer. One of them uses an `Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz` with six cores, and two other computers have an `Intel(R) i7-7567U` CPU. All three computers use 16GB of RAM. The machines were connected through a router to easily demonstrate what happens when agents get disconnected from each other.

The timeline summarizing the experiment is shown in Fig. 16. The agents within a group are assumed to always be able to communicate. To demonstrate the ability to handle network interruptions, disconnection and re-connection between groups is simulated. Computers used

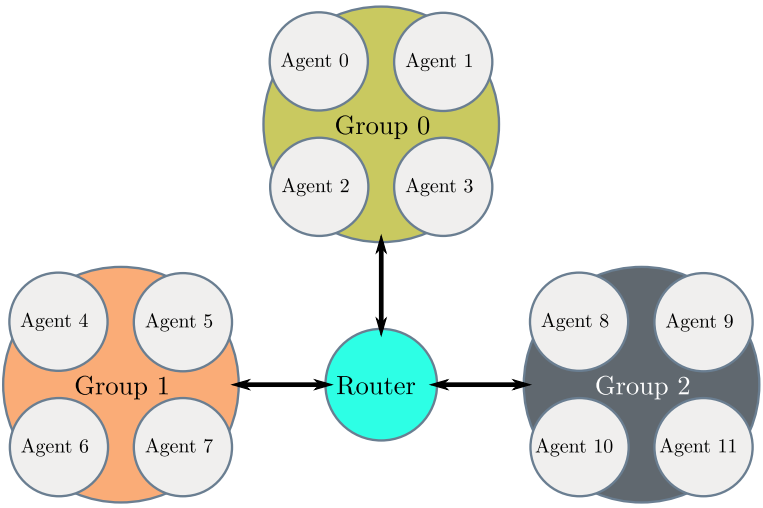


Fig. 15 The experimental setup in a mission involving 12 agents distributed over three computers

for *Group 1* and *Group 2* get disconnected for some period of time from the router as shown in red in Fig. 16. When both *Group 1* and *Group 2* are disconnected, then *Group 0* is essentially disconnected as well (i.e. Disconnection A, B, and C overlap in the figure). This results in creating up to three distinct groups that elect their own merge masters (c.f. the orange color in the figure) until the connection is reestablished and a single merge master for all three groups is elected. During the course of the experiment, several agents make simultaneous changes to the RDF Graph, which need to be synchronized. The new revisions, which are the result of the changes, are marked by blue arrows.

The results show that the agents are quick to synchronize the RDF Graphs with each other (c.f. the unsynchronised times in Fig. 16) even in the context of multiple concurrent edits or connection and disconnection behavior.

Each agent received an average of 1959 messages, out of which 1584 (approx. 81%) were status messages (see Sect. 3.1.5). Agents received the maximum of 105 messages/s and the average of 2.25 messages/s.

The final Graph of Revisions of the experiment presented in Fig. 17 shows the relation between the different revisions and their creators (numbers in boxes). The red areas show the revisions that were created during the three disconnections: A, B and C (c.f. bottom of Fig. 16). The turquoise area shows an excessive number of merges that were computed during the reconnection. This happens in particular at the end of *Disconnection B*, when *Group 0* and *Group 1* reconnect. The merge master, Agent 5, follows an eager strategy for merging, and starts the process as soon as it receives revisions from *Group 0*. After an election occurs and Agent 0 is selected, the same problem repeats. In normal operation, the merge eagerness allows for faster propagation of knowledge between agents, but in future work, different strategies for when to merge will be investigated.

4.5 Evaluating the maximum revision creation rate

In this experiment, the scalability of the RDF Graph synchronization mechanism is tested, where many agents make concurrent changes to several documents. The evaluation presented

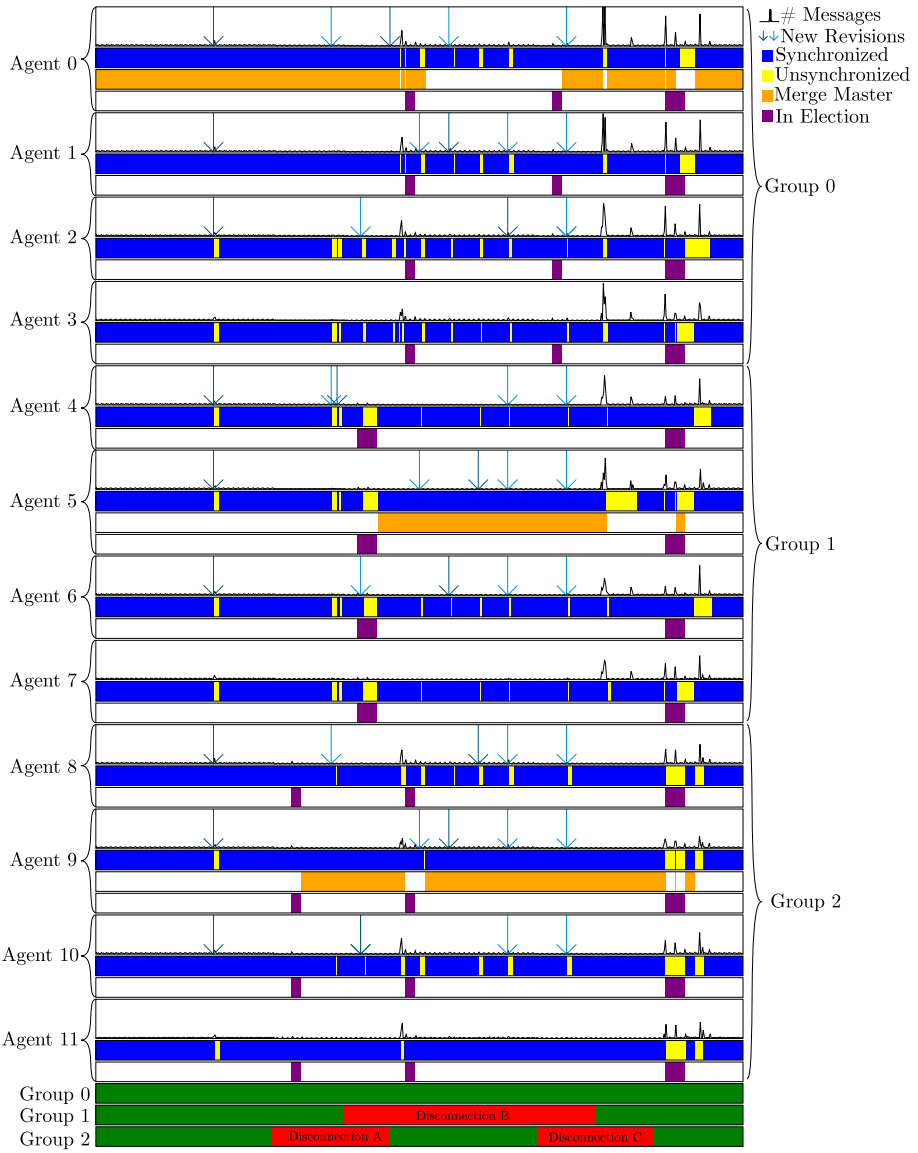


Fig. 16 Experimental results: The top row of each agent shows arrows to indicate when new triples are inserted into the agent's RDF Graph and the plots to indicate the number of messages exchanged at any given time. Blue indicates when the agent is synchronized with the merge master and yellow when it is unsynchronised. Purple indicates an election period. Orange indicates when an agent has the role of a merge master. Green indicates when a team computer is connected to the router and red when it is disconnected (Color figure online)

