# Decision Making for Teams of Mobile Robots

Dissertation for the acquisition of the academic degree

**Doktor der Naturwissenschaften (Dr. rer. nat.)**

Submitted to the
Faculty of Electrical Engineering and Computer Science
University of Kassel, Germany

by Andreas Witsch

Kassel, February 2016

**Advisor:**
Prof. Dr. Kurt Geihs, University of Kassel

**Date of Defense:** 23rd August 2016

# Contents

# List of Figures

# List of Tables

# Abstract

In the past years, we could observe a significant amount of new robotic systems in science, industry, and everyday life. To reduce the complexity of these systems, the industry constructs robots that are designated for the execution of a specific task such as vacuum cleaning, autonomous driving, observation, or transportation operations. As a result, such robotic systems need to combine their capabilities to accomplish complex tasks that exceed the abilities of individual robots. However, to achieve emergent cooperative behavior, multi-robot systems require a decision process that copes with the communication challenges of the application domain.

This work investigates a distributed multi-robot decision process, which addresses unreliable and transient communication. This process composed by five steps, which we embedded into the ALICA multi-agent coordination language guided by the PROViDE negotiation middleware. The first step encompasses the specification of the decision problem, which is an integral part of the ALICA implementation. In our decision process, we describe multi-robot problems by continuous nonlinear constraint satisfaction problems. The second step addresses the calculation of solution proposals for this problem specification. Here, we propose an efficient solution algorithm that integrates incomplete local search and interval propagation techniques into a satisfiability solver, which forms a satisfiability modulo theories (SMT) solver. In the third decision step, the PROViDE middleware replicates the solution proposals among the robots. This replication process is parameterized with a *distribution method*, which determines the consistency properties of the proposals. In a fourth step, we investigate the conflict resolution. Therefore, an *acceptance method* ensures that each robot supports one of the replicated proposals. As we integrated the conflict resolution into the replication process, a sound selection of the distribution and acceptance methods leads to an eventual convergence of the robot proposals. In order to avoid the execution of conflicting proposals, the last step comprises a *decision method*, which selects a proposal for implementation in case the conflict resolution fails.

The evaluation of our work shows that the usage of incomplete solution techniques of the constraint satisfaction solver outperforms the runtime of other state-of-the-art approaches for many typical robotic problems. We further show by experimental setups and practical application in the RoboCup environment that our decision process is suitable for making quick decisions in the presence of packet loss and delay. Moreover, PROViDE requires less memory and bandwidth compared to other state-of-the-art middleware approaches.

# Acknowledgments

I want to thank my supervisor Prof. Dr. Kurt Geihs, who formed a scientific environment that gives all researchers the freedom to follow their interests and develop research ideas. The distributed systems laboratory was already a key part of my university education, where I found my mentors Roland Reichle, Philipp Baer, and Hendrik Skubch, who guided me as an undergraduate student and in my early days as PhD student. Their enthusiasm and inspiration made this work possible. In particular, their encouragement for the RoboCup Middle Size league team Carpe Noctem stimulated my curiosity about all types of scientific topics in the area of robotics research. Additionally, Carpe Noctem became a social institution for me, where I found numerous like-minded people, most notably: Tim Schlüter, Till Amma, Stefan Triller, Mike Bui, Martin Wetzel, Kai Liebscher, Kai Baumgart, Jens Wollenhaupt, Janosch Henze, Fridolin Gawora, Claas Lühring, Christof Hoppe, Andreas Scharf, Stefan Jakob, Eduard Belsch, Tobias Schellien, Lukas Will, Dennis Bachmann, Thore Braun, Michael Gottesleben, Nils Kubitza, Lisa Martmann, Phileas Vöcking, Paul Panin, and all the RICE exchange students. Thank you for the great time and all your efforts to advance and promote the great spirit of RoboCup.

In the same fashion, current and former colleagues supported me and the Carpe Noctem team. I want to thank Dominik Kirchner and Daniel Saur, who pushed me with their optimism and attitude. Stefan Niemczyk and Stephan Opfer gave me inspiration with their relentless enthusiasm and methodical way of working. I also want to highlight the support of Christoph Evers, Harun Baraki, Michael Wagner, Michael Zapf, Diana Reichle, Tareq Haque, Huu Tam Tran, Nugroho Fredivianus, Thao Van Nguyen, Alexander Jahl, and Nguyen Xuan Thang. My time with the VENUS project gave me comprehensive insights into other research disciplines, which I owe to all the great personalities working there. I also thank all colleagues that supported the everyday work such as Thomas Kleppe, Inken Poßner, Heidemarie Bleckwenn, Iris Rosbach, and all the others.

The best start for every education is a good school. I had the opportunity to attend the Ursulienenschule Fritzlar, the greatest school I could have ever hoped for. I also want to remark the importance of all my friends, particularly Philipp Karius and Florian Seute, who helped me find a good work-life balance when facing gloomy prospects. Of course, I have to thank my parents and siblings, who always supported me selflessly. Last but not least, I thank my wife Annalisa, who was at my side all the time and even changed her country of residence to make this work possible. You give me the spirit to move mountains and my sweet son Martin.

# Part I

# Preliminaries

# 1 Introduction

In recent years, a wide variety of robots has been constructed. This holds for many areas of our life such as research, industry, or even private environments. For instance, autonomous cleaning or lawn mower robots are used in many households. In the same way, the industry uses autonomous robots to deliver packets within warehouses or inspect leaks of gas pipes [72]. The *DARPA Robotics Challenge* [48] showed the ability of robot systems to solve complex tasks in generic settings. As a consequence, the participants developed highly complex humanoid robot systems to cope with the diversity of the problem domain. However, when investigating the challenges independently, present technologies allow for the construction of specialized robots that are able to solve these tasks even more convincingly. For example, the car driving challenge, which no humanoid robot could pass until today, has already been accomplished by the *Google* car [71] despite facing public road traffic. In order to solve more complex problems with the performance of specialized robots, the next logical evolution step is a team composed of heterogeneous collaborative robots.

Teams of specialized robots are able to perform tasks that exceed the capabilities of the individuals. This shifts the main challenges from mechatronics to computer science, as comprehensive *communication*, *coordination*, and *decision* frameworks are essential to exploit the potential of robotic teams. A central goal of such frameworks is the achievement of emerging behavior based on coherent robot decisions, e. g., to assign tasks in disaster scenarios regarding search, emergency supply, and rescue victims, while facing a weakened communication infrastructure. In recent years of research, many frameworks that tackle practical *communication* issues have been proposed. These borrow ideas from distributed systems research. For example, the usage of a communication middleware such as the Robot Operating System (ROS) [134] has become a common approach on which many of today's robotic systems rely on. This process is part of an ongoing standardization effort that enables plug-and-play solutions of robot components. Moreover, it facilitates the interaction between different robot vendors developed by independent manufacturers.

The development of cooperative multi-robot behaviors is a complex and time-consuming task [138]. Accordingly, many research activities investigate coordination approaches for teams of cooperative robots. *A Language for Interactive Cooperative Agents* (ALICA) [157] is such an approach, which is geared towards highly dynamic domains, while facing unreliable communication and sensory noise. ALICA robots estimate each other's actions based on their observations and refine these estimations through communication. This allows the coordination of tasks, execution states, and common *decisions* despite of lost or delayed network messages. In this work, we present a unified solution to achieve *coherent decisions for teams of autonomous robots* designed for the usage in coordination frameworks such as ALICA.

## 1.1 Motivation

Considering the amount of different environments today's robots are used in, generic modeling frameworks require a decision mechanism that is flexible enough to handle almost arbitrary situations. Many industrial systems make use of a central decision component that observes robots [18] or assigns tasks [14, 31, 101], which are then executed autonomously. Such a central approach forms a *single point of failure*, which can cause two possible problems: First, a robot can lose its connection to the central coordinator, e. g., due to limitations in the communication range or a hardware malfunction. In this case, the majority of the team can continue its operation. If communication range limitations cause the broken connection, fail-safe procedures may navigate the robot back to re-engage the communication range. In the second case, the central component might break down. As a result, the coordination among the robots is completely shut down. To address this problem, the central components are usually designed with redundancy, e. g., mirrored on a second host that can replace the original one. In either case, the question arises how to operate in unknown environments where cooperation has to be formed almost ad hoc, e. g., for search and rescue scenarios. Thus, we conclude that centralized approaches are inflexible and problem dependent.

An important factor for the coordination within teams of robots is to achieve coherent decisions. This allows different robots to execute individual tasks while achieving an emerging behavior. However, when applying a decentralized approach, we have to face another kind of problem: How can we achieve coherent decisions in a team of robots with potentially conflicting decisions? This problem has been examined extensively by the distributed systems community in the domain of fault-tolerant systems, which led to solutions such as consensus algorithms [35, 90, 123]. These aim especially at achieving replication transparency for server farms. In scenarios with autonomous mobile robots, configuration changes of the team composition have to be considered, where robots break down, leave the team, or new robots join.

Although distributed consensus algorithms address configuration changes partially, they cannot be adapted to situations where swift decisions are more important than consensus guarantees. This is particularly the case in fast changing environments such as in robotic soccer. These domains show that consensus is not always required. However, the communication overhead of consensus algorithms hampers fast decisions. Hence, decision procedures of robots have to be adaptable to the intended application domain and situation of the robot. This work aims for a comprehensive solution that realizes a flexible decision process for generic robot coordination frameworks supported by a negotiation middleware.

## 1.2 Problem Statement

Our approach is led by the observation that coherent decisions are mandatory to achieve cooperative and emerging behavior. However, in many cases consensus is not mandatory to achieve coherent robot decisions. For instance, if robots work on a cooperative construction project partitioned into subtasks, decisions concerning single subtasks do

not need consensus among all robots. Even the robots working on the same subtask do not always need consensus, e. g., if building material needs to be delivered to a certain construction area, the common decision only needs to ensure that all the required materials arrive in time. Hence, robots do not need to agree on the exact object that should be picked up next in order to solve the task.

According to the well-known results of Fischer, Lynch, and Paterson [57], consensus cannot be guaranteed if only a single *faulty robot* is present. Therefore, consensus algorithms focus on guaranteeing safety at the price of liveness. Additionally, they comprise a high amount of communication overhead and latency. Vice versa, standard "best effort" communication middlewares do not provide support for conflicts among the published data. Rather, the conflict resolution is delegated to the robot software developer. The key research question solved by this work is:

> *How can robots make coherent decisions adapted to the current situation and task?*

Generic and comprehensive coordination frameworks for teams of robots require a flexible decision process that allows for adapting the communication in a data-centric fashion. Furthermore, particular steps of the decision process need to be exchangeable, adaptable, or left out according to the problem-specific needs, e. g., to achieve fast decisions in highly dynamic environments or to ensure agreement if required so. However, the developer needs much experience in the design of distributed systems in order to integrate a robot decision process into the application. To support developers of multi-robot systems, we implement our decision process as middleware into the ALICA framework and answer the question:

> *How can a middleware framework support a multi-robot decision process?*

A key feature of decision-making is the computation of proposals that solve the targeted problem. The question many scientists in robotics research try to solve is:

> *How can robots compute decision proposals?*

This work focuses on two different aspects of this question: First, an efficient solver for continuous nonlinear constraint satisfaction problems (CNCSP) is presented, which can handle generic and domain-independent problem descriptions. Second, we present a generic interface that allows for exchanging the solution strategy. This enables the developer to select the most efficient solution strategy for a given problem. For example, although path-planning problems can be represented as CNCSP and the presented solver is able to find a solution for this type of problem, we cannot rely on the domain knowledge about physical relationships to improve its search heuristic. Hence, a dedicated path planner can compute the solution more efficiently. Furthermore, a dedicated approach simplifies problem description by implicitly relying on domain specific knowledge.

## 1.3 Scenarios

In the following, we describe the scenarios examined in this work. We consider them as archetypes to derive requirements for our approach, as these scenarios cover a broad range of challenging problems for multi-robot decision processes.

### 1.3.1 RoboCup Soccer

The RoboCup organization is performing yearly robotic competitions with the goal to compare research approaches of the participants and stimulate research in the fields of artificial intelligence, object recognition, mechatronics, and multi-robot coordination. RoboCup tournaments are divided into several leagues with different research focuses. This work specifically refers to the robotic soccer section, which has the overall goal to beat the soccer world champion with a team of fully autonomous robot soccer players in a match that complies with the official FIFA rules by 2050. The main research goals "concern cooperative multi-robot and multi-agent systems in dynamic adversarial environments" [142].

The primary reasons for choosing soccer as a challenge in RoboCup are first the clearly specified and structured environment, composed of white lines, black obstacles, and a colored ball. This forms an upper limit to the object recognition challenge and allows a strong focus on coordination problems. Second, due to the static field, it provides a reproducible setting to compare different approaches. Third, the objects in this environment provide a high dynamic. Hence, approaches need to be reactive and computationally efficient, which is a characteristic that distinguishes from typical laboratory settings. Finally, soccer is a well-known and popular sport. Thus, it requires little explanation of the basic rules and fosters public visibility.

In this work, we refer to the RoboCup Middle Size League (MSL) [162], which provides the heaviest (up to $40\,\mathrm{kg}$) and fastest (up to $5\,\mathrm{m/s}$) robots among all robotic soccer leagues. Each team consists of five fully autonomous robots playing on an $18\,\mathrm{m} \times 12\,\mathrm{m}$ sized field with a standard FIFA ball of yellow or red color. An industry PC, laptop, or other on-board computer forms a fully distributed setting that controls the team of robots. IEEE 802.11a or IEEE 802.11b WIFI are the only allowed inter-robot communication technologies. Human interaction during the game is restricted to the referee, who sends game commands to the robots. These describe the current game situation such as throw-in, kick-off, penalty, or free kick.

Most robots in the MSL have a size of $52\,\mathrm{cm} \times 52\,\mathrm{cm} \times 80\,\mathrm{cm}$ and are driven by wheels arranged to a holonomic drive, allowing them to move in arbitrary directions. Additionally, they are equipped with an electronic or pneumatic kicking device, which is able to kick the ball with a speed of over $12\,\mathrm{m/s}$. The main sensor to observe the environment is a camera pointing towards a convex mirror on top of the robots, allowing an omni-directional view. This allows for detecting obstacles, lines, and ball positions with a frequency of around $30\,\mathrm{Hz}$ and up to a distance of $6\,\mathrm{m}$ to $10\,\mathrm{m}$ depending on the light conditions.

The RoboCup soccer scenario is particularly interesting for this work, as it requires swift and coherent reactions of the robots to cope with the high dynamic of the game. Furthermore, the WIFI connections at tournaments are often error prone and led in exceptional cases to packet loss rates of $50\,\%$, which we partially observed at the world championships 2013. These are mostly induced by the large amount of appearing access points and network traffic. Due to the frequency overlap of the WIFI channels, the lower transmission layers cause packet collisions, although the participants are connected to a different access point. In conclusion, a fault-tolerant decentralized system design is required.

### 1.3.2 Autonomous Driving

The *DARPA Grand Challenge* [30] that took place in 2004, 2005, and 2007, generated research efforts in almost all automobile manufacturer companies and many universities worldwide. The initial challenges in autonomous driving were in efficient and reliable sensor data processing and single-robot decision making. Therefore, autonomous cars are equipped with expensive laser rangefinders, radars, cameras, and LiDARs. To process the high amount of data, autonomous cars are provided with powerful computational units with the goal to extract the position on the street, obstacles, and road signs. Many results could be applied successfully in intelligent driver assistance systems such as parking assist systems or lane departure warning systems. By combining all these technologies, the *Google* car [71] did the first fully autonomous trips in public road traffic in 2012.

In parallel, the automobile industry developed communication standards for vehicles and infrastructure of public road traffic. These technologies are called *Car2Car* and *Car2X* communication, respectively. Here, a wide agreement on the lower layers of the communication protocol like IEEE 802.11p [36] exists. This standard specifies the data link and physical layer for vehicular environments focusing on high movement speeds of the participants. Mostly, Car2Car communication is realized as *vehicular ad hoc networks* (VANETs). Besides the fact that VANETs operate at $5.9\,\mathrm{GHz}$, they behave similar to other IEEE 802.11 standards with respect to packet loss and delay, as shown by Bilgin and Gungor [21].

As a next step, car manufacturers investigate cooperative cars. This investigation is caused by the insight that some accidents can only be prevented by cooperation between road users, e. g., due to incomplete or faulty information about the environment of an autonomous vehicle. Hence, cooperation is mandatory to increase safety [177].

In contrast to RoboCup, the environment of autonomous driving scenarios provides new characteristics that allow deriving additional requirements for our approach: First, autonomous cars usually interact with each other only during a limited period, i. e., in dynamic teams. Second, the presence of a common communication infrastructure cannot be assumed. Note, some autonomous cars use cellular networks, which provide a good coverage but are still not area-wide available. Furthermore, the available bandwidth is shared among all devices, which can lead to a bottleneck. Finally, autonomous cars have to achieve a high degree of safety to avoid accidents and causalities.

### 1.3.3 Emergency Response

As a third scenario, we investigate emergency response operations. Recent years showed a growth of catastrophic disasters in number and complexity [70]. An example for this is the earthquake in the pacific ocean near Japan in March 2011, which caused a tsunami and the nuclear accident in Fukushima. One characteristic of this accident was the lack of information and broken communication infrastructure, which increased the scale of the disaster. We assume that the application of rescue robots could have stinted the catastrophe, backed up by the following quote:

> *“Close study of the disaster’s first 24 hours, before the cascade of failures carried reactor 1 beyond any hope of salvation, reveals clear inflection points where minor*

> *differences would have prevented events from spiraling out of control."* – IEEE Spectrum [163]

Unknown disaster environments with damaged communication infrastructure also imply new challenges for decisions of cooperative robots. Most notably, we have to assume that a viable communication link between the robots is only rarely available. However, since many decisions have only regional impact, the global quorum is not required to make a decision. Instead, agents can build their decisions based on a local quorum, which can be assumed to be within communication range. Here, the dynamic identification of such a local quorum is a key challenge.

Another important challenge in emergency response scenarios – considering the mobility of autonomous robots – is the distribution of knowledge about past decisions. For example, when a robot engages a communication island of a group of robots, which execute a task cooperatively, we need to determine the relevant past decisions for the engaging robot to enable reasoning regarding future decisions.

## 1.4 Requirements

In this work, we aim for a comprehensive robot decision process that is covering a broad range of robotic domains and scenarios, as described in the previous sections. As summarized in Table 1.1, these scenarios vary in agent connectivity, required decision time, incidence of team decisions, and safety requirements.

| Scenario | Robot Connectivity | Decision Speed | Team Decision Incidence | Decision Safety |
|---|---|---|---|---|
| RoboCup Soccer | Dense | Fast | ca. once per second | Low |
| Autonomous Driving | Medium | Fast | ca. once per minute | High |
| Emergency Response | Sparse | Medium | ca. once per minute | High |

**Table 1.1:** Targeted application domains and their properties.

We aim for a multi-purpose decision process for autonomous mobile robots. To foster the ease of use, we need an implementation that is included within a general robot control framework. Hence, our approach inherits the requirements from multi-agent coordination frameworks. Other requirements follow directly from the needs of the previously described scenarios:

**Full Distribution** In many scenarios, agents have to divide into sub-groups that try to achieve partial goals. Therefore, agents may operate separated locations. In conclusion, a communication link to a central decision making component cannot be guaranteed, especially when considering agent mobility in a range-limited communication setting. Hence, a cooperative decision making approach has to be fully decentralized.

**Incomplete Knowledge** In particular, large-scale or dynamic environments with many agents interacting with a potentially high number of relevant world entities make

it almost impossible to establish a coherent shared world model. This is caused by the bandwidth limitations of common network interfaces that prohibit all agent observations to be continuously shared within the team. As a result, agents cannot rely on a shared knowledge base to make common decisions.

**Support for Dynamic Team Configurations** Today's wireless communication interfaces have limitations in the communication range. Moreover, the environment constrains the communication range of robotic systems. For example, obstacles such as walls or uneven terrain may result in reflections and interference in the hardware layer, which may reduce the communication range of these devices drastically. Since autonomous mobile robots change their locations to implement their tasks, they can engage or leave each other's communication range. As a consequence, a distributed robot decision process needs the ability to adapt to new team configurations: Engaging agents have to be discovered and incorporated into the decision process to allow for decisions, which are coherent in a local sub-group.

**Robustness** Robot observations are usually noisy and error prone. Consequently, the decisions made upon these observations suffer from the same issues. In some cases, this might lead to conflicting decision proposals. Hence, one of the key requirements in distributed decision processes is the ability to resolve these conflicts to achieve robust coherent decisions.

**Fault Tolerance** Like in every distributed setting, network errors might occur, which usually results in lost data packets. A robot decision process needs to address the problem of missing or delayed packets such that agents can detect network errors and resulting conflicting decisions. This is necessary to enable a robot to correct wrong decisions to preserve a minimum level of safety properties.

**Availability** Another key aspect for distributed decision-making follows from the mobility of agents and their limited communication abilities especially with a low inter-robot connectivity like in emergency response scenarios, i. e., robots can move out of their communication range. Here, we need to ensure that agent decision data provides a high *availability*, e. g., when communication is not possible anymore.

**Efficiency** In scenarios like robotic soccer, where the environment changes quickly, swift reactions are mandatory. Thus, the decision process needs the capability to make fast decisions. Therefore, management operations need to be efficient to avoid the injection of high latencies. Additionally, the number of communication rounds needs to be restricted with respect to the needs of the given problem setting of the robots.

**Adaptability** Decision-making in a distributed setting always involves a trade-off between the safety and efficiency of the applied process. However, in our opinion, this trade-off requires a problem-specific response: In some time-critical situations, we might ignore safety properties in favor of faster decisions and vice versa. Hence, the decision process requires a mechanism that is adaptable to a given problem setting.

In summary, a generic multi-robot coordination framework requires a distributed decision method that can change its safety properties and decision velocity adapted to the given situation. Such a framework needs support for robustness and fault tolerance against network errors with the ability to handle unreliable and range-limited communication.

## 1.5 Approach

This work provides a comprehensive solution to realize a decision process for teams of autonomous robots. Our solution is inspired by a profound and flexible human *consensus decision-making* process from sociology [166], which guides the generation and negotiation of solution proposals. This process is adjustable to the problem setting, e. g., to achieve quick decisions or enable large number of participants and proposals. Therefore, it forms an ideal basis for this work.

For teams of mobile robots, we derived a five-step process: Determination of the problem statement, proposal computation, proposal distribution, conflict resolution, and value decision. Thereby, we identified two major advances for mobile robots compared to the sociological process: First, to speed up the overall process, robotic participants are generally not allowed to declare reservations. Hence, we focus on coherent solutions instead of consultation. Second, we propose an adequate middleware support for each decision step. This middleware supports the implementation of multi-robot decisions to satisfy the requirements described in Section 1.4.

The first decision step determines the problem statement. Here, we assume a common multi-agent program, which is distributed among the agents at the time of deployment. This program describes all problem definitions that might appear during runtime. However, impreciseness of robot observations might lead to incoherent problem statement. This idea is implemented by ALICA, which we use as a framework to express the problem statement.

The determination of proposals – performed in the second step – requires a generic solver that is able to compute solutions for a wide variety of robotic problems. We assume continuous nonlinear constraint satisfaction and optimization problems to be a suitable and extensive problem class for most robotic applications. However, considering the required response times of many robotic systems, former solution approaches require a critical amount of computation time to find solutions for this problem class. We present an efficient but incomplete solver that is able to cope with common problem instances of robotic domains in a reasonable time. Note, generic solvers rely on heuristics, which are less specialized to a specific problem setting. Thus, generic approaches cannot provide performances similar to those of specialized solvers, e. g., for planning, task allocation, or trajectory generation. In conclusion, the exchange of the solution technique can increase the speed of the solution process. Consequently, we provide an interface, which allows the integration of problem-specific solution algorithms.

For the final three decision steps, we aim for a versatile solution that explicitly supports the distributed decision process. In particular, support for the handling of concurrent and possibly conflicting proposals is required. Thus, we developed a data-centric communication middleware called PROViDE (*Proposal Replication for Value DEtermination*) that steers the negotiation of decision values in a replicated data space for proposals.

In order to enable applications that enforce swift decisions, only small latencies can be tolerated. Hence, there is a need to observe the network to quickly compensate lost network packets. Furthermore, the limitations in communication range of mobile robots hinder the availability of proposals and can lead to non-persistent decisions, e. g., because proposals may disappear from the global data space. However, as proposals usually require only little amount of data, their replication can usually be afforded. Additionally,

replication speeds up the data accessibility, as read operations can be realized on local data.

For practicability reasons, a decision process must be embedded in a coordination framework, which is able to distribute tasks among the robots and to implement control loops that execute robot behaviors. In other words, a multi-agent decision requires an adequate execution layer. Therefore, we embedded our decision process into the ALICA coordination framework, which provides domain independence without a central coordinator. Furthermore, we inherit ALICA's locality principle that allows the determination of a validity scope for decisions. Note that our solution is designed for the usage within ALICA but its general concept can be transferred to other coordination languages. We define the following assumptions and limitations for our solution:

1. PROViDE requires an intelligent routing protocol. In case no direct connection between two agents within the same *group* is possible, the routing protocol relays the relevant network packets to ensure that all robots can be reached. We call such a group the current *team* of a robot.
2. Our solution assumes that no corrupted packets can be transmitted – vice versa successfully delivered network packets are assumed to be correct. In standard wireless communication technologies such as IEEE 802.11g [173], this is usually achieved by checksums that are appended to network packets and allow the detection of faulty bits.
3. The described decision process does not guarantee safety of its decisions. Hence, bad network conditions can lead to conflicting decisions among the robots. It is our goal to achieve a sufficiently high probability for coherent decisions in typical robotic application domains. The results of Fischer, Lynch, and Paterson [57] make this limitation inevitable.
4. The goal of our approach is to achieve coherent decisions. However, this does not necessarily imply that agreement is present.

## 1.6 Contributions

The main contribution of this thesis is a comprehensive middleware, which supports a decision process for autonomous mobile robots in adverse and dynamic environments. Additionally, we make several contributions to subareas of distributed decision-making:

1. To provide conflict resolution, the decision process explicitly distinguishes decision values from solution proposals of individuals.
2. The decision process systematically separates the determination of robot decisions values from the algorithms that implement the decisions.
3. To compute decision proposals, we combine a DPLL algorithm [47] with an incomplete and distributed constraint satisfaction solver based on local searches. Thus, we achieve considerable speed enhancement for the solution of many continuous nonlinear constraint satisfaction problems.

4. We embed the process within a multi-robot coordination language, which forms a systematic approach for the execution and implementation of distributed decisions.
5. Our solution is guided by the PROViDE middleware, which encompasses a virtual shared data space, supports the publish-subscribe concept, and provides clock synchronization.
6. The integration of PROViDE into ALICA allows an explicit specification of the trade-off between safety and efficiency for the developer adapted to the current situation of the robot.

## 1.7 Structure of this Work

This work is structured in three parts. The first part states the foundations for our work and brings our work in line with existing scientific results. Afterwards, the second part describes the solution and the steps of the multi-robot decision process in detail. The last part provides an assessment of our solution that shows its properties and applicability. The content of the following chapters can be summarized as follows:

**Chapter 2:** We explain the basic terms for this thesis and give an overview of common communication techniques and organization of multi-agent systems.

**Chapter 3:** We present the coordination language ALICA, which hosts our decision process and supports the specification of multi-agent decision problems.

**Chapter 4:** An analysis and a comparison of existing approaches with our solution are presented.

**Chapter 5:** The presentation of our solution starts with an overview of the multi-agent decision process.

**Chapter 6:** Before negotiation can take place, agents need to compute proposal values that can be shared within the team. We give examples of how proposals can be determined and present a generic constraint satisfaction solver in detail.

**Chapter 7:** The PROViDE middleware that executes the negotiation of value proposals is presented. Here, we also describe the theoretical properties of our solution.

**Chapter 8:** After the description of the concepts, we give an overview of the PROViDE reference architecture.

**Chapter 9:** For the implementation of the negotiated decision, our decision process requires a sophisticated execution framework. Chapter 9 explains the integration of PROViDE into ALICA.

**Chapter 10:** The assessment starts with a runtime analysis of the present constraint satisfaction solver to show its general applicability and computational efficiency.

**Chapter 11:** Some properties of PROViDE require practical evaluation. In particular, we analyze the latency, bandwidth requirements, scalability, and impact of packet loss on the negotiation process.

**Chapter 12:** To show the versatility of the negotiation process and middleware, we present application examples, which highlight the practical value of our approach.

**Chapter 13:** We conclude with a summary and future work.

# 2 Foundations

This chapter describes the foundational knowledge for this work and is composed of three parts. First, we present multi-agent systems, types of teamwork, and their organization structures. Second, we summarize the most common types of data distribution methods for multi-robot systems. Third, we conclude with basic definitions for replication in distributed systems.

## 2.1 Multi-Agent Systems

In general, there is no universally accepted definition for the term *agent*. In this thesis, we rely on the definition of the *rational agent* by Russell and Norvig [145] that is also used in the context of multi-agent systems by Woolridge [180, p. 15]:

> *"An agent is a computer system situated in some environment and that is capable of autonomous action in this environment in order to meet its design objectives"*

We see a *robot* as a physical instance controlled by an agent, which allows the agent to interact with the physical world through sensors and actuators. In the research field of robotics, many publications use the term *robot* as an instance of an *agent*. Nevertheless, we interpret both terms equally in favor of a better readability. Both concepts – robot and agent – can also appear as *participants* in a communication process. Note that several economic theories reject the model of a rational agent [98, p. 72], as some agents are not fully controlled by their underlying logic, e. g., human beings, who also incorporate an emotional level.

According to Woolridge [180, p. 105], single-agent systems are rare in practical applications. In particular, most agents interact with parts of their environment that are again intelligent agents. This classifies such a system as a multi-agent system (MAS). For example in the *DARPA Robotics Challange* [48], robots interacted with human users who supported the robots with their knowledge to facilitate the tasks.

Multi-agent systems are controlled by various mechanisms and can appear in different organization forms. In the next sections, we will describe the most important concepts of multi-agent systems in the context of this thesis.

### 2.1.1 MAPE-K Cycle

In practice, most rational agents rely on the *MAPE-K cycle* [78, 97] (see Figure 2.1), which is a model to characterize the processing cycle of autonomous agents. *MAPE-K* is an acronym for each component of the processing step: Monitor, Analyze, Plan,

Execute, and Knowledge. Similar to other rational agents, the control loop perceives the environment through sensors and processes the acquired data in order to extract more abstract information such as positions, velocities, or other states of objects in the environment. Based on the results of this analysis, agents plan and decide next steps and realize them through their actuators. These actuators manipulate the environment of the agent in order to fulfill the agent's task. The communication interface of a robot does also act as an actuator or a sensor, respectively, as it is able to generate or sense electromagnetic pulses.

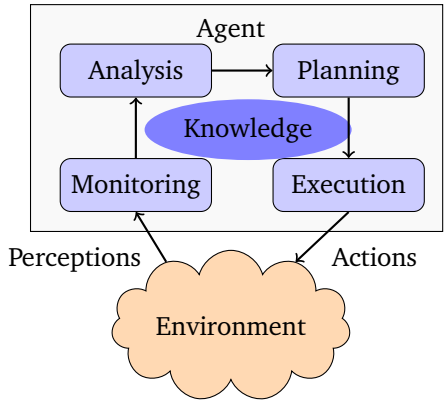



**Figure 2.1:** Agent-environment relation (*MAPE-K Cycle*).

The whole *MAPE-K* process is supported by domain-specific knowledge, which is mostly provided by a human expert. In other cases, it can result from machine learning techniques such as reinforcement learning or Bayesian inference, which use the agents' experience to improve its behavior. Hence, these techniques often rely on a utility function that measures the *usefulness* of a certain state or action. In the context of this work, we consider other agents as a source of knowledge, e. g., by their transmitted observations or decision data.

### 2.1.2 BDI Agents

*BDI agents* adopt the decision theory of "human practical reasoning" by Bratman [23] to compute their actions and decisions. Here, BDI is the abbreviation for *belief*, *desire*, and *intention*, which are the key components in BDI architectures:

**Beliefs** encode the assumed informational state of the environment, the agent itself, and other agents. Beliefs are usually stored in the *Beliefset* or *Beliefbase*. In the robotics domain, the data container is usually referred to as *world model*. The most common examples for beliefs refer to sensor readings. Yet, beliefs can also be higher level information as a result of sensor fusion processes as in [137].

**Desires** describe on a high abstraction level what the agent tries to achieve. For example, the desire of a soccer robot is to win the game, an autonomous car tries to transport its passengers to a certain location, and an emergency response robot aims for the rescue of disaster victims. Moreover, desires can be more precise. In that case, they

describe mandatory sub-goals for successful operations. In this context, goals denote a desire that is currently addressed by the agent, e. g., to perform a pass to a robot, which is then able to score a goal. In this thesis, we use a slightly different notion of goals, as described in Section 3.7.

**Intentions** describe actions the agent decided to execute, e.g., the execution of a certain plan or behavior. In the theory of BDI agents, plans are understood as a sequence of actions with the goal to implement at least one intention of the agent.

BDI agents received much attention in the research community [23, 135], resulting in the development of many BDI frameworks [24, 46, 76]. However, these frameworks did not gain acceptance in the area of robotics. One reason might be the fact that the implementations follow strict rules based on a multi-modal logic, which does not provide the required usability. Furthermore, BDI agents do not support multi-agent systems without any extensions. Nevertheless, multi-agent frameworks such as ALICA borrowed ideas from the BDI architecture while trying to make them more applicable for practical applications.

### 2.1.3 Cooperation in Multi-Agent Systems

The level of cooperation in a multi-agent system is an important criterion for its classification. There are two main notions to describe agents that cooperate with each other:

**Cooperative** describes agents that follow a common goal. More precisely, they do not have any "egoistic" goals. This is the strictest form of cooperation, which is usually used in sports, e. g., in robotic soccer.

**Collaborative** denotes agents that do not necessarily follow a common goal but support each other. In many cases, the agents form a temporary coalition to achieve a goal, which is annulled afterwards. More rarely, agents have compatible goals leading to an emerging behavior. For instance, companies that work on a common project typically use collaboration as form of interaction. Here, the major goal of companies remains the growth of their own profits. This can lead to the end of their collaboration if the collaboration shrinks their revenue.

Furthermore, some interacting multi-agent systems do not cooperate. These can be classified as:

**Neutral** interaction characterizes agents that follow their own goals without considering the other agents in their decisions. As a result, interaction only occurs by chance, e. g., when two independent cleaning robots perform collision avoidance.

**Antagonistic** behavior refers to agents that have competitive goals. Usually, this means that only one agent is able to achieve the intended goal, e. g., in competitive games where only one participant can win such as chess or Ludo.

These categories cannot been seen as distinct from each other. For example, robotic soccer, where each team uses a cooperative strategy to win the game, is also antagonistic, as both teams compete with each other to win the game. Furthermore, the same domain can also be modeled using a different paradigm. For example, in human soccer, we can

observe *antagonistic* behavior when single players try to score a goal on their own instead of increasing the chances by passing the ball to a better positioned teammate.

This thesis examines cooperative as well as collaborative systems. Therefore, our solution considers proposals from all agents participating in the same team. The selection mechanism for the final value determines whether a collaborative or cooperative decision is made.

### 2.1.4 Organization of Multi-Agent Systems

Cooperative or collaborative agents underlie an organizational structure, which specifies a set of relationships, roles, or authorities. In special cases, the structure changes during runtime, e. g., adapted to the goal [77]. Furthermore, in some systems, the organization is rather implicit and informally represented.

As described by Carley and Gasser [33] or Sandholm et al. [148], the organizational structure and their realization can have an important impact on the performance of the system. Researchers agree on the fact that no general structure optimal for all situations exists [33]. In the following, we give an overview of the most important organization forms in multi-agent systems, which are described in full detail by Horling and Lesser [77]:

**Hierarchies** are organized as a tree. Agents close to the tree root have a higher priority than agents that are more distant. Usually, this also implies different requirements for the perspective of agents: Agents with a high priority have a more global perspective than agents with low priority. Interaction does only appear between connected agents and not across the arms of the tree. The main advantage of hierarchies is their clear structure that provides an ideal basis for divide and conquer algorithms. However, hierarchies also induce a single point of failure at the root node. Lastly, a tree structure is rigid, which makes adaptation to new tasks challenging.

**Holarchies** describe agents interacting in groups that can be controlled by one or more groups of a higher priority. This idea was inspired by the universe, which is divided into galaxies, solar systems, planets, and moons. It is also the most common organization of enterprises, which distinguish companies, groups, divisions, etc. The boundaries between the *holons* are less strict, which allows more autonomy and interactions across the tree.

**Coalitions** are formed as a temporary group of interacting agents, which makes these groups suitable for collaborations. They usually follow one or more common goals and disperse afterwards if no common goals exist anymore. Interactions between groups do not usually appear. Although nesting is possible, most coalitions follow a rather flat hierarchy.

**Congregations** are similar to coalitions with the difference that they are designed for a long-term cooperation. This organizational form benefits from complementary capabilities of the participants, as this facilitates the discovery of a valuable composition.

**Teams** are formed by agents to achieve a common goal. Agents can have own (sub-) goals that contribute to the goal the team agreed on. Within the team, many structures

are used to interact with each other and agents can take any role that supports the common intention. Multi-agent teams usually have an explicit representation of shared goals, mutual beliefs, or common plans.

**Societies** unite agents without a common goal. Instead, each agent follows their own goal and is guided by rules, norms, or laws that can achieve emerging behavior and limit the actions of the agents.

**Marketplaces** create a producer-consumer system, where some agents *sell* shared resources, tasks, services, or goods, which can be requested by *buying* agents. Hence, markets usually form a competitive environment. Additionally, sometimes agents with a dedicated role called *auctioneer* might guide the bidding process.

**Federations** group agents with common characteristics together. One distinct mediator or broker guides each group and interacts with others. Hence, federations provide autonomy under a central government. The unique characteristic of federations is the mediator, who is in charge of the communication and therefore uses status information, task requests, and capability descriptions to interact with other groups. This is particularly helpful for heterogeneous groups of agents.

**Matrix** organizations have several supervisors that are able to delegate, guide, or apply pressure. Hence, agents need enough autonomy to deal with possibly conflicting instructions.

**Compounds** describe the overlapping of different organization systems, e. g., when the form changes with respect to the currently executed task.

The decision process presented in this thesis is mainly designed for organizations with tight cooperation and flat hierarchies such as teams, federations, matrices, marketplaces, and congregations. However, we also introduce *scopes* that support the usage in hierarchies, holarchies, or compounds. The usage in societies is possible but not our targeted application, as no common decisions are required.

## 2.2 Data Distribution

The key ingredient for a multi-agent decision process is the distribution of data to realize communication. Therefore, our solution includes a middleware that facilitates and automates the data dissemination. This allows the decision process to abstract information from socket-based data exchange. In the following sections, we present the most important data distribution paradigms most modern communication middlewares rely on. Thereby, we conclude each section with a short analysis of their impact on the robotic domain and assess their appropriateness for our approach.

### 2.2.1 Message Passing

Message passing is a simple data distribution paradigm. It is mostly realized using a publish-subscribe pattern, which describes a producer-consumer relationship. Here, the middleware allows to instantiate a publisher or a subscriber object, which provides the

functionalities to transmit or receive messages, respectively. The *message* type and *topic*, which are specified at the application layer, identify message offers and subscriptions. This setting allows a $1 : n$ relation between publisher and subscriber, i. e., $n$ subscribers can receive messages from one single publisher.

It is common to specify a *callback function* for the message reception, which is called when a transmission has been received. These callbacks are often realized by a *thread pool* that allows assigning an idle thread to process an incoming message. In case all threads are busy when a message arrives, the message is stored temporarily in a message buffer until a thread becomes available. In case this buffer reaches its memory resource limit, messages have to be discarded or the "publish function call" needs to wait until new memory space is available. Therefore, such a callback mechanism fosters an asynchronous processing of communication events, similar to *interrupt service routines*.

The *message broker* is the core of almost all modern message passing middlewares. Its main function is to establish and maintain connections between publishers and subscribers. In particular, this includes the message transport from the subscriber to the publisher. Depending on the used middleware product, the message broker provides functionality for routing, service discovery, prioritization, or other quality of service features, e. g., time-based filtering for fast producers and slow consumers. Another important task of the message broker is encryption if a security service is present.

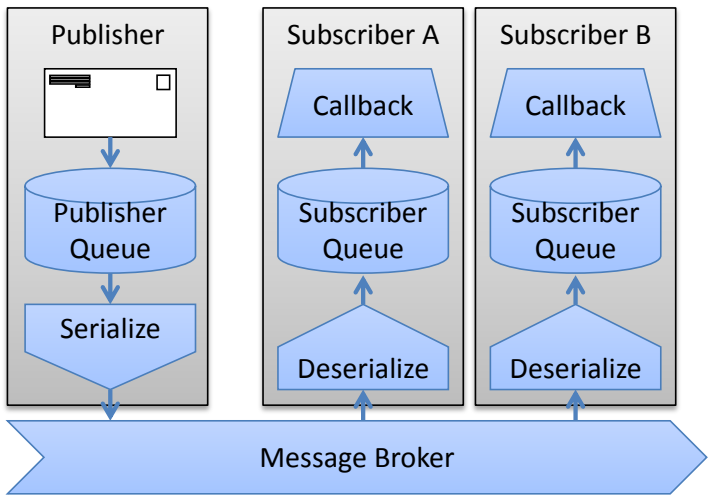



**Figure 2.2:** Message passing example with one publisher and two subscribers.

The ability to marshal messages for transmissions is another important feature of many message passing middlewares. Therefore, these middlewares provide an – ideally platform-independent – interface definition language (IDL). Such a language allows the developer to specify the relevant messages for the application. Afterwards, the message definitions are transformed into source code using an IDL compiler [117], which supplies serialization and deserialization methods [15]. An example of how a middleware can realize message passing is shown in Figure 2.2.

Summarized, the simplicity and loose coupling between message producer and consumer

are the advantages of message passing programming models. Due to the model transformation that is performed by the IDL compiler, it is easy to integrate platform independence by adding platform-specific code generation templates. The major disadvantage of message passing is its low level of abstraction. Hence, it provides a low application development support. Nevertheless, in robotic applications, message passing is currently the most popular programming model (cf. [134]), as its simplicity seems to outweigh the disadvantages. Note that our solution uses message passing as base technology. However, messages do not carry any semantic meaning of their information. In particular, they do not have a clear distinction between a value proposal and the final decision.

### 2.2.2 Remote Procedure Calls

Remote procedure calls (RPC) realize inter-process communication with distribution transparency for function calls. Hence, the developer can call a function at a remote server instance as if it was locally available. RPCs follow the *client-server model* [168, p. 42-55] and are usually implemented with the following procedure:

1. The client possesses a function stub with the same signature as the function that has to be called.
2. In case of a function invocation on the client, it sends a request to a known server, including the name of the called function and the serialized function parameters.
3. The server invokes its local function implementation with the deserialized parameters of the request.
4. When the execution is completed, the server serializes the return values and sends them back to the client.
5. After receiving the response from the server, the client deserializes the return value and delivers them to the function stub.

To generate the function stub, the middleware uses the interface definition encoded in an IDL [117]. In contrast to message passing (cf. Section 2.2.1), the IDL includes the relevant data types of the parameters, return values, and function signatures. In some frameworks, such as the *Common Object Request Broker Architecture* (CORBA) [117], the IDL can provide the complete object structure to provide transparency for an object-oriented programming model.

An interesting topic in the context of RPC middleware products is their error handling when network packets are lost or duplicated. Therefore, each RPC underlies one of the following error semantics:

**Maybe** does not provide any error handling. Hence, the developer cannot know whether the function is invoked or not, or if it is invoked multiple times. *Maybe* is the typical error semantic of vanilla UDP calls.

**At-least-once** remote procedure calls are executed at least once, even if packet loss occurs. For example in cases of packet duplication, it is possible that a single function call causes more than one method invocation. This error semantic is usually acceptable

for *idempotent* function calls where multiple executions do not cause side effects but provide the same result, e. g., for stateless services.

**At-most-once** guarantees that the function is not executed more than once. When packet loss occurs, e. g., due to broken servers, it is possible that the function cannot be invoked. This error semantic is of particular interest for information transmitted continuously. Here, many applications do not require a retransmission of outdated data.

**Exactly-once** ensures exactly one method invocation no matter which type of error occurs. Although this is the most desirable error semantic, it is generally not realizable considering all possible error types.

Note that these error semantics can be assigned to the delivery mechanism in asynchronous message passing middlewares. Due to the extensive use of an IDL for RPCs, this paradigm provides more development support and a higher level of abstraction than message passing. Modern middlewares provide extensive support for directory and naming services as well as security layers, life-cycle control, and support for concurrency [117]. In the field of robotics, most interactions are executed asynchronously and unidirectional. Thus, RPCs are only rarely used in the field of robotics although being part of the widely used *Robot Operating System* (ROS) [134]. For our decision process, we consider RPCs as inadequate, since a request-response semantic seems too inflexible for the needs of an adaptable communication protocol.

### 2.2.3 Distributed Shared Memory

Distributed shared memory (DSM) is derived from the common memory of computer clusters. It allows processes that are distributed over the network to access the same memory, e. g., by declaring, deleting, reading, or writing variable values. To realize this pattern, two approaches exist: First, DSM can use a central component that is managing the memory by granting read or write access. Second, the data can be replicated in the network to provide fault tolerance. Replication induces the problem of achieving proper consistency guarantees. A more detailed discussion of replication can be found in Section 2.3.

In order to access the DSM, processes requires a *memory manager* that provides access methods for reading and writing. Furthermore, the memory manager locates the requested data in the network and hides the communication protocol for data access, i. e., when a process writes or reads data that is physically located on a remote computer, the memory manager queries this data via the network and sends, respectively, a write request to the host. In this context, some implementations provide caching functionality to allow a more efficient data access. However, caching can be seen as some form of replication, which can again cause consistency problems [100].

The goal of most DSM systems is to extend the locally available memory to a large virtual memory space. Hence, the major advantage of today's implementations is their scalability. Furthermore, the implicit distribution can be handled easily by the developer, who does not need send or receive primitives. As already mentioned, the major disadvantages are the

provided consistency properties and the problems with concurrent access. Furthermore, the access to remote data induces higher delays than local data.

In multi-robot systems, DSM is a rarely used approach. In our opinion, this is because hardly any implementations focusing on the robotics domain are available. Therefore, many developers of multi-robot systems do not seem to be used to the DSM paradigm. Nevertheless, we claim that DSM can provide the simplicity that is requested by the developers, caused by the implicit data distribution. The solution presented later in this thesis explicitly addresses the drawbacks of DSM in robotic systems, especially considering unreliable communication and concurrency issues.

### 2.2.4 Data-Centric Dissemination

Middlewares with data-centric dissemination use an orthogonal communication concept than the ones presented before. Instead of sending messages via specified communication channels, data-centric approaches usually manage data within a *distributed shared memory* based on the *contained parameters* [126]. Briefly, a sender transmits its data into some kind of "middleware cloud", which allows all participants manipulation or withdrawal.

Besides an identifier and type, the parameters mostly describe quality of service features such as validity time, rate of publication, rate of subscription, or reliability. On the one hand, this allows the infrastructure to be aware of these parameters and use them to improve routing, error handling, or other network features. On the other hand, the middleware can use parameters to invalidate, cache, or filter the data. Altogether, data-centric approaches allow the middleware to be smarter concerning its data handling.

The most popular standard for data-centric middlewares is the *Data Distribution Service* (DDS) [125], which has, among others, been implemented by *OpenSplice DDS* [150] coined by *PrismTech*[1]. This implementation uses a publish-subscribe architecture to address data in a shared memory. This shows that data-centric dissemination can be combined with other communication paradigms.

Although data-centric communication was adopted in 2004, only a few suitable standards exist besides DDS. However, recent research efforts are making progress in this direction [12]. In particular, the ROS community is currently pushing a DDS-based ROS implementation [143]. Our solution can be seen as a data-centric negotiation middleware using a shared memory view to address decision data.

## 2.3 Replication

The concept of replication describes the approach of distributing data copies on different machines in the network. Generally, this approach has two major advantages: First, it increases the reliability of data. For example, if one copy vanishes from the network, e. g., due to a server crash, it can easily be replaced by one of the others. Furthermore, it is easier to detect corrupted data and provide fault tolerance. Second, replication is used to

[1] http://www.prismtech.com/dds-community

improve the performance of data access. For example, when many clients try to access replicated data, their requests can be distributed to one of the copies.

For the usage of replication in multi-robot scenarios, the two advantages manifest slightly differently. On the one hand, the increased reliability of data leads to an increased availability. This is particularly interesting when considering agents that might break down or leave the communication range. On the other hand, replicating the data to all available robots leads to local accessibility. Hence, robot requests are not influenced by the delay of the present network. Since we identified *availability* and *efficiency* as key requirements requirements for multi-agent decision processes, replication forms the base technology for our solution.

The advantages also include two disadvantages that have to be addressed: First, the data copies need to be kept up to date if changes occur, and second, agreement on certain values has to be ensured. The following sections describe these issues in detail.

### 2.3.1 Consistency

The main issue with replication in distributed systems is consistency of data copies. In this work, we define consistency according to Tanenbaum and Steen[168, p. 291]:

> *"Informally, this means that when one copy is updated, we need to ensure that other copies are updated as well; otherwise the replicas will no longer be the same."*

The "level of consistency" provided by a distributed system is classified by a *consistency model*, which is a contract between the processes and data store. This contract describes in which order processes read – mostly concurrently – write operations of others. Therefore, each application requires a definition regarding which behavior is acceptable for concurrent writes. The most relevant consistency models are:

**Strict Consistency** is the strongest consistency model, ensuring that *every* read operation returns the result of the most recent write operation. Hence, write operations have instantaneous effect on the complete data store. For the distributed case, strict consistency cannot be achieved, as global time would be required to determine the most current write operation.

**Sequential Consistency** ensures that all processes recognize the read and write operations in the same sequential order. This means that interleaving between concurrent operations is allowed as long as the interleaving follows the same order at all processes. As a consequence, sequential consistency does not allow to determine the most recent write operation.

**Causal Consistency** requires causally related write operations to be read by all processes in the same order. In contrast, the order of concurrent write operations might differ on remote machines. In this context, event $A$ has a causal relation to event $B$ if event $A$ *caused or influenced* event $B$, otherwise both events are called *concurrent*.

**FIFO Consistency** or **PRAM Consistency** ensures that all write operations of a single process are read in the same order by all other processes. However, concurrent write

operations may be seen in a different order by different processes. This allows shared memory applications to apply write operations of all processes to the memory in parallel. Hence, write operations of a process do not need to be stalled until other processes have finished their operations.

**Weak Consistency** holds if synchronization variables that can be used to ensure consistency for *critical sections* exist. Three properties hold for synchronization variables: First, synchronization variables must at least be sequentially consistent. Second, access to synchronization variables is only granted if all other write operations have been completed. Third, access to the data store is only possible if no operations on synchronization variables are pending [50].

**Eventual Consistency** aims for data stores where concurrent value updates are rather rare, and consistency violations can either be resolved or tolerated. *Eventual consistency* does not give a guarantee that replicas are consistent at any point in time. Instead, the underlying protocol of the system tries to converge to a state where replicas are consistent and when no updates take place.

Distributed system developers appreciate strong consistency guarantees. However, this usually comes with the price of a higher communication overhead and makes transactions slower. Recent research concluded from these facts that the consistency model should be chosen with respect to the given problem setting [170].

We further want to highlight the importance of the term *causality*. In the environment of distributed decision-making, a decision might causally depend on other decisions, e. g., when an autonomous car decides to cross an intersection, other cars should consider this decision in their own plans. For those cases, we have to ensure that the "trace of causally dependent decisions" can be considered by the decision process. The only exceptions to this rule apply when the system can tolerate or resolve inconsistencies.

The knowledge base for causally dependent decisions does not necessarily require causal consistency for its entire data. In Section 12.4, we show how to realize this use case with PROViDE. Aside from that, the decision process presented in this thesis supports *eventual consistency*, as other models imply that the guarantees hold for operations on multiple variables. If we relax the latter condition and regard variables as independent memory instances, our middleware solution provides *FIFO consistency*.

### 2.3.2 Consensus

In computer science, the term *consensus* has been essentially coined by the research on fault-tolerant systems. Thus, consensus is a valuable technology for distributed decisions in the presence of replication. In this work, we define a consensus problem as follows:

> In the consensus problem, *"each process has an initial value and all correct processes must agree on a single value"* [86, p. 513]

This definition does not imply that all robots (processes) consider the chosen value as the best solution. However, they agree to support this value and to ignore their "doubts". To this end, our definition conforms to consensus in sociology [25].

According to this definition, a solution for the consensus problem has to achieve three *safety* requirements [88]: First, the chosen value has to be one of the proposals (*non-triviality*). Second, a value can only be chosen once (*stability*), which implies that a process cannot change its value selection after it has been chosen. Third, all processes chose the same value (*consistency*). Note that here the term *consistency* does not refer to the consistency of replicated data but to the choice of different processes.

The safety properties ensure that only *correct* values can be chosen. In particular, this avoids arbitrary or unpredictable behavior among the processes. Applied to robot decision processes, this means that no uncoordinated behavior is tolerated. However, a further requirement is called the *liveness* property, which guarantees that the robots can act productively together: If a value has been proposed, then it will eventually be chosen.

Most consensus algorithms also make assumptions on the considered possible faults. Most notably, usually no byzantine faults are considered, which refers to agents that are still communicating and transmitting correct messages that deliberately try to sabotage the interaction [93]. Furthermore, agents are assumed to be able to fail, disappear, restart, operate asynchronously, and have arbitrary speeds. Moreover, the message transmissions can take arbitrarily long and result in duplicated or lost packets. Nevertheless, messages cannot be corrupted.

Solutions for solving the consensus problem usually involve four types of agents: *clients*, *learners*, *acceptors*, and *proposers* [90]. Depending on the implementation, one agent can perform multiple of these roles or even all together. When clients intend to propose a new value to eventually achieve consensus on, they transmit an according message to one of the proposers. The proposer tries to distribute the value among all acceptors. Once consensus has been established among all acceptors, a learner can eventually learn this value and deliver an according information to a requesting client.

Fischer, Lynch, and Paterson [57] proved that it is impossible to reach consensus in an asynchronous distributed system if only a single process is faulty. The key reason for the so-called *FLP impossibility* is the fact that it is not possible to distinguish between a very slow process or communication link and a crashed process. More precisely, the authors prove the existence of a *critical step*, which is the decision for a consensus value. This critical step can be performed by either a single or multiple independent *proposer* processes. If multiple processes decide independently for a consensus value, the decisions can occur with arbitrary permutations, which prohibits a well-defined transition. Hence, one proposer could declare consensus on a certain value, while another one does not. In case only a single proposer performs the critical step, it is still not possible to decide whether a response to the client is only slow or the proposer crashed. For the same reasons, no other process can take over the work of this distinguished proposer. As a result, it may happen that other processes such as the client wait forever.

Many robotic frameworks do not tackle the consensus problem explicitly. Instead, they simply assume synchronous communication links among the agents realized with some timeout. As this timeout is essential to detect broken robots and communication links, our approach is addressing this issue explicitly, see Section 7.3.3 for details.

### 2.3.3 State Machine Replication

Most distributed systems use replication with the goal to ensure availability in the presence of faulty processes [34, 79, 165]. Here, a faulty process can be replaced by an operative replicate, which is created through *state machine replication*. Thereby, the algorithm ensures consensus on one state of a finite-state machine, which usually receives inputs and produces outputs. Hence, almost all programs can be represented as a state machine. Moreover, most behavior modeling frameworks for autonomous robots are realized as state machines.

State machine replication allows for building highly reliable state machines. Here, all servers have the same state machine implementation and a log of all past commands is replicated. Whenever consensus on a new value in the log has been established, it can be applied to the state machine. As this ensures that all state machines receive the same commands in the same order, the state machines have the same state and will produce the same outputs. Thus, in case of a crash, the remaining servers can continue to provide service. Figure 2.3 depicts state machine replication for three servers.

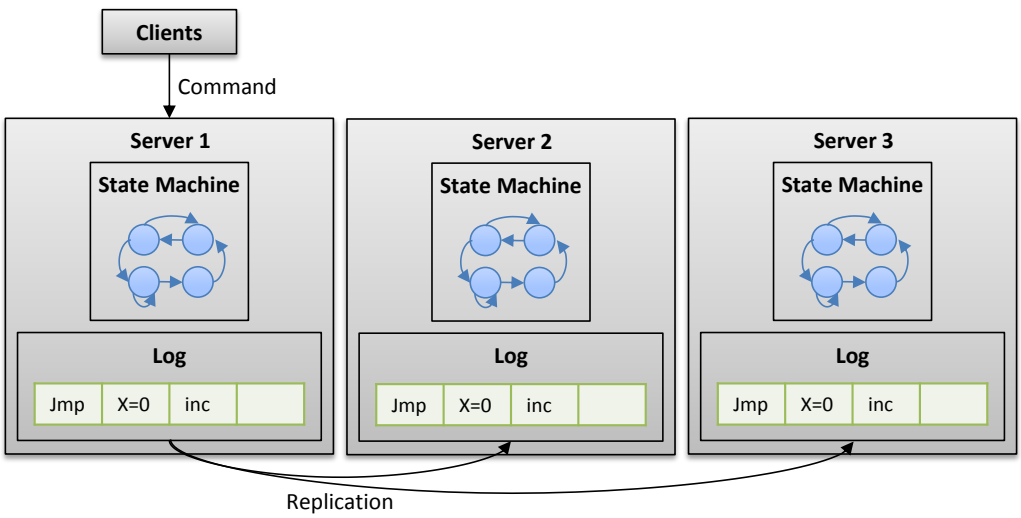



**Figure 2.3:** Example of state machine replication with three servers.

Despite its theoretical capabilities to coordinate multi-agent systems (cf. Chapter 3), state machine replication is only rarely used in the robotics domain. The main reason is the overhead of replicating the log, which is not necessary for most applications. Instead, it suffices to distribute the current states of all robots. In other cases, a central coordinator is elected to ensure coherent decisions [159]. However, both methods can be adapted to the given problem setting, resulting in a more efficient implementation with less communication overhead.

### 2.3.4 Coherent Decisions

Practical robotic applications do rarely require the safety or liveness properties guaranteed by consensus algorithms. Instead, they can rely on *coherent* decisions and still achieve emergent and coordinated behavior. A common definition for coherence in computer science is originated from the field of parallel computing. Here, cache or memory coherence is given if the read operation of CPU $A$ always returns the value written by CPU $B$ and whenever the operation on CPU $B$ is the most recent write operation [168, p. 17].

Since this thesis concerns decision processes of multi-robot systems, we conceive coherence from a more general perspective. We call two decision values $x$ and $y$ coherent if the resulting behavior during execution leads to the same result. As robots act in a continuous world, there is no general measure for "same result". More formally, we assume a world of an acting robot that depends on a vector of continuous states $\vec{s}$. These states describe positions of all relevant objects and their dynamics. The decision values $x$ and $y$ are coherent if the Mahalanobis distance (defined as $D_M(\vec{s_x}, \vec{s_y}, S) = \sqrt{(\vec{s_x} - \vec{s_y})S^{-1}(\vec{s_x} - \vec{s_y})}$) of the world state vectors $\vec{s_x}$ and $\vec{s_y}$ is lower than some threshold $\delta_c$ with $\delta_c > 0$. Here, $\vec{s_x}$ and $\vec{s_y}$ describe the world after following decision $x$ and $y$, respectively.. As the dimensions within the state vector are generally neither independent nor have equal importance, the covariance matrix $S$ characterizes the distance measure. According to this definition, the simplest example of coherence is equality, as the Mahalanobis distance of equal vectors is always equal to zero.

A more complex example appears in standard situations of the RoboCup MSL. Here, a common strategy foresees that three robots of a team try to cover their opponents to prevent them from receiving a pass. Therefore, each robot is assigned to a position that is nearby an opposing robot and within the line of sight to the ball. When the robots decide for such a covering strategy, we can achieve coherence if the three most important positions are covered. It is important to ensure that no more than one robot tries to cover each distinct position. Otherwise, one opposing robot will be uncovered, which would encourage the opponents to pass the ball to this uncovered robot. Hence, the consequent situation after the decision has been executed is completely different, resulting in a big value of $\delta_c$. To decide for coherent covering of positions, we at least require agreement on the assignment of robots to the opponents they are meant to cover. However, slight deviations of the cover positions based on the unequal knowledge bases of the robots will still result in almost the same game situation.

In this example, a less strict decision policy can be used in favor of enabling the robots to adapt faster to the movement of their opponents: If the decision process would be based on consensus, the adaptation to a movement of an opponent would require restarting the complete decision process for every position change. In contrast, our approach allows the robots to adapt their positions independently without the use of communication, which allows faster decision changes and less bandwidth usage.

# 3 ALICA

Modeling cooperative behavior for autonomous teams of robots is a challenging and complex task. In particular, the robots need sophisticated mechanisms to synchronize their actions, assign tasks, and resolve conflicts considering the requirements of the targeted environment. For example, in the RoboCup MSL, agents have to act in a very dynamic environment. At the same time, the robots have to consider that network errors can cause packet loss and delay. Such environments require a framework that supports developers by fostering modular and maintainable software.

Since 2008, the research department "Distributed Systems" of the University of Kassel has been developing the *ALICA* (A Language for Cooperative Interactive Agents) [157, 159–161] framework, which provides a formally defined language to specify multi-agent behavior. Thereby, it considers dynamic environments and imperfect network communication. As our multi-robot decision process is embedded in ALICA, we describe ALICA in the next sections in detail. We first explain the syntax and semantics of ALICA. Afterwards, we describe the first-order language components of ALICA, on wich our decision process relies. Finally, we give an overview of the conflict resolution mechanisms that are implemented in ALICA.

## 3.1 Syntax

In this section, we describe the syntax of ALICA programs, which form a set of instructions that are executed on each agent participating in the team. Such a *program* contains basic single-agent action elements called *behaviors*, as described in Section 3.1.1. In order to describe sequences of behavior selections and robot tasks, behaviors are arranged in a higher-level concept called *plan*, see Section 3.1.2. A further concept of ALICA aims for the synchronization of actions when transitioning between behaviors, which is described in Section 3.1.3. Furthermore, a *role* is assigned to each agent that specifies its adequateness for the execution of tasks. A more detailed description of the role concept can be found in Section 3.1.4. Another language element advances ALICA to a first-order language by parameterizing plans with variables that are used as a further coordination mechanism and are described in Section 3.1.5.

### 3.1.1 Behaviors

As mentioned before, the atomic activities of ALICA agents are called behaviors. In practical cases, behaviors are usually realized as control loops that move an agent from

one place to another, grasp something in their environment, explore an unknown area, or the like. As behaviors need an interface that allows controlling the actuators of the agent, they are usually implemented in a high-level programming language such as C# or C++. Furthermore, behavior iterations are triggered by a user-defined event, e. g., when new sensor data is available or after a certain time period. Despite this triggering mechanism, ALICA does not regulate the internal structure of behaviors, which allows the developer to implement arbitrary control structures, e. g., a path-planning algorithm for obstacle avoidance.

In order to restrict the execution of reasonable behavior actions, ALICA allows for specifying *pre-* and *runtime conditions*. The precondition has to hold before a behavior is allowed to be executed. Hence, it specifies the initial valid world state required by the behavior, e. g., a robot can only grasp something if no other object occupies its actuator. In contrast, a runtime condition has to hold during the complete execution of the behavior, e. g., an autonomous car should not pass a crossing if no collision-free trajectory exists. Furthermore, a behavior can be configured with a postcondition, which describes the change in the agent's environment performed by the behavior. The main intention of these three conditions is the possibility to apply planning algorithms on a set of behaviors with a description of the final world state. Classical algorithms such as STRIPS planners [56] are capable of solving this problem class.

In order to address execution failures, ALICA behaviors have a termination state. It identifies whether the behaviors have been executed successfully or a failure occurred. In this context, failures are present if the implemented controller is not able to reach the control goal within the given constraints. This can for example allow a ball interception behavior in robotic soccer to signalize that a robot will not be able to catch the ball.

### 3.1.2 Plans

ALICA plans are a generalization of behaviors, which form a composition of single-agent actions to multi-agent finite-state machines. Each state machine represents one *task* that is executed by one or more agents, describes behavior sequences, and points to the initial state. Hence, a plan is a collection of state machines, which aim for common goals and are logically related to each other. Considering a standard situation in robotic soccer, a plan could include one task for a pass executor, one task for the pass receiver, and one task for two defending robots.

The directed edges connecting ALICA states are called *transitions*. Transitions contain a condition that changes the state of an agent from the origin state to the destination state. States can have more than one outgoing transition. Generally, the transitioning behavior is undefined if more than one outgoing transition becomes *true* simultaneously. However, this can be avoided by adding a conjunction to each condition with the complement of all other conditions that belong to the same state.

A plan state can host one or more ALICA plan elements such as behaviors or other plans. As a consequence, a hierarchical structure of plans can be implemented forming a special case of hierarchical task networks (HTN), which is a research matter of planning algorithms [112]. The basic advantage of such behavior descriptions is the decomposition

of the team behavior into small pieces. This leads to a better maintainability of the program and breaks down the complexity.

As plans are a generalization of behaviors, they inherit the concept of success and failure states. These states are modeled as terminal states of each state machine. Additionally, plans also inherit the notion of *pre-*, *runtime*, and *postconditions*.

Another component of a plan is a utility function, which computes the utility of a plan for a given assignment of robots to tasks in the currently known situation. A more detailed discussion on the task assignment can be found in Section 3.2.3.

Besides the task assignment, the utility function is used to select the best plan for a given situation from a set of possible plans. In ALICA, such a set of plans is called *plantype*. In practical cases, plantypes group plans that achieve the same goal or compatible goals. For example, in robotic soccer, different implementations of free kick plans could be grouped to a plantype. Here, the different plans could vary in their aggressiveness. In this case, the utility function determines the most promising plan during runtime. These could consider the opponents' maximum velocity to determine which strategy gains the most space towards the opposing goal.

An example of an ALICA program for agents that act in an emergency response scenario is shown in Figure 3.1. Note that each ALICA program is characterized by a single master plan, which represents the root node of the *plan tree*.

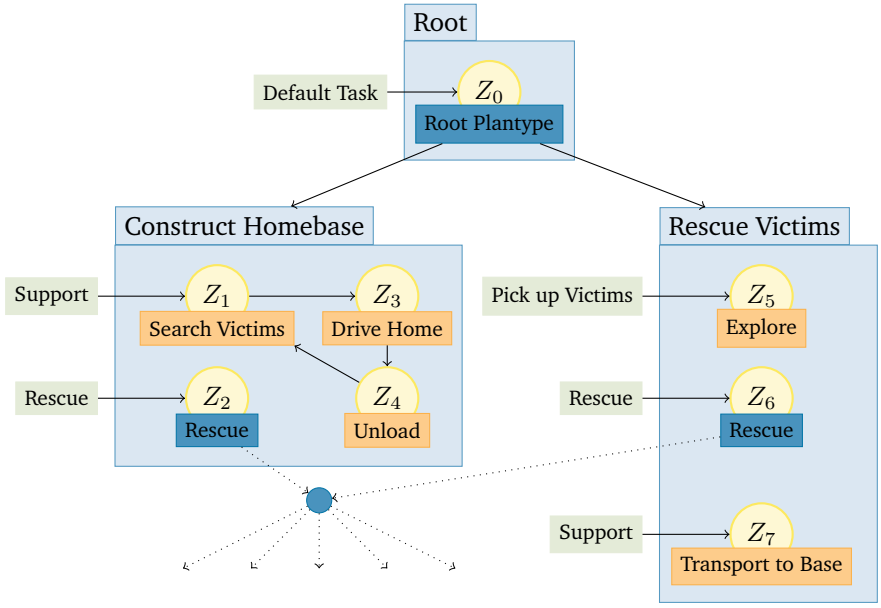



**Figure 3.1:** Example ALICA plan hierarchy for an extraterrestrial exploration mission composed of behaviors (orange), tasks (green), plans, and plantypes (blue).