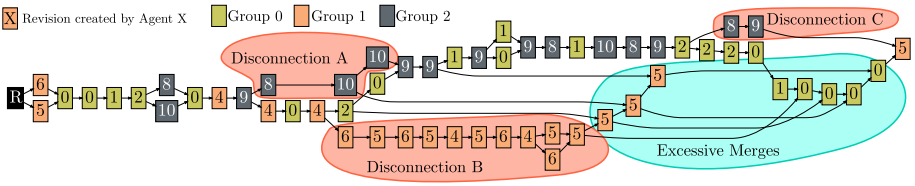


Fig. 17 The final Graph of Revisions for the experiment. Numbers indicate which agent created the revision. The colors of the boxes show the three groups of agents. Three disconnection periods are marked with red areas. The turquoise area indicates the period of excessive merges (Color figure online)

in this section is focused on finding the limits on how fast the changes created concurrently by multiple agents in multiple RDF Graphs can in practice be merged and synchronized without unnecessary delays. In other words, the interest is in the *practical* rate of changes below which the merge master can perform its task in a timely manner. Above this limit the queue of changes grows without the possibility to be merged on time, before more changes are created. There exists a wide range of factors which influence this maximum rate: the number of RDF$^{\oplus}$ Documents, the number of revisions per document, the number of participating agents, communication links' properties, agent's hardware etc. For this experiment, focus is on the factors that directly affect the computational cost of merging, such as the number of documents, the amount of changes to the documents and the number of participating agents.

Agents perform changes to the RDF Graphs concurrently and locally as they carry out their mission tasks in real-world environments. In a general case, the agents' KDB Managers create new revisions for each change in a document which is then sent to the merge master. In practice, it is the performance of executing the tasks of the merge master which limits the rate at which the RDF Graphs can be merged and synchronized in a timely manner. This issue is related to the Never Synchronized Problem described in Sect. 3.1.4 from which one can conclude that the rate of publishing of local revisions needs to be smaller than the time it takes to perform the merge by the master. Also, note that the fact that rebase is used does not solve this practical bottleneck. It only handles the behavior of an agent regarding local revisions while waiting for the result of a merge.

Effectively, if agents create changes to shared documents above the maximum rate at which a merge master can perform its tasks in a timely manner, a delay will be introduced in the synchronization process. The delay duration will vary depending on the amount of excessive changes that still need to be incorporated. In an extreme case, when all agents always create changes over the maximum rate, the delay will keep growing and the full synchronization will never be achieved. However, in a practical setting agents join and leave missions as they have limits on how long they can operate (e.g. fuel, battery power), and the maximum load imposed on the merge master will vary over time. After a period of too frequent updates, when the rate becomes lower and manageable by the master, the changes will eventually be merged, thus the information will be synchronized among all agents. One way to improve the maximum synchronization rate would be to include the computational performance of the agents when electing a new merge master, but this will be pursued in future work.

With the above in mind, the evaluation was performed using the following procedure loop:

- Each agent creates a new revision with at most C insertions or deletions of triples for each document.
- Each agent broadcasts its revisions to others and waits for the master to finish the merge.
- The master performs the merge procedure.

To isolate and evaluate the performance of the tasks of the merge master, in this experiment the agents are *not allowed* to make local changes until the merge master finishes the merge. When the merge master finishes the synchronization process and sends the new revision, the cycle may then start again. In the experiment, all documents are merged by a single master. This is a worst-case scenario as, in general, separate documents can have different merge masters thus the merge performance can be improved.

Using the procedure described above the influence of the following parameters was evaluated: the number of agents $N = \{2, 5, 10, 20\}$, each making $C = \{10, 20, 30, 40, 50\}$ changes resulting in N revisions to each of $M = \{1, \ldots, 20\}$ RDF$^{\oplus}$ Documents. The procedure loop described above is performed 30 times, all agents run on one computer equipped with a six core `Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz`. The influence of the communication links' properties is excluded in this experiment in the name of reproducibility between single merges in the loop. Average time for 30 iterations is recorded and reported in Figs. 18 and 19.

The average time to perform a merge for 2, 5, 10, and 20 agents is presented in Fig. 18. For each case the plots show the time as a function of the number of documents and changes per revision. For example, for $N = 2, M = 20, C = 40$, during the merge period, local changes can not occur faster than every 2 s, as the merge master would not be able to keep up. Figure 19 presents the same results but allows to evaluate how the system scales with the number of agents.

The performance shown appears to be sufficient in typical use cases as agents usually generate smaller changes at much lower rates. The merge is sporadic in a typical mission because there is a longer period of time between creating new revisions. For example, in the case study presented in Sect. 4.4 involving 12 agents, the merge master only performed 13 merges. Even in the biggest case, where $N = 20, C = 50, M = 20$ the merge procedure took only 60 s.

4.5.1 Discussion of results

To provide additional examples and put the results of the experiment presented above in an extended and practical context, three possible use cases are considered.

Use case 1: dataset discovery In [29], a dataset concept is introduced. This encapsulates a set of low-level data with its description in the form of metadata stored in an RDF Graph. Each agent adds the metadata to a shared RDF Graph while they collect the data during exploration missions, which last 5 to 20 min. Considering that each agent creates two revisions in the RDF Graph, assuming that N agents collaborate, $2N$ revisions would be required. The first revision would be created at the start of the scanning mission and the second one at the end once the dataset is completed. In this scenario, agents work with a single RDF Graph.

The results in Fig. 18 show that for 20 agents, for a single document, if all agents make a change simultaneously, the merge takes $2.9s$, meaning that agents can make up to 0.3 changes per second on average. In this scenario, each agent makes at most 2 new revisions per 5 min, which is well below what the system can handle. Extrapolating these results using the fact that the merge and rebase algorithms scale quadratically in the worst-case, the RGS$^{\oplus}$ System should be able to handle missions of this type with up to 143 participating agents.

Use case 2: victim detection In this scenario, a fleet of UAVs is considered that explores an operational environment and searches for victims of a natural or human-made disaster. Each UAV adds the location and selected information about the identified victims in a single document. Each victim is represented by two triples, one for the victim's location and one for the state (injured, unconscious, hungry, etc.).