### 3.1.3 Synchronizations

In general, ALICA provides loose synchronization for agents that transition from one state into another in order to reduce the required amount of communication. This limits not only the required amount of bandwidth, but also increases the tolerance towards network errors. However, in some scenarios this might not suffice to guarantee the successful execution of a cooperative plan. For example, consider two robots in an emergency response scenario that try to lift a stretcher simultaneously to carry a disaster victim safely to a rescue area. In the presence of network errors, such robots need a tight coupling of their actions to ensure that the lift operation is not executed asynchronously, which could cause the victim to fall off the stretcher. Hence, ALICA provides a language element called *synchronization*, which ensures that two robots perform a state transition simultaneously. An example plan for two robots solving a lifting task synchronously is depicted in Figure 3.2.

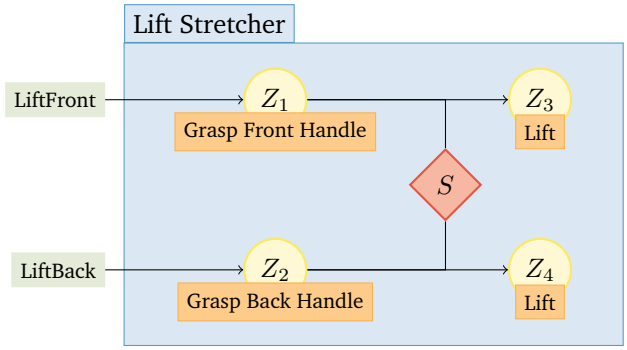



**Figure 3.2:** ALICA plan for two robots to synchronously lift a stretcher.

Synchronizations ensure that all involved agents, waiting in a certain set of states, execute the outgoing transition at the same time. Vice versa, either all agents perform the state transition or none of them. This mechanism follows the idea of joint intentions [99]. To this end, synchronizations are a form of a cooperative decision. Therefore, the realization of synchronizations can be implemented by our solution, as presented in Part II of this thesis.

### 3.1.4 Roles

As mentioned in the section before, tasks describe a specific part of a plan that implements a certain sub-goal of one or more agents. Furthermore, the task assignment is determined by a utility function. This mechanism allows to model arbitrary assignments of agents to tasks. However, the development of proper utility functions is a difficult task, which ALICA tries to simplify with the concept of *roles*. ALICA assigns one or more roles to each agent and these roles are determined based on the capabilities of the agents. Additionally, each task includes preferences, which enforce certain roles to execute the task or reject the task execution.

In many applications, this mechanism is sufficient to determine an adequate task assignment. For example, in robotic soccer, offensive robots need to move over longer

distances, while defensive robots travel less in the field. As a consequence, it is preferable to use fast robots as attackers and slow robots as defenders. Of course, these preferences should not prevent defending robots from attacking an opponent if they are in a good position to do so. Therefore, the preferences are modeled as part of the utility function and can be overwritten by problem-specific conditions such as the distance to an attacking opponent. The resulting three-tier architecture to assign a task to a robot is shown in Figure 3.3.


**Figure 3.3:** Three-tier task assignment architecture of ALICA.

ALICA does not require a specific role allocation algorithm. Nevertheless, the algorithm should rely on the capabilities of the agents instead of dynamic components of the environments. As a result, roles are expected to change only rarely, as the capabilities of the robots should only change rarely as well. This could be the case if a self-healing and monitoring framework such as RoSHA [83] detects a malfunction of one of the robot devices, e. g., the kick mechanism. In contrast, task allocations can change quickly according to the environment state, e. g., the robot closest to the ball. Hence, ALICA does not provide any conflict resolution mechanisms for the role assignment. However, role changes require a sophisticated team decision process like the one presented in this work in order to eventually achieve consistent role changes.

### 3.1.5 Plan Variables

The ALICA elements described so far are restricted to certain behaviors, i. e., behaviors execute the same actions in order to achieve a particular goal. We could only overcome this problem by exploiting the fact that behaviors are implemented in a high-level programming language, which allows internal states. For many scenarios, this can lead to complex behavior implementations or a state explosion problem [38], e. g., if a robot needs to navigate to three different points in a given world coordinate system, we have to develop three different specialized behaviors. However, all three behaviors host the same algorithm and controller to reach their goal. This leads to much code duplication and diminishes the maintainability. Furthermore, there would be no explicit method to coordinate these points with other robots, e. g., to ensure that each point is only reached once. Therefore, ALICA allows to parameterize plans and behaviors with *variables*. This lifts the propositional form of ALICA (also called *pALICA*) to a first-order language.

In practical scenarios, variables mostly describe the goal that a behavior or plan should achieve such as a target movement position, an object to be grasped, or a region to explore. This allows to divide the computation of agent goals from the algorithm to reach them. As a result, fewer behaviors are necessary to solve similar problems.

In the original version of ALICA, the variable values are determined by collecting the constraints and optimization functions from all active conditions in the plan tree, i. e., runtime, pre-, postconditions, and transitions can contribute conditions called *constraints* for the value determination. These are composed to a constraint optimization problem,

which can be computed by a distributed solver (see Section 3.4 and Section 3.5 for more details). Figure 3.4 shows how a plan for searching and rescuing a disaster victim could be realized using variables. The major advantage of this example is the fact that more than one robot could execute the *Search and Rescue* task, as long as a decision mechanism is present ensuring that at most one robot tries to rescue a single victim.

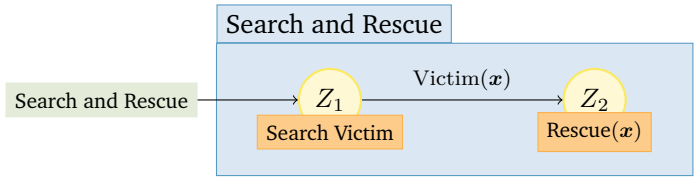



**Figure 3.4:** Plan for the search and rescue of disaster victims using plan variables.

ALICA supports two types of variables: *Plan variables* and *agent variables*. As a major difference, the number of plan variables is statically defined by the plan. For instance, if we assume that $x$ in the plan of Figure 3.4 is a plan variable, the behavior *Rescue* does provide the same value for $x$ to all agents executing it. More precisely, if two robots are in the state $Z_2$, both help to rescue the same victim. In contrast, $x$ can be defined as an agent variable for all agents executing the task *Search and Rescue*. This allows to assign one disaster victim as a goal to each of these agents. Hence, the number of values depends on the number of agents in $Z_2$, which is dynamically computed during runtime. Nevertheless, the active constraints have to hold for all values of $x$ delivered to *Rescue*.

Variables play an important role for our work, as they are the manifestation of multi-agent decisions in ALICA. Variables are not restricted to any data type, which makes them ideal to store the data for decisions. Furthermore, ALICA variables have a unique name, which identifies the decision issue. Moreover, they allow the usage of coordination mechanisms, e. g., to achieve consensus on their values. Finally, variables represent the link between the decision and its execution.

## 3.2 Semantics

In the following sections, we give an overview of the semantics of ALICA, which define well-formed ALICA programs (a formal definition of well-formed plans can be found in [157, p. 46]). One important reason to give a formal definition of the ALICA semantics is that they enable proofs of ALICA programs for properties such as liveness and safety. However, a formal definition is beyond the scope of this work. Instead, we focus on the key features and principles that are relevant for robot decision processes. A complete definition is given in [157, p. 51].

### 3.2.1 Principles

The semantics of ALICA underlie three fundamental principles, which aim for the ease of use, robustness, and adaptability. Additionally these principles allow the usage of ALICA

in new domains forming a general-purpose framework for multi-agent coordination. In the following, we state these principles in detail:

**Domain Independence** ALICA focuses on the robotic domain. However, it is applicable in almost every multi-agent domain, as it makes only few assumptions about the environment and its representation. In particular, it is possible to apply closed or open world assumptions as basis for decisions. As a consequence, the inference logic to evaluate conditions or utility functions requires a domain-specific implementation that fits to the assumptions of the world model.

**Autonomy** The key requirement for dealing with unreliable communication is autonomy. ALICA performs computations and inferences redundantly on all machines that act in the same plan to introduce fault tolerance. This allows detecting and resolving possible conflicts. In consequence, ALICA is the ideal framework to implement team decision processes like the one presented in this thesis. However, ALICA assumes agents that are not communicating for a certain amount of time to be malfunctioning and excludes them from the team. As our approach explicitly tackles large-scale scenarios in which continuous communication is usually not present, we have to weaken this assumption. We present the details in Chapter 9.

**Locality** Redundant computations for all expressions of all agents in the ALICA team would lead to potentially unmanageable complexity. Therefore, ALICA agents exploit the hierarchical form of the plan structure to define scopes for relevant conditions. The plan or state, in which an agent is currently located, identifies these scopes. Furthermore, agents track decisions of other agents that are located in the same plan in order to realize redundant computations. In the solution presented in this thesis, we exploit the same mechanisms to define scopes for agents to reduce the required communication bandwidth, see Section 7.5.2 for details.

### 3.2.2 Agent Model

With respect to the MAPE-K cycle of Section 2.1.1, ALICA is located in the planning phase of an agent. To provide a separation from the other computation steps, ALICA requires a mediation language $\mathcal{L}$, which evaluates conditions and infers the final commands that are computed within behaviors and executed by the actuators. As this logic is not part of ALICA but realized by a higher programming language such as C++ or C#, code stubs are generated. Thus, the application developer implements these expressions externally. This means, ALICA provides source code stubs for each condition, utility function, constraint, and behavior. The ALICA *engine* automatically calls the according component when an evaluation is necessary with respect to the current program or agent state according to the operational ALICA rules, which are specified in [157, p. 93ff].

The configuration of an ALICA agent is specified by the tuple $(B, \Upsilon, E, \mathrm{R}, G)$, with $B$ being the belief base, $\Upsilon$ the plan base, $E$ the execution set of active behaviors, $\mathrm{R}$ the set of roles an agent currently holds, and $G$ a storage of currently valid constraints. ALICA programs and the operational rules operate on this tuple and thereby change the configuration of the agent.

Belief base $B$ defines the world model the agent currently believes in. $\mathcal{L}$ refers to the belief base to describe and evaluate conditions, constraints, and utility functions. In order to track agents and comprehend their decisions in the same scope, ALICA requires all agents to have coherent mutual belief concerning all relevant expressions referring to $B$. Further assumptions, which are beyond the scope of this work, can be found in [157, p. 55ff].

A plan base contains the current plan states and tasks of an agent. This structure is implemented as a tree structure, which allows efficient transitions between states. The formal definition of ALICA describes the plan base as a set of plans, tasks, and state triples, see [157, p. 54ff] for details.

### 3.2.3 Task Allocation

Multi-robot task allocation (MRTA) is one of the most complex and central problems for multi-agent coordination. The problem consists of three main components: First, a set of tasks has to be assigned to a set of agents. Second, a set of conditions forms constraints for the assignments, which restrict the number of possible ones, e. g., to ensure that a grasping task is executed by a robot with a gripping device. Third, a utility function determines the value of an assignment. For instance, in robotic soccer, a robot close to the ball should execute an attacking task instead of a robot with a long travel distance. ALICA induces a fourth component to the task assignment problem, as the final utility value decides which plan within a plantype is executed.

The robotics community classifies task allocation problems according to three characteristics: The amount of tasks a robot can execute in parallel, the amount of robots that need to execute one tasks, and the time extension of the assignment. ALICA addresses a multi-task robot (MT) multi-robot task (MR) assignment problem with instantaneous assignment (IA), which is strongly $NP$-hard [65]. Hence, an ALICA agent can execute multiple tasks in parallel, or multiple agents can execute one task. Furthermore, the assignment does only cover the current set of tasks and robots and does not consider future assignments. The only exception to this rule arises when the system developer models such predictions within the utility functions. However, ALICA does not generally consider time-extended assignments. Encoding them by the utility function would probably result in a violation of the soft real-time properties of ALICA.

According to the locality principle, ALICA computes the task assignment for each plan of the plan tree separately. This reduces the problem to $n$ single-task (ST) robot multi-robot task problems, with $n$ being the number of current plan tree levels. Nevertheless, the ST-MR-IA problem is unfortunately still $NP$-Hard [65].

To cover a close to optimal task allocation, the assignments are computed recursively, beginning at the root node of the plan tree. In case a plan cannot compute a valid assignment for the given robots, ALICA propagates this error to the parent plan, which in turn has to find a different assignment. Invalid assignments can appear if the allocation constraints are not satisfied. This means, an assignment is invalid when too few or too many robots need an assignment for the plan, or the plan pre- and runtime conditions do not hold. A formal definition for valid assignments is stated in [157, p. 68f].

### 3.2.4 Utility Functions

In order to address the complexity of the task assignment problem described in Section 3.2.3, ALICA requires its utility functions $\mathcal{U}$ to provide a heuristic $H$. This heuristic allows to estimate the maximum expected utility of partial assignments to speed up the search for an optimal assignment. Moreover, utility functions are a weighted sum, where each summand computes its contribution to the heuristic. Hence, the utility function $\mathcal{U}(p)$ for plan $p$ is computed as follows:

$$\mathcal{U}(p) = \begin{cases} -1 & \text{if } \mathrm{pri} < 0 \\ w_0 \ \mathrm{pri} + \sum_{1 \leq i \leq n} w_i \, f_i & \text{if the team works on } p \ , \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

with $f_i$ denoting the summands, $w_i$ the corresponding weights, and $\mathrm{pri}$ a priority summand, which is part of the utility function by default and encodes the priority for ALICA roles to execute a certain task. ALICA requires the weights to sum up to one $\sum_{i=0}^{n} w_i = 1$ and all summands $f_i$ to map to $[0..1]$. This ensures that the utility function is also limited to the range of $[0..1]$, except for negative priorities. According to this equation, the heuristic is computed as $H(p) = w_0 \, h_{\mathrm{pri}} + \sum_{1 \leq i \leq n} w_i \, h_i$ if the priorities are not violated and where $h_i$ denotes the heuristic for the summand $i$.

To perform the task allocation, ALICA uses an $A^*$ search, which determines the best assignment and plan choice of plantypes. Therefore, robots are successively assigned to tasks (*expansion*) according to the highest heuristic value given by $H$, which allows for evaluating partial assignments. In order to result in an optimal assignment, the heuristic needs to be admissible $\forall n, \mathcal{U}(n) \leq H(n)$. In other words, $H$ never underestimates the utility value. However, precise estimates of the final utility value by the heuristic can speed up the task allocation significantly. The properties of the algorithm follow the $A^*$ search, as described by Hart, Nilsson, and Raphael [74].

### 3.2.5 Operational Rules

The operational rules of ALICA describe how a multi-agent program is processed. Formally, each rule transforms the agent tuple vector if a certain condition holds. We distinguish these rules into three classes: base rules, plan execution rules, and repair rules. In the following, we give an intuitive overview of these rules.

The two base rules are required to set up the ALICA program and sense the environment. Therefore, the *initialization rule* is processed when the program starts its execution. Here, each agent initializes with an empty execution set and allocates in the initial state of the default task in the top-level (root) plan of the plan tree with an empty constraint store. If new information is available, the *sensing rule* adds the new sensor data to the belief base, which is the base for all inference of $\mathcal{L}$.

The plan execution rules specify how the plan is executed if no failures occur. ALICA defines five execution rules:

**Transition Rule:** If the condition of an outgoing transition, which is not part of a synchronization, from one of the current states of an agent holds, the agent removes

all plans below this state from the plan base. Furthermore, the plan below the destination state of this transition is added to the plan base of the agent. In practical implementations, this role successively applies for all current states starting from the root node of the plan tree.

**Synchronized Transition Rule:** If the transition is part of a synchronization set, the agents transition to the next state if all agents have the mutual belief that the conditions of all transitions in the synchronization set hold. As a consequence, either all agents perform the state transitions or none of them. The transition process itself is defined equally to the transition rule.

**Allocation Rule:** As ALICA agents have to work on a task when executing a plan, the *allocation rule* is performed whenever no valid task assignment for one of the current agent states is present. Therefore, ALICA calls the task allocation algorithm and adds the initial state of the newly computed task for the agent to its plan base.

**Behavior Success Rule:** In case a behavior announces success of its actions, the *behavior success rule* removes this behavior from the current execution set of the agent. Hence, its postcondition has to hold.

**Task Success Rule:** Each plan can be defined with *success states*. If such a state is reached by an agent, the task completion is stored in the belief base of the agent. This particularly affects the validity of the current task assignment and thus requires communication with other agents.

Due to the complexity of robotic systems, the dynamics of robotic environments, and the imperfectness of robots and their software, failures can occur. ALICA handles such failures by trying to execute a different behavior, plan, role, or task allocation. The ALICA repair rules address these issues:

**Behavior Abortion Rule:** Each behavior within the current execution set of an agent can signal a failure. In this case, the behavior is removed from the execution set and the *failure count* is increased. This failure count allows for the implementation of recovery strategies, which address the given domain and problem setting.

**Behavior Repair Rules:** The two behavior repair rules (*redo* and *propagation rule*) represent the default failure handling mechanism. The behavior redo rule reinserts failed behavior into the execution set if the failure count equals one. For failure counts greater than one, the failure propagation rule increases the failure count of the plan containing the failed behavior.

**Plan Abortion Rule:** A plan is aborted if either an agent transitions into a failure state, the runtime condition becomes invalid, or no valid assignment is present. As a consequence, the failed plan and all child elements are removed from the plan tree.

**Plan Repair Rules:** Plan failures can be repaired by restarting, replacing, or propagation of the failure. Therefore, ALICA provides four rules: First, the *top fail rule* is executed if a failure occurs in the top-level plan and resets the whole ALICA program by applying the initialization rule. Second, the *plan redo rule* is executed if an agent fails to perform a task but a valid assignment is given, which still allows fulfilling the plan, e. g., because pre- and runtime conditions do hold. Hence, the agent restarts the execution of the task and increases the failure count of the plan. Third, the *plan*

*replace rule* triggers the task allocation to find a more appropriate assignment in case the failure count equals one and the plan or one of its behaviors has been aborted. Fourth, the *propagation rule* aborts the parent plan by increasing its failure count and removes the failed plan with all child elements from the plan tree if the failure count is two or greater.

**Allocation Failure Rule:** Due to the *allocation failure rule*, the plan is aborted and the failure count is increased by one if no valid task assignment can be found by the task assignment algorithm. Hence, this rule might trigger conditions leading to the execution of the plan repair rules.

**Adaptation Rule:** If the utility value of a task assignment is low and an assignment with a higher utility can be expected to be found (defined by a plan-specific measure called similarity weight) for the current situation, the *adaptation rule* triggers a new task allocation.

**Role Allocation Rule:** Similarly, the *role allocation rule* triggers a new role allocation process if the capabilities of team members or the whole composition of the team changes. Due to the adaptation rule and role preferences, this can result in new task allocations for the currently executed plans.

The transition rules add the constraints associated with the condition to the constraint store. In contrast, if task allocation or reallocation takes place, the constraint store is updated to enclose constraints that belong to plans that are currently executed. Following the same logic, the constraint store is cleared when the *top fail* or *initialization rule* is called. No other rules affect the constraint store.

A formal definition of these rules can be found in [157, p. 93ff] and [157, p. 155ff]. In the context of this work, it is important to note that the operational rules do not explicitly address failures caused by inconsistent belief bases. However, the ALICA implementation provides several mechanisms to handle conflicts, which are described in Section 3.6.

## 3.3 Team Estimation

A key ingredient to compute a valid task assignment is the estimation of the agents cooperating in a team. This refers to all active ALICA agents within communication range. To estimate the present agents, in ALICA, all agents broadcast their plan tree iteratively. The basic idea is to exclude an agent from the estimated team if more consecutive messages from this agent than expected from typical packet loss rates of the network connection get lost. Therefore, ALICA assumes a cumulative Poisson distribution for lost packets:

$$p(k \geq n) = 1 - e^{-\lambda} \sum_{i=0}^{n-1} \frac{\lambda^i}{i!}, \tag{3.2}$$

where $\lambda$ is the mean value, $k$ the number of packets sent within a burst of lost plan tree messages, and $k$ the number of consecutively lost messages. Using this equation and a probability threshold $p_{\text{down}}$, the number of consecutive messages can be determined, which satisfy $p_{\text{down}} < p(k \geq n)$. If this condition does not hold, the respective robot is excluded from the team.

In the robotic soccer domain, ALICA sends plan trees with approximately $10\,\mathrm{Hz}$. To determine $p(k \geq n)$, a mean burst length $\lambda$ needs to be estimated. Based on empirical evaluations under bad network conditions in RoboCup tournament situations, we empirically determined $\lambda = 10$. Assuming $p_{\mathrm{down}} = 10^{-3}$, an agent is excluded after 22 consecutively lost messages and after $2.2\,\mathrm{s}$, respectively. This model does not consider communication delay.

## 3.4 Constraints

ALICA does not impose strict rules on the representation of constraints. However, constraints are defined to be expressed in $\mathcal{L}$ and describe restrictions to some free variables $\vec{x}$. As ALICA is implemented in C++ and C#, respectively, it also delivers a standard method to describe constraint formulas based on a modified version of the automatic differentiation (AD) library by Shtof [154]. It allows to express nonlinear functions in inequality equations, which can be combined with conjunctions and disjunctions. Our approach to define multi-agent goals uses the same foundations. A more detailed definition can be found in Section 6.4

A constraint problem in ALICA is defined by the *unfolded constraints* for all variables that are queried in $\mathcal{L}$. To unfold a constraint, ALICA recursively searches all active constraints in the store, which are relevant to determine the queried variables. Hence, the resulting problem requires the determination of free variables, which provides a dependency of some constraints within the store, e.g., if we consider a positioning problem. Constraints can determine $n$ robot positions as a two-dimensional vector with the goal to place each robot within communication range. Here, we would formulate $n$ constraints to ensure that the distance between the positions is less than the communication range: $\forall i \forall j ||\vec{x}_i - \vec{x}_j|| < c_r$, where $c_r$ is the communication range of the agents and $\vec{x}_i$ the free variable determining the position of agent $i$. In this scenario, an agent will probably only query for its own target position. However, to compute a solution for this problem, all constraints are relevant.

Constraints can also include a utility function, which describes the benefit provided by a certain solution. This allows for a more precise choice among the constraint solutions.

## 3.5 Constraint Satisfaction Problem Solver

For each variable query, ALICA unfolds the relevant constraints to form the problem description. This problem description is the base for a constraint satisfaction problem (CSP) solver to determine a value for the free variables. Although the solver shipped with ALICA is exchangeable, for implementation reasons, it needs to be able to deal with *AD descriptions* for constraints.

The CSP solver of ALICA mainly relies on local searches. Therefore, it transforms the conjunctions, disjunctions, and inequalities of the AD constraint representation into a *membership function*, as known from fuzzy logic theory. This membership function returns values greater than one for all valid solutions. In contrast, invalid variable values evaluate

the membership function to values below zero and give an estimation of the violation of the constraint. Based on this estimation, the solver performs gradient descent computed by the AD library to find a local optimum in this error space.

The membership function transformation does usually not provide a convex error landscape. Hence, local search-based approaches cannot guarantee to converge to a valid solution. In order to increase the probability of finding a solution, the solver restarts from different start points. The resulting approach is still incomplete but shows high success in finding solutions for difficult problem classes quickly, as shown in [158]. Note that the solver to compute solution proposals presented in this thesis also uses the same techniques, see Section 6.5 for details.

## 3.6 Conflict Resolution

All decisions within ALICA require the presence of a coherent mutual belief base in order to ensure consistency among agents. However, in many practical scenarios such as robotic soccer, the environment changes dynamically, which poses problems in ensuring coherency. To address potentially conflicting decisions, ALICA provides some mechanisms that we analyze in the next sections. Thereby, we highlight drawbacks that we address with our solution in Part II.

### 3.6.1 Transitions

Caused by the *transition rule*, ALICA agents change their plan state if the condition of an outgoing transition of the current state is evaluated as *true*. These conditions are expressed in $\mathcal{L}$ and rely on consistent belief bases. Since ALICA provides operational rules to withdraw a transition decision, a conflicting execution of the transition rule occurs if no mutual belief is present. Hence, it is the responsibility of the application developer to either establish a mutual belief or describe a fault-tolerant plan. Note that triggering a transition back to the source state of a detected conflict can be used to model fault tolerance. Here, ALICA's capability to establish a mutual belief concerning the current states of the agents forms a profound basis for the implementation of such a conflict detection.

A different solution to coherent transitioning in ALICA is the usage of synchronizations. Synchronized transitions ensure agreement before transitioning to the next state in order to avoid conflicts in advance. However, this comes at the price of a three-way handshake protocol, which requires $n^2$ messages, where $n$ is the number of agents. Moreover, synchronizations are restricted to transitions within the same plan. Furthermore, synchronizations do not provide any decision knowledge such as a mutual belief concerning some world entity.

The fact that ALICA was originally designed to address *cooperative* teams of agents prohibits the possibility of a single agent making a decision for other agents of the team, unless it provides a mutual belief to achieve a common decision. This also holds for transitions between states, which cannot be triggered by a third-party agent, except a whiteboard for mutual belief as decision base is present. However, our decision process extends ALICA

to additionally support collaborative settings including agents that are not controlled by ALICA.

### 3.6.2 Task Allocation

Each ALICA agent computes the task allocation independently. Consequently, inconsistent mutual beliefs can lead to conflicting task assignments among the agents. In this context, a conflict is present if multiple agents in the same plan either rely on a different task assignment or select a different plan within a plan type. In contrast to transitions, ALICA iteratively checks the task assignment for consistency. Hence, if a conflict is detected, the allocation decision can be changed to resolve the conflict. To achieve this, ALICA provides two ingredients: a conflict detection algorithm and a conflict resolution method, which avoids a conflict of the new assignment.

To detect conflicts, ALICA agents rely on the iteratively sent plan tree information. If agents receive plan tree information, they will update their belief base with the received data. During the next execution of the allocation rule, they check whether the computed assignment is consistent with the received plantrees in the knowledge base and update the knowledge base according to their local data. Hence, a conflicting task assignment leads to a cycle, where the local task allocation overwrites the received plan tree data and vice versa. If the number of such cycles exceeds a certain threshold, the agent assumes a conflict, e. g., in robotic soccer, we regard more than two cycles as a conflict.

The conflict resolution follows the idea of a leader election among the relevant agents. Therefore, ALICA relies on the Bully algorithm [63] to select the agent with the highest unique identifier as a leader. ALICA appends the election messages to the proposed task assignment, which allows immediately overwriting the local task assignment by the winner's proposal. A detailed description of this approach can be found in [159].

The major drawback of this election-based approach is the fact that a predefined agent wins the election without incorporating situation knowledge. Consequently, the winning agent might not be the one with the highest confidence. Summarized, all task allocations have to be coherent with the allocation of the agent with the highest identifier. Thus, a single agent with a low identifier cannot propose allocation decision to others, unless it provides a mutual belief.

### 3.6.3 Variables

Assuming that no conflicts are present in the plan trees of a team of ALICA agents, all agents infer the same CSP for a given variable query. However, the constraints incorporate references into world entities, which again have to be a mutual belief. Here, conflicts can develop due to two problems: The incomplete solution strategy does not find a common solution deterministically for all agents or mutual beliefs concerning the world entities are not present.

To avoid conflicts due to the nondeterminism of the local search solver, the server distributes valid solutions within the network. During the computation of a solution,

the solutions received from other agents serve as potential starting points. This method provides two strong points: First, it incorporates a weak form of negotiation, as all agents eventually receive the same solutions and therefore use the same start points. As a result, the solver might behave deterministically and eventually converges to a common solution. Second, it decreases the computational time to find a solution, as in many cases old solutions are still valid at a later point in time. Moreover, many problems require only small changes of old solution values to become valid. According to our definition of Section 2.3.4, this might result in coherent decisions despite the presence of inconsistencies in the world model. A detailed analysis on the solver properties can be found in [157, p. 207ff]

Concerning true mutual belief, the solver does not provide any support to the application. As a consequence, especially for cases which are only rarely computed, ALICA requires a more sophisticated and uniform solution to avoid conflicts for solver variables. We regard such a solution as a key contribution to this thesis.

## 3.7 Goals of Multi-Agent Systems

For the representation of goals in multi-agent systems, we can distinguish two major forms: *explicit* or *implicit* goal representation. Explicitly represented goals are a natural representation form where a desired state of the agent's environment is encoded as a terminal state for a certain entity. For example, an explicit goal can be the target position of an object. The goal can mostly be represented in a single object instance. In some rare cases, the goal description can also refer to a non-physical entity such as a mental state. A typical example for non-physical goals occurs in games where the agent has the overall goal to win the game, which might not necessarily correspond to a single final situation, e. g., the winning team of a soccer game needs to have a higher score than the other team at the end of the game.

On the one hand, the memory efficiency of explicit goals is their major advantage, as it allows them to exchange and negotiate with other robots. On the other hand, explicit goals do not encode any information about how the team can reach their goal. Therefore, a single goal can be too abstract to achieve a tight cooperation among agents, e. g., when considering the goal to win a soccer game. To tackle this issue, sub-goals can be introduced, which allows a simpler inference of the next steps the agent will take. For example, a soccer robot can have the goal to pass to a certain point on the field, which allows other agents to predict the next actions and adjust accordingly. In some sense, this makes sub-goals to a part of the solution.

Implicit goals cannot be encoded within a single variable. They are usually used when agents execute a fixed plan, such as an ALICA plan. Such a plan or state machine describes a set of actions, which implicitly encode goals to manipulate the environment or a non-physical state. Thus, the description allows to directly infer the future actions of an agent, which is particularly useful for tight cooperative coupling such as driving in a formation or as a convoy. However, to the goal of an agent can only be inferred by predicting the environmental state until the last action is executed. In some cases, this is not even possible, e. g., when uncertainties are involved. This makes the support in long-term tasks difficult for other agents.

A further disadvantage of implicit goals is the difficulty of their negotiation between agents. In practical cases, this is usually realized by communicating the plan of the agents and negotiating the described actions or the whole plan. Approaches such as ALICA assume that all relevant plans are distributed at deployment time. Here, it is sufficient to exchange unique identifiers to implement a negotiation algorithm. However, for dynamically generated plans, ALICA provides plan serialization to enable network transmissions. Note, dynamically generated plans usually appear in planning algorithms, which are also in the focus of ALICA [149].

## 3.8 Summary

This chapter describes the multi-agent language ALICA, which provides a basis to describe and execute the behavior for teams of robots in various domains such as robotic soccer, autonomous car driving, or lunatic exploration missions. In particular, we highlighted the need for a unified decision process that allows for coherent decision making in ALICA. This would enable ALICA programs to realize not only cooperative but also collaborative behavior while reducing the possibility of conflicts due to inconsistent belief bases. Additionally, our decision process requires an execution layer, which allows to perform actions to implement a common decision. As ALICA already provides robustness against network delay and packet loss, it forms an ideal basis technology for our decision process.

# 4 Related Work

The main contribution of this work is a *middleware-supported decision process suitable for small teams of autonomous mobile robots*. Generally, we could not identify a single framework supporting all requirements for the targeted applications. However, several research areas and projects provided valuable contributions. The most relevant areas of research are agent decision processes, robot middleware frameworks, object and tuple spaces, and consensus algorithms. In this chapter, we will highlight the most important related work and identify their major drawbacks. As baseline, we rely on the requirements that we identified in Section 1.4.

## 4.1 Multi-Agent Decision Processes

Most multi-agent coordination frameworks rely on an implicit decision process, i. e., without any specified rules addressing the resolution of proposal conflicts. However, there are some theories and frameworks that provide an explicit description of how agents make common decisions. The most notable approaches are described in the following sections.

### 4.1.1 Multi-Agent Markov Decision Processes

The most common model of multi-agent decision processes in the reinforcement learning research community is the multi-agent Markov decision process (MMPD). Here, the agents behave as an ideal team, i. e., they do not follow individual motivations and act together towards a common goal. From the perspective of reinforcement learning, the main challenge of such architectures is to generate a policy that leads to the best team performance in a stochastic game setting. According to the taxonomy of Pynadath and Tambe [133], MMDPs are characterized by two main properties: the observability of the environment and the restrictions on the communication. Pynadath and Tambe [133] distinguish four classes of observability:

**Individually Observable Environments** allow all agents to gather all relevant information from the environment.

**Collectively Observable Environments** provide all relevant information to at least one agent. Hence, all relevant knowledge is present in a shared knowledge base.

**Collectively Partially Observable Environments** hide some relevant information from all agents prohibiting them to gather all relevant knowledge.

**Unobservable Environments** do not allow any observations for the agents at all, as in open-loop systems.

The capability of the agents to perform communication provides the key influence on the ability to distribute relevant data within the team. Pynadath and Tambe [133] identify the following scenario classes:

**Free Communication** allows transmitting arbitrary messages without any costs, thereby transforming an MMDP to an MDP.

**General Communication** includes limitations or costs for each transmission, e. g., in cases of simulated bandwidth limitations or network delay.

**No Communication** prohibits message passing completely.

Pynadath and Tambe [133] also analyzed the complexity of the different MMDP forms. They concluded global optimality to be $P$-complete problem if all individual agents have full observability and $NP$-completeness otherwise. Collectively observable environments provide the complexity of $NEXP$-complete except for free communication, which simplifies the problem to $P$-completeness. The same holds for collectively partially observable environments unless free communication forms a $PSPACE$-complete problem.

According to Boutilier [22], MMDPs are defined as a tuple $\langle \mathcal{A}, \{a_i\}_{i \in \mathcal{A}}, \mathcal{S}, \Pr, R \rangle$, where $\mathcal{A}$ is the set of agents, $a_i$ represents the set of finite actions for agent $i$, $\mathcal{S}$ symbolizes a set of finite states, $\Pr$ forms a probability distribution that models the probability to transition from one state into a different one given a certain joint action, and finally $R$ expresses a reward function. In this setup, MMDP algorithms focus on the search of a policy that maximizes the utility function.

Most solutions for generating an optimal policy for MMDPs are based on dynamic programming methods such as [73]. It iteratively simulates policies in the targeted environment and prunes them until only a single policy remains. However, this approach does not address the problem that commonly occurs within a distributed setting. Roth, Simmons, and Veloso [144] address this issue by establishing joint beliefs to transform a multi-agent *partially observable Markov decision process* (POMDP) into a single-agent POMDP. The resulting approach, coined with the term *DEC-COMM*, performs reasoning on a tree of joint beliefs that allows inferring the need to use some communication process to integrate local observations into the team belief.

The goal of MMDPs is to infer policies that compute a set of joint actions for a given set of states. Hence, most algorithms compute proposals for possible actions. However, to our knowledge, none of the approaches provide a sophisticated solution that properly addresses conflicting observations for this setting. Although most papers do not explicitly state this issue, they commonly assume that there are consistent observations of all agents. In conclusion, MMDPs address a different level of multi-agent decision-making processes that could replace our proposed CNSMT solver approach for scenarios involving numerous explicit uncertainties. Due to the ability to learn communication actions, we consider algorithms such as *DEC-COMM* as a *meta-decision process* that could extend PROViDE by adapting the distribution parameters, e. g., in case their specification cannot be determined during compile time.

### 4.1.2 Joint Intention Theory

The joint intention framework was developed by Cohen and Levesque[40, 41]. It allows the implementation of a common mental state among all cooperating agents. Such *joint intention* appears if all agents jointly commit to the same intention and mutually know that all agents execute it. As the term intention usually determines a *team action*, we regard it as a common decision. Additionally, the joint intention framework defines the following terms:

**Beliefs** denote facts that are assumed to hold, i. e., *beliefs* represent the current world state according to the knowledge of an agent and after the elimination of wishful thinking and competing opinions.

**Goals** describe the most desired world state of an agent. Similar to beliefs, the notion of goals presumes that conflicting goal choices have been resolved.

**Mutual Beliefs** refer to the beliefs owned by an agent about the beliefs of other agents in an infinite conjunction, i. e., mutual assumptions of all agents in a team.

Another key ingredient of the joint intention framework is *commitment*. If agents establish a joint intention, they have to commit to the corresponding team action. In case one of the agents wants to abandon the commitment to the joint intention, then prior to that, the agent has to communicate the termination of the joint intention. As a result, joint intention theory allows reasoning in a framework that provides a high level of safety.

Contrary to the joint intention theory, our decision process does not explicitly distinguish between goals and beliefs. However, PROViDE can be used to establish joint intentions, beliefs, and goals at the same time. This means that it is possible to negotiate all types of facts independently from the *possible-world* semantic. Such activity is driven by the observation that the establishment of mutual beliefs on all world facts –independent from their current relevance – requires unnecessary overhead. At the same time, we claim that some decisions require mutual beliefs as their base knowledge. PROViDE requires neither commitment to any decision nor a joint agreement on decisions. At the price of general safeness, PROViDE is more flexible and adaptable to the problem setting. Moreover, we clearly distinguish between the final value decisions from a proposed value, which makes conflicts or contradicting observations visible at the application level. Finally, our decision process does not require mutual agreement on a decision value, which is necessary for making decisions in the absence of one or more agents.

### 4.1.3 Shared Plan Theory

As the name suggests, the shared plan theory [68, 69] establishes agreement on the execution of common plans. Like in ALICA, a plan is a hierarchical group action structure that describes a strategy to achieve a set of goals. Hence, a plan includes beliefs for a set of actions and sub-actions. This allows the decomposition of complex tasks into basic capabilities to facilitate their implementation.

In contrast to the joint intention theory where a number of agents share a mental state, each agent disposes an intentional attitude coined with the term *intending that*. This

attitude always refers to an action or joint action of the collaborators of the agent. The *intentions that* follow four axioms that regulate conflict avoidance. They ensure that the agents realize the propositions they believe to intend and follow basic actions, which fulfills the intention they believe in. Thus, these axioms are constraints for the rational actions of agents. Shared plan theories rely on similar communication basics as joint intention theory but describe the way to achieve a common goal instead of the goal itself.

In shared plan theory, agents are not required to commit to any decision. The lack of joint intentions limits the possibility to reason about coordinated behavior. Moreover, the shared plan theory provides no explicit goal representation, which hinders their explicit negotiation. We conclude that decision-making is the selection of a proper plan or action without any explicit support for environments with unreliable or transient communication.

### 4.1.4 STEAM and Teamcore

*Shell for TEAMwork* (STEAM) [167] is a model to implement teamwork designed for closing the gap between teamwork theories and practice. Therefore, STEAM uses joint intention and shared plan theory to achieve coordinated behavior. In particular, STEAM assigns sub-teams of agents to a hierarchical shared plan structure similar to ALICA. The major difference between them is that STEAM agents generally establish a joint intention before acting together. This makes STEAM agents behave lazy when the communication appears unexpectedly delayed or error prone. Like ALICA, STEAM defines a set of operational rules. These are formulated as domain-independent Soar rules [113].

STEAM establishes joint intentions before each execution of a team plan in order to avoid conflicting decisions. The middleware implementation that realizes joint intentions for STEAM is called Teamcore [132], which is also capable of incorporating heterogeneous external agents through proxies. Thus, external agents such as human collaborators can receive and execute tasks and provide information that is required for the coordination. Such a design enables the implementation of Teamcore wrappers for mobile devices for example laptops or smartphones, to provide the human interaction.

Teamcore uses the *knowledgeable agent resources manager assistant* (KARMA) to operate with third-party agents. KARMA is able to locate agents with respect to organizational requirements and assigns tasks to them. Pynadath and Tambe [132] coined their abstract plan specification model with the term *team-oriented programming*. To a certain extent, PROViDE in ALICA adopts the role of Teamcore in STEAM, i. e., it is responsible to set up joint and make a decision, respectively. This forms the foundation for coordinated behavior. In contrast to Teamcore, PROViDE allows an adaptation of its communication protocol to the needs of the given problem setting. Hence, a common decision does not require a joint intention, this also includes scenarios that require reactive agent behavior, e. g., in robotic soccer. In the same fashion, PROViDE does not dictate a specific conflict resolution method but allows the developer to specify how conflicting proposals are treated. Moreover, although our decision process is implemented as middleware core for ALICA, it is not designed for a specific coordination language.

### 4.1.5 BITE

The Bar Ilan teamwork engine (BITE) by Kaminka and Frenkel [80] divides its decision and execution process into three execution structures: Firstly, a tree-like structure similar to hierarchical task networks [169], which represents a global execution plan similar to the *master plan* in ALICA. A second structure describes the organization hierarchy of individuals and robots in sub-team memberships. This refers to the hierarchical task structure to provide a team-wide allocation of robots and sub-teams to behaviors. The resulting approach corresponds to the methodology of ALICA's task assignment algorithm. The third structure describes the *social interaction behaviors*, i. e., realizations of explicit communication between agents. This could be implemented using various methods, e. g., a voting algorithm.

A major drawback of BITE is the fact that it forces a successful negotiation before any physical action can take place. As a result, BITE is not appropriate for domains that require swift reactive behavior. Nevertheless, it provides an adaptable architecture that supports the exchange of the synchronization mechanism during runtime. In contrast, other approaches allow the implementation of conflict-handling mechanisms, although such a mechanism is not built in by default.

Another disadvantage of BITE is the lack of an explicit representation of common goals. Therefore, no separation between the goal determination and the executing behavior is possible. This propositional structure might cause a state explosion if the robots have to solve numerous tasks. In contrast, PROViDE postulates a distinction between the elementary capabilities and the decision method. We discuss this issue in Section 6.1.

### 4.1.6 Collective Decision-Making and Swarm Robotics

One key characteristic of swarm robots and other self-organized systems are their limitations in perception and communication capabilities. As a result, in many applications, no single agent has the knowledge to make decisions for the team. Instead, agents oftentimes rely on biologically inspired weak indicators, such as pheromones or observations of their local neighborhood. Hence, these limitations require sophisticated decision and control mechanisms that solve the trade-off between decision speed and accuracy for achieving emergent behavior [59].

Collective decision-making processes are usually inspired by either natural systems or opinion dynamics [172]. Opinion dynamics are part of the statistical physics research and focus on the dynamics of physical systems. The approaches rely on a stochastic process. This process is partinioned into rounds and some kind of leader agent is chosen in each round. This leader applies a decision rule on a set of decision possibilities [32]. A decision process inspired by natural systems is more common. Most of the approaches follow the decision-making process of honeybees [127] or ant colonies [28, 49]. Broadly speaking, these approaches send out some agents to explore different decision options and apply a decision-making process based on the gathered information. This commonly proceeds by following the most supported decision identified by the highest density of pheromones or other indicators.

As decision-making is one of the fundamental aspects in swarm robotics, a wide variety of approaches is investigating this direction. However, all of these approaches assume that agents underlie strict limitations in communication and world knowledge. As a result, the opinion of a single agent is valued according to the number of supporters, i. e., a single agent can affect the group but does not make a decision. This makes the adaptation of these methods to new scenarios complex. Furthermore, the exploration of possible decisions is rather inefficient in practical scenarios.

### 4.1.7 Summary

All presented multi-robot frameworks have in common that they are capable of making decisions for teams of agents. However, none of these approaches explicitly addresses the problem of unreliable communication. Moreover, they cannot adapt to a given problem setting, e. g., for cases where an agreement is less important than providing swift reactive behavior. Hence, these approaches are rather inadequate for scenarios such as robotic soccer. It is also notable that all approaches provide some support for incomplete world knowledge. These approaches partially assume that the application software explicitly exchanges required data. We summarize the capabilities of the presented approaches in Table 4.1 regarding the requirements of Section 1.4. Our analysis of JIT, SPT, MMDP and collective decision-making is not focused on a specific implementation. Thus, properties of approaches that are not explicitly cited may be different.

| | JIT | SPT | STEAM | MMDP | BITE | CDM[1] |
|---|---|---|---|---|---|---|
| **Distribution** | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |
| **Incomplete Knowledge** | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ |
| **Changing Team Configuration** | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ |
| **Robustness** | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| **Fault Tolerance** | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| **Availability** | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| **Efficiency** | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| **Adaptability** | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |

✗ = no support ✗ = very limited support ✓ = partial support ✓ = full support

**Table 4.1:** Properties of agent decision processes following the requirements defined in Section 1.4.

[1] Collective Decision-Making

## 4.2 Robot Middleware Frameworks

The robot middleware community is taking a different perspective on multi-agent coordination. Frameworks such as Orocos [29], CLARAty [174], or MIRO [171] use event-based behavior coordination to allow agent decoupling. Here, the events are triggered by either communication or timer events that are mostly realized as remote procedure calls. This provides an insufficient decoupling between the initiator and receiver of an event. Some approaches such as Orocos [29] and MIRO [171] are relying on CORBA [117] or Ice [155], respectively. Hence, both use TCP-based connections, which we claim to be inadequate for environments where network errors have to be expected, according to our results of Section 11.3. In contrast, CLARAty [174] has been developed for communication of NASA rovers. It explicitly supports unreliable communication and can operate in either centralized or decentralized mode. As the approach is directed at a single-agent communication, it does not provide automatic configuration.

Today, the most common communication concept of robot communication middlewares is publish-subscribe due to its higher degree of decoupling. Examples for middlewares based on message passing are RoboFrame [130], Spica [15], or ROS [134]. Although ROS is clearly the most used communication middleware for robotic applications, it provides only view middleware features. The current version of ROS is only shipped with the possibility for message passing and remote procedure calls on a single central machine. Note that such a machine is managed by a *roscore* process that sets up the connections between remote processes and a host, which also allows to remotely connect with a robot. However, this mechanism is not suitable for real distributed applications. In contrast, RoboFrame and Spica are explicitly designed for distributed systems and offer the capabilities to deal with unreliable communication in diverse RoboCup events where standard WiFi communication channels suffer from the bandwidth limitations.

In the literature, we can find a wide variety of further robotic communication middlewares that are not mentioned here. Elkady and Sobh [53] have recently conducted a survey including a detailed overview. Although some of the mentioned frameworks provide fault tolerance to some extent, they focus on broken connections and packet loss. Thus, they lack the capability of handling conflicting transmissions or negotiation processes. More precisely, none of them resolves or identifies conflicting value proposals.

### 4.2.1 Real-Time Database

The *real-time database* (RTDB) [13] implements a *distributed blackboard* that was originally developed for robotic soccer environments by the MSL team CAMBADA. The blackboard is able to fully replace a robot middleware. RTDB memory consists of two areas: global memory area containing information that is shared over the network with all other agents within communication range, and a private memory area that keeps data, which is only locally relevant a specific agent. The latter addresses the storage of current motion commands or configuration data, which are thus accessible by all processes on the local agent.

The database only provides two access operations: read and write. Data values are addressed with an integer-based identifier and the identification of the agent, which

distributes the value. Concurrent write operations by remote processes are not possible, since agents can only perform write operations concerning their own address space in the memory. Contrary to this, different processes are allowed to address the local memory area. In order to guarantee atomicity, a single dedicated real-time process handles these write operations. This process acts as scheduling mechanism for access operations. The real-time properties of this task are realized by the *realTime application interface for Linux* (RTAI) [19].

The key underlying assumption of the RTDB is frequently updated shared data. The authors conclude that the data has to be transmitted to all other agents iteratively, e. g., with a frequency of 10 Hz. To make this possible, the authors make the assumption that the amount of shared data is considerably small compared to the bandwidth provided by the communication interface. This assumption is suitable for dynamic environments with only view-relevant objects such as robotic soccer. However, the RTDB might unnecessarily exceed the available bandwidth in static domains with a high amount objects of such as emergency response scenarios.

The data distribution of PROViDE borrows several ideas from the RTDB. In particular, both approaches provide automatic replication and only allow write access to the values written by a local agent. Nevertheless, RTDB does not include a methodical approach to negotiate a value among the agents. PROViDE is more adequate for large domains, due to our scoping concept, which determines the relevance of objects in the current situation. Additionally, the concept of change subscriptions of PROViDE enables for a broader variety of applications, as automatic reactions to communication events is possible.

### 4.2.2 Data Distribution Service

The Data Distribution Service (DDS) [125] is a standard developed by the Object Management Group (OMG) [116] that defines the organization of publish-subscribe systems. DDS targets are real-time systems that require quality of service (QoS) specification in a data-centric [126] setting to balance a predictable transmission behavior and communication efficiency. Here, the defining aspect is the decoupling between anonymous information producers (publisher) and consumers (subscriber) in space, time, and flow. Each DDS process – called *participant* – runs in different address spaces on possibly different computers and can act simultaneously as a publisher and a subscriber. The information is divided into three types:

**Signals** identify information that is changing continuously over time such as sensor data. The DDS usually sends this type of information with best-effort transmissions, as they become outdated quickly.

**Streams** send snapshots of a data value that requires interpretation in the context of previous snapshots, if necessary in video streams. Here, a reliable transmission is assumed, e. g., for the key frames of a video stream.

**States** are assumed not to change within a fixed time interval, and usually, only the most current value is required by consumers. Hence, the DDS transmits the most current value reliably, as it is usually not acceptable to wait until the value changes again.

One of the unique selling points of the DDS is its broad range of supported QoS features. Each information is offered with a set of QoS features by publishers, and each request is offered with a possibly different set by subscribers. In case offer and request do not match, the DDS provides the QoS that matches best to the request but can be provided by the publisher, e. g., if an information is offered with *best effort* but requested with *reliable* transmissions, the middleware will only provide *best-effort* quality. The most important QoS features of the DDS are:

**DEADLINE** specifies the period within which a new data sample is expected by a subscriber or transmitted by a publisher, respectively.

**LATENCY_BUDGET** represents the hint for a maximum acceptable latency for the transmission.

**TIME_BASED_FILTER** defines the minimum separation between two consecutive data transmissions.

**CONTENT_BASED_FILTER** filters the data by an SQL-like statement.

**PRESENTATION** allows setting coherent or ordered access to a specified scope. Note, coherent access means grouping a set of values to a single value.

**DURABILITY** controls whether data is stored for delivery to new data subscribers.

**OWNERSHIP** determines whether more than one publishers are allowed for the given topic.

**LIVELINESS** defines the validity after which a data entity is deleted unless it is refreshed

**PARTITION** is the list of partitions the entity is allowed to connect with.

**RELIABILITY** switches between best-effort transmissions, where packets may be dropped or lost, and reliable transmissions.

**DESTINATION_ORDER** allows specifying whether messages are ordered by the sender or receiver time stamp.

PROViDE also borrows several QoS features for its own implementation from the DDS. First, the LIVELINESS feature of the DDS encodes the validity time of variables. Second, PROViDE controls the transmission reliability according to the distribution method. Furthermore, the durability of PROViDE proposals is inherently given by its persistence properties. Finally, PROViDE's decision methods facilitate any kind of filtering, as provided by the DDS.

There are three major differences among the approaches. Firstly, the DDS does not store local copies of data. Instead, each data set has to be requested from the remote *data writer*, which does not provide fast accessibility compared to a local copy. Note that DURABILITY QoS is not available for *data readers*. Secondly, although QoS features the provided filters reliability and presentation provide some support for coherent features, no problem-specific negotiation of distributed values is built in. Thus, robots using the DDS cannot be sure that all relevant robots already read a specific data entity. Thirdly, the lightweight architecture of PROViDE reduces the latency compared to modern DDS implementations. Note that the DDS is also designed to have small communication overhead, but our evaluation in Section 11.1 shows significant higher latencies in comparison to PROViDE.

### 4.2.3 Summary

Robot communication middlewares facilitate the communication between robot components. Therefore, their strengths lie in distribution and tolerance against network errors, although some older middleware products rely on TCP connections that are not applicable in environments with high packet loss rates. Besides the RTDB, none of the presented approaches provides any support for conflict avoidance, detection, or resolution.

As none of the presented approaches implements a decision process, the requirement regarding capability of handling incomplete knowledge is not applicable to robot communication middlewares. As shown in Table 4.2, the only drawback of the DDS is its lack of conflict negotiation. Although the authors of the DDS do not propose a decision-making process, DDS implementations provide ideal solutions for our addressed environments. At the same time, the DDS is the only approach that emphasizes adaptability to different problem settings.

| | Event-Based Middlewares | Publish-Subscribe Middlewares | RTDB | DDS |
|---|---|---|---|---|
| **Distribution** | ✓ | ✓ | ✓ | ✓ |
| **Changing Team Configuration** | ✓ | ✓ | ✗ | ✓ |
| **Robustness** | ✗ | ✗ | ✓ | ✗ |
| **Fault Tolerance** | ✓ | ✓ | ✓ | ✓ |
| **Availability** | ✗ | ✗ | ✓ | ✓ |
| **Efficiency** | ✓ | ✓ | ✓ | ✓ |
| **Adaptability** | ✗ | ✗ | ✗ | ✓ |

✗ = no support ✗ = very limited support ✓ = partial support ✓ = full support

**Table 4.2:** Properties of robot middlewares following the requirements defined in Section 1.4.

## 4.3 Object and Tuple Spaces

PROViDE maintains a repository of proposals that resembles distributed object spaces. Object spaces are an associative memory paradigm for distributed computing. As PROViDE borrowed some ideas from object spaces, we describe the most relevant approaches in the following sections.

### 4.3.1 Linda and JavaSpaces

Linda [11] is the first known implementation of a distributed object space and was developed in 1986. The original version of Linda operates on a list of tuples that can be composed of arbitrary elementary data types, examples are strings, integers, or real values. Therefore, the authors introduced the term *tuple space*. Here, the first component of a tuple refers to the logical name that addresses this tuple. To operate on these tuples, Linda provides a small set of operations:

**Out** writes a tuple into the tuple space.

**In** reads a tuple and subsequently deletes it.

**Read** performs a pure read operation – the tuple remains in the tuple space.

**Eval** spawns a parallel process that evaluates the parameters of *eval* and writes it into the tuple space.

Linda maintains a copy of the tuple space on each client. Therefore, read operations are based on local memory access. Write and delete operations are communicated via broadcast to all participants. In case a node fails, the transmission is repeated until it is successfully acknowledged. In the meantime, the operation (*in* and *out*, respectively) blocks the process. Under the assumption that the participants are known, this guarantees consistent replication. In robotic scenarios, a broken agent could block others for infinite long time.

In case two agents delete a tuple simultaneously, a two-phased protocol grants access to one of the requests while the other returns an error. To realize this protocol, Linda relies on the S/Net bus as a semaphore that grants access in a *first-come, first-served* manner. Hence, Linda does not explicitly support the negotiation of conflicting proposals. In particular, its implementation does not consider that participants dynamically appear and disappear. In contrast, PROViDE uses a three-step negotiation process that can be adapted to the given problem setting. This allows to consider disappearing or engaging agents.

Following the Linda model, numerous tuple and object spaces were developed for almost every common programming language. The most famous implementation is JavaSpaces [61], which could achieve commercial success as being part of the Jini[2] [115] technology. The main contribution of Javaspaces is that it was developed to allow simultaneous access on different elements of complex object types, i. e., a process can perform a write operation on the first entry of an array while another operates on the second entry. However, each atomic element is stored on a single host instance. This instance is responsible for realizing the synchronization. As a consequence, if this instance vanishes from the network, its data is not accessible anymore. In PROViDE, we use replication to enable continuous access to the data, although robots disappear from the network.

### 4.3.2 Tuple Centres Spread over Networks

The authors of *tuple centres spread over networks* (TuCSoN) [120, 121] identified a disadvantage of standard tuple spaces: they are *information driven*, meaning that the

[2] also known as *Apache River*

agent coordination is mostly based on information that is available within a shared data space. Hence, tuple spaces do not distinguish between information and its representation. To overcome this issue, TuCSoN incorporates reactions to communication events into the standard tuple interface. The authors introduced the notion of a tuple center, which describes a tuple space enriched with an *event-based* behavior mechanism. Note that the *in* operation of tuple spaces is blocking, which also enables to react to communication events.

To describe coordination protocols for TuCSoN, the authors developed a tuple language [119] based on a first-order logic called ReSpecT. On the one hand, this language provides the language elements known from Linda *in*, *out*, and *read* in blocking and non-blocking versions. On the other hand, operators are allowed to define tuple events called *reactions*. A reaction refers to a tuple operation that is executed in the *pre-* or *post-phase* of a communication event. It is also possible to restrict the reaction to successful or failed tuple operations. Reactions are triggered by one out of four supported manipulation events:

1. The manipulation affects a specified tuple
2. executed by a specific agent,
3. by usage of a given operator
4. with the operation concerning a targeted tuple center.

As a summary, ReSpecT and TuCSoN aim for a more efficient implementation and a cleaner design of coordination protocols. The mechanism of reactions is similarly included in the PROViDE middleware by proposal change subscriptions. However, we assume that the robot control loops execute the most coordination actions. Therefore, the subscription mechanism of PROViDE is less fine-grained. Note that PROViDE can emulate the event types supported by ReSpecT, as proposal properties allow the inference of the information on the event type. In case of TuCSoN, it relies on an underlying architecture similar to Linda. Hence, it suffers from the same drawback in scenarios that incorporate mobility and range-limited communication. This means that it neither distinguishes between proposals and the actual tuple value nor supports the negotiation of conflicting values.

### 4.3.3 Linda in a Mobile Environment

*Linda in a mobile environment* (LIME) [111] extended the idea of Linda by introducing federated tuple spaces for groups of connected mobile agents, e. g., for mobile sensor networks [43, 44]. Linda creates a single, global, and persistent tuple space that is incapable of handling the mobility of an agent that leaves or engages the communication infrastructure. In contrast, LIME explicitly addresses physical and logical mobility of the participating agents. The fundamental assumption for this framework is the fact that mobility makes the presence of a global and persistent tuple space impossible. Therefore, each agent maintains at least one own tuple space, which transparently migrates during its movements. Moreover, *virtual federated tuple spaces* are dynamically created on each host that merges all accessible agent tuple spaces, which are called *host-level tuple space*. In the same fashion, a second layer creates another *federated tuple space*, which is the union of all host-level tuple spaces. As a result, LIME produces the illusion of a global persistent tuple space, which is accessible with the same operations as proposed by Linda.

Although LIME provides location transparency, each tuple is associated with a single agent. Hence, by default, each *out* operation writes a tuple into the local tuple space of the agent. Vice versa, *in* and *read* search the target tuple in the federated tuple space by forwarding the request to all associated agents. This is a behavior contrary to Linda where tuples are stored at each participating agent. Note that the authors of Linda also experimented with similar implementations [11], but to our knowledge, they never published any results.

LIME also realizes direct access to remote tuples. Here, remote write access performs two operations: First, LIME injects tuples into the local tuple space and moves the tuple to the target location afterwards. If the destination agent is currently not connected, the tuple remains in the local tuple space and is marked as *misplaced tuple*. As part of the agent engagement process, LIME transfers all misplaced tuples if their destination becomes available.

Similar to tuple centers, LIME adopts the concept of *reactions*. Reactions represent a method that is called when a tuple with a predefined layout appears in the tuple space. Such an *event* is initiated either by a write operation or due to an engagement of an agent with misplaced tuples. Note that federated tuple spaces or remote agents do not provide these actions, i. e., the current location must always be local. Hence, LIME introduces *weak reactions*, thereby allowing reactions for tuples on remote agents. Both reaction concepts only differ by the fact that LIME executes standard reactions synchronously after a new tuple is detected, while weak reactions are called asynchronously and eventually after the tuple appears. This concept of reactions corresponds to the variable change subscriptions of PROViDE. Note that LIME only allows reactions to newly appearing tuples, while PROViDE allows subscriptions to arbitrary change operations.

Overall, the functionality of LIME appears similar to PROViDE for local variable management with remote access. However, PROViDE additionally allows the replication of proposal values. Thus, the proposal distribution of PROViDE is a more adaptive solution that mediates between LIME and Linda, as both distribution methods are possible. Additionally, PROViDE is an explicit solution to negotiate concurrent data access, whereas LIME does not address the problem. Instead, read and write operations always refer to a specific tuple space. The accessing function call specifies either explicitly or implicitly the destination agent. Thus, it lacks an explicit conflicting detection and resolution mechanism.

### 4.3.4 Physically Embedded Intelligent Systems

A physically embedded intelligent system (PEIS) [27, 146] refers to any device that incorporates computation and communication resources. Hence, the authors of PEIS developed a middleware acting as collaboration layer for heterogeneous devices such as robots or those operating in the IoT (Internet of Things). Therefore, the PEIS middleware core supports a dynamic model for self-configuration implemented in a distributed shared memory model. Due to the small memory footprint and a minimalistic API, PEIS is also suitable for small devices such as wireless sensor nodes. Therefore, the authors also provided an implementation for TinyOS called *Tiny PEIS kernel*. PEIS automatically detects new devices and is able to manage infrastructure as well as ad-hoc communication. Similar to LIME, PEIS detects dynamic groups of interconnected agents forming federated tuple spaces in a fully decentralized middleware.

Each PEIS tuple belongs to a namespace that identifies its *owner*, while each PEIS component can access tuples of arbitrary owners in the network. However, remote read operations require a subscription for the respective tuple, while local tuples are directly accessible. As a result, if a PEIS component writes a tuple remotely, it has to wait for an acknowledgment before reading the tuple value, otherwise, it might operate on an outdated value. Since each tuple belongs to exactly one owner, a single process instance schedules the access operations depending on their order in the receiving network buffer.

One important feature of PEIS participants is their built-in routing protocol, which ensures the automatic transmission to all subscribed nodes in the network. Nevertheless, PEIS suffers from the same drawbacks as LIME: Since a tuple is located at a single agent, it disappears from the tuple space when the owner disengages the network. Additionally, PEIS does not distinguish between value proposals and the final value decision. Although this mechanism is by definition free of conflicts, it does not investigate negotiations for conflicting value proposals, e. g., in case of contradicting observations. A further drawback is the fact that PEIS is not able to adapt its communication protocol to the needs of the application domain. In particular, the authors state that the protocol is based on TCP/IP, which is not suitable for robotic scenarios with unreliable network connections, as shown in Section 11.3.

### 4.3.5 Summary

All tuple and object spaces provide almost similar support with respect to our requirements. However, this class of approaches does not explicitly aim for the realization of team decision processes. Therefore, the *incomplete knowledge requirement* is not applicable. Two notable drawbacks of all presented approaches are their lack of robustness against conflicting value proposals and the adaptability of their communication behavior. We assume the requirement of location transparency to be the root cause for both drawbacks. However, location transparency is also the main strength of these approaches due to their full distribution and the ability to handle changing team configurations. An overview of the capabilities provided by the different approaches is given in Table 4.3.

## 4.4 Consensus Algorithms

A common method for achieving mutual agreement on replicated data are consensus algorithms, which we regard as the native form of distributed decision-making. As described in Section 2.3.2, consensus algorithms aim for two main properties: safety and liveness, i. e., if a consensus on a value is present, no process can chose a different value later on, and if a value is proposed by a process, it will eventually be chosen in the future. According to the results of the famous *FLP theorem* [57], a single faulty process is sufficient in an asynchronous distributed system to make simultaneous liveness and safety impossible. Nevertheless, in practice, many algorithms provide good liveness performances while guaranteeing safety. The following sections present the most common consensus algorithms.

| | Linda | JavaSpaces | TuCSoN | LIME | PEIS |
|---|---|---|---|---|---|
| **Distribution** | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Changing Team Configuration** | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Robustness** | ✗ | ✗ | ✗ | ✗ | ✗ |
| **Fault Tolerance** | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Availability** | ✓ | ✓ | ✓ | ✗ | ✗ |
| **Efficiency** | ✗ | ✗ | ✗ | ✓ | ✗ |
| **Adaptability** | ✗ | ✗ | ✗ | ✗ | ✗ |

✗ = no support ✗ = very limited support ✓ = partial support ✓ = full support

**Table 4.3:** Properties of object and tuple spaces following the requirements defined in Section 1.4.

### 4.4.1 Paxos

In computer science, the problem of distributed consensus [88] was introduced by Leslie Lamport, which resulted in the first and most commonly applied solution called Paxos [34, 89, 90]. Lamport introduced three main roles for Paxos processes:

**The Proposer** proposes a value, attempts to achieve agreement, and interacts with the client.

**The Acceptor** stores the replicated values.

**The Learner** infers values that provide consensus.

Additionally, at least one of the proposers can temporarily take over the role as leader, who authorizes the distribution of proposal values. Note that multiple leaders can coexist, without violating the safety properties of the protocol.

The Paxos protocol is composed of four phases: *prepare, promise, accept request, accepted*. The former two perform a leader election that eventually identifies a single leader. The latter two ensure that an agreement on a single value is reached. In order to achieve consensus, a proposer must receive a majority of acknowledgments in both phases. Otherwise, at least two proposers would act concurrently causing the protocol to restart. We will provide a more detailed explanation of the Paxos protocol in Section 12.5.

There are a number of extensions and optimizations to Paxos, which are specifically for problems that require agreement on multiple values [34]. The goal for this *multi-Paxos* algorithm is to replicate a fault-tolerant log of values. Therefore, the negotiation of each log entry instantiates a new Paxos instance. Multi-Paxos implementations need to cope with several badly documented implementation issues. In particular, Chandra, Griesemer, and Redstone [34] refer to configuration changes of the server setup, e. g., for acceptors that newly appear or disappear from the network.

Paxos provides a clear distinction between a value proposal and an agreed value. Hence, it explicitly considers conflicting proposals. Contrary to PROViDE, Paxos only selects a single value without any explicit criteria. More precisely, the agreement process determines the elected proposal only by the order of arriving messages. As shown in Section 12.5, Paxos can be implemented by the PROViDE middleware. Thereby, PROViDE is more adaptable to the problem setting. For example, PROViDE can reach agreement based on one of multiple criteria. In contrast, consensus in Paxos is always based on the order of received messages. To achieve this flexibility, PROViDE does not guarantee the safety properties of consensus algorithms. We assume that many robotic applications do not require consensus and rely on a reactive behavior execution, which Paxos handles inadequately slow. Moreover, Paxos requires at least a majority of agents to be within communication range for achieving a *Quorum* to meet the decision. As a consequence, Paxos is unable to make decisions in environments with sparse network connectivity such as in emergency response scenarios.

### 4.4.2 Chandra-Toueg

The key component of the Chandra-Toueg consensus algorithm [35] is an *eventually strong failure detector* for faulty processes. Such a failure detector monitors a list of processes that are suspected to have crashed. The term *eventually strong* refers to a detection procedure that guarantees the detection of all crashed processes. However, some processes are identified as crashed processes although they are still operating. Additionally, the failure detector can correct its false classification later on. The properties of such eventual strong failure detectors are satisfied by selecting a timeout value for missing acknowledgment to detect crashed processes.

The algorithm uses a rotating coordinator, which changes in every round. When a round is initiated, all processes send a message to the coordinator including their presumed proposal number and a time stamp. The coordinator selects the proposal with the highest time stamp and transmits it via broadcast to all processes. Once a process receives the message, it replaces its own proposal by the received one and replies with an acknowledgment. It is possible that the failure detector presumes the coordinator to have crashed. In that case, the process replies with a *negative acknowledgment* (NACK). Then, the coordinator waits until a majority of positive acknowledgment is received before sending a broadcast message containing the final value decision. Finally, all processes that receive this value decision for the first time forward the message to all other processes.

Similar to Paxos, this algorithm requires a majority of processes to be functioning. However, Chandra and Toueg [35] also describe enhancements for failure detectors with a higher precision that allow consensus for $n - 1$ faulty processes in an asynchronous network with $n$ processes.

Similar to the other consensus algorithms, the authors also identified the need for a distinction between possibly conflicting proposal values and the final decision as a key component for a consent decision. However, due to the three communication rounds, the algorithm requires a high amount of messages, which makes it impracticable for robotic systems. It also lacks in flexibility, since it cannot be adapted to environments requiring fast reaction times such as robotic soccer.

### 4.4.3 Raft

The development of the Raft algorithm [122, 123] was initiated by the insight that earlier consensus algorithms such as multi-Paxos are difficult to understand and their documentation is on a rather abstract level [34]. As a result, the drawback of erroneous implementations leading to the inability of satisfying the safety property appears. The authors concluded that an algorithm, which is easy to understand and implement, is required.

Each Raft process has one of the following roles: *leader*, *follower*, or *candidate*. Usually, only one leader responsible for the consensus decisions exists. In that case, all other processes are followers. Followers are passive processes that only serve as replicated data store and react to the requests of the leader. Similar to Chandra-Toueg, Raft divides its operation into terms, which emulate logical clocks. Each term initiates with a leader election among all candidate processes. To become a leader, all processes use heartbeat messages similar to PROViDE as failure detection mechanism. If a process detects a leader crash, it becomes a candidate process and participates in the election phase. To ensure that the election selects only one leader, the leader requires a majority of votes during the election phase. In case of a split vote, the process restarts.