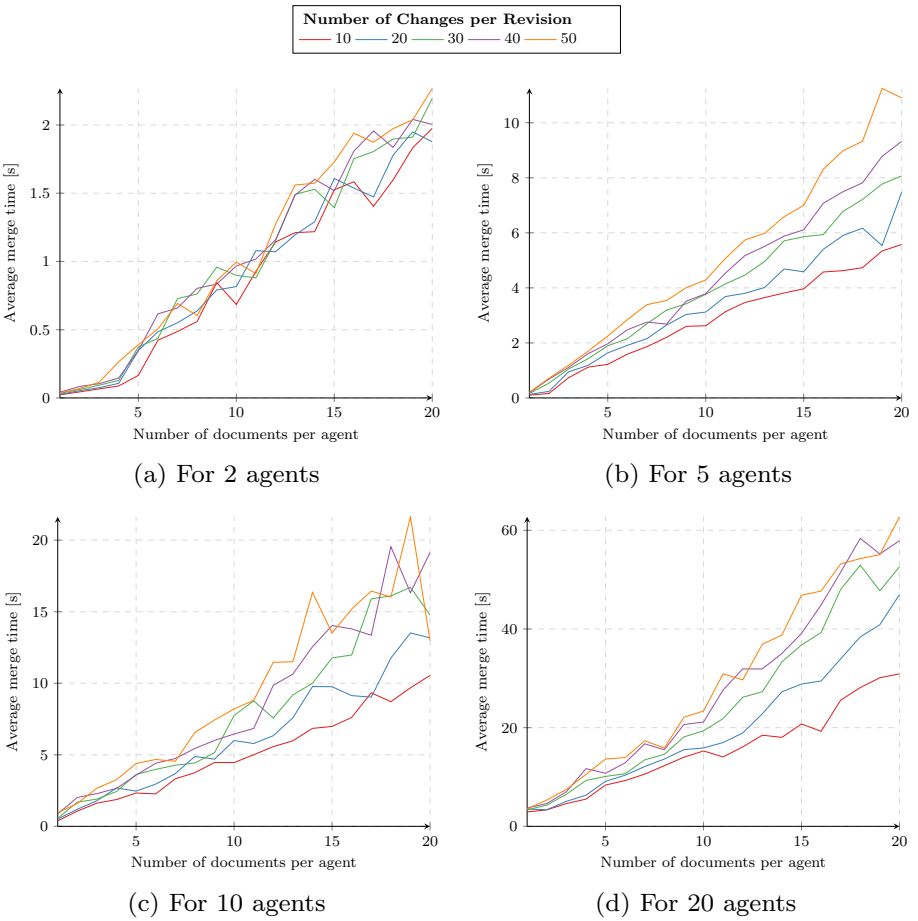


Fig. 18 Results of the experimental evaluation with focus on the scalability aspects of the RDF$^{\oplus}$Framework. Plots depict the average merge time as a function of a number of documents per agent for 2, 5, 10 and 20 agents involved, with a varying number of changes in each document revision

Consider the mission is performed in the most dense city in the world which is Manila[7] with 41515 person per square kilometer (i.e. 0.04 person per square meter). One can assume that the UAVs are flying at an altitude of 13.7m and are equipped with cameras using a wide-angle lens with 60 deg field of view. Each image acquired by a single UAV covers an area of approximately $15.8 \times 15.8 \approx 250m^2$. Assuming each UAV flies at 15.8m/s speed, it can scan $250m^2/s$. Ideally, this would result in detecting $250 \times 0.04 = 10 people/s$ in the city of Manila. Consequently, each UAV would generate $2 \times 10 = 20\ RDF\ Triples/s$. Instead of creating a new revision per one detected victim, we buffer the information until we detect 5 victims which results in creating $10/5 = 2$ revisions per second.

According to the results presented in Fig. 18, this type of mission would be feasible when using up to 20 agents working on the same RDF Graph. During a rescue operation, it is critical to conclude the exploration as quickly as possible. Given example of rescue mission in Manila performed using a small fleet of 20 agents, would result in rather long exploration

[7] Source: https://en.wikipedia.org/wiki/List_of_cities_proper_by_population_density.

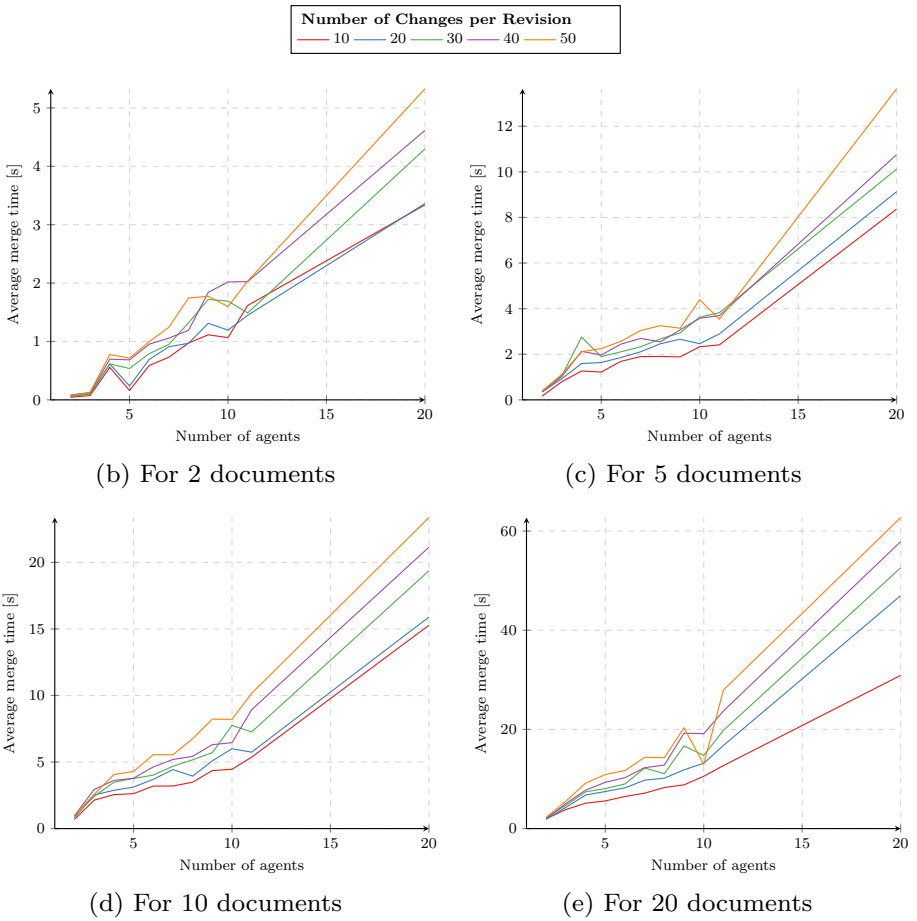


Fig. 19 Results of the experimental evaluation with focus on the scalability aspects of the RDF$^{\oplus}$Framework. Plots depict the average merge time as a function of a number of agents involved for 2, 5, 10 and 20 documents, with a varying number of changes in each document revision

time of $2h20m$ (Manila has a surface of $42.88km$, and each agent covers $250m^2/s$). Since 20 agents is the limit, in this scenario, for collaboration on a single document, to use more agents a divide and conquer approach is used, and several documents covering smaller areas would be created.

Use case 3: exploration with a fleet of UAVs In this scenario, consider N UAVs scanning an area with color and thermal cameras, both delivering images at a rate of $10Hz$. Additionally, locations of the platforms are recorded at $30Hz$. The UAVs collaborate on three documents: one for color cameras, one for thermal cameras, and one for location data ($M = 3$).

Each image is represented by three RDF Triples: one to specify which agent acquired the image, one to specify the acquisition timestamp, and one to point to the location of image data. Since images are delivered at a rate of $10Hz$, and there are three triples used, agents will create 30 changes per second for each of the two camera sensors. Therefore the total number of changes to the documents about the camera data is 60 per second.

The location information is represented by three RDF Triples: one to specify whose agent location it relates to, the timestamp, and the actual position value. In that case, agents will create 90 changes per second to represent the location information. The total number of changes the agents will create per second is $60 + 90$, that is 150 changes per second in the three documents ($C = 150$).

The agents collaborate on three documents, and they report images as well as locations twice per second which results in revisions with 30 and 45 changes, respectively.

According to Fig. 18, this would work up to 10 agents. With 20 agents, it would take more than 2 s to merge the changes created during one second. As before this limitation can be overcome using a divide-and-conquer approach.

4.6 Exploration scenario

In this section, we present an example use of the RGS$^{\oplus}$ in a real-life scenario where three agents execute a collaborative exploration mission. The agents involved include a ground operator and two UAVs. One of the UAVs is a DJI Matrice 600[8] equipped with a 3D LIDAR sensor from Velodyne.[9] The other UAV is a DJI Matrice 100[10] equipped with a 3D LIDAR sensor from Ouster.[11] Additionally, both UAVs are equipped with an Intel NUC computer[12] that hosts the necessary software. The communication between agents and software modules is handled by the ROS [46] middleware.

Each agent runs a KDB Manager with the RGS$^{\oplus}$ System. Agents synchronize two RDF Graphs. The first graph encodes the agent's capabilities. The information includes the specification of the agent's sensors and flight capabilities, and it is used in the task allocation process and mission planning.[13]

The second RDF Graph that agents synchronize includes information about the acquired sensory datasets. This graph is used for dataset discovery and follows the same RDF Model as in *Use Case 1* described in Sect. 4.5.1. A detailed description of the model and the sensory dataset concept is presented in [29]. In short, the HFKN Framework stores a sensory dataset in two parts. Part 1 contains the actual sensor data (i.e. raw sensor or processed data), which generally can be of high volume. Part 2 includes the metadata associated with the sensory dataset (i.e. Part 1). The metadata is small in size and contains information such as the type of data, the geometry of the explored area, the time when it was acquired, and a list of agents that currently store the content.

The separation allows the agents to efficiently share the information about available sensor data by automatically synchronizing the RDF Graphs using RGS$^{\oplus}$. The actual dataset content is stored on a subset of agents, and its synchronization is handled by a separate process designed to handle an exchange of high-volume data.

In this field robotics experiment, the ground operator requests a 3D model (i.e. a point cloud) for two overlapping areas in the environment. This requires the exploration of two regions and the fusion of the acquired LIDAR data into a single point cloud. Two exploration missions are executed using the two UAVs. An overview of the missions is presented in

[8] https://www.dji.com/se/matrice600-pro.

[9] https://velodynelidar.com/products/puck/.

[10] https://www.dji.com/se/matrice100.

[11] https://ouster.com/products/scanning-lidar/os1-sensor/.

[12] https://www.intel.com/content/www/us/en/products/sku/214614/intel-nuc-10-performance-kit-nuc10i7fnkn/specifications.html.

[13] The RDF Model will be described in a forthcoming paper.

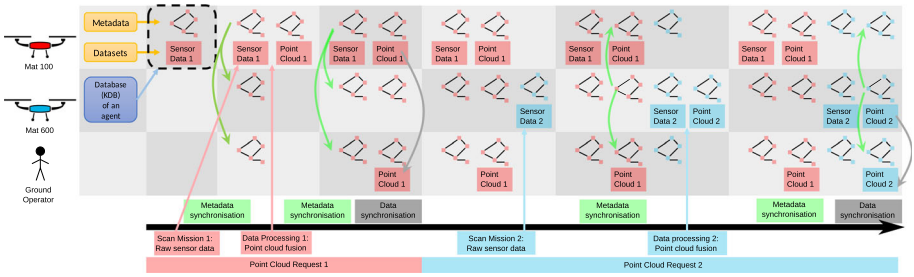


Fig. 20 Timeline for dataset creation and exchange. Each row corresponds to the knowledge of an agent over time. Graphs representing metadata are stored in an RDF$^{\oplus}$ Document, while boxes represent raw sensor data. The green and gray arrows represent metadata and raw sensor data synchronization, respectively (Color figure online)

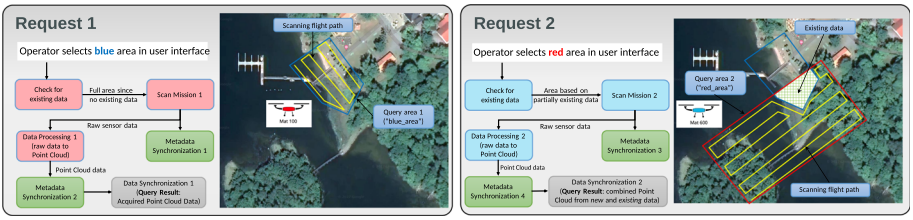


Fig. 21 Overview of the two collaborative missions. On the left, the first part of the mission is presented, where the blue region is selected by the operator. On the right, the second part of the mission is depicted, where the operator selected the red area. Diagrams show the flow of the execution associated with the missions, and the automatic synchronization of metadata is highlighted in green (Color figure online)

Fig. 21. Figure 20 depicts the timeline for dataset creation and the synchronization using the RGS$^{\oplus}$ System and the dataset exchange mechanism.

In the first mission, the ground operator selects an area (i.e. blue area in Fig. 21) in the user interface and executes a data request query for a point cloud data type. The system checks for existing data in the metadata RDF Graph. Since no existing data is available to answer the operator's request, a scanning mission for the region is automatically generated and delegated to a DJI UAV (i.e. Matrice 100) to acquire raw laser data, which is saved in the UAV's KDB. After the scanning mission is performed, the data is combined into a point cloud covering the blue area. The data is then synchronized between the DJI platform and the ground operator and it is displayed in the user interface.