After a successful election, the leader processes all requests from clients for consensus decisions. Since only one leader exists, the sequential processing of buffers in the network stack determines the order in which requests are processed. Leaders can only effectively establish consensus if they can successfully replicate the value to at least the majority of their followers.

Similar to Paxos, the Raft algorithm requires the majority of agents to be present within communication range to meet a decision. Hence, it is not suitable for environments with sparse connectivity. Moreover, establishing a mutual decision requires agents to be within communication range to the leader. Raft reduces the distributed problem of consensus to a distributed election problem. Since PROViDE can realize distributed elections, we consider Raft as a consensus application similar to the Paxos application presented in Section 12.5.

### 4.4.4 Summary

The most common application of consensus algorithms is the fault-tolerant replication in server farms. As a result, they support distributed settings while providing high availability and fault tolerance. Aside from RTDB, consensus algorithms are the only approaches that are clearly distinguishing between a value proposal and the finally chosen value. Thus, consensus algorithms are robust against conflicting proposals. However, the decision criteria are not explicitly specified. Instead, the decision depends on the order of arriving network packets. Nevertheless, the major drawback of all presented approaches is their inefficiency. This is mainly caused by the two communication rounds for establishing consensus. Additionally, none of the approaches disposes an adaptable communication protocol. Hence, fast decisions in the presence of unreliable communication are not possible. Table 4.4 shows the supported requirements in more detail.

| | Paxos | Chandra-Toueg | Raft |
|---|---|---|---|
| **Distribution** | ✓ | ✓ | ✓ |
| **Changing Team Configuration** | ✗ | ✓ | ✓ |
| **Robustness** | ✓ | ✓ | ✓ |
| **Fault Tolerance** | ✓ | ✓ | ✓ |
| **Availability** | ✓ | ✓ | ✓ |
| **Efficiency** | ✗ | ✗ | ✗ |
| **Adaptability** | ✗ | ✗ | ✗ |

✗ = no support ✗ = very limited support ✓ = partial support ✓ = full support

**Table 4.4:** Properties of consensus algorithms following the requirements defined in Section 1.4.

# Part II

# Solution

# 5 Robot Group Decision Process

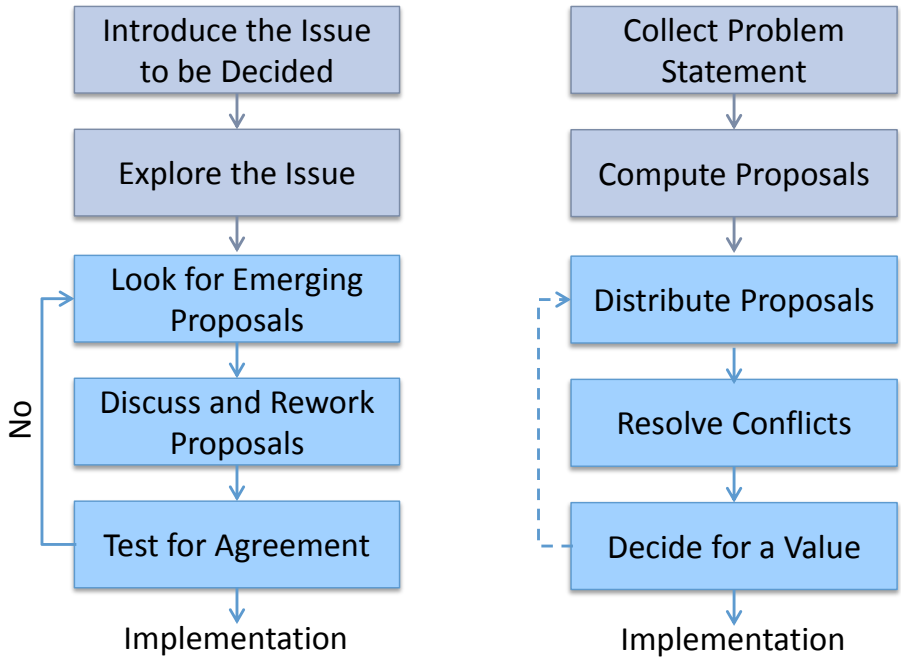
This chapter describes our decision process for teams of autonomous mobile robots. Section 5.1 illustrates the typical decision process for human collaborators, which inspired our solution. In Section 5.2, we explain our modifications to apply this process on teams of robots. The resulting process first identifies the problem and computes possible solution proposals, as described in Section 5.2.1 and Section 5.2.2, respectively. Afterwards, the agents initiate a negotiation process for all proposals, which we explicitly support with a communication middleware, see Section 5.2.3. This explicit separation of decision values from solution proposals in a multi-agent decision process forms a key contribution of this work.

## 5.1 Decisions in Human Teams

Decision processes are a well-studied subject of sociology [58, 81, 166]. Due to the human diversity, social decision processes are mostly adjustable to the composition of the group, the issue, the communication medium, and other parameters. The resulting flexibility builds the motivation to develop an approach for teams of robots inspired by such a human consensus decision process. Therefore, we adopted the profound and widely applied human decision process discovered by Susskind, McKearnen, and Thomas-Lamar [166], which is depicted in Figure 5.1 a. The process contains five steps:

1. Participants share relevant information and identify the key questions to introduce the issue to the group.
2. The group analyzes the issue and determines possible solution ideas. Thereby, the first elimination of bad solutions takes place.
3. Emerging proposals are collected and summarized, which further reduces the amount of proposals. This solution list should address people's key concerns.
4. The remaining proposals are discussed, clarified, and amended. Every participant should have the possibility to contribute or propose amendments.
5. Afterwards, the agreement is checked. This can lead to four kinds of concerns: Fundamental disagreement, reservations, agreement, or consensus.

In case Step 5 passes without any protests, the implementation of the decision starts. Otherwise, the process goes back to Step 3 and tries to resolve the concerns. Thus, the decision process assumes the participants to be willing to compromise, which iteratively reduces the number of proposals until a joint resolution is found.




**(a)** Simplified Decision Process in Sociology **(b)** PROViDE Decision Process

**Figure 5.1:** Comparision of the decision process to sociological decision processes. Blue boxes indicate the presence of communication.

## 5.2 Decisions in Teams of Autonomous Mobile Robots

In order to adequately adopt the decision process for robotic applications, we introduced two changes, as depicted in Figure 5.1 b. In particular, most current agents cannot apply reasoning on their knowledge to identify and summarize emergent proposals. Some research approaches on this topic exist, for example [137]. These require additional information, which allows the interpretation of the data by a machine, e. g., semantic data annotations and domain ontologies. However, the processing algorithms of these solutions are currently too slow for dynamic application domains such as robotic soccer. We therefore replaced this step by a simple selection mechanism. Here, the agent applies a selection criterion on every new proposal. As a result, the agent either rejects the new proposal or accepts it as its own.

The second change concerns the final decision rule: In human consensus decision-making, the negotiation process proceeds iteratively until only a single proposal remains, which each participant accepts. Consequently, the negotiation process is time consuming if the participants cannot eliminate their amendments. Since rational agents do not tolerate stubbornness of individuals but ground their decisions on facts, a common solution can be identified when all facts are present. This allows our decision process to support quick decision-making, which is again required in dynamic domains such as robotic soccer. Therefore, we regard a linear process as default method for multi-agent decisions.

Nevertheless, a feedback loop, as shown in Figure 5.1 b, can be applied optionally for the implementation of algorithms, which require multiple communication rounds such as consensus algorithms (see Section 12.5). The application developer can implement this feedback, which allows to adjust the trade-off between safety and liveness adapted to the problem setting.

### 5.2.1 Identification of the Problem Statement

Following our decision process as depicted in Figure 5.1 b, the first step is the specification of the problem statement and collection of relevant facts. Here, we use the constraint unfolding algorithm of ALICA for the extraction of the problem statement. The algorithm collects the relevant constraints of the current ALICA program. As the ALICA program is distributed at the deployment-time on all robots, it is ensured that all robots process the same problem structure. Nevertheless, conflicting problem definitions are still possible, as the problem parameters are determined at runtime. This means inconsistent observations of the environment usually lead to conflicting problem definitions.

As an example, we consider a positioning problem in RoboCup. Here, two robots decide for two target blocking positions in an opposing free kick situation. The given game situation defines a set of problem constraints for the blocking positions: According to the rules, only one robot is allowed to be inside the own penalty area. Since all parameters of such a constraint are predefined – for example the location of the penalty area – the constraint description must be equal for all agents if they execute the same ALICA program. Another rule predetermines that no robot is allowed to operate inside a three meter radius around the ball. In this case, the constraint is parameterized with the current ball position, which depends on each robot's observations. If robots make incoherent observations, problem descriptions become inconsistent. A possible solution for this problem might be to ensure agreement on the ball position before defining the problem parameters. However, this would slow down the decision process, as all relevant world entities would require agreement. Domains with many entities would especially require much communication overhead, e.g., in emergency response scenarios.

Most state-of-the-art robot control frameworks do not define problem specifications explicitly to identify robot decisions. Rather, decisions are encoded implicitly by the current state or action, which an agent executes, e. g., through the current ALICA state. Due to the implicit nature of the problem specification, reasoning methods become almost inapplicable. Additionally, developers of robotic behaviors have to face disadvantages in contrast to the explicit representation:

**No separation of problem specification and solution** Most robotic frameworks do not separate the problem specification from the *solution strategy*. This results in monolithic implementations of different atomic behaviors, which solve almost similar problems, i. e., behaviors that execute the same robotic skill are implemented multiple times to realize the different decisions. Experienced developers address this problem by introducing skill libraries. Hence, they reintroduce explicit problem descriptions, which determine the parameters for the library functions. However, such libraries reduce the maintainability for agile development groups in contrast to a comprehensive approach including explicit problem definitions.

**No separation of a decision and the behavior to reach it** Many robot behaviors try to implement similar decisions. A typical example is a behavior that moves the robot to a specific point on a global coordinate frame. Without a separation of the goal definition and the solution strategy, developers tend to duplicate the code of a behavior, which implements a similar decision. As a result, their application could provide multiple behaviors that move the robot to a certain point. In a robotic soccer scenario a covering behavior should be a parameterized version of the basic skill to approach a point on the field rather than the implementation of a second position controller.

**No reusability** Behaviors relying on implicit problem specifications tend to be overspecialized, e. g., in an emergency response scenario, a victim-grasping behavior represents a specialization of a general grasping behavior, which is parameterizable with a target object. Here, the overspecialized victim-grasping behavior is obviously less reusable for other tasks.

**Combination is difficult** Implicit problem definitions can only be combined if they are independent from each other, e. g., if one addresses a movement and another one a kicking action of a soccer robot. In contrast, the combination of several constraint satisfaction problems can be realized by a simple conjunction or disjunction.

### 5.2.2 Computation of Solution Proposals

The second step of the decision process is the computation of solution proposals. Therefore, every robot requiring a decision for the previously identified problem attempts to compute a solution proposal, e. g., in the RoboCup blocking scenario, both of the two blocking robots compute both target positions by solving the common constraint satisfaction problem. This requires an efficient solution technique, which enables fast response times. This work provides a generic approach that is applicable for a wide range of common robotic problems. In Chapter 6, we present nonlinear constraint satisfaction solver based on satisfiability modulo theories and incomplete local search. We use a unique combination of solution techniques, which allows to solve many typical problems of the robotic domain in a reasonable time.

Although nonlinear constraint satisfaction problems are very generic problem descriptions, the solver component is exchangeable. Moreover, our decision process facilitates a problem-specific solver selection, i. e., a specialized solver selection adapts the proposal determination to the problem class. Such problem-specific solvers are advantageous, as they can rely on problem-specific assumptions and heuristics to reduce the computation time. For example, a path-planning solver can make assumptions about the dynamics of the involved robots and the environment. To realize exchangeability of the solver, we extend the implementation of ALICA as explained in Chapter 9.

Many problem solving algorithms are developed by research communities, which use domain-specific knowledge to efficiently compute solutions for problems such as path planning, motion control, or reinforcement learning. Their efficiency in the targeted problem domains makes these approaches also valuable for many multi-agent systems. However, the majority of approaches does not explicitly address distributed settings, which

makes the integration into multi-agent systems a complex and time-consuming task. As our decision process supports the distribution of possible solution problems, it serves as a framework for their negotiation in a distributed setting.

A minority of approaches such as the ALICA gradient solver [158] consider distributed settings: They exchange (partial) solutions, which are used as initial seeds for local search algorithms. Thus, assuming a coherent problem specification, the solver converges to a common solution, as shown in [157, p. 208]. For instance, in the RoboCup blocking scenario, one robot computes the blocking positions and transmits them to the other robot, which then starts its gradient search by the received positions. In this case, both robots are most likely computing the same target positions if they search a solution for a coherent problem. Nevertheless, agreement on a common solution proposal is not guaranteed and depends on the processing schedule of the problem solvers and order of network transmissions. Hence, these types of solvers can still benefit from our decision process.

### 5.2.3 Negotiation

The final three steps imply a multi-tier communication protocol, which is simplified by a sophisticated middleware implementation that can cope with our requirements of Section 1.4. In Chapter 7, we introduce the core component of the decision process: the PROViDE middleware. This middleware unifies the decision process and facilitates the application development. Considering the RoboCup blocking scenario, both blocking robots might compute conflicting solution proposals, e.g., if both robots try to block the same position. Here, PROViDE collects both proposals in a distributed shared data space, tries to resolve possible conflicts, and achieves a decision for one of the two proposals.

PROViDE performs three logical process steps: proposal distribution, conflict resolution, and value decision. Therefore, in the *proposal distribution* step, a fault-tolerant and adaptive communication protocol replicates proposals to all agents. This forms a shared knowledge base for solution proposals, similar to tuple spaces.

The communication protocol also covers the conflict resolution process. Therefore, agents react to the appearance of new proposals. More precisely, they can decide to overwrite their own proposal according to the proposal knowledge base, which indicates their support for the new proposal. We coined this decision criterion *acceptance method*, which determines whether an agent supports a new proposal value. As agents may communicate their proposal changes, conflicting proposals are eliminated if the distribution process is successful and the acceptance method constitutes consistency. The exact mechanism and the conditions that ensure conflict resolution are described in Section 7.4.

The last step of the decision process implements a value *decision method*, which applies whenever the application requests a variable value. Since the conflict resolution does not guarantee the complete elimination of conflicting proposals, the final step forces a decision to prevent the negotiation process from stalling. This is equivalent to the human decision process, which uses either a democratic or an autocratic decision to select one of the remaining proposals.

ALICA determines the problem specification and implements the decisions. Hence, we integrated PROViDE as a middleware core to realize the communication and multi-agent

decision process [176]. Since ALICA variables are mostly used for explicit decision value representation, they are the intended negotiation object.

# 6 Proposal Computation

As described in the previous chapter, the second step of our decision process determines proposals for decision values. In this chapter, we first classify descriptions for typical robot decision issues in Section 6.1. As famous teamwork theories such as joint intention theory and shared plan theory are only rarely used in robotic applications, we give an overview of common practices in Section 6.2 and Section 6.3. Here, we explain that many applications use static behaviors that aim for a fixed goal, which cannot be dynamically adapted to new problem settings. In contrast, our approach foresees behaviors that are parameterizable with a decision value, e. g., a target location for a grasping behavior. This fosters the reusability of behaviors and facilitates reasoning and planning methods due to an explicit goal representation. However, a generic and expressive language that allows the description of common decisions is required.

Constraint satisfaction and optimization problems are versatile descriptions of multi-agent goal values. In Section 6.4, we give a detailed formal definition of the problem class we propose for multi-agent scenarios. Since this problem class is computationally expensive, we combine the solution techniques of satisfiability modulo theories, gradient descent, and interval propagation methods, as described in Section 6.5. The approach provides two major advantages: First, the unique combination of different techniques allows to solve relevant nonlinear problems in a reasonable time, which is appropriate for many robotic settings. Second, the approach can be extended by new theory solvers to address an even broader range of problems, e. g., for linear arithmetics, bit vectors, or quantifiers. Thereby, the only demand to the problem setting is a logic formula on the highest degree of abstraction.

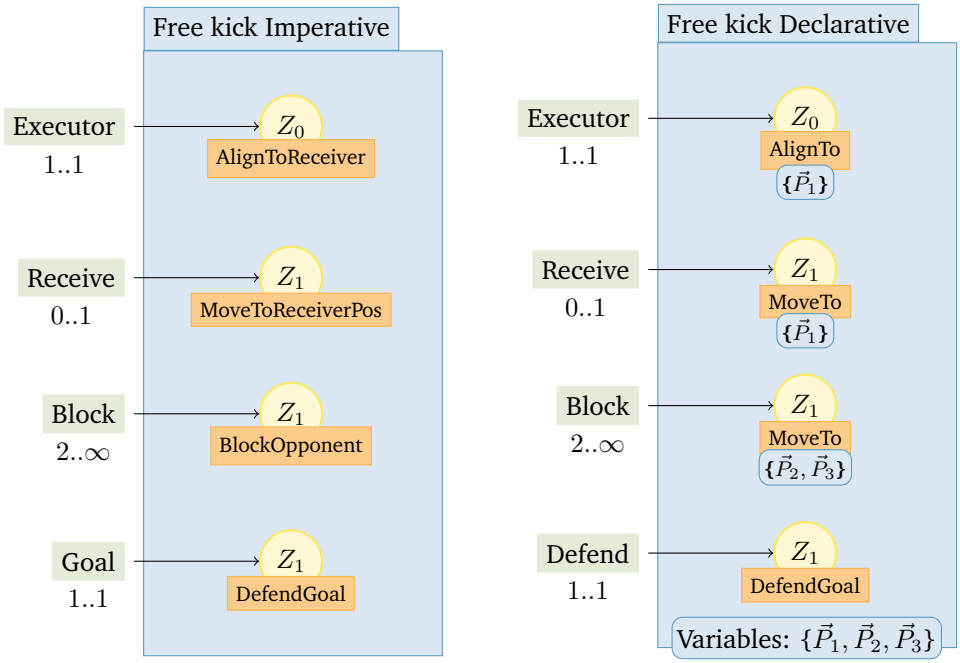
In order to demonstrate the efficiency of the presented solver, Chapter 10 provides an evaluation for a set of selected problems. These give an intuition of the problem structures fostering the efficiency of our approach and compare the results with other state-of-the-art solutions.

## 6.1 Problem Descriptions of Multi-Agent Decisions

In order to describe multi-agent decisions, we can identify two core approaches: imperative and declarative problem descriptions. The optimal description choice strongly depends on the type of problem and the experience of the developer with the selected method. One advantage of imperative descriptions is the fact that computer scientists are more familiar with them, as the most common high-level programming languages such as C++, Java, and C# are based on an imperative programming paradigm [26]. In contrast, declarative methods do not require knowledge about the solution procedure but the precise problem

description. For example, combinatorial problems such as the $N$-queens problem [54] can be described intuitively by using a declarative description without knowing a solution algorithm.

To clarify the difference between imperative and declarative descriptions, we use an example from robotic soccer. More precisely, we assume a positioning problem in a free kick situation, which is modeled in ALICA. Here, the referee places the ball at the position where a foul has been committed. Afterwards, the referee uses a graphical software interface to transmit a free kick command to all robots, which identify the referee's decision. As a consequence, the ALICA program transitions into a free kick positioning plan, which is shown in Figure 6.1. The plan positions four robots: One *executor*, who has to perform a pass of at least one meter, one receiver, who is placed at a distance of at least two meters from the ball and outside of the opposing goal area, and two blocking robots, which are meant to block the line of sight of two opposing robots to the ball. Additionally, the goalkeeper has to defend the goal, which does not necessarily require coordination with other agents. In the most trivial case, we position the robot on the line of sight between the ball and the center of the goal.




**(a)** Imperative description. **(b)** Declarative description.

**Figure 6.1:** Positioning plan for a free kick situation in RoboCup.

Figure 6.1 a shows an ALICA plan with an imperative problem description for a free kick situation. Here, we compute the receiver position by projecting the ball's position 2 m towards the center of the opposing goal. Additionally, we have to consider situations, where this projection is inside the opposing goal area. In that case, we turn the projection by 180 °, which allows the receiving robot to get a good shooting position after receiving

the pass. At the same time, the *Executor* task robot is placed close to the ball and aligns to the robot in the *Receiver* task. Finally, at least two robots can execute the *Block* task, where each robot randomly choses an opponent and moves within the line of sight between this robot and the ball. The major problem for this implementation is to ensure that each robot covers a different opponent. Therefore, we rely on an auction to negotiate the covered robots, e. g., based on the bully algorithm [63].

When examining the imperative solution, we can draw similar conclusions as Skubch [157]: First, propositional ALICA programs, which represent the imperative implementation, do not provide coordination support for the target position aside from the task allocation. This is caused by the absence of an explicit problem representation. Second, the solution lacks in generality: Each behavior needs a problem-specific implementation. Hence, we implemented separate behaviors for each task although each behavior uses a positioning controller. As a result, the size of the code base increases, which reduces the maintainability. Third, the architecture of ALICA does not generally force a separation between the computation of decisions and the algorithms to implement them. This again reduces the maintainability.

As shown in Figure 6.1 b, ALICA also allows for modeling the same plan in a declarative fashion. Therefore, we can determine the four agent positions with three target points: The pass destination and two blocker positions. Therefore, we translate the position description into formal constraints:

$$\begin{aligned}
&\mathrm{dist}(P_n, \mathrm{BallPosition}) > 2\,m && \forall n \in \{1,2,3\} && (6.1)\\
\wedge\,&\neg\mathrm{InsideArea}(P_n, \mathrm{OpponentGoalArea}) && \forall n \in \{1,2,3\} && (6.2)\\
\wedge\,&\Bigg[\bigvee_{opp \in \mathrm{Opponents}} \mathrm{dist}(P_n, opp) < 1\,m \\
&\quad \wedge \mathrm{InsideArea}(P_n, \mathrm{LineOfSight}(opp, \mathrm{BallPosition}))\Bigg] && \forall n \in \{2,3\} && (6.3)\\
\wedge\,&\mathrm{dist}(P_2, P_3) > 1\,m, && && (6.4)
\end{aligned}$$

where $\mathrm{dist}$ is the Euclidean distance between two points, and $\mathrm{InsideArea}$ a predicate, which determines whether the point given as first parameter is inside the area given with the second parameter. The predicate $\mathrm{LineOfSight}$ determines the corridor spanned by the two parameter points. Here, the belief base provides a list of opponents denoted by $\mathrm{Opponents}$, the ball position $\mathrm{BallPosition}$, and the location of the opponents' goal area $\mathrm{OpponentGoalArea}$.

The declarative representation reveals four advantages: First, the problem specification is similar to the natural language description of the problem. This is an indication of a more intuitive representation. Second, the robots only require two different behaviors to realize the positioning: *AlignTo*, which aligns the robot to a given target point while moving close to the ball, and *MoveTo*, which moves the robot to a certain target point while being aligned towards the ball. In contrast to a behavior such as *MoveToReceiverPos*, these behaviors also provide a better reusability in other plans, which inherently increases the maintainability and decreases the size of the source code base. Third, the plan can be extended more easily for different numbers of agents, as an additional agent only increases the set of target points and requires the Equation 6.4 to be extended for more than two blockers.

Fourth, this solution provides an inherent distinction between the problem description and the atomic behavior, which implements the problem solution. As a consequence, an explicit problem and solution representation is present, which allows us to use a generic negotiation algorithm that avoids conflicts similar to our decision process, as described in Chapter 5.

Although we claim many robotic problems to have similar characteristics, the example does not allow a generalization of the advantages regarding other problem settings. We also want to remark that the developer needs to identify the complexity of the described problem, as the solution process of some problems requires too much computational time. Thus, the capabilities of the problem solving algorithm have a strong influence on the favorable description.

## 6.2 Proposal Computation by Problem Solvers

We define the term *solver* as an algorithm capable of computing solutions for problem expressions such as constraint satisfaction problems. Almost all solvers rely on some kind of search algorithm. These use a search space that covers all relevant assignments for the variable values. One or more of these values may have characteristics that satisfy the conditions of a valid solution. These conditions are usually described by constraints encoded in the problem description.

The simplest approach for search algorithms is the brute-force approach. Here, the solver checks all possible value assignments for their validity. Hence, in finite spaces, the algorithm is ensured to find all possible solutions when all possible values are tested. However, as most problems allow too many possible assignments, a brute-force approach is usually too computationally expensive for most problem domains. Most continuous search spaces are typical examples for this property.

More advanced approaches try to exploit the structure of the search space through a *heuristic*. Broadly speaking, a heuristic describes a measure to identify probably good search directions. For example, gradient descent methods usually use the gradient direction as a heuristic to determine a new assignment. We call methods that start from some initial seed and try to improve the assignment iteratively until a solution is found *local search methods*. Another class of algorithms performs searches in a graph structure, such as $A^*$ [74]. Here, the heuristic indicates which next value assignment is potentially close to a solution. Tree searches have a high relevance in game-theoretic problems, playing chess [66], or task allocation, see Section 3.2.4.

## 6.3 Common Practice

With the term *decision-making*, the field of artificial intelligence refers to the computation of proposal values. It has a central role in artificial intelligence research, a wide range of approaches that aim for various kinds of decision issues has been developed [145]. Here, many researchers deliberately ignore problem-specific knowledge, as they aim for general-purpose approaches. Vice versa, other parts of the research community investigated related

fields such as control theory to develop algorithms that are dedicated to specific problem classes, e. g., path planning [95]. These approaches exploit knowledge of their intended use case to outperform general approaches in solution quality and computational efficiency.

In practice, the performance of domain specific algorithms encourages developers to combine multiple mechanisms in their robotic systems and query the most promising approach when a decision issue appears. This method follows our idea for a solver selection in ALICA, as described in Section 9.1. However, the diversity of tasks today's robotic systems have to fulfill increases systematically. For example, autonomous vacuum cleaners execute only two main tasks: cleaning and recharging. In contrast, we can expect future household robots to support in almost all areas of housework. Nevertheless, we claim that static tasks are solvable by static behaviors, i. e., a robotic engineer can identify a set of solution algorithms for all problems of such a domain.

As long as the complexity stays relatively low, general-purpose solvers are sufficient. In these cases, the costs of integrating specialized solutions are not necessary. Therefore, we propose a multi-purpose problem solver, which is suitable for the most typical problem classes of robotic systems. The next sections describe the proposal computation step of our decision process, which consists of a problem solver that addresses the most common small-scale robotic problems. Thereby, we specifically considered the computational efficiency to make the solver applicable in fast and dynamic environments.

## 6.4 Continuous Nonlinear Constraint Satisfaction and Optimization Problems

The declarative example from Section 6.1 and the requirement to develop a multi-purpose problem solver allows to infer the targeted problem class. Here, we have to face an assignment problem with the goal to assign values to $n$ variables of the set $X$. Since robots operate in the physical world, we have to deal with real-valued and continuous domain ranges for the variables, respectively.

The solution is determined by a propositional formula $\phi$ with the variables $P$. Each $p_i \in P$ identifies a constraint $c_i = f_i(\vec{x}) \circ_i g_i(\vec{y})$, where $\circ$ is one element of the set of supported operators defined as $\{<, >, \leq, \geq, =, \neq\}$. Furthermore, it holds that all $x_i \in \vec{x}$, $y_i \in \vec{y}$ are element of $X$. $f$ and $g$ are arbitrary functions $\mathbb{R}^k \mapsto \mathbb{R}$. Here, we explicitly include nonlinear functions such as $\{\sqrt{}, \sin, \cos\}$. We see nonlinear constraints as highly required in robotic systems, as they enable descriptions of geometric relationships between objects and the goal values, e. g., the distance between two points or the orientation angle of the robot in a global coordinate frame.

An interpretation is the valuation function $v\colon X \mapsto \mathbb{R}$, which is extended to $P \mapsto \{\top, \bot\}$ by

$$v(p_i) = \begin{cases} \top & \text{if } v(f_i(\vec{x})) \circ_i v(g_i(\vec{y})) \\ \bot & \text{otherwise} \end{cases}$$

Based on these components, we call the targeted problem class *continuous nonlinear constraint satisfaction problem* (CNCSP) according to [158] and [179]. According

to Richardson [140], this problem class is in general undecidable. Hence, all existing solvers can either not guarantee termination or stop their search under certain termination criteria. If we consider $\phi$ as a SAT problem of constraints, it has also been proven that this partial problem is $NP$-complete [42]. In order to address this problem class nevertheless, our solver combines incomplete local searches, as described in Section 6.5. This approach allows to address many small real-world problems in robotic scenarios without the need for domain knowledge or a manually tweaked heuristic.

We extended the problem class to a *continuous nonlinear constraint optimization problem* (CNCOP) [178]. Here, a utility function $U$ is additionally optimized, which allows the characterization of the solution quality. The utility function is defined as

$$U(\vec{x}) = \sum_{I} u_i(\vec{x}), \tag{6.5}$$

where $u_i$ are again arbitrary possible nonlinear functions for which $U(\vec{x}) > 0,\ \forall \vec{x} \in \mathbb{R}^k$ holds. As described in Section 6.5.5, we require this condition to combine $U$ and $\phi$ to a single expression.

The following equation gives an example for a utility function, which extends the positioning problem of Section 6.1:

$$U(\vec{x}) = \sum_{i=2}^{3} \text{maxDistance} - \text{dist}(\vec{P}_i). \tag{6.6}$$

The resulting problem description prefers blocker positions that are close to the ball over positions that are far away.

## 6.5 Carpe Noctem Satisfiablility Modulo Theories Solver

CNCSPs and CNCOPs are specific variants of satisfiability modulo theories (SMT) problems [17]. In general, SMT problems are decision problems for a logical formula that incorporates background theories to solve first-order logic expressions with equality expressions. Most approaches transform the logical formula into an abstract satisfiability problem, which we refer to as *propositional* problem level. Afterwards, a valid assignment is determined on (partial) solutions of the propositional problem by applying specialized theory solvers (T solver). We classify this assignment problem as *theory* problem. Unlike our approach, most SMT solvers only support decidable theories such as bit-vector or linear arithmetic theory [60, 110].

An interesting example for a theory solver is based on *Presburger arithmetic* by Presburger [131]. Presburger proved the completeness of all first-order problem expressions over natural numbers with additions. If a solution for such a problem exists, Presburger arithmetic is able to find it. Otherwise, the algorithm can prove that no solution exists.

Our approach for the proposal computation follows the general idea of SMT Solvers. However, the advantage is the incorporation of a T solver that can handle nonlinear geometrical problems, which appear frequently in robotic scenarios. In the next sections, we describe the combination of interval propagation and incomplete local searches with the techniques of the SMT research community.

### 6.5.1 DPLL Core

Similar to SMT approaches, our problem solving algorithm is based on the Davis-Putnam algorithm. In 1962, the algorithm was extended by Martin Davis, Hilary Putnam, George Logemann, and Donald W. Loveland to the DPLL algorithm [47], which is named according to the last names of its inventors. Listing 6.1 shows the main loop of the algorithm. The DPLL algorithm receives a set of logical clauses as input and computes whether an assignment that evaluates all clauses to $\top$ exists. Therefore, the algorithm tries to find an assignment to the *propositional variables* of the clauses to prove the existence of a solution.

The DPLL algorithm consecutively assigns values to unassigned variables, which are selected by a heuristic in the *Decide* function. Afterwards, all clauses are checked for conflicts, i. e., clauses that evaluate to $\bot$. To perform this check, the *Propagate* function assigns all literals that are implied by the former assignments, i. e., if a clause does only have a single unassigned literal and is not already satisfied, this literal has to be assigned to $\top$ to avoid a conflict. This procedure terminates when all implications are assigned.

Afterwards, the *Propagate* function evaluates the assignment and identifies conflicts. In case a conflict is detected, the *ResolveConflict* function performs backtracking to the last assignment decision[1]. The algorithm inverts this assignment to attempt conflict resolution. As a result, unsatisfiable problems cause backtracking of the conflict resolution to the first assignment decision for both possible assignments. In this case, an inconsistency is present, which proves the unsatisfiability of the expression. In contrast, satisfiable problems allow at least one assignment to all literals, which does not result in a conflict.

```
while true do
    if Decide()=∅ then
        return true
    end
    while Propagate() do
        if !ResolveConflict() then
            return false
        end
    end
end
```

**Listing 6.1:** Davis-Putman algorithm.

In order to apply the DPLL algorithm, we transform the CNCSP into a set of clauses. Therefore, we first substitute the arithmetic expressions by *propositional variables*, which serve as a placeholder. Hence, we solve the expression on an abstracted logical layer before investigating the arithmetic complexity. During the execution, the DPLL algorithm computes (partial) solutions, which can be transformed back into a set of conjuncted arithmetic expressions, which serve as input for the T solver.

In a second transformation step, the expression needs to be converted into a conjunctive normal form (CNF). Through the application of *De Morgan's* laws and negation elimination,

[1] In contrast to robot decisions, assignment decisions only refer to the assignment to a propositional variable during the solution process as *true* or *false*.

every logic expression can be converted into a CNF. Therefore, we rely on a naive recursive implementation, which might result in an exponential number of clauses [145].

### 6.5.2 Propositional Satisfiability

Borrowed from the implementation of Minisat [52], our solution starts with the extraction of *unit clauses*. This term refers to clauses that only consist of a single literal. Unit clauses are used to define prior knowledge, as they require an assignment to $\top$ to satisfy the input expression.

The solution process is guided by a history of assignments, which enables the implementation of the backtracking procedure and identifies the current candidate solution. Furthermore, a list of *decision levels* that counts the number of decisions and refers to the elements in the assignment history of each decision level is maintained. Here, decision level $0$ classifies all assignments inferred from prior knowledge, i. e., unit clauses.

According to the DPLL algorithm, our solution consists of a main loop that executes the algorithm until it either proves unsatisfiability or finds a valid assignment. To identify inconsistencies within the unit clauses, the loop starts with a propagation step. It identifies all implications from prior knowledge, assigns to the decision level, and identifies possible conflicts.

The *Propagate* algorithm is borrowed from the Chaff SAT solver of Moskewicz et al. [109]. The algorithm initially assigns *watcher* to two randomly chosen literals of each non-unit clause. The *Propagate* method checks all *watchers* whether a past assignment (*assumption*) or an implication evaluates the referenced literal to $\bot$. If the clause is not already satisfied, one watchers are moved to another literal. This literal has to be either unassigned or $\top$. In case the clause has no such literal at disposal, a conflict is present and the method returns *true*. If the clause does only contain a single unassigned literal, it has to be assigned to $\top$ to avoid a conflict. As such an assignment is an implication, it does not create a new decision level. This method proceeds until all watchers refer to a valid literal. Afterwards, the method returns *false*, as no conflict is present and all implications are assigned.

If the *Propagate* method detects a conflict at decision level 0 or the conflict cannot be resolved, the SAT problem is not satisfiable. In contrast to a SAT solver, our algorithm cannot guarantee the detection of unsatisfiable problems, as the conflict may result from the incomplete T solver. Thus, instead of returning unsatisfiability, we delete all clauses that are unsafe, i. e., implied from the conflict resolution method. *Unsafe clauses* are inferred by the T solver or are implied by a clause that has been inferred based on an implication caused by the T-solver. The only case where our solver can deduce unsatisfiability is if a conflict is detected although the list of unsafe clauses is empty. However, since we address robotic problem settings, we can assume that the developer does not aim for unsatisfiability proof.

The *ResolveConflict* method performs a decision backtracking procedure to the last assignment, which is not an implication, inverts this assignment, and learns a new clause that ensures that the same conflicting assignment decisions cannot occur twice. Since the propagation method checks the switched assignment, the algorithm can iteratively backtrack the conflict to an earlier decision level if the flipped literal caused a conflict in a

prior propagation step. If the clause learning function infers a unit clause, which conflicts with another unit clause, it is again incapable of inferring unsatisfiability. Hence, we delete all unsafe conclusions and continue with the algorithm, unless the list of unsafe clauses is empty.

The clause learning algorithm follows the GRASP scheme [156], which is also part of Minisat [52]. The algorithm identifies the last *unique implication point* (UIP). Therefore, a clause is learned iteratively. This clause is a collection of all assignments, which are a *reason* that caused the unsatisfiability of the current assignment. The *reason* for a literal assignment is an *assumption* or a clause that implied the assignment by propagation. In the first case, the assumption is added to the learned clause. Here, an assumption refers to an assignment based on a decision. In the latter case, the literals, which forced the implication by the propagation algorithm, are marked as a reason and added to the learned clause. This process repeats until a unique reason for each literal remains, the last UIP. The resulting clause prohibits the same conflict from occuring twice, which speeds up the search process and drives the backtracking procedure. Note that the efficiency of the learning procedure is problem specific. Modern SAT solvers rely on empirical studies, which are based on common problems such as the problem sets of the SAT competition[2]. In empirical experiments with problems of the robotic domain, we observed only a minor influence on our solution performance. We assume that this is caused by the simplicity of propositional problems of robotic domains compared to the problems addressed by state-of-the-art SAT solvers.

After detecting and resolving all conflicts, a partial propositional solution is present. Therefore, the T solvers can be queried for an assignment on theory level. We incorporate two approaches: One relies on interval propagation, while the second uses local search techniques to determine a solution. As shown by Fränzle et al. [60], recursive interval search can be applied to realize a complete solver. However, for non-convex constraints, the algorithm lacks in efficiency, as we will show in Chapter 10. Nevertheless, we use a fast version of the interval propagation algorithm to determine intervals for the local search, which in some cases can speed up the local search process and increase the probability of finding a solution. All partial solutions are checked for a partial solution on the theory level by the local search solver.

In case the local search solver can find a solution, it is cached as starting point for later searches. When computing a partial solution for a more complex statement the former cached results are close to a solution in many cases, as the constraint already partially holds. In case the T solver cannot find a solution, we add an unsafe clause, which contains the inverted current assignment. Afterwards, the propositional conflict detection and resolution are called. This enables the learning of a new propositional clause from the conflict on theory level and avoids similar assignments in the future.

The next step of the solver checks whether the current assignment is a solution for the complete problem. This is the case if all clauses are satisfied. As a result, the solution is the last cached value. If unsatisfied clauses exist, a new assumption can be made, i. e., a new decision level is added and an unassigned literal is chosen and satisfied. The most common decision heuristic of modern SAT solvers is called *variable state independent decaying sum* (VSIDS) [109]. Here, each propositional variable has a counter for each polarity.

[2] `http://www.satcompetition.org/` (accessed 2015-06-29)

When a new clause is learned, the counter of all included literals is increased. Here, the VSIDS selects for each decision the unassigned literal with the highest counter. As a result, this decision mechanism tries to satisfy as many clauses as possible. The Minisat solver relies on an alternative approach, which selects the assignment with the highest activity. The activity increases whenever an assigned variable is involved in a conflict. Hence, the variable with the highest conflict potential is assigned over time. Nevertheless, in our experiments, the VSIDS approach showed better performances, which may be caused by the simple propositional structure of robotic problems compared to the problem instances of the SAT competition.

As a final step, our algorithm reduces the database for learned clauses after every $1000$ decision. The goal of this operation is to speed up the propagation by deleting rarely required clauses. Therefore, each clause has a counter, which increases whenever the clause is involved in a conflict. Additionally, the counter decays over time to prevent clauses that were initially required from persisting in the database. If the number of clauses exceeds a certain threshold, the least active clauses are removed. An overview of our algorithm is shown in Listing 6.2.

```
while true do
    while Propagate() do
        if decisionlevel = 0 or !ResolveConflict() then
            if !DeleteUnsafeConclusions() then
                return false;
            end
        end
    end
    if !IntervalPropagate(decisions) or !LocalSearch(decisions, intervals, lastSolution) then
        continue
    end
    if Satisfied(decisions) then
        return true
    end
    Decide()
    ReduceClauseDB()
end
```

**Listing 6.2:** Extended DPLL main loop using local search and interval propagation as incomplete solution theories.

Many state-of-the-art SAT solvers perform restarts periodically. Here, all decisions are reverted but the learned clauses remain unchanged. Since the learned clauses induce prior knowledge, the initial problem is simplified and the solver avoids being stuck in adverse assignments. However, the number of decisions in our targeted types of problems is rather low. Thus, we excluded restarts from our algorithm, as we could not identify a contribution to efficiency.

### 6.5.3 Interval Propagation

The currently most efficient implementation of *interval propagation* (IP) methods within an SMT solver, called iSat, was developed by Fränzle et al. [60]. The iSat solver transforms the problem expression into a tree-like structure, where each operator and operand is represented as a node. More precisely, each operator is the parent of its operand nodes. In the same fashion, the parent appears as an operand of higher order operators. A solution interval is propagated downwards from the root to all leaves and, vice versa, upwards along this tree structure [178].

During the propagation, the solution interval contracts according to the inspected expression. In case of iSat [60], the propagation is refined by splitting the interval until a solution is found or the intervals collapses. The solution interval describes an outer boundary of the solution. To reduce the search space for the interval propagation as well as for the local search, the application developer specifies maximum domain ranges as a solution interval. In most cases, these are inherently defined by the problem domain, e. g., when computing target positions, the target area can be specified, i. e., blocking positions of soccer robots should be within the boundaries of the soccer field.

During the propagation, the CNSMT solver a solution interval for each node. For example, for the expression $0 < \sin(x)$, the root node is the $<$ operator, with two operands $0$ and $\sin(x)$, where $x$ is again an operand of the $\sin$ operator. Hence, the expression includes $4$ intervals. Initially, the intervals are set to their maximum possible range: The constant $0$ has the interval $[0, ..., 0]$, the variable $x$ the domain interval,$<$ operator $\in [1, ..., 1]$, and $\sin \in [-1, ..., 1]$. During the downward propagation, each node recursively updates its interval with the intervals of its child nodes. The $<$ operator requires to be *true*, which is mapped onto the interval $[1, ..., 1]$. Here, we propagate the lower boundary of the left operand to the upper boundary of the right operand and vice versa. This results in the interval $[0, ..., 1]$ for the $\sin$ node, which does not shrink the boundaries of $x$ due to the periodicity of the $\sin$ function.

The upward propagation follows the same scheme with the exception that the algorithm starts at all leaves and propagates the intervals recursively to the root. As an example, we consider the expression $\sqrt{x^2 + y^2} < 2$. Here, we start again with the downward propagation. We can assume domain value ranges of $[0, ..., 18]$ for $x$ and $[0, ..., 12]$ for $y$ corresponding to the size of a soccer field in the RoboCup MSL. In the first upwards step, we have to raise the intervals with the power of two: $x \in [0, ..., 18^2]$ and $y \in [0, ..., 12^2]$. The sum of $x^2$ and $y^2$ leads to the sum of the respective intervals. After computing the square root, we receive $\sqrt{x^2 + y^2} \in [0, ..., \sqrt{18^2 + 12^2}]$. Since this expression has to be lower than $2$, the interval of the full expression is $[0, ..., 2]$. If we perform a downward propagation afterwards, the algorithm concludes: $x, y \in [0, ..., 2]$.

The previous example shows that multiple propagation steps are required to achieve convergence. Therefore, the algorithm performs alternating downward and upward propagations until the sum of absolute interval changes deceeds a certain threshold. Nevertheless, most expressions do not require more than $2$ downward and upward propagations. Solvers such as iSat [60] use a refinement method that splits the computed intervals to increase the precision. On the one hand, this allows a refinement until the intervals converge to a solution. On the other hand, this leads to a combinatorial explosion

of possible interval splits and a high computational complexity. Hence, we did not consider interval splitting for our approach.

The CNSMT solver calls this interval propagation process once in each iteration of the main loop and once in the initialization phase. This allows the early determination of unit clauses inferred from collapsed intervals. These provide prior knowledge in the propositional problem. Additionally, the stored intervals determine the domain ranges for the search phase. Furthermore, collapsed intervals form safe clauses for the propositional problem and therefore allow inference of unsatisfiability of simple problem classes. For more complex problem classes, for example non-convex problems, the inferred variable intervals determine boundaries for the seeds of the local search.

### 6.5.4 Local Search

The local search algorithm is based on the approach of Skubch [158]. In the general scheme, Skubch [158] transforms a CNCSP expression into a *fitness function*, which is evaluated to values greater $1$ if all constraints are satisfied. Otherwise, the function returns a value below $0$, which indicates a degree constraint violation. In some sense, this approach defines suitable membership functions similar to a fuzzy logic, which allow the inference of the adequateness of a solution. To find a solution for the generated function, local search operates on multiple random start points. The major advantage of this approach is its high performance in practical in practical CNCSP instances, as shown in [158] and [157]. However, this comes at the price of incompleteness. Hence, the algorithm cannot distinguish whether no solution exists or the searching time was insufficient. However, our experiments suggest that many practical problems can be solved correctly in less than 100 ms on current state-of-the-art CPUs.

The usage of incomplete local search in our SMT scheme provides two major advancements: First, we divide the potentially complex problem expression into smaller sub-problems with potentially lower complexity. Second, the solution process for problems with a complex logical structure can be simplified on the propositional level, as shown in Section 10.1. In particular, disjunctions are removed from the problem statement. In some cases, this can simplify the error landscape from non-convex to convex problems.

The formula transformation is divided in two parts. First, one of the following rules applies for each inequality constraint:

$$\mathcal{T}(a < b) = <^*(a, b) \tag{6.7}$$

$$\mathcal{T}(a > b) = <^*(b, a) \tag{6.8}$$

$$\mathcal{T}(\neg(a < b)) = \leq^*(b, a) \tag{6.9}$$

$$\mathcal{T}(\neg(a > b)) = \leq^*(a, b) \tag{6.10}$$

$$\mathcal{T}(\neg(a \leq b)) = <^*(b, a) \tag{6.11}$$

$$\mathcal{T}(\neg(a \geq b)) = <^*(a, b) \tag{6.12}$$

This step normalizes the inequalities to the two ”membership functions” $<^*$ and $\leq^*$. These

functions are defined as:

$$<^*(a,b) = \begin{cases} 1 & \text{if } a < b \\ b - a & \text{otherwise} \end{cases} \tag{6.13}$$

$$\leq^*(a,b) = \begin{cases} 1 & \text{if } a \leq b \\ b - a & \text{otherwise} \end{cases} \tag{6.14}$$

$$\tag{6.15}$$

As a result, we retrieve a list of necessary solution conditions, which appear in conjuncted form. In the second step, the formula representing the problem expression is created by combining the inequality constraints with

$$\Sigma_\wedge(a,b) = \begin{cases} 1 & \text{if } a = 1 \wedge b = 1 \\ \min(0,a) + \min(0,b) & \text{otherwise} \end{cases} \tag{6.16}$$

$$\mathcal{T}(p \wedge q) = \Sigma_\wedge(\mathcal{T}(p), \mathcal{T}(q)). \tag{6.17}$$

Although other promising representations are possible, our solution uses a sum-based combination for the conjunctions. Briefly, it returns the fitness value of the more violated constraint. Skubch [158] compared the performance to a combination with $\min$ operator known from the Gödel t-Norm of fuzzy logic [107]. However, in a majority of the tested problem instances, the sum-based approach showed a better performance.

As the design of the $<^*$ and $\leq^*$ functions indicate an error measure for the degree of constraint violation, we can perform gradient ascent methods as local search techniques to find a function value greater zero. Therefore, our algorithm first picks a starting seed $\vec{s}$ from a *seed generator* within the solution ranges that are determined by the interval propagation. Afterwards, we compute the gradient vector $\vec{g}_i$ for this seed, where $i$ denotes the $i$th iteration. The derivations for the conjunction operator and the functions $<^*$ and $\leq^*$ are given by:

$$\frac{\delta}{\delta x_i}\Sigma_\wedge(a,b) = \frac{\delta}{\delta x_i}(a+b) \tag{6.18}$$

$$\frac{\delta}{\delta x_i} <^*(a,b) = \begin{cases} 0 & \text{if } a < b \\ \frac{\delta}{\delta x_i}(b-a) & \text{otherwise} \end{cases} \tag{6.19}$$

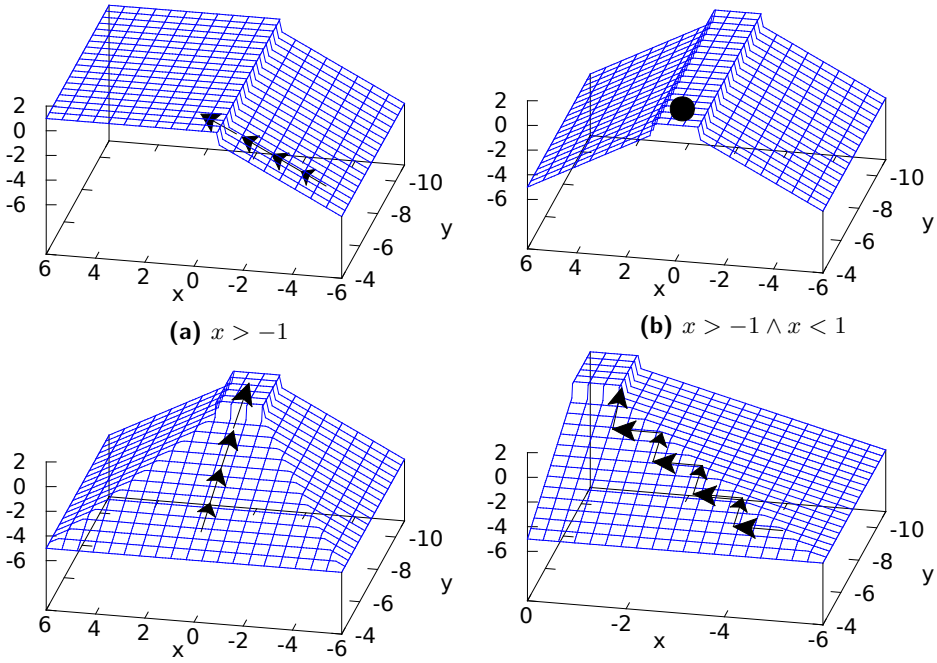
$$\frac{\delta}{\delta x_i} \leq^*(a,b) = \begin{cases} 0 & \text{if } a \leq b \\ \frac{\delta}{\delta x_i}(b-a) & \text{otherwise} \end{cases} \tag{6.20}$$

The derivation of the problem formula is computed by using a auto differentiation library of Shtof [154]. Note that the library relies on a tree-like representation and the visitor design pattern to compute the derivation. We used the same data structures to implement the recursive interval propagation algorithm.

The update step of the gradient ascent approach relies on the resilient propagation (Rprop) algorithm of Riedmiller and Braun [141], which was originally developed for the training of neural networks. A standard gradient ascent rule would update the current evaluation point with a learning rate by: $\vec{s} + \alpha\vec{g}_i$. In contrast, Rprop assumes that the gradient length

does not allow conclusions on the optimal step length. Hence, a fixed problem-specific step length is assumed: $\vec{s} + \gamma\,\mathrm{sgn}(\vec{g}_i)$. In order to travel long distances, $\gamma$ increases by a factor $\eta^+$ if $\mathrm{sgn}(\vec{g}_i) = \mathrm{sgn}(\vec{g}_{i-1})$. Vice versa, Rprop assumes that a local optimum has been bypassed if $\mathrm{sgn}(\vec{g}_i) \neq \mathrm{sgn}(\vec{g}_{i-1})$, and decreases $\gamma$ with a factor $\eta^-$ to facilitate convergence. In general, the values of $\eta$ are problem-specific. Nevertheless, in many problems, the values $\eta^+ = 1.2$ and $\eta^+ = 0.5$ showed good performances and are therefore usually chosen as default parameters for our approach.

Various differet update rules for gradient descent/ascent problems exist. In particular, the Levenberg-Marquardt (LM) algorithm [104] achieves fast convergence for a broad range of problem instances. The LM algorithm exploits the Jacobian to estimate the step length, which requires second-order derivation of $\vec{s}$. However, in our particular case, the computation of the second-order derivation is a computationally expensive operation, since the function is a rather complex expression and good approximations cannot be assumed. Therefore, we do not further investigate the LM gradient descent algorithm in the context of this thesis.




**(a)** $x > -1$

**(b)** $x > -1 \wedge x < 1$

**(c)** $x > -1 \wedge x < 1 \wedge y < -9$

**(d)** Gradient trace without problem decomposition

**Figure 6.2:** Example of problem decomposition and gradient ascent with (a,b,c) and without (d) propositional problem decomposition. In both cases, the approach starts from the initial seed $(-5, -5)$.

Since the local search with gradient ascent is incomplete, our solver needs a restart limit to assume unsatisfiability of the current expression. Due to the efficient problem decomposition on the propositional level the problem complexity for the solver increases only slowly, which makes cached solutions – despite in the possible presence of new

constraints – to a strong indicator of where a new solution might be found. Figure 6.2 shows the problem decomposition of our approach for the problem:

$$\underbrace{x > -1}_{A} \wedge \underbrace{x < 1}_{B} \wedge \underbrace{y < -9}_{C} \tag{6.21}$$

For this example, we assume that the gradient solver starts with an initial seed $\vec{s}_0 = (-5, -5)$. If the solver is queried once for the propositional problem $A \wedge B \wedge C$, the gradient ascent needs to perform $8$ update steps, as shown in Figure 6.2 d. Due to the propositional problem decomposition, it is possible that the solver is first queried for a solution of $A$, then $A \wedge B$, and finally $A \wedge B \wedge C$. In this case, the solution of $A$ results in $\vec{s}_4 = [0.08, -5]^T$ as a result of 4 update steps, see Figure 6.2 a. When the solver is queried in the next partial problem $A \wedge B$, it can directly return the cached solution $\vec{s}_4$, which already satisfies this problem (Figure 6.2 b). The final query again requires $4$ update steps to compute the final solution, as shown in Figure 6.2 c. Overall, the number of computed update steps is again $8$. However, the crucial difference is that the solver needs to evaluate a less complex function for the first $4$ update steps. Assuming a constant step length for Rprop, the number of update steps is equal for all possible starting points in this error landscape. We conclude that such a problem can be computed more efficiently by using the described SMT technology. In practice, this efficiency allowed us to reduce the number of restarts before assuming unsatisfiability to two.

### 6.5.5 Constraint Optimization

As mentioned before, our solver allows the extension of the CNCSP to a global optimization problem (CNCOP). This is particularly useful to determine the best solution among all solutions satisfying the constraints. Considering the RoboCup blocking problem, a situation where two robots have to block two of the four opponents can appear. In that case, a utility function prefers to block opponents close to the ball, as these are more likely to attack the ball.

To address the CNCOP, we exploit the fact that the transformed constraint function has an upper bound of $1$. Hence, arbitrary utility functions with a lower bound of 0 can be added to this function and will be optimized by the gradient-based T solver. Nevertheless, we have to consider two issues: First, we have to take the incompleteness of the solver into account, which does not provide a guarantee to compute the global optimum. Second, the solution of the CNCSP does not need to identify all possible propositional solutions. However, the function optimum search requires the search within all possible propositional solutions. To achieve this, the solver stores the solution with the highest utility and adds a new clause, which excludes the found solution from the search space. Since this new clause makes the problem unsatisfiable at the propositional level, the solver continues its search to find a new solution. Finally, after all propositional solutions have been analyzed, the solver returns the solution with the highest utility value. This method is also referred to as branch-and-bound [94].

Since our solution is only executing two restarts per propositional solution, the optimization does not cover complex utility search space well. In contrast, we ensure optimality at the propositional level, as we can analyze all possible propositional solutions.

### 6.5.6 Coherent Proposals

The presented solver computes proposal values on each agent individually. Due to inconsistent knowledge bases, the problem statement might differ between the agents. Hence, the computed solutions for the robots can also deviate from each other. One reason for a deviation is the fact that agents compute the problem formula during runtime under consideration of their observations. Hence, the problem descriptions themselves could differ if no mutual beliefs on the foundations are present. A further reason is the structure of most CNCSPs, which usually refers to multiple solutions, as the problem space is continuous and the constraints do not shrink it to a single solution. Due to the random seeds of the local search solver, the distributed execution fosters deviations in the final solution. For the same reason, symmetric problems can result in symmetric solutions. For example, considering a blocking problem in robotic soccer with two opponents ($O$ and $P$) and two blockers ($A$ and $B$), a possible solution could be that robot $A$ blocks opponent $O$ and robot $B$ blocks opponent $P$. Except the constraint describes a fixed assignment of robots to opponents, a valid solution would also be an assignment of $A$ to $P$ and $B$ to $O$.

Although our decision process aims for common solutions in later steps, the seed generator provides an ideal basis to achieve coherent solutions at the theory solver level. Therefore, the solver posts the final problem solutions to a *variable synchronization module*, which distributes the solutions among all agents. When the seed generator is queried, it first returns the solution having most supporting agents. Here, agents use the same solution proposal if the Euclidean distance between the solution factors deceeds a certain problem-specific threshold. As all agents apply this method, equal starting seeds are likely. This in turn increases the probability to converge to a common solution. However, since the problem statement may differ, convergence is not guaranteed.

Note that the idea of seed distribution was originally described by Skubch [157]. The major difference of [157] is the fact that the gradient solver analyzes the problem expression with a higher number of restarts. This allows the verification of more seeds rather than only of the most supported one. An even more sophisticated approach, which is conform with the SMT technology for achieving convergence, is presented by [129]. Here, the agents exchange partial solutions, which can increase the convergence probability if the problem changes only concern parts of the statement. We consider such a support of partial solution exchange as future work.

## 6.6 Summary

In this chapter, we first presented multi-agent problem description methods and distinguished between explicit and implicit problem description as well as declarative and imperative description. We further showed the advantages of declarative problem descriptions due to the inherent division of the algorithm to compute and implement a decision. Furthermore, declarative representations simplify the incorporation of exception rules, as shown by the RoboCup blocking example. We also explained advantages that hold similarly for propositional and first-order logic expressions. Afterwards, we gave an overview of the common practice of proposal computation algorithms, which mostly

rely on search algorithms that are specialized for a specific problem class to provide the necessary performance for real-world robotic systems.

In Section 6.4, we defined the problem class addressed by our decision process and highlighted its general undecidability, which is caused by the presence of nonlinear, continuous, and non-convex constraints. The main part of the chapter explained our solver, which combines three different solution techniques into a *satisfiability modulo theories* solver: First, a DPLL-based SAT solver, which computes solutions on the propositional level of the problem. Second, we use an interval propagation algorithm, which is able to shrink the solution interval and allows the identification of unit clauses. Third, the CNSMT solver incorporates a gradient ascent-based local search solver that is able to compute the solution for nonlinear expressions. The incorporation of incomplete solution techniques enhances the solution speed in contrast to state-of-the-art approaches. We demonstrate this in Chapter 10.

Section 6.5.5 extends our solver by the capability of solving optimization problems by completely discovering the propositional solution space and adding utility functions with a lower bound to the gradient optimization problem. Nevertheless, , the solver does not answer the question of how robots can ensure to have agreement on their computed solution proposals. In this context, we consider the solver solutions only as proposals other agents can accept or reject. However, the reasons of accepting or rejecting a proposal are strongly problem specific. In order to follow the idea of a general-purpose decision mechanism, we claim that a middleware that supports the proposal negotiation process and is adaptable to the typical robotic problem setting is required.

# 7 Proposal Replication for Value Determination

In the last decades of computer science in the research field of distributed systems, a wide variety of algorithms to achieve agreement for decision data among cooperating servers or agents were investigated. In Chapter 4, we highlighted drawbacks of these approaches when mobility and dynamic environments come into play, e. g., in emergency response, robotic soccer, or autonomous driving scenarios. Most notably, existing approaches are not flexible enough to allow adapting adequately to new problem settings. For example, strong consensus algorithms such as Paxos are not suitable for situations in which swift decisions are essential. However, domains with quickly changing environments such as robotic soccer face exactly this challenge.

In this chapter, we describe and analyze the PROViDE middleware, which implements the three negotiation steps of our decision process: proposal replication, conflict resolution, and value decision. The primary goal of PROViDE is the negotiation of conflicting decision proposals within a team of mobile robots to achieve coherent decisions. For the robotic domain, we can derive middleware-specific requirements for PROViDE:

**Efficiency** As stated before, many middleware frameworks are too heavy weight and inefficient for the needs of mobile robots. This requirement is twofold: First, the latency induced by the middleware has to be low, and second, the accessibility of the data should be robust against the absence of robots. Hence, we claim proposal replication to be a key requirement to comprehend decisions locally and quickly.

**Flexibility** The diversity of scenarios where today's robots are used requires a whole set of communication mechanisms that adjust to the requirements of the problem setting. Due to the diversity of different robotic domains, we claim that no single coordination protocol can cover all possible situations a robot can face. In particular, caused by the mobility and imperfect communication, each protocol implements a trade-off between safety, liveness, and decision time. As a result, the communication protocol and conflict handling mechanism require the flexibility to cope with the needs of the concrete problem setting. Furthermore, we need to address the transient communication behavior of robots, which can appear and disappear within communication range.

**Adaptivity** During runtime, the mobility of robots causes changes in the communication channel regarding throughput and connectivity. Consequently, our middleware needs to detect these changes and adapt its underlying protocols, i. e., detecting after which time the middleware assumes packet loss for missing acknowledgments.

**Extensibility** We expect future research activities to develop new communication protocols and data distribution frameworks, e. g., to tackle communication in domains that

current approaches do not consider. Thus, a software architecture should allow to plug in other protocols.

The rest of this chapter is structured as follows. To describe PROViDE and the underlying negotiation scheme, we give an overview of three decision steps in Section 7.1. Here, robots first replicate their value proposals among the team members. Thereby, value proposals are assigned to shared variables, which are described in Section 7.2. Afterwards, Section 7.3 describes the set of predefined but extensible distribution protocols to replicate value proposals to other robots. In a second decision step, conflicting proposals are negotiated, as described in Section 7.4. The last decision step implements the final decision, which results from the replicated proposals and produces eventually coherent values. Afterwards, we describe in Section 7.5 how the present team members are detected and how variables, which are inconsistent and relevant for current decisions, are identified. Section 7.6 analyzes additional considerations for robots acting in ad hoc networks.

## 7.1 Negotiation with PROViDE

PROViDE implements a database with replicated value proposals. Each proposal is assigned to exactly one decision variable, e. g., the position of a disaster victim in an emergency response scenario is assigned to the variable describing the goal for the next rescue operation. Similar to a linear human group decision process, we first collect all possible solutions and order them according to a quality criterion. Finally, we select the one solution by applying a second criterion such as a majority voting. As we assume that all agents rely on the same decision criteria, this may result in coherent values in a distributed setting. As shown in Figure 7.1, this negotiation process includes three steps: proposal distribution, conflict resolution, and value decision:

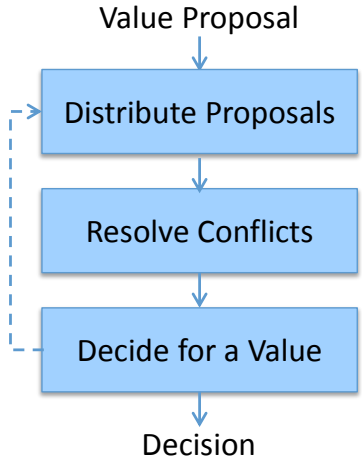



**Figure 7.1:** Negotiation scheme of PROViDE.

1. **Distribute Proposals** We distinguish between the chosen *decision value* and a solution *proposal* of each robot: Every robot can contribute at most one proposal for each

problem, which is replicated among the team, e. g., target blocking positions in a RoboCup scenario. The proposal value can change over time although this contradicts the safety property of consensus algorithms. Exactly one of the present proposals is selected as the decision value, which must not necessarily be the proposal of the local robot. If no valid local proposal is present, the empty value $\varnothing$ is chosen.

The *distribution method* for the replication process can be adapted to specify the consistency properties, i. e., it describes the effort to keep the proposal replica consistent. Thus, it specifies the trade-off between decision safety, liveness, and decision time. For example, in a RoboCup scenario we require rather fast decisions due to the dynamic environment, whereas autonomous car driving scenarios need a much higher level of safety.

2. **Conflict Resolution** Whenever a robot receives a new proposal, it checks for a conflict with its own current proposal. The robot adopts one of the present proposals as new own proposal, e. g., the most recent proposal or the one with the highest priority. According to distributed consensus literature [90], we call this selection *acceptance method*, since it decides on the acceptance of values as own proposal. We will show that the design of the communication protocol and the acceptance method may tend to eliminate unequal proposals. In conclusion, a suitable combination of both methods implements conflict resolution.

3. **Decide for a Value** In order to select a proposal as current variable value, a *decision method* is applied. Similar to human decision processes, this method does not necessarily rely on a democratic (majority-based) voting as used by distributed consensus algorithms. Instead, it can be useful to choose the most reasonable, most current, or only unanimously agreed upon values. As the decision method usually waits for the termination of the proposal distribution step, it strongly influences the speed of the whole decision process. In case of a conflict in the RoboCup domain, the robots could for example select the proposal of the robot with the highest confidence.

In contrast to human group decisions or classical consensus algorithms such as Paxos, we consider a feedback loop as optional. Moreover, it requires the manual implementation of a feedback rule, which we consider as exception in robotic domains. However, for this reason, we cannot generally provide guarantees for consensus on the final value. For the exceptional cases where feedback is required, it can be implemented trough the usage of *proposal change subscriptions*, e. g., to implement consensus algorithms. Here, multiple communication rounds implement the safe consent decision. These communication rounds follow two phases – prepare and accept – of the Paxos algorithm. We present an according implementation using PROViDE in Section 12.5. To ensure non-trivial decisions for PROViDE, the selected decision method is restricted to choose values that are supported by at least one proposal. The subscription mechanism also allows for a publish-subscribe communication mechanism and lifts PROViDE to a general-purpose communication middleware in addition to its negotiation capabilities. In the following sections, we describe the negotiation principles in detail. Thereby, we introduce a rule set, which is implemented by PROViDE and helps to show the properties of the negotiation process.

## 7.2 Shared Variables

Shared variables structure the distributed data space of proposals. Hence, a robot $n$ can address all decision variables $x \in X$ of a local database $DB(n)$. Each shared variable groups the replicated proposals $\mathrm{VP}_n(x)$ as basic data entities. These proposals are stored persistently for the lifetime of PROViDE even though the proposing agent disappears from the network. This property realizes persistence of the value decisions and follows the *persistence rule*. It also induces the problem of persistent conflicts if an agent is not able to change its own proposal replications, e. g., due to being outside the communication range.

We assume that proposal types are not restricted to scalars, but they can be vectors, lists of values, points, or other complex data types. In particular, a proposal value $V(\mathrm{VP}_n(x))$ can refer to all types of serializable objects. Thus, multiple value decisions can be made upon a single variable to unify their negotiation to a single decision process. Typical examples for proposal values are target movement positions, object identifiers for grasping operations, or sensor observations.

According to the *proposal update rule*, only agent $a$ is allowed to change or permit the change of $\mathrm{VP}_a(x)\ \forall\ x \in X$. Vice versa, locally replicated proposals of other robots only change if a *change command* has been received implicating the permission for change. Therefore, the following condition holds at any point in time:

$$T(\mathrm{VP}_n(x) \in DB(a)) \leq T(\mathrm{VP}_n(x) \in DB(n)) \quad \forall x \in X, n \mid n \neq a, \tag{7.1}$$

where $T(\mathrm{VP}_n(X) \in DB(m))$ denotes the decision time of the proposal of robot $n$ in the database of robot $m$. In other words, a replicated proposal of the robot $n$ is either the current proposal of $n$ or a proposal that he had before. The latter case always follows from delay or disturbances of the communication channel. In conclusion, the choice of weak proposal replication methods fosters outdated proposals, e. g., because lost messages are not detected and retransmitted.

A further property of proposals is the guarantee of monotonic changes, i. e., that the current proposal of on agent $a$ of an agent $b$ is always the most current proposal, which is sent by $b$ and received by $a$. It follows:

$$Lt(\mathrm{VP}_b(x) \in DB(a)) = \max(Lt_{\mathrm{Snt}}(\mathrm{Rcv}_a(\mathrm{VP}_b(x)))), \tag{7.2}$$

where $\mathrm{Rcv}_a$ denotes that the proposal has been received by agent $a$ and $Lt_{\mathrm{Snt}}$ the logical time referring to the moment the proposal has been sent.

Figure 7.2 shows an example of the data distribution for three robots in an emergency response scenario, where robot $a$ and $b$ operate as a team and robot $c$ operates alone (as a team). This separation usually happens because robot $c$ moved out of the communication range. An exemplary case for this is to rescue a disaster victim and transport it to a base camp. The figure shows that Robot $c$ is disconnected before the robots $a$ and $b$ proposed a new value for $x_1$ at time $l$. Therefore, the replica of robot $c$ is not updated and might be inconsistent. However, the robots can make their decisions based on the most current local replica. If the two teams can reestablish a connection, PROViDE will automatically restore the consistency of the proposals according to the distribution method.