In the second mission, the ground operator selects another area (i.e. red area in Fig. 21) in the user interface and executes a data request query for a point cloud data type. As existing data in the overlapping region is available to partially answer the request, some results can be reused and shown directly to the user. A scanning mission for the missing part of the red area is automatically generated and then delegated to a DJI UAV platform (i.e. Matrice 600). After this scanning mission is finished, the data is combined into a point cloud covering the red area, including data previously gathered. The data is then synchronized between the DJI UAV platform and the ground operator and it is displayed in the user interface, as shown on Fig. 22.

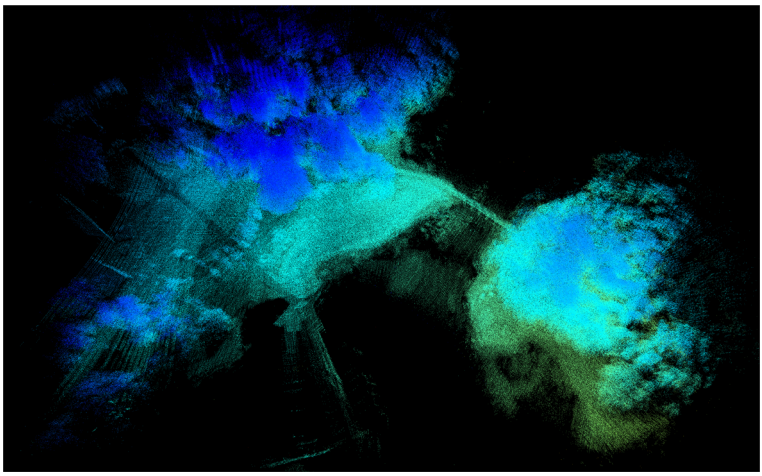


Fig. 22 Final results of the exploration

5 Related work

RDF Graph Synchronization A naive approach to RDF Graph Synchronization is for agents to exchange their complete RDF Graphs and integrate the respective triples. However, this leads to a significant communication overhead, as every time a change is made, the entire document has to be exchanged. To address this problem, RDFSync [55] works by clustering an RDF Graph and associating each cluster with a unique hash. These clusters are very similar to the revisions used in the RGS$^{\oplus}$ System, except that in RDFSync they are created from the full RDF Graph using a clustering algorithm, while in the RGS$^{\oplus}$ System, the revisions are a by-product of the evolution of the document. In RDFSync, when two agents want to synchronize an RDF Graph, they exchange their list of clusters and hashes. Then they request from each other the missing clusters, which leads to more efficient communication than exchanging the entire document.

RDFSync is a one-to-one algorithm where agents synchronize with each other in pairs. Such an approach does not scale well to large multi-agent scenarios with many changes to the graph. The algorithm used in the RGS$^{\oplus}$ System solves this problem by introducing a graph of revisions, which allows to very efficiently decide which revisions are missing. That implies there is no need to exchange the hashes representing all the changes, just the hash of the current revision, and then backtrack to a common revision.

A many-to-many synchronization algorithm for RDF Graphs has been proposed in [5]. C-Set allows each agent to broadcast their changes to all the other agents, and it offers a guarantee of synchronization no matter in which order the changes are received. However, in C-Set, deleted triples remain in the store and have to be garbage collected, which is an operation that has to be completed simultaneously by all the agents. This synchronous operation is problematic in case of unreliable communication, or with a dynamic set of agents.

RDF Graph Versioning A more advanced solution to synchronization of RDF Graphs is to use a versioning system and to exchange the revisions between the agents. A natural approach for versioning RDF Graph is to use a text-based versioning system, such as Git, as presented in [3, 4]. Such a system encodes the source code as text files, where the deltas between two different code revisions are added, deleted, and merged text lines. While the

RDF Graphs could also be encoded as text, using such a representation becomes problematic. Two semantically identical graphs can be encoded in an infinite number of ways depending on the actual text format used, for example, using different orders of RDF Triples or the number of spaces between the terms. A system that uses text-based representation would essentially create new revisions based on these encoding differences even if the two versions of the RDF Graph are identical. A simple solution would be to adhere to a very strict encoding of an RDF Graph (order, spaces, etc.), which is the solution adopted in [3], but even such a solution could lead to merge conflicts that cannot be resolved automatically. Such a conflict would occur if two branches change the same triple. A more efficient approach is to use a specialized system for synchronization and versioning control of RDF Graphs, such as the RGS$^{\oplus}$ System or the Quit Store presented in [4].

R&Wbase [57] is a Git inspired system for versioning of RDF Graphs. Unlike Git, which encodes revisions as textual differences, the R&Wbase encodes revisions as RDF Deltas [11]. This approach was extended in [4] to handle conflict resolution as the Quit Store. The RGS$^{\oplus}$ System is an adaptation of Quit Store and R&Wbase to fully autonomous agents. RGS$^{\oplus}$ System borrows the conflict-free three way merge behavior (see Sect. 3.5) of the Quit Store, while providing a merge algorithm that is linear in the number of triple changes, without being affected by the number of triples already existing in the graph. Additionally, in the RGS$^{\oplus}$ System, blank nodes have unique identifiers that allow for unambiguously identifying them (see Sect. 3.1.2). The identifiers are propagated with RDF Deltas, while in R&Wbase approach there is a risk of inconsistency when blank nodes are used. The distribution mechanism in R&Wbase relies on a statically selected central server, while we propose a protocol for selecting a master agent (see Sect. 3.2.4), and the distribution of revisions is fully peer-to-peer.

Distributed and Federated Database Systems The main challenge in Federated Database Systems (FDS) [9, 51] is to provide a unified query mechanism that hides data inconsistencies. This is often called the *database integration* problem [43]. The main challenge in Distributed Database systems [18, 40, 42] is the consistency problem where schema definitions and data should be consistent and equal across the different database instances.

Traditional database technology provides common techniques for storing and querying data. For databases running on a single computer and using smaller amounts of data, data consistency is less of an issue. But this does not scale for large numbers of users and large amounts of data. This has led to the development of Distributed Database Management Systems (DDBMS) [42]. Homogeneous DDBMS are systems where the schema definitions and data should be consistent and equal across the different database instances. A common approach to solving this consistency problem is to use database replication [40], where a master has write permissions on a subset of the data, and when changes occur, they are propagated to the slaves. Homogeneous DDBMS also solve the problem of load balancing between servers. In the case of Big Data, when a single computer cannot store all of its data in memory, it is necessary to use a Heterogeneous DDBMS approach, such as Spanner [18] or Dynamo [23]. Here, the system controls which server stores which data depending on user needs and system requirements. In [18], the authors propose a dynamic system to lock tables so that any instance can write on a subset of the table schema. This approach is also capable of handling inaccuracies in the timestamping of transactions. In [23] the data is replicated across multiple hosts. Dynamo trades off consistency for availability, where data is guaranteed to be eventually consistent, that is all updates reach all replicas eventually. Generally, these systems rely on a central server or policy for handling the spread and distribution of data. Their goal is to optimize the efficient accessibility of data by end-users and deal with load balancing between servers.

In Homogeneous and Heterogeneous DDBMS, the control on the availability and writing of data is left to the system. In Federated Database Systems (FDS), each database instance is autonomous, in the sense that there is no assumption as to the individual schema and the availability and location of where specific data resides. What is of particular focus is the development of a common query mechanism across heterogeneous databases. Such systems have to deal with many types of data inconsistencies, such as naming of concepts, precision, schema alignment, etc.

FDS attempts to provide unified query mechanisms that hide data inconsistencies in order to deal with the *database integration* problem [43]. A solution to the distributed mapping of integrated answers to queries was proposed in [51], where the system is required to be static, and each database instance must remain in the federation. A more dynamic approach was recently proposed in [9], where the FDS is implemented as a graph. When a query is executed, it is propagated along the graph and at each node the results are aggregated, correcting the inconsistencies incrementally.

The RGS$^{\oplus}$ System shares some ideas from both DDMS and FDS technologies. A challenge in DDMS is usually to guarantee consistency of the data, while the RGS$^{\oplus}$ System does not require complete consistency of the semantic data stored in RDF Graphs. A form of *weak consistency* is guaranteed as discussed in Sect. 3.5. With FDS, the schema (i.e. structure of the data) and concepts (i.e. the meaning of the data) are the same for all agents. For this work, we assume that each agent has full autonomous control of what kind of data it stores. Agents can join and leave the federation at any time. This is different from previous work [51] with FDS where an agent can leave a federation only after obtaining a permission. In the application scenarios targeted, communication is assumed to be unreliable. Consequently, the approach proposed in this paper is designed to handle dynamic changes in the federation structure without any notice.

Semantic Web and Robotics The choice of representation for information and knowledge used in the RGS$^{\oplus}$ System, RDF Documents/Graphs, is inspired by Tim Berners-Lee's vision of a Semantic Web [2, 12, 32], where web pages in the World Wide Web are annotated with representations of their semantic content in the form of collections of RDF Triples (RDF Documents) and ontologies and linked across the WWW. These information structures can be reasoned about using logical inference mechanisms such as those based on Description Logics [6], in addition to related powerful ontology inference mechanisms such as OWL [6, 39]. The modern equivalent of these ideas has resulted in standardization [48] and the linked-data [58] research area of which knowledge graphs are a prime example. Many additional tools and technologies have been developed since the original idea of the Semantic Web was proposed and these are used in the backbone of many companies' knowledge-intensive products.

More recently, there has been a trend toward leveraging Semantic Web ideas with robotics [35–37, 41, 44, 53]. Many existing ontologies are available and useful for describing robotic systems [16, 49] and sensing for robotics [41].

Several frameworks using Semantic Web technologies have been implemented on robotics systems. The OpenRobots Ontology (ORO) [35] presents a processing framework leveraging Semantic Web technologies to enhance communication between robots and humans. ORO makes use of a *common sense* ontology, an events system, a model of the cognitive state of other agents, and a long-term memory. In [37], the authors present a method to connect low-level sensor data and high-level semantic information using RDF and an ontology for low-level information. One of the most mature frameworks combining Semantic Web technologies and robotics is *KnowRob* [7, 54], a knowledge processing framework for robotic agents.

KnowRob supports reasoning over semantic information while taking into account planning processes.

All of this related work where RDF technologies are used to represent the knowledge in a robot or an agent is the motivation for the use of RDF Graphs in the RGS$^{\oplus}$ System and why a system for synchronizing those graphs among a set of agents is being considered.

Communication and Multi-Agent Systems Multi-Agent Systems (MAS) [52, 59] have direct relevance to our work since we view each robotic system as an agent with capabilities to plan and reason. In MAS, the topics of distributed and common knowledge are central to logical approaches to multi-agent problem solving and inference [31]. In the RGS$^{\oplus}$ System, it is assumed that each agent has its own local knowledge in the form of RDF Documents/Graphs that are distributed across agents. The multi-agent system consisting of a team of agents has common knowledge in the form of the shared, synchronized RDF Documents/Graphs.

Communication among agents is also an important topic in MAS. A common approach for communication in MAS is to use an implementation of the Agent Communication Language (ACL) such as FIPA ACL [45]. ACL contains mechanisms for agents to query other agents and transfer *beliefs* between them. This provides an infrastructural mechanism for belief transfer but also requires a semantics for beliefs such as that used in the BDI approach [14, 47]. While the RGS$^{\oplus}$ System does not strictly use FIPA ACL, in the sense, that it does not use the standard messages, its communication protocol is inspired by those of FIPA ACL, and its messages can be expressed as FIPA TELL/ASK utterances.

One of the challenges with such multi-agent systems that is related to belief transfer is for agents to determine what agents to ask for certain information required during a mission. Sending a broadcast is not always practical, since for large numbers of agents, this approach could degrade the available bandwidth of the communication network. A potential solution involves using matchmaker agents [17] that hold information about which agents can answer questions. The approach used in the RGS$^{\oplus}$ System is much different because each agent's public/shared information is synchronized, so the collective knowledge from a team is accessible directly to each agent. Consequently, each agent can then internally query such information.

6 Conclusion

The RGS$^{\oplus}$ System for the synchronization of RDF Graphs among a dynamic set of agents has been proposed as the basis for a pragmatic knowledge management system. RGS$^{\oplus}$ provides a fully automatic approach to the RDF Graph Synchronization problem and is robust to unreliable communication through dynamic centralization. RGS$^{\oplus}$ extends existing synchronization approaches by providing a discovery mechanism that allows the handling of changes to the team of agents, whether they come online, go offline, or leave the communication range. The discovery mechanism also handles the automatic incorporation of changes created in concurrent branches. It builds a history of changes made to the RDF Graph. When two agents make concurrent changes to an RDF Graph, the changes are combined by a *merge* or *rebase* operations by a dynamically selected merge master. Once all the agents have received all the revisions, their local RDF Graphs become identical, and the knowledge can be considered to be synchronized. The RGS$^{\oplus}$ System is constructed to ensure that knowledge is shared as quickly as possible between agents and can be partially retrieved even before full synchronization has been achieved. This is because knowledge can be accessed by agents as soon as a revision has been received even before all merge operations are completed.