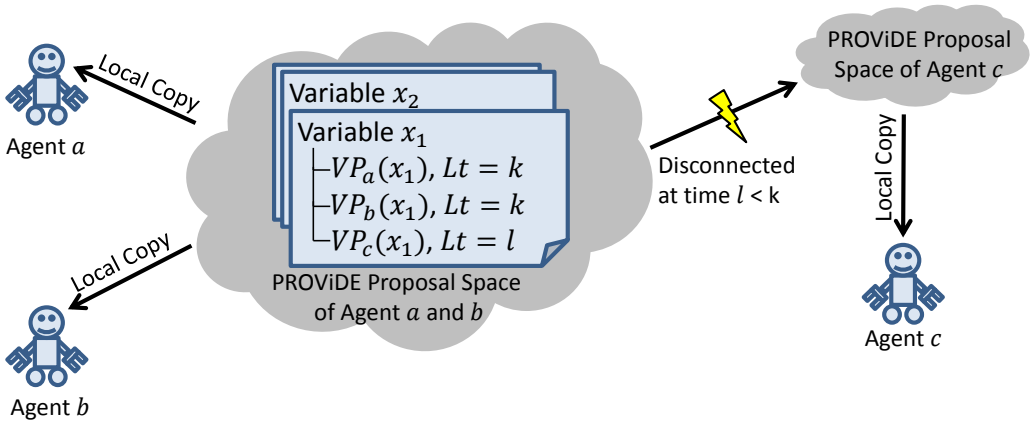



**Figure 7.2:** Example for the data distribution in the PROViDE proposal space with three agents.

### 7.2.1 Attributes of Replicated Proposals

Distributed value proposals in PROViDE contain the following attributes:

- scoped variable name,
- value,
- logical timestamp,
- decision time,
- validity time,
- distribution method,
- and acceptance method.

The name of a proposal matches the variable to which it belongs. A variable name includes a scope, which is structured like the UNIX file system, e. g., ”/mission/discovery/goalArea”. In this example, the variable ”goalArea” is located within the scope ”discovery”, which in turn is a sub-scope of ”mission”.

As explained before, each proposal has a logical timestamp, which we denote by $Lt(\mathrm{VP}_n(X))$. It allows to order proposals by the point of time at which their distribution has been initiated. In particular, the logical timestamp is required to ensure the condition of Equation 7.2. Therefore, each agent maintains a Lamport clock [92], which increases its time monotonically whenever a new proposal transmission appears. Furthermore, whenever a message with a higher timestamp is received, this higher value overwrites the local value.

Due to the monotonic progress of the logical clock, Lamport’s happened-before relation [92] holds for two messages $m_1$ and $m_2$ from the same agent:

$$Lt(m_1) < Lt(m_2) \Rightarrow m_1 \rightarrow m_2 \vee m_1 \mid m_2. \tag{7.3}$$

If an agent distributes two consecutive proposals and the first one is outpaced during the transmission, we can detect this event after both messages have been received. In this case, we can discard the message with the lower logical timestamp. Additionally, the list of logical timestamps within the proposals of each variable represents a special form of vector time [105], which allows the identification of causal relationships between variables. For example, if all logical timestamps of a variable $x$ are lower than the timestamps of the variable $y$, we can infer a causal dependency as $Lt(x) \rightarrow Lt(y)$. Similarly, we can check whether all agents acknowledged a new proposal. Here, all proposals of the other agents perform an update and therefore have a higher logical timestamp. The update mechanism is realized by the acceptance method and is described in Section 7.2.3. Note that a proposal update does not necessarily imply a value change, which makes a logical timestamp mandatory to detect causal relationships.

The proposal decision time $T(\mathrm{VP}_n(x))$ is the actual wall-clock time, which determines the *age* of the current proposal value. As a result, only the sending agent can assign this timestamp. The proposal decision time relies on a synchronous clock, which is available to all agents. We describe the synchronous clock implementation of PROViDE in Section 7.3.6.

The validity time $T_v(\mathrm{VP}_n(x))$ specifies an absolute time for the invalidation of a proposal. This timestamp again refers to the time provided by the synchronous clock. The validity time aims for time-limited decisions. For example, considering an agent in an emergency response scenario executing a victim rescue task, we can assume that it will leave the communication range of the other robots. Therefore, the assignment of the rescue task to the robot needs to be persistent, i. e., other agents trust that the agent commits to the task and executes it successfully although communication is currently impossible. As this method prohibits the detection of an agent fault, an occuring fault would prevent the victim from being rescued. The validity time allows the application developer to specify the required time for the intended rescue mission. In case the robot is not able to rescue the victim within that time, the decision becomes invalid and another agent can take over the task.

Additionally, proposals have two attributes referring to the distribution and acceptance method. These determine how the proposal is distributed, respectively, provide a guideline for other agents when a proposal is accepted. We assume these values to be static for each variable. For example, considering autonomous cars that decide which car is allowed to pass a crossing first, we can assume that the same distribution and acceptance rules hold for all instances of the decision.

### 7.2.2 Distribution Method

The distribution method describes the protocol for keeping replicated proposals consistent. It adjusts the required bandwidth, consistency guarantee, availability, and fault tolerance adapted to the decision issue. For instance, robotic soccer requires swift decisions as the environment changes quickly due to the dynamics of the moving objects such as the ball or other robots. Moreover, the domain does not require high safety constraints because no direct human interaction takes place. Consequently, the domain is inherently fault tolerant to a certain extent. In contrast, in an emergency response scenario, uncoordinated decisions can result in deaths of undiscovered or not rescued victims. At the same time, the

usually weak communication infrastructure does not allow continuous negotiation once a robot leaves the communication range. Thus, sophisticated decisions need to be made, which implies that all conflicts have to be resolved before a robot leaves the communication range. A more detailed discussion concerning the necessity of a problem-specific trade-off of consistency properties can be found in Terry [170]

PROViDE proposes four distribution methods (also referred to as consistency levels) $L(x)$, which can be chosen for each variable individually:

**Local Variable Management (Level 0)** indicates no replication of newly added proposals. Here, proposals are not replicated automatically to other robots, which allows an instantaneous execution of a decision. The standard use cases for this method are individual decisions that have no relevance to other agents. Nevertheless, proposals with local variable management can be set and queried by other agents, to either actively request the intention of the robot or for decisions that require guaranteed consistency for a specific point in time.

At the same time, the lack of replication leads to a poor accessibility for other robots due to the communication delay. Furthermore, access is coupled with the network connection of the proposing agent, i. e., the proposals disappear when the agent loses its connection. Vice versa, new proposals might appear in the proposal space when an agent engages the network. This behavior is similar to distributed tuple spaces such as LIME [111], which explicitly consider the mobility of agents.

**Fire and Forget (Level 1)** realizes eventual consistency, i. e., proposal consistency is achieved among the robots with a probability greater zero. This method follows the idea of optimistic replication [147] for situations in which replication success is not mandatory. Thus, this method is beneficial for the robotic soccer scenario where proposals have a short validity time or receive updates with a high frequency in relation to the communication delay. Performance is the major advantage of the *fire and forget* replication method, which does not correlate with the number of connected agents. However, the lack of an explicit feedback that indicates, whether the proposal replication is successful is a drawback. As a consequence, fire and forget cannot be used when such knowledge is required, e. g., to set up a mutex variable (unless a packet loss rate of 0 % can be assumed, or the domain provides fault tolerance, see Section 12.2).

**Monotonic Updates (Level 2)** ensures that a proposal change is successfully replicated to all other robots. At the same time, a proposal-distributing robot receives an acknowledgment of the other agents including their own proposal. This allows proposing agents to reason about remote knowledge bases, e. g., whether all agents agree on a certain proposal. Variables that are distributed with the monotonic updates method have a *monotonic reads* distribution (sometimes referred to as session guarantee) for proposal updates. This means robots receive a subset of all proposal writes in a monotonic order. Hence, two consecutive read operations of the replicated data result in either the same or a more recent proposal for the second read operation. The replicated proposals become more up to date over time. As a consequence, if a robot cannot receive a proposal due to not being able to communicate, it has to receive all necessary updates once it re-engages the team communication process.

We consider monotonic updates as the standard method for proposals that are not changing iteratively but require negotiation. As an example, we assume an autonomous car that distributes a proposal indicating it wants to pass an intersection. When the car waits until all other nearby cars confirm the proposal, the intersection can be passed safely. Otherwise, it will receive a concurrent proposal, which allows to either change the own intention or to wait for an intention change of the concurrent traffic.

**Monotonic Accepts (Level 3)** is the logical extension of monotonic updates to all proposals. While the monotonic updates method only allows consistency for proactive decisions of agents, monotonic accepts also ensures consistency of proposals that are created due to the selected acceptance method. This method allows for an eventual establishment of a joint intention, where every robot is aware of the fact that all other robots know all proposals. Here again, the probability for the establishment of a joint intention increases over time. The major drawback of the monotonic accepts method is its usually high demand for bandwidth implying a bad scalability with the number of agents.

In contrast to consistency level 0, which does never automatically distribute proposals, the consistency levels one to three only transmit messages if a proposal change occurs. However, in some domains, it is typical to realize a mutual belief by iteratively transmitting proposals. For example, in the middle size league, it is common to transmit the world model information about detected ball and robot positions with a certain update frequency as for example $10\,\text{Hz}$. This has two main reasons: First, due to the continuous environment of the robots and the sensor noise, position values change almost always when new sensor readings appear, i. e., values are likely to change frequently. And second, the chosen transmission frequency is usually lower than the frequency of the sensor readings, which reduces the required bandwidth [13]. In PROViDE, we similarly allow limiting the updates of proposals to achieve such a behavior for frequently changing variables.

### 7.2.3 Acceptance Method

As mentioned before, each PROViDE agent supports at most one proposal for each variable. This proposal is by default the *empty proposal*. The *acceptance method* decides about the support of proposals from other agents, i. e., this method can *accept* a new proposal by replacing the former selection with the new one. In the same way, a new proposal can be rejected. To ensure that an agent transmits acknowledgments including the proposal it supports, the acceptance method applies whenever a change in the proposal database occurs (*acceptance rule*). Note that new replicated proposals are always added to the proposal database although they are not accepted. In this case, the own proposal remains unchanged.

In order to specify the condition under which a proposal is accepted, the developer can select a condition $\circ$ for an ordering relation $<_o$ for all proposals that belong to the same variable:

$$<_o = \{(\mathrm{VP}_i(x), \mathrm{VP}_j(x)) \in |\mathrm{VP}(x)| \times |\mathrm{VP}(x)| \ : \ \mathrm{VP}_i(x) \circ \mathrm{VP}_j(x) \wedge i \neq j\}\,, \tag{7.4}$$

where $|\mathrm{VP}(x)|$ denotes the set of proposals for $x$. The acceptance function $P$ sorts a set of proposals for variable $x$ by $<_o$ to select the highest ordered proposal for the local agent $a$:

$$\mathrm{VP}_a(x) = P(\mathrm{VP}_a(x), \mathrm{VP}_b(x), ..., \mathrm{VP}_n(x)) \tag{7.5}$$

$$= \max(\mathrm{VP}(x) \in <_o). \tag{7.6}$$

As $P$ is executed independently on all agents, we require $<_o$ to be a strict ordering relation if equal acceptance decisions are required, i. e., $\forall i, j(\mathrm{VP}_i(x), \mathrm{VP}_j(x)) \in <_o$ or $(\mathrm{VP}_j(x), \mathrm{VP}_i(x)) \in <_o$. In other words, two proposals are never equal, which ensures that $P$ returns the same result for the same set of replicated proposals on each agent. To facilitate the construction of strict ordering relations, we assume each agent to have a unique identifier such as an IP address. The incorporation of this identifier as ordering criterion allows the transformation of non-strict ordering relations into strict ordering relations. Here, proposals that cannot be ordered by the main criteria are ordered by the identifier value.

In some cases, the ordering relation considers the confidence of the distributing agent. Therefore, the proposals have to include confidence values as an additional payload. Hence, this type of ordering relations is defined in a problem-specific way on the application level. PROViDE implements a set of standard acceptance methods that suffice for the typical robotic applications:

**Happened-After** selects the proposal with the highest logical timestamp $Lt(\mathrm{VP}(x))$. If the timestamps of two proposals are equal, the proposal from the robot with the lower identifier is selected. We define this relation as:

$$<_{Lt} = \{(\mathrm{VP}_i(x), \mathrm{VP}_j(x)) \mid Lt(\mathrm{VP}_i(x)) < Lt(\mathrm{VP}_j(x)) \vee (Lt(\mathrm{VP}_i(x)) = Lt(\mathrm{VP}_j(x)) \wedge i < j)\}. \tag{7.7}$$

We consider this as standard selection method, which selects the causally newest proposal. This proposal is not necessarily the most recent proposal with respect to the wall-clock time. However, the stable behavior makes it to a suitable choice without the presence of other domain knowledge.

**Most-Recent** behaves similar to *happened-after* but relies on the proposal decision time:

$$<_T = \{(\mathrm{VP}_i(x), \mathrm{VP}_j(x)) \mid T(\mathrm{VP}_i(x)) < T(\mathrm{VP}_j(x)) \vee (T(\mathrm{VP}_i(x)) = T(\mathrm{VP}_j(x)) \wedge i < j)\}. \tag{7.8}$$

The main drawback of this method is that no common global clock can be assumed. Although PROViDE implements clock synchronization internally, a synchronization error with the magnitude of the network delay might still be present. As a result, this method is useful for weakly synchronized distributed shared memory implementations, e. g., to negotiate target passing positions in RoboCup.

**Priority** relies on the priority $\mathrm{pri}$ of all agents defined at compile time:

$$<_{\mathrm{pri}} = \{(\mathrm{VP}_i(x), \mathrm{VP}_j(x)) \mid \mathrm{pri}(i) < \mathrm{pri}(j)\}, \tag{7.9}$$

where $\forall i, j\ \mathrm{pri}(i) \neq \mathrm{pri}(j)$. The priority usually relies on the task of the agent, e. g., in hierarchical agent organizations or to favor the proposals of agents with the best observation capabilities.

**Collection** is the simplest form of acceptance methods. Here, the local agent $a$ never accepts new proposals other than its own:

$$<_c = \{(\mathrm{VP}_i(x), \mathrm{VP}_j(x)) \mid j = a\}. \tag{7.10}$$

Consequently, all remote proposals are collected forming a list. The typical application example is the realization of a distributed election: Each agent proposes a value that indicates its appropriateness to become the leader. If a single maximum proposal exists, the proposing agent becomes the leader. Otherwise, the procedure repeats with a new variable and with all winning agents until only a single agent remains. Additionally, *collection* is also the method of choice to realize a majority-based decision.

The acceptance method applies independently of the selected distribution method. Moreover, if the distribution method demands feedback from the receiving agents as for example in consistency level two and three, the feedback includes the selected proposal. This eventually allows reaching homogeneous proposals, i. e., all agents propose the same value. However, this property depends on the selected distribution method, acceptance method, and the amount of network failures. Overall, the acceptance method contributes to the conflict discovery and resolution. We examine this topic in Section 7.4 in detail.

### 7.2.4 Value Decision

After applying the proposal replication and the partial resolution of proposal conflicts due to the acceptance method, PROViDE applies a *decision method* on the remaining proposals. If no proposal satisfies the decision condition, the value decision procedure returns an empty proposal indicating that the decision process is incomplete. PROViDE delivers four built-in decision methods:

**Majority Voting** makes a fault-tolerant decision by choosing the proposal that is accepted by a majority of agents. Here, we determine the majority among the robots that are currently within communication range. If no proposal is present for a certain agent, we assume that the agent does not support any proposal. As a consequence, a majority can only exist if more than 50 % of the agents successfully distributed their proposals. Since the majority has to support at most one value, this method provides a weak form of fault tolerance. Due to the presence of network errors and delay, consensus cannot be assumed [93]. This approach creates a special form of an *early stopping mechanism* for *eventual consensus* according to Cheng and Tsai [37]. In particular, the probability to converge to a consent value increases with a higher consistency level and with the time passed since the last value has been proposed. However, this property comes at the price of an increased decision delay, as packet loss and network delay impose an upper bound for the possible decision time.

**Unanimous Agreement** only returns a value decision if all agents agree upon the same proposal value. Although a majority voting already provides some level of fault tolerance among all connected agents, some applications require all agents to conclude the same value. For example, if mutual exclusion is required, not a single unauthorized agent may enter the critical section. This additionally requires

consistency level 3. As a result, unanimous agreement is the slowest decision method, since a positive response to the proposals of all other agents is required. Hence, a single agent with an interrupted connection can block the decision process.

**Weak Agreement** is similar to unanimous agreement except for the fact that only agents within communication range need to agree on the value proposal. This allows for decision-making in scenarios where some agents regularly act outside of the communication range of the others. An application example is an autonomous car that proposes its future trajectory through a crossroad. In that case, the car may not execute the trajectory until all nearby cars confirm not to interfere. If only one single car proposes a different trajectory, the proposal is invalid. Note that this method bears the risk that a re-engaging agent breaks the unity. A new negotiation process should either be initiated when agents appear or a consistency level of 2 or higher is required, which ensures that each contradicting proposal will be replicated.

**Own Proposal** always returns the own proposal as decision value. This allows for fast decisions but provides no fault-tolerance capabilities. The main assumption of this decision method is that the acceptance method resolves all conflicts over time. Thus, the decision converges to a common value, e. g., to the proposal value with the highest priority when choosing the priority acceptance method. In the same fashion, selecting the most recent acceptance method leads to the value decision with the most recent decision time.

Enabling quick decisions is the main purpose of this decision method. They result from the fact that a newly proposed value directly leads to a change of the final variable value. However, as the value is chosen before the negotiation process of the acceptance and distribution method is completed, the resulting value choices may be unstable or contradicting to the proposals of the other agents. For these reasons, this decision method is ideal for domains such as robotic soccer where decision safety plays a minor role.

As the decision method is application specific, PROViDE allows the manual implementation of new decision methods. Here, the decisions do not necessarily rely on the amount of supporting agents. In particular, new decision methods may rely on payload knowledge such as confidence values.

### 7.2.5 Interference between Consistency, Acceptance, and Decision Method

From the implementation point of view the distribution, acceptance, and decision method can be chosen independently of each other. However, some combinations do not provide additional value to the decision process. For example, local variable management makes an acceptance and decision method obsolete, as no values of other agents can appear in the proposal storage.

The fire and forget method can generally be combined with all provided acceptance methods and still produce a meaningful result. Yet, a decision method that requires more than two proposals to be present does only make sense if we assume that multiple agents propose a value over time. Meaning, if we assume that only a single agent makes proposals,

the majority or unanimous agreement method cannot infer a decision value. Additionally, unanimous agreement is unlikely to be successful unless the proposed values change only rarely. The only reason to transmit the same proposal value repeatedly is to overcome a possible packet loss. However, this scenario can be tackled by using monotonic updates more efficiently.

The monotonic updates and the monotonic accepts distribution method force all agents to send their own proposals at least once. Therefore, we can assume that all decision methods may be able to determine a value. For collection-based decisions, we recommend the usage of monotonic updates, which provides the same guarantees as monotonic accepts but requires less bandwidth. The compatibility of the provided methods is summarized in Table 7.1.

| | Happened-After | Most-Recent | Priority | Collection |
|---|---|---|---|---|
| $L(x) = 0$ | Own Proposal | Own Proposal | Own Proposal | Own Proposal |
| $L(x) = 1$ | Majority* | Majority* | Majority* | Majority* |
| | Own Proposal | Own Proposal | Own Proposal | |
| $L(x) = 2$ | All | All | All | Majority |
| | | | | Unanimous / |
| | | | | Weak Agreement |
| $L(x) = 3$ | All | All | All | None |

**Table 7.1:** Compatibility chart for combinations of consistency, acceptance, and decision method. Each cell includes all recommended decision method for a given acceptance and distribution method. The * indicates compatibility for the case where all agents send frequent proposal updates.

## 7.3 Proposal Replication

The communication protocols of PROViDE implement the replication and distribution mechanisms. Additionally, they invoke the acceptance method if necessary. In the following sections, we first describe the assumptions about the network properties with a network model. Subsequently, we derive communication protocols for the four presented distribution methods. In this context, we have to detect packet loss due to network failures. Therefore, we assume a timeout for pending acknowledgments, which we drive from a probability model. The same model controls the probability estimation for the consistency of the replicated proposals. Afterwards, we explain the clock synchronization mechanism implemented in PROViDE. Finally, we describe variable change subscriptions that enable feedback-based decisions and message passing.

### 7.3.1 Communication Model

The communication model for the network protocols can be derived from the general requirements of the decision process described in Section 1.4. Since we designed PROViDE

for the usage in multi-agent systems, we assume the presence of a low-level communication infrastructure, which realizes point-to-point communication. Hence, the link, internetwork, and transport layer follow an asynchronous communication model, e. g., through the realization with UDP.

Asynchronous communication does not provide any knowledge about whether a transmission is successful. Because robots could act in environments with harmed communication infrastructure, we have to assume faulty network communication, i. e., network transmission can get lost or corrupted. The latter case is averted by checksums at a lower communication layer. As a result, corrupted transmissions can be discarded and appear as additional lost transmissions to the higher communication layers. Additionally, we assume that no byzantine faults [93] occur.

A further property of asynchronous communication is the lack of ordered data control. Generally, transmissions can appear in arbitrary order at the receiver, as they can outpace each other. Furthermore, according to the results of Zheng et.al. [181] and Daniel et.al. [51] for UDP and other *packet-based networks*, we assume the network delay to follow a Laplace distribution:

$$P(D = t|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|t - \mu|}{b}\right), \tag{7.11}$$

where $P(D = t|\mu, b)$ is the probability of the network delay $t$ assuming a mean $\mu$ with a diversity $b$. Additionally, we assume symmetric communication links, i. e., that the network delay and packet loss rates between two agents $a$ and $b$ are independent from the direction of the transmission. This simplification is only relevant for the mathematical models that underly the packet loss detection mechanism. This mechanism optimizes the transmission time within unreliable networks. However, it does not affect other properties of the protocols. The same holds for the simplifying assumption that the probability for lost network packets $P(L)$ is uniformly distributed. This contradicts practical observations in usual computer networks where packet loss usually occurs in bursts [157, p. 135]. This implies a Poisson distribution, which models the length of the bursts. However, since no appropriate values for the Poisson parameters can be estimated during normal operation and no evidence for the occurrence of a burst exists, a uniform distribution is a reasonable simplification.

The communication model further assumes that all agents have access to a logical unicast and broadcast service. Meaning, they can transmit a packet to a single robot within the same group of connected agents or to all agents in this group. If a robot is connected to the group of agents $p$, we assume that either a direct peer-to-peer connection to all other agents or a routing protocol, which implements a logical peer-to-peer connection, exists. If $p$ engages another group $q$, a new joint group $\{p, q\}$ is formed. Since no agent within a group has a distinct role, we also call a connected group of agents *team*. In order to facilitate routing mechanisms, PROViDE can tolerate the duplication of network transmissions. This allows the usage of flooding mechanisms, which cannot avoid that network packets arrive multiple times.

### 7.3.2 Protocol

The communication protocols of PROViDE replicate value proposals and transmit acknowledgments, which give feedback about successful replication. Accordingly, the protocols transmit two types of network messages: commands and acknowledgments. Commands are initiated by a new proposal that has actively been assigned at application level and needs replication. PROViDE transmits all command messages via broadcast to all other agents within communication range. Acknowledgments can be divided into the two subgroups of standard *acknowledgments* and *short acknowledgments*. While *short acknowledgments* only confirm the message arrival at a receiver, standard *acknowledgments* additionally include a proposal of the receiver, which indicates whether the proposal has been accepted.

When a robot receives a command message, it adds the new proposal to its data store, i. e., if a proposal of the transmitting robot for the given variable is already present, this proposal is overwritten accordingly. Afterwards, the ordering relation of the acceptance method could be violated. Therefore, PROViDE invokes the acceptance method to update the own supported proposal. Since the locally supported proposal has been consistent before, the invocation can only have two possible effects: Either the own proposal remains unchanged or the new proposal is accepted.

All transmitted messages include a logical timestamp, as suggested by Lamport [91]. This logical timestamp is stored in the proposal payload. PROViDE uses this timestamp to perform a message consistency and duplication check. Therefore, PROViDE checks whether the related proposal is already present in the data store by comparing the logical timestamp of the present proposal with that of the received one. If both are equal, the network infrastructure or the underlying communication protocol duplicated the message. If the new proposal has a lower timestamp, we can assume that a more recent message has outpaced the received message. In neither case causes the message changes of the data store.

The fire and forget distribution method terminates after the execution of the acceptance method. In contrast, monotonic updates and monotonic accepts confirm the reception of command messages by broadcasting an acknowledgment, which realizes a two-way handshake protocol ($L_2$ *notification rule*). All agents that receive the acknowledgment also process it, i. e., the included proposal is added and updated in the local proposal space.

If the initiator of the proposal command message does not receive a response after a time $t_r$, it assumes packet loss of either the command message or the acknowledgment. The value of $t_r$ is dynamically adjusted based on the current delay estimation and an a priori probability $P(D_e < t_r)$ that encodes the threshold of packet loss. The underlying probability model is described later in Section 7.3.3.

Lost packets are compensated by the initiator, who resends the original message. In order to distinguish a message duplication through the routing process from a retransmitted message, all messages contain a unique identifier. If the receiving agents detect the same logical timestamp on a message with a different identifier, its local proposal is already up to date but the acknowledgment transmission has been lost. In that case, the local proposal remains unchanged and the acknowledgment is resent.

After the acknowledgment arrives at the initiator, it updates the received proposal in its local proposal data store. Once all agents in the current team acknowledged the commanded message to the initiator $a$, the monotonic updates distribution method is completed and:

$$Lt(\mathrm{VP}_a(x)) >_o Lt(\mathrm{VP}_n(x)) \quad \forall n \in \mathcal{A} \wedge n \neq a \tag{7.12}$$

holde, where $\mathcal{A}$ refers to the currently connected group of agents. This forms a necessary condition for consistency. By additionally checking the reception of acknowledgments from all agents, the initiator can check the consistency of its most recent distributed proposal.

Due to the iterative resending of lost packets, the monotonic updates method eventually converges to a consistent state for all proposals that are distributed by command messages. However, the acknowledgments are only guaranteed to arrive at the initiator. Meaning, acknowledgments are not guaranteed to be received by other agents. Therefore, the monotonic accepts method expects an additional short acknowledgment to check for a packet loss among the receiving agents. This realizes a three-way handshake, which is a common method to establish a joint intention ($L_3$ *notification rule*).

Short acknowledgments follow the same logic as regular acknowledgments: They confirm the reception of an regular acknowledgment. Therefore, they confirm the logical timestamp and unique ID of the original message. We can identify two differences: First, short acknowledgments do not include a proposal, and second, they do not cause changes in the proposal data store, which makes them inherently tolerant towards message duplications. The main reason for the usage of short acknowledgments is to save bandwidth, as the broadcast mechanism of the previous protocol steps already ensures that all agents receive the proposals of all other agents. An example for the PROViDE communication protocols with two agents is shown in Figure 7.3.

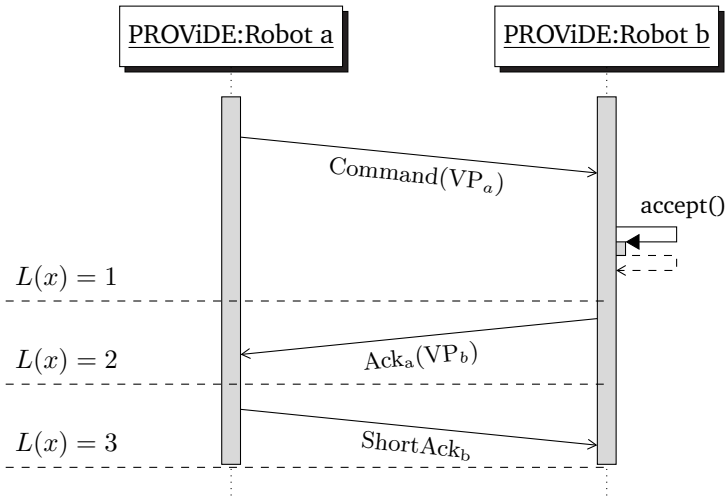



**Figure 7.3:** PROViDE distribution protocol for exchanging data using the *monotonic accepts* distribution method. The termination for weaker distribution methods is indicated with the dashed lines.

The communication protocols provide a *fail* termination criterion: Message retransmissions are limited to a threshold $c_r$. If more than $c_r$ transmissions failed, the destination agent is excluded from the current team estimations. This criterion limits the communication bandwidth of disappearing agents to an upper bound.

The resend mechanism of our protocol is related to the idea behind the transmission control protocol (TCP). The major difference is the explicit support of mobility through loosely coupled communication links. For this reason, we avoid session management for individual transactions. The only session management overhead is induced by the team detection mechanism, which requires periodically transmitted *heartbeat* messages. The team detection is explained in Section 7.5 in detail.

### 7.3.3 Resend Time

As mentioned before, packet loss is detected by the absence of expected acknowledgments for a time greater than $t_r$. By using the assumptions of our communication model, we can determine $t_r$ dynamically at runtime by estimating the parameters of the probability-based network model. As a result, the protocol can adapt to the present network delay to increase the distribution efficiency in the presence of packet loss. Therefore, we first estimate the probability of receiving an acknowledgment for the proposal of robot $a$ for a given time interval $t$, which denotes the timespan between the current point in time and the message transmission. As the probabilities of consistency for all value proposals is independent:

$$P(C) = \prod^{\mathrm{VP}} P(A_{\mathrm{VP}_i}) \tag{7.13}$$

holds, where $P(C)$ is the probability for consistency of the whole replicated data store and $P(A_{\mathrm{VP}_i})$ the probability for consistency of a single proposal $\mathrm{VP}_i$. To compute $t_r$, we are specifically interested in $P(A_{\mathrm{VP}_a})$. This probability depends on the packet loss and delay and is determined independently for each communication link. We assume the network delay to follow a Laplace distribution. Hence, we can use a maximum likelihood estimation for the diversity $b$ and mean $\mu$. PROViDE measures the delay for the last $N$ heartbeat transmissions and computes the mean with $\mu = \sum_{i=1}^{N} \frac{\delta t_i}{N}$ and the diversity with:

$$b = \frac{1}{N} \sum_{i=1}^{N} |\delta t_i - \mu|, \tag{7.14}$$

with $\delta t$ denoting a delay measurement. Assuming that no packet loss occurs, we can compute the cumulative distribution to determine the probability whether a packet arrived after a transmission time $t$:

$$P(D_s \leq t|\bar{L}) = \int_{-\infty}^{t} P(D_s|\mu, b), \tag{7.15}$$

where $P(D_s \leq t|\bar{L})$ is the probability for a delay of at least $t$ for a transmission under the assumption that no packet loss ($\bar{L}$) occurred. With the same scheme, we can compute the expected delay $D_e$ for the time until we receive an acknowledgment as:

$$D_e = D_s + D_a + C_p, \tag{7.16}$$

where $C_p$ denotes a processing offset that is required by the receiving robot to process the command message and initiate the acknowledgment transmission, and $D_a$ is the probability variable that describes the time to transmit the acknowledgment. As we assume synchronous communication delays for each link, $D_s = D_a$ holds. To compute the sum of the probability variables, we have to determine the convolution of their probability distributions:

$$P(D_e|\mu, b) = P(D_s|\mu, b) * P(D_s|\mu, b) * P(C_p) \tag{7.17}$$

$$= \frac{1}{4b}(1 + \frac{|t - \Delta|}{b}) \exp\left(\frac{-|t - \Delta|}{b}\right), \tag{7.18}$$

where $\Delta$ is the offset with $\Delta = 2\mu + c_p$, and $*$ is the convolution operator. A proof for this result can be found in Appendix A. The cumulative distribution of Equation 7.17 describes the probability to receive the acknowledgment until a point in time $t_r$ after sending the initial command message:

$$P(D_e \leq t_r|\mu, b) \tag{7.19}$$

$$= \int_{-\infty}^{t_r} P(D_e|\mu, b) \tag{7.20}$$

$$= \begin{cases} \frac{1}{4}e^{\frac{t_r - \Delta}{b}}\left(2 - \frac{t_r - \Delta}{b}\right) & \text{if } t_r \leq 2\mu \\ \frac{1}{4}e^{-\frac{t_r - \Delta}{b}}(-2 - \frac{t_r - \Delta}{b}) + 1 & \text{otherwise} \end{cases} \tag{7.21}$$

Hence, we can solve this equation for $t_r$:

$$t_r = \begin{cases} \Delta + bW_{-1}\left(\frac{-4P(D_e \leq t_r)}{e^2}\right) + 2b & \text{if } P(D_e \leq t_r) \leq 0.5 \\ \Delta - bW_{-1}\left(\frac{4(P(D_e \leq t_r) - 1)}{e^2}\right) + 2b & \text{otherwise} \end{cases}, \tag{7.22}$$

with $W_{-1}(x)$ being the *Lambert W function*, which computes the inverse function of $f(w) = we^w$. Due to this result, we can specify an a priori probability for $P(D_e \leq t_r)$, which determines the certainty that a packet should have arrived if no packet loss occurs. The influence of $P(D_e \leq t_r)$ on the resend time is exemplary depicted in Figure 7.4.

The value $t_r$ can be seen as a *congestion window* similar to the way in which it is used by TCP. The major advantage in contrast to TCP is the fact that this window size changes dynamically based on online delay samples. As a consequence, our protocol is able to adapt to unexpected communication delays.