A major effort has been placed on incorporating resiliency into the algorithms and the resulting engineered system to deal with difficult problems associated with unreliable communication among agents, out-of-range issues, and agents entering and leaving mission environments. This resiliency has been demonstrated in a set of experimental evaluations performed in simulation and elsewhere in field robotics experimentation. The results of the experiments show that when disconnections occur, agents' RDF Graphs remain synchronized within their respective sub-groups (defined by remaining connections). Additionally, after communication is regained, the RGS$^{\oplus}$ system ensures that the synchronization process continues and respective RDF Graphs are synchronized among all involved agents. Results of empirical evaluations in simulation have been presented, showing the performance and scalability of the proposed RGS$^{\oplus}$ System and its applicability to realistic real-world deployments.

A theoretical analysis of the algorithmic complexity of the algorithms used in the RGS$^{\oplus}$ System has also been provided. The two main algorithms (merge and rebase) used during the synchronization process have been shown to have quadratic complexity. Additionally, the complexity analysis has been validated in a number of experiments.

The theoretical and empirical evaluations confirm that the RDF$^{\oplus}$ System is suitable for sharing low bandwidth semantic knowledge among dynamic teams of agents operating in challenging and realistic environments.

Funding Open access funding provided by Linköping University.

References

1. Ahmetaj, S., David, R., Polleres, A., & Šimkus, M. (2022). Repairing SHACL constraint violations using answer set programming. In *The semantic web—ISWC 2022: 21st international semantic web conference, virtual event, October 23–27, 2022, proceedings* (pp. 375–391). Springer.
2. Allemang, D., Hendler, J., & Gandon, F. (2020). *Semantic web for the working ontologist: Effective modeling for linked data, RDFS, and OWL* (3rd ed.). New York: Association for Computing Machinery.
3. Arndt, N., Radtke, N., & Martin, M. (2016). Distributed collaboration on RDF datasets using GIT: Towards the quit store. In *Proceedings of the 12th international conference on semantic systems (I-SEMANTICS)* (pp. 25–32). https://doi.org/10.1145/2993318.2993328
4. Arndt, N., Naumann, P., Radtke, N., Martin, M., & Marx, E. (2019). Decentralized collaborative knowledge management using git. *Journal of Web Semantics, 54*, 29–47.
5. Aslan, K., Molli, P., Skaf-Molli, H., & Weiss, S. (2011). C-Set: A commutative replicated data type for semantic stores. In *RED: Fourth international workshop on resource discovery. Heraklion, Greece*. https://hal.inria.fr/inria-00594590
6. Baader, F., Calvanese, D., Mcguinness, D., Nardi, D., & Patel-Schneider, P. (2007). The description logic handbook: Theory, implementation, and applications.
7. Beetz, M., Beßler, D., Haidu, A., Pomarlan, M., Bozcuoglu, A., & Bartels, G. (2018). Know rob 2.0—a 2nd generation knowledge processing framework for cognition-enabled robotic agents (pp. 512–519). https://doi.org/10.1109/ICRA.2018.8460964
8. Bellare, M., & Rogaway, P. (1996). The exact security of digital signatures-how to sign with RSA and RABIN. In *International conference on the theory and applications of cryptographic techniques* (pp. 399–416). Springer.

9. Bent, G., Dantressangle, P., Vyvyan, D., Mowshowitz, A., & Mitsou, V. (2008). A dynamic distributed federated database. In *Proceedings of the 2nd annual conference of the international technology alliance*.
10. Berners-Lee, T., & Connolly, D. (2004). Delta: An ontology for the distribution of differences between RDF graphs. In *World wide web* (p. 3). http://www.w3.org/designissues/diff
11. Berners-Lee, T., & Connolly, D. (2004). Delta: An ontology for the distribution of differences between RDF graphs. In *World wide web* (p. 3). https://www.w3.org/DesignIssues/Diff
12. Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The semantic web. *Scientific American, 284*(5), 34–43.
13. Bizer, C., & Schultz, A. (2009). The berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS), 5*(2), 1–24.
14. Bratman, M. (1987). *Intention, plans, and practical reason*. Cambridge, MA: Harvard University Press.
15. Brickley, D., Guha, R. V., & McBride, B. (2014). RDF schema 1.1. W3C recommendation 25, 2004–2014.
16. Carbonera, J. L., Fiorini, S. R., Prestes, E., Jorge, V. A. M., Abel, M., Madhavan, R., Locoro, A., Gonçalves, P., Haidegger, T., Barreto, M. E., & Schlenoff, C. (2013). Defining positioning in a core ontology for robotics. In *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems (IROS)* (pp. 1867–1872). https://doi.org/10.1109/IROS.2013.6696603
17. Chen, J. R., Wolfe, S. R., Wragg, S. D. (2000). A distributed multi-agent system for collaborative information management and sharing. In *Proceedings of the ninth international conference on information and knowledge management* (pp. 382–388). https://doi.org/10.1145/354756.354844
18. Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Woodford, D. (2013). Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*. https://doi.org/10.1145/2491245
19. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms* (2nd ed.). Cambridge: The MIT Press.
20. Cyganiak, R., Hyland-Wood, D., & Lanthaler, M. (2014). RDF 1.1 concepts and abstract syntax. W3C proposed recommendation.
21. Cyrille Berger. (2023). RGS+: RDF graph synchronization library for collaborative robotics [software]. https://gitlab.liu.se/lrs/kdb-gs-docker
22. de Champeaux, D. (1983). Bidirectional heuristic search again. *Journal of ACM, 30*(1), 22–32. https://doi.org/10.1145/322358.322360
23. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., & Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review, 41*(6), 205–220.
24. Denning, P. J. (2006). Hastily formed networks. *Communications of the ACM, 49*(4), 15–20.
25. U.S. Department of Commerce and National Institute of Standards and Technology: Secure Hash Standard—SHS: Federal Information Processing Standards Publication 180-4. CreateSpace Independent Publishing Platform, USA (2012).
26. Doherty, P., Kvarnström, J., & Szalas, A. (2012). Temporal composite actions with constraints. In *Proceedings of the 13th international conference on principles of knowledge representation and reasoning (KR)* (pp. 478–488). AAAI Press.
27. Doherty, P., Kvarnstrom, J., Wzorek, M., Rudol, P., Heintz, F., & Conte, G. (2015). HDRC3: A distributed hybrid deliberative/reactive architecture for unmanned aircraft systems. In Valavanis, K. P., Vachtsevanos, G. J. (Eds.), Handbook of unmanned aerial vehicles (pp. 849–952). Springer. https://doi.org/10.1007/978-90-481-9707-1_118
28. Doherty, P., Heintz, F., & Kvarnström, J. (2013). High-level mission specification and planning for collaborative unmanned aircraft systems using delegation. *Unmanned Systems, 1*(1), 75–119. https://doi.org/10.1142/S2301385013500052
29. Doherty, P., Berger, C., Rudol, P., & Wzorek, M. (2021). Hastily formed knowledge networks and distributed situation awareness for collaborative robotics. *Autonomous Intelligent Systems, 1*(1), 1–29.
30. Duerst, M., & Suignard, M. (2005). RFC 3987: Internationalized resource identifiers (IRIs). RFC 3987 (proposed standard). http://www.ietf.org/rfc/rfc3987.txt
31. Fagin, R., Halpern, J. Y., Moses, Y., & Vardi, M. Y. (1995). Reasoning about knowledge.
32. Hendler, J. (2001). Agents and the semantic web. *IEEE Intelligent Systems, 16*(2), 30–37. https://doi.org/10.1109/5254.920597
33. Knublauch, H., & Kontokostas, D. (2017). Shapes constraint language SHACL. W3C recommendation.
34. Lassila, O., & Swick, R. R. (1999). Resource description framework (RDF) model and syntax specification. Recommendation 22 February 1999 REC-rdf-syntax-19990222, W3C, Cambridge, MA. http://www.w3.org/TR/REC-rdf-syntax/