### 7.3.4 Probability of Consistency

The same model from Section 7.3.3 allows the estimation of the probability whether consistency for a new proposal is given. This enables applications to achieve faster or more simultaneous decisions, as distributing agents do not need to wait for acknowledgment messages to presume consistency. Here, the application assumes consistency after the probability exceeds a certain threshold. For example, a robot in RoboCup with the task to perform a pass can execute its pass kick if the probability for a consistent replication of the destination point exceeds 75 %. In that case, the passing robot does not need to wait for

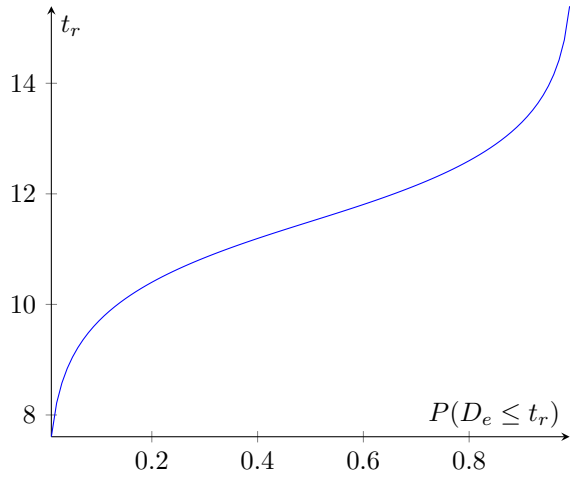



**Figure 7.4:** Resend time $t_r$ for a divergence $b = 0.75$ and a mean network delay of $\mu = 5$ ms relative to the choice of $P(D_e \leq t_r)$.

the reception of an acknowledgment of the pass receiver despite the presence of packet loss.

Besides the transmission delay, the probability of consistency also has to consider the probability of packet loss. Assuming that packet loss is distributed uniformly, a simple estimate of the probability for packet loss to a robot $b$ counts lost and transmitted packets and computes $P(L_b) = 1 - \frac{||r||}{||s||}$, where $||s||$ is the number of expected packets and $||r||$ the number of received acknowledgments by $b$. This estimation requires constant packet loss rates. Since robots can act in inhomogeneous environments and their communication range is limited, we rely on an estimation that attaches more importance to more recently lost packets. More precisely, we assume an exponentially decreasing relevance of lost packets:

$$w(t_p) = e^{-\lambda t_p}, \tag{7.23}$$

with $\lambda$ adjusting the decreasing rate to the application domain, and $t_p$ describing the time that has passed since a packet was expected. As a result, we estimate the probability of packet loss as:

$$P(L_b) = 1 - \frac{\sum_r w(t_r)}{\sum_s w(t_s)}, \tag{7.24}$$

where $t_r$ and $t_s$ denote the time that has passed since an occured message-receiving event $r$ and an expectected message-receiving event $s$, respectively. The overall probability of a consistent value proposal at robot $b$ and at time $t$ after sending a proposal is

$$P(D \leq t, \bar{L}) = P(D_s \leq t|\bar{L})(1 - P(L)) \tag{7.25}$$

The resulting probability of packet loss for a single variable proposal to $b$ is

$$P(A_{\mathrm{VP}_b}) = P(D_s \leq t_m|\bar{L})(1 - P(L)) \sum_{i=0}^{m} P(L)^i \tag{7.26}$$

with $m$ representing the number of resent packets computed by $\lfloor \frac{t}{t_r} \rfloor$ and $t_i = t - t_r i$ the time that has passed since the $i$th packet was sent. For fire and forget consistency, $m$ is limited to $1$, since no messages are resent. Note that $t$ does only refer to the time where both robots are part of the same team. The resulting probability is depicted in Figure 7.5. Naturally, the arrival of an acknowledgment ensures that a distributed proposal has successfully been transmitted. Thus, the probability for consistency increases to $1.0$.

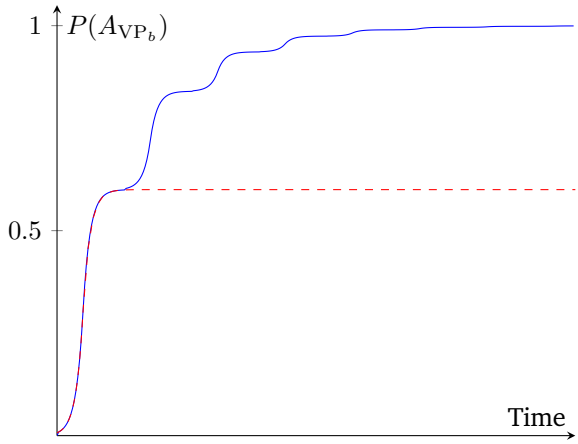



**Figure 7.5:** The blue graph shows the probability distribution for the consistency of a single proposal $P(A_{\mathrm{VP}_b})$ assuming an estimated 40 % packet loss and $P(D_e \leq t_r | \mu, b) = 0.9$. The red curve shows the probability distribution without resending lost packets.

From Equation 7.26 follows

$$\lim_{t \to \infty} P(A_{\mathrm{VP}_b}) = 1 \tag{7.27}$$

for $P(L) < 1$, since $\lim_{t \to \infty} P(D_s \leq t_m | \bar{L}) = 1$ and $m \to \infty$ for $t \to \infty$ if at least one distribution method of at least consistency level 2 is selected. We can conclude that the provided protocol converges to a consistent replication state over time.

### 7.3.5 Remote Proposal Access

As already mentioned in Section 7.2.2, agents can remotely access proposals of other agents. The main intention of this feature is the manipulation of proposals that use local variable management. Since no replication is incorporated, PROViDE enables strong consistency through a central component. For example, in an emergency response scenario, a robot might execute a transportation task to bring a victim to a first aid center. Since the robot can execute this task without the help of others, it does not communicate its goal and stores the goal data locally without automatic distribution. A second robot can request the goal information of the incoming robot to prepare the first aid measures. It can also set a new destination point if the targeted first aid center does not have the capacity to treat an additional patient.

The remote proposal access mechanism imitates the approach of the LIME framework [111], where each agent maintains an own data space. In contrast to classical

tuple spaces such as Linda [11], PROViDE and LIME explicitly tolerate the mobility of agents, i. e., the agents within communication range, form a federated space of data. Vice versa, if they leave their communication range the data space decomposes into several smaller data spaces according to the location of the data. The major difference to LIME is the data addressing: Here, proposals of PROViDE must be addressed with the variable name and the robot identifier. In contrast, LIME discovers all variables in the federated tuple space and does not require a robot identifier. As a result, the union of tuple spaces can cause conflicts.

To access a proposal remotely, an agent can sent read and write requests. Despite the usage of unicast instead of broadcast, PROViDE transmits remote access requests with the same protocol as the monotonic commands distribution method. Therefore, the read and write request, respectively, is transmitted with unicast to the destination agent, which applies the requested operation. Afterwards, the acceptance method is invoked to avoid violations of the ordering relation. The process concludes by the transmission of an acknowledgment message, which includes the final proposal value.

The remote proposal access mechanism is able to deal with lost network packets or duplication. To overcome the latter issue, we again rely on a logical timestamp, which is part of each transmission. By storing the most recent known logical timestamp of each agent, we identify duplicates by having a smaller timestamp than the stored value. This is especially important for multiple consecutive write operations, where the duplicates lead to a change of the operation order.

The requesting agent detects lost network packets similarly to the distribution protocol: If no response is received within the threshold $t_r$, an error is assumed. Hence, the detection depends on the estimation of the network delay, as described in Section 7.3.3. In case of the detection of packet loss, the message is retransmitted. This procedure repeats until an acknowledgment is received. If a maximum count of retransmissions is exceeded, the access operation has failed and returns an error. PROViDE executes access operations by default in non-blocking mode. Thus, the application developer is not required to spawn threads manually. Every response of an agent is automatically stored in the local proposal data storage. This newly created replicate is not automatically kept consistent by PROViDE. The presence of the replicate indicates the success of the access operation.

### 7.3.6 Clock Synchronization

In PROViDE, we require synchronous clocks for three purposes: Firstly, they are needed for the determination of the proposal decision time, which identifies the most recent value proposal, for example within the most-recent acceptance method. Secondly, the validity check of proposals is based on the validity time property of proposals. To enable simultaneous invalidation, this process also relies on the synchronized clock. Thirdly, the synchronous time serves as an external service on application level. Thus, decisions that rely on this time can achieve a tight synchronization of agent actions. As example, we can consider the lifting of a rescue stretcher carrying a human victim by two rescue robots. Here, the lifting operation requires simultaneous execution in order to prevent the victim from falling off the stretcher. Humans usually solve such a task by mutually tracking the actions of their supporter. However, this requires a sophisticated approach to realize these

actions, which usually requires complex sensor processing methods. However, if we rely on the synchronized clocks of PROViDE agents, we can avoid this complexity. Therefore, the agents decide on a future point in time at which the simultaneous lifting action starts. Assuming successful transmission before the proposed time, the common action can be synchronized according to the precision of synchronous clocks.

The distributed systems research community provides a wide variety of clock synchronization algorithms such as NTP [139]. In particular, the popular *precision time protocol* (PTP) showed strong results in achieving precise clock synchronization [136]. Nevertheless, both approaches assume a client-server-based architecture, i. e., if we would rely on these approaches, we have to assume the existence of a central time coordinator, which is able to synchronize the clocks of all agents regularly in order to ensure that their internal clocks do not diverge critically for a given application. However, we also consider scenarios with fully distributed settings where agents can leave the communication range for an unspecified amount of time.

We require a distributed approach that is robust against engaging and disengaging agents. Many clock synchronization algorithms, which support this property, are based on distributed averaging such as [84]. For the estimation of a synchronized time, PROViDE implements an approach similar to Malada et. al. [102, 103]. Here, the agents periodically send messages that contain the identifier of the sending robot and its local time $t_l$, which we use to estimate the network delay and an "averaged" synchronized time. This allows the compensation of the network delay and the computation of the moving average by

$$t_l(n)_{\text{new}} = t_l(n)_{\text{old}} + c_g \sum_{m \in \mathcal{A}} \left( t_l(n)_{\text{old}} - t_l(m) \right), \tag{7.28}$$

where $t_l(n)$ is the time delta of robot $n$ between its local clock and the distributed time, and $\mathcal{A}$ is the set of all other robots. The parameter $c_g$ is a control gain that can reduce overshooting. As these methods can achieve time offsets $o_t \ll 1\,\text{ms}$, we simultaneously estimate the current communication delay to all other robots within communication range with:

$$t_d = t_l(n) - t_l(m), \tag{7.29}$$

where $m$ denotes the robot identifier. The algorithm is able to synchronize clocks more precisely than 1 ms. Since the communication delays in wireless network communication are usually higher than this value, we provide sufficient precision for our targeted problem domains. The synchronized clocks might still diverge when agents cannot maintain their communication or do not pass an initial synchronization procedure. A disadvantage of this approach is an additional message transmitted iteratively, e. g., every 3 s. Additionally, engaging agents usually induce non-monotonic points of discontinuity in the time estimation. In other words, backward jumps can appear in the synchronous time. To avoid these, we slow down the time until we assume synchronicity. A similar solution is presented in [102]. More complex approaches are beyond the scope of this work.

### 7.3.7 Variable Change Subscriptions

To query variable and decision values, the robotic application has to poll the data store, i. e., the value request requires explicit action. However, in some applications, a notification

(pushing) mechanism is a beneficial solution to speed up the reactions on communication events. For example, if the application waits for the arrival of a new value, continuous polling induces additional overhead to set up the value query and the lookup of the value in the data store. Furthermore, the frequency $f_p$ of the value polling limits the average reaction time of the application to $\frac{1}{2f_p}$ according to the Nyquist-Shannon sampling theorem [151], i. e., the average reaction time for a typical polling frequency such as 30 Hz is $16.67$ ms, which can have a notable impact in dynamic domains such as robotic soccer.

To overcome the aforementioned issues, PROViDE includes a notification service, which allows the subscription of callback functions for a variable or a proposal value. PROViDE invokes these functions whenever the according value has been changed by the acceptance method, due to command messages or a local change caused by the application. This method invocation applies before the network response is transmitted, which minimizes the reaction time. In particular, the callback function can modify the own proposal before it is distributed, to implement complex network protocols using multiple communication rounds such as Paxos, see Section 12.5. Here, PROViDE uses the implementation of a feedback loop for the decision process through value change subscriptions.

As variables and proposals can be addressed by a scope and a name, the notification service can also be seen as an implementation of a publish-subscribe mechanism: The scope and name of the variable identify the topic, and the new value represents the message payload. This makes the communication capabilities of PROViDE comparable to classical communication middlewares such as ROS. The major difference to classical communication middlewares is the fact that PROViDE runs as a single instance on each agent. Thus, communication between different processes of the same agent is not supported.

It is important to note that the pool of threads executing the callback is limited. Thus, callback functions should only execute computationally fast operations, similar to interrupt service routines. Otherwise, other value updates can be blocked or even dropped if the message queues of the network protocol and the network stack overflow.

## 7.4 Conflict Resolution

The negotiation of conflicting proposals is one of the key features of PROViDE. We can show that under certain conditions this negotiation process is able to resolve conflicts if appropriate distribution, acceptance, and value decision methods are chosen. Therefore, we first specify the definition of a conflict in terms of our decision process. Since PROViDE agents are only allowed to change their own proposal, the agents that are the *sources of the conflict* have to identify the necessity to change their proposal to resolve the conflict. Afterwards, we describe the convergence properties of the conflict resolution.

### 7.4.1 Types of Conflicts

In our decision process, we define a conflict as a state where at least two agents cannot infer the same value for a variable at the same time. In other words, a conflict is present when no consistent mutual belief exists. A special situation is given for non-strict ordering

relations such as the combination of the collection acceptance method and a value decision method, which makes a value selection depending on local information. This is for example the case for the own proposal value decision method. As non-strict ordering relations do not result in a total order of proposals, selective choices do not provide any convergence guarantees. Therefore, we consider these combinations as incompatible, see Section 7.2.5. For a compatible combination of decision methods, we can derive two conditions that can cause a conflict: First, inconsistent proposal replications can cause a conflict. Second, a conflict arises if at least one proposal of at least one agent violates the ordering relation defined by the acceptance method.

A strict ordering relation is described by exactly one proposal being ordered higher than all other proposals. We can formally define the proposal that should be proposed by all agents if no conflict is present.

**Definition 7.1.** *Favorite* and *winner*: The agent with the highest ranked proposal is

$$w = \underset{a}{\operatorname{argmax}}(\mathrm{VP}_a(x) \in DB(a)) \quad \text{w.r.t.} \quad <_o . \tag{7.30}$$

We call the proposal $\mathrm{VP}_w(x)$ *favorite* and the agent $w$ the *winner*.

According to Section 7.3.4, the following theorem holds:

**Theorem 7.1.** The probability for consistency in PROViDE converges to $1$ over time if the probability for packet loss $P(L) < 1$ and the resend count $m \to \infty$.

The latter implies a distribution method of at least level $2$. Otherwise, PROViDE does not detect and resend possibly lost packets. Thus, the probability for conflicts caused by inconsistent replicates converges to $0$ over time.

For the second conflict condition, we can distinguish two different types of conflict states: local conflicts and global conflicts:

**Definition 7.2.** A *global conflict state* (GCS) is present if and agent $a$ with a lower-ordered proposal than the *favorite* $\mathrm{VP}_w(x)$ w.r.t. the relation $<_o$ exists:

$$\exists a \exists b \exists x (\mathrm{VP}_a(x) \in DB(a) <_o \mathrm{VP}_w(x) \in DB(b)) \tag{7.31}$$

As a consequence, when an agent changes its proposal, a global conflict state is always present until the protocol has consistently replicated the new value. All proposals have to converge to a single value in order to resolve the conflict. Otherwise, at least one agent violates $<_o$. Since not all distribution methods involve the distribution of all proposals such as level 0 and 1, these methods do not allow conflict detection for all agents. This condition does not imply that all proposals of all agents are consistently replicated or satisfy $<_o$ if no conflict is present. For example, an agent $b$ might store a proposal $\mathrm{VP}_a(x) \in DB(b) <_o \mathrm{VP}_w(x) \in DB(w)$ as long as $\mathrm{VP}_b(x) \in DB(b) = \mathrm{VP}_w(x)$.

The goal of the conflict resolution is to converge to a common global decision. However, as distributed systems do not provide a global view, only *local conflict states* can be detected and only by the agent hosting the conflict.

**Definition 7.3.** A *local conflict state* ($\mathrm{LCS}(a)$) is present if a higher-ordered proposal of another agent w.r.t. $<_o$ exists in the local data store of an agent $a$:

$$\exists x(\mathrm{VP}_a(x) \in DB(a) <_o \mathrm{VP}_w(x) \in DB(a)) \tag{7.32}$$

Again, this condition does not imply a conflict for a third-party proposal that violates $<_o$. Besides the fact that we have to presume a strict ordering relation, a second requirement for the conflict resolution needs to be considered: In order to allow all agents to detect the conflict, they have to rely on the same acceptance method. Therefore, we assume that a variable is always distributed with the same acceptance method. This can be achieved by a simple name convention, e. g., using the acceptance method as prefix of the variable name.

### 7.4.2 Sources of Conflict

The term source of conflict describes the agents maintaining the proposal that contradicts the *favorite*:

**Definition 7.4.** *Sources of conflict*:
All agents of the set $SoC$ are called *sources of conflict*, where $SoC$ is defined as:

$$SoC = \{a \in \mathcal{A} \mid \exists x \exists n(\mathrm{VP}_a(x) \in DB(a) <_o \mathrm{VP}_n(x) \in DB(n))\}. \tag{7.33}$$

From these definitions, we can conclude that, for strict ordering relations, each agent distributing a new favorite proposal initially becomes the *winner*. In turn, all other agents become a source of conflict, as their own proposal violates $<_o$. We can further conclude the propagation of global to local conflicts:

**Lemma 7.1.** Each GCS for a variable $x$ converges to a $LCS(b) \quad \forall b \in SoC$ with $\mathrm{VP}_w(x) \in DB(b)$ if $L(x) > 1 \wedge P(L) < 1 \wedge <_o$ is a strict ordering relation.

*Proof.* PROViDE sends belief updates infinitely often until they are acknowledged. According to Theorem 7.1, $\lim_{t\to\infty} P(A_{\mathrm{VP}_w}) = 1$ holds if $P(L) < 1$. Thus, each proposal $\mathrm{VP}_w(x)$ will be successfully replicated to each agent $b$ with a conflicting proposal. Afterwards, $\mathrm{VP}_w(x) \in DB(b) <_o \mathrm{VP}_b(x) \in DB(b) \quad \forall b \in SoC$ holds. □

An example for the conflict resolution is shown Figure 7.6. Here, two agents $a$ and $c$ simultaneously start to distribute a new proposal value for the variable $x$, which causes a *global conflict state*. Since we assume the *happened-after* acceptance method, agent $b$ accepts the proposal of agent $c$, to resolve the local conflict. In turn, agent $a$ only stores the proposal of agent $c$, as its own proposal has a higher logical timestamp and is therefore the *favorite*. Vice versa, the other agents become a *source of conflict* until they accept the proposal of agent $a$, as marked in purple. In the same fashion, the proposal of agent $a$ causes a local conflict at both other agents. Here, the agents resolve the conflict locally by the application of the acceptance method. Afterwards, the global conflict is resolved. Since all messages are acknowledged, agent $a$ is also able to detect the successful resolution of the conflict. The acknowledgment sent via broadcast by agent $a$ for the proposal of agent $c$ could be received by both other agents, which would also trigger the acceptance method and resolve the conflict. However, it is important to note that the arrival of this acknowledgment at agent $b$ is only ensured by the monotonic accepts distribution method.

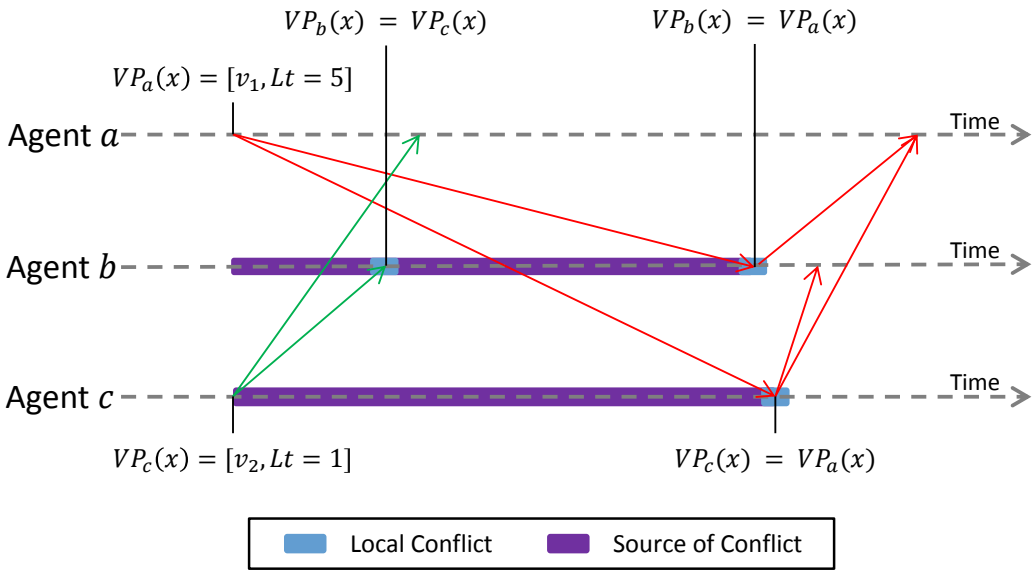



**Figure 7.6:** Example of conflict resolution using *monotonic commands* distribution and *happend-after* acceptance method: Agent $a$ and agent $c$ simultaneously make a new proposal, which leads to a global conflict. The higher logical timestamp of agent $a$ makes its proposal to the *favorite*. After the arrival of $\mathrm{VP}_a(x)$ at agent $c$, it detects the violation of the strict ordering relation and resolves the conflict at the other agents with the acknowledgment including its updated proposal.

### 7.4.3 Proof: Convergence of the Conflict Resolution

From the previous sections, we can conclude that the conflict resolution method of PROViDE converges for a variable $x$ if the following assumptions hold:

1. The chosen distribution method for $x$ is at least level 2.
2. The number of resendings is not limited to an upper bound.
3. It is possible to transmit packages among all agents in the team, i. e., $P(L) < 1$. This also implies that no participating agent is broken and incapable to respond.
4. New proposals always respect the acceptance method locally. Thus, an agent never commands a new proposal that has caused a local conflict.
5. The same strict ordering relation $<_0$ for $x$ is used by all agents.
6. The agents stick to all rule definitions of PROViDE, i. e., no byzantine faults are present.
7. Distorted messages can be detected and rejected, e. g., by a checksum.

According to the proposal update rule, only agent $a$ is allowed to permit a change of $\mathrm{VP}_a(x)$ in the data storage of all agents. Therefore, conflict resolution can only be successful if agent $a$ is able to detect the conflict, changes its proposal, and redistributes it to all other agents. In Section 7.4.1, we stated that a local conflict $LCS(a)$ can be detected by agent $a$. Using 7.1, we can conclude:

**Lemma 7.2.** All agents that are a *source of conflict* can detect their conflicting proposal at some point in time.

This point in time occurs when the *favorite* has been successfully replicated to the sources of conflict. Since the *acceptance rule* ensures that the same strict ordering relation is used and the acceptance method is called whenever a proposal has been successfully replicated, the agent accepts the *favorite*. This resolves the local conflict at the sources of conflict. Moreover, the proposal must be either directly transferred by a command message or the arrival of a command message is pending.

When the command message of a proposal origin (the *winner*) arrives at the *source of conflict*, PROViDE acknowledges the command message according to the $L_2$ *notification rule*. In case the acknowledgment transmission gets lost, the origin will resend the command until a successful transmission occurs. The probability of such a transmission according to our communication model is $P(A_{\mathrm{VP}_w})^2$, as two replication processes have to be successful. Due to the fact that $P(A_{\mathrm{VP}_w})$ converges to 1 over time, the same holds for $P(A_{\mathrm{VP}_w})^2$. Likewise, the successful transmission resolves the local conflict at the origin. In conclusion, all proposal conflicts are resolved at their origin and their *sources of conflict*. As a result, all global and local conflicts are resolved, as the assumption of a strict ordering relation ensures that all agents have either accepted the *favorite* proposal or have to be a *source of conflict*. □

## 7.5 Team Organization

The communication protocols described in the previous sections assume that all agents in the same team have knowledge of the team composition. Therefore, we require an estimation of the set of agents that are currently within communication range. Furthermore, we try to reduce the required bandwidth by stopping the distribution protocol when agents that disappeared are detected. In order to keep the guarantees provided by the distribution methods, the distribution protocol continues its operation when an agent re-engages later on. To optimize the data synchronization process for engaging agents, PROViDE uses scopes to identify proposals that are still relevant and require and update to restore their consistency. The next sections describe the processes of team estimation, agent disengagement, and agent engagement in detail.

### 7.5.1 Team Estimation

The goal of the team estimation procedure is to determine the set of agents within communication range. Therefore, we borrowed the basic idea from ALICA [157], which uses *heartbeat* messages that are iteratively sent via broadcast by each robot to indicate its presence. As payload, these messages contain the identifier of the sending agent and the data for the time synchronization according to Section 7.3.6. Thus, these messages simultaneously estimate the connectivity, packet loss rate, network delay, and clock synchronization data.

We assume an agent to be broken, incapacitated, or out of communication range if the number of expected but not received consecutive messages falls below a certain threshold. Since we assume the events of packet loss to be independent from each other, the probability $P_{\mathrm{dc}}$ for exactly $n$ consecutive packet losses follows a binomial distribution:

$$P_{\mathrm{dc}}(n = k) = \binom{n}{k} P(L)^k (1 - P(L))^{n-k}, \tag{7.34}$$

accordingly the probability for at least $n$ packet losses is:

$$P_{\mathrm{dc}}(n \leq k) = \sum_{k=0}^{\lfloor x \rfloor} \binom{n}{k} P(L)^k (1 - P(L))^{n-k}. \tag{7.35}$$

Note that the independence of packet loss is a simplification, which does not hold for practical scenarios, where packet loss usually appears in bursts. According to [157, p. 135], a Poisson distribution leads to a more realistic approximation:

$$P_{\mathrm{dc}}(n = k) = \frac{\lambda^k e^{-\lambda}}{k!}. \tag{7.36}$$

However, the estimation of the average burst length $\lambda$ of such a distribution is not practicable and error prone in unknown environments for two reasons: First, if the time interval between two consecutive messages is too long, the estimation of $\lambda$ becomes imprecise. Vice versa, short time intervals increase the traffic unnecessarily. Second, agents move around, which influences the properties of bursts including $\lambda$. In contrast, the assumption of the independence of packet loss events does only require an estimation of the parameter $P(L)$, which reduces the amount of required samples to achieve a sufficiently precise estimate. Moreover, the packet loss events can be analyzed independently. As a consequence, the estimation is more robust against changing communication conditions.

A further influence that can be considered is the geographic characteristics of the environment. Hence, a more precise model could estimate the probability by using the current positions of the robots. In particular, positions close to the boundaries of the communication range are more likely to cause an agent disengagement then areas with good connection quality. However, such an approach would require to gauge the targeted environment to estimate the distribution. The only targeted scenario of this thesis, which provides a known environment allowing such measurements, is the RoboCup domain. Due to the fact that the wireless devices have a higher communication range than the maximum distance on a RoboCup field, the resulting distribution does not incorporate beneficial knowledge for our estimation. Caused by the lack of practical relevance in the targeted environments, we did not further investigate this approach.

### 7.5.2 Scopes

As already mentioned in Section 7.2.1, the name of a proposal assigns it to a variable and a scope. PROViDE uses a naming convention that addresses a proposal in a certain scope by a single string. More precisely, each proposal is uniquely addressed by its scope, name, and

distributing agent. Hence, PROViDE allows for multiple occurrences of the same variable name that belongs to a different scope.

Besides proposals and variables, agents also have one or more scopes that describe their current informational state to encapsulate relevant information for a certain (partial) objective, e. g., a list of hierarchical plans that are currently executed. Such a plan hierarchy creates multiple sub-scopes, e. g.,

```
/RescueMission/RescueVictim/
```

where the agent is on a rescue mission with the current task to rescue a victim. If the agents negotiate a variable *Position* that identifies the target positions for the rescue mission, PROViDE automatically uses the current scope as namespace:

```
/RescueMission/RescueVictim/Position
```

Here, the scope identifies the relevance of the variable *Position*. For example, when the agent picked up the victim and returns to the rescue area, it leaves the sub-scope *RescueVictim*. Afterwards, the agent still responds to commands and acknowledgments of other agents that concern the variables in the scope. However, the agent does not actively continue distributing proposals with this scope, i. e., if an agent, who does not have a consistent replica of the current value proposal, engages, the consistency will not be restored. A detailed description of the engagement behavior follows in Section 7.5.4. This has no influence on the negotiation process of other agents that distribute new positions of the victims, i. e., agents negotiate all variables within the current scope of at least one present agent.

ALICA plan variables can have a bijective mapping to PROViDE variables if the plan containing the variables is set as a scope. The unique identifier of ALICA plans is not their name but their generated number. This makes it difficult for human observers to read the scope names.

### 7.5.3 Agent Disengangement

An agent disengagement occurs when an agent vanishes from the team, e. g., caused by a connection loss. This arises due to three reasons: First, the team estimation component identifies a connection time-out. Second, if the number of retransmissions exceeds a certain threshold value, we assume an agent disengagement that has not yet been detected by the team estimation mechanism. As a consequence, the according replication process is aborted and only restarted when the agent re-engages. Third, a disengagement can be evoked by the robotic application. The main reasons for this can be an active announcement of an agent that presumes its disengagement in the near future, or due to observations of other agents. This is particularly useful to allow the robot application to incorporate its knowledge about the intention of the robot to increase the precision and efficiency of the decision process. An example for this is a robot executing a behavior that moves it out of the communication range.

Due to the *persistence rule*, an agent disengagement does not cause changes of any local proposals. This property is important to store the most recent proposals to indicate the

last intention of robots, e. g., a proposal value that identifies a certain victim. Additionally, the validity time specifies the expected time to rescue the targeted victim. If this time is exceeded, the proposal becomes invalid and the other robots can infer that the victim still requires a rescue operation. The same mechanism applies to most other goals that require operation outside the communication range. Summarized, this property is used to keep task assignments persistent such as in ALICA.

The value decision strategies weak agreement and majority voting use the estimated team. Thus, although no proposal change occurs, value decisions could change. As a result, an agent disengagement can still influence the behavior of other agents.

### 7.5.4 Agent Engangement

We describe an agent engagement as the event when an agent appears within communication range and joins the team. The main method to detect newly appearing agents is the team estimation algorithm, which adds a agent to the team when it observes the first message from this agent and resets the counter of consecutively lost messages. As an alternative, PROViDE allows manually adding new agents at application level. Here, domain knowledge can be used to achieve a faster detection of new agents, e. g., in some scenarios, robots can detect their teammates by visual perception over a longer distance. Furthermore, some frameworks such as ALICA provide an own team estimation method.

In case a robot $n$ discovers a robot $m$ with the intention to join the team, the local proposal data store checks whether it maintains variables that underlie the following inconsistency conditions:

$$L(x) > 1 \tag{7.37}$$

$$Lt(\mathrm{VP}_n(x)) > Lt(\mathrm{VP}_m(x)) \tag{7.38}$$

$$\mathrm{VP}_n(x) \in DB(n) \neq \mathrm{VP}_m(x) \in DB(n) \tag{7.39}$$

$$\mathrm{Scope}(x) \subseteq \mathrm{Scope}(n) \tag{7.40}$$

where $\mathrm{Scope}$ represents the scope of a variable or a robot. This means PROViDE only considers inconsistency if the distribution method is at least monotonic commands. The variable $x$ still has to be relevant for agent $n$, which we identify by the variable scope, which has to be a sub-scope of $n$. Finally, the proposal is only inconsistent when a change of the proposal that has not yet been successfully replicated to $m$ occurs. Such a *failed proposal replication* can be identified by the logical timestamp of $\mathrm{VP}_n(x)$, which is greater compared to $\mathrm{VP}_m(x)$. Additionally, agent $n$ checks whether the proposal value actually changed. This prevents retransmissions of proposals that changed multiple times but have been negotiated to the initial value.

If the inconsistency conditions hold, robot $m$ missed a command message of robot $n$ and is in an inconsistent state. As a reaction, robot $n$ sends an according notification in form of a new command message. This message informs about the current proposal value and triggers the conflict resolutions as described before. Since the consistency level of $x$ is at least $2$, agent $m$ responds in turn with an acknowledgment. The distribution method monotonic accepts also confirms the notification acknowledgments with the transmission of a short acknowledgment. This procedure only differs from the standard communication

protocol by the fact that messages are not transmitted via broadcast, i. e., all robots of the team rebuild consistency separately. Note that both robots restore consistency only for their currently chosen scope.

In Section 7.4.3, we showed the convergence of our negotiation mechanism. When we also consider the presented optimization for mobile agents, the convergence does only hold for proposals that satisfy the inconsistency conditions. Thus, convergence additionally requires Equation 7.40 to hold. In contrast, the other conditions have already inherently been part of the convergence proof. Since $L(x) > 1$ is a prerequisite, equal proposal values cannot cause a conflict, and a higher logical timestamp follows directly from the fact that a new value is proposed.

We propose this engagement mechanism for four reasons:

1. Robots do not need to track all uncompleted operations to ensure consistency, as it can be determined by the described inconsistency conditions.
2. Joining robots do only process the most current robot proposals when changes occur and at least one robot is in the respective scope
3. We reduce the number of messages to the minimum number that still maintains consistency
4. This method implicitly enhances the local team estimation.

The disadvantages of this approach result from each agent's independent reestablishment of consistency. As a result, if multiple agents discover a new agent within a short period, the communication channel might be flooded with proposal update messages, which might cause collisions. To overcome this problem, we propose to assign distinct time slots to each agent, which are dedicated for update messages. This allows for collision avoidance and reserves bandwidth for other messages.

## 7.6 Considerations for Ad Hoc Networks

Standard communication technologies do not ensure that all agents are able to communicate with all other agents in the same way, especially in scenarios with ad hoc networks such as autonomous car driving or emergency response domains. As an example, we consider three agents $a$, $b$, $c$, and links between agent $a$ and $b$ and between agent $b$ and $c$ exist. In this case, all agents operate within the same team according to our definition. However, there is no communication link between the agents $a$ and $c$. To establish such a communication link, agent $b$ has to forward relevant network packets in both directions. The problem of identifying which agent has to forward which packet to which of his neighbors is called *routing problem*.

The ideal solution to this problem would first require the identification of the network topology and then computing the path among each pair of agents and identifying optimal broadcasting agents, respectively. The optimality of the latter