35. Lemaignan, S., Ros, R., Mösenlechner, L., Alami, R., & Beetz, M. (2010). Oro, a knowledge management platform for cognitive architectures in robotics. In *IEEE/RSJ international conference on intelligent robots and systems (IROS)* (pp. 3548–3553). https://doi.org/10.1109/IROS.2010.5649547
36. Lemaignan, S., Warnier, M., Sisbot, E., Clodic, A., & Alami, R. (2016). Artificial cognition for social human-robot interaction: An implementation. *Artificial Intelligence*. https://doi.org/10.1016/j.artint.2016.07.002
37. Lim, G. H., Suh, I. H., & Suh, H. (2011). Ontology-based unified robot knowledge for service robots in indoor environments. *IEEE Transactions on Systems, Man, and Cybernetics—Part A: Systems and Humans, 41*(3), 492–509. https://doi.org/10.1109/TSMCA.2010.2076404
38. Mallea, A., Arenas, M., Hogan, A., & Polleres, A. (2011). On blank nodes. In *International semantic web conference* (pp. 421–437). Springer.
39. McGuinness, D. L., Van Harmelen, F., et al. (2004). OWL web ontology language overview. *W3C Recommendation, 10*(10).
40. Moiz, S. A., Sailaja, P., Venkataswamy, G., & Pal, S. N. (2011). Database replication: A survey of open source and commercial tools. *International Journal of Computer Applications, 13*(6), 1–8.
41. Neuhaus, H., & Compton, M. (2009). The semantic sensor network ontology. In *AGILE workshop on challenges in geospatial data Harmonisation* (pp. 1–33). Hannover, Germany.
42. Özsu, M. T., & Valduriez, P. (2020). Principles of distributed database systems.
43. Parent, C., & Spaccapietra, S. (1998). Database integration: An overview of issues and approaches. *Communications of the ACM, 41*(5), 166–178. https://doi.org/10.1145/276404.276408
44. Persson, J., Gallois, A., Bjoerkelund, A., Hafdell, L., Haage, M., Malec, J., Nilsson, K., & Nugues, P. (2010). A knowledge integration framework for robotics. In *International symposium on robotics (ISR)* (pp. 1–8).
45. Poslad, S. (2007). Specifying protocols for multi-agent systems interaction. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*. https://doi.org/10.1145/1293731.1293735
46. Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., & Ng, A. (2009). ROS: An open-source robot operating system. In: *Proceedings of the IEEE international conference on robotics and automation (ICRA) workshop on open source robotics. Kobe, Japan*.
47. Rao, A. S., & Georgeff, M. P. (1991). Modeling rational agents within a BDI-architecture. In: Allen, R. F. J., Sandewall, E. (Eds.), *Proceedings of the 2nd international conference on principles of knowledge representation and reasoning (KR'91)* (pp. 473–484). Morgan Kaufman.
48. RDF Standards. (2014). https://www.w3.org/RDF/. Accessed June 2021.
49. Schlenoff, C., Hong, T., Liu, C., Eastman, R., & Foufou, S. (2013). A literature review of sensor ontologies for manufacturing applications. In *Proceedings of the IEEE international symposium on robotic and sensors environments (ROSE)* (pp. 96–101). https://doi.org/10.1109/ROSE.2013.6698425
50. Seaborne, A., Manjunath, G., Bizer, C., Breslin, J., Das, S., Davis, I., Harris, S., Idehen, K., Corby, O., Kjernsmo, K., et al. (2008). SPARQL/update: A language for updating RDF graphs. *W3c Member Submission, 15*.
51. Sheth, A. P., & Larson, J. A. (1990). Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys (CSUR), 22*(3), 183–236. https://doi.org/10.1145/96602.96604
52. Shoham, Y., & Leyton-Brown, K. (2009). Multiagent systems—algorthmic, game-theoretic, and logical foundations.
53. Stenmark, M., Malec, J., Nilsson, K., & Robertsson, A. (2015). On distributed knowledge bases for robotized small-batch assembly. *IEEE Transactions on Automation Science and Engineering, 12*(2), 519–528. https://doi.org/10.1109/TASE.2015.2408264
54. Tenorth, M., & Beetz, M. (2015). Representations for robot knowledge in the knowrob framework. *Artificial Intelligence*.
55. Tummarello, G., Morbidoni, C., Bachmann-Gmür, R., & Erling, O. (2007). RDFSync: Efficient remote synchronization of RDF models. In K. Aberer, K. S. Choi, N. Noy, D. Allemang, K. I. Lee, L. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber, & P. Cudré-Mauroux (Eds.), *The semantic web* (pp. 537–551). Berlin: Springer.
56. United Nations Office for Coordination of Humanitarian Affairs: 2010 Haiti earthquake response: An after action review of response. https://www.insarag.org/images/stories/Documents/Haiti_AAR_Meeting/Printed_Haiti_Book_Low_Definition_Version.pdf. Accessed June 2021 (2010).
57. Vander Sande, M., Colpaert, P., Verborgh, R., Coppens, S., Mannens, E., & Van de Walle, R. (2013). R&wbase: Git for triples. In *LDOW*.
58. Wood, D., Zaidman, M., Ruth, L., & Hausenblas, M. (2014). Linked data: Structured data on the web.
59. Wooldridge, M. (2009). An introduction to multiagent systems (2nd edn).

60. Zeginis, D., Tzitzikas, Y., & Christophides, V. (2007). On the foundations of computing deltas between RDF models. In K. Aberer, K. S. Choi, N. Noy, D. Allemang, K. I. Lee, L. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber, & P. Cudré-Mauroux (Eds.), *The semantic web* (pp. 637–651). Berlin: Springer.
61. Zeginis, D., Tzitzikas, Y., & Christophides, V. (2011). On computing deltas of RDF/S knowledge bases. *ACM Transaction on the Web*. https://doi.org/10.1145/1993053.1993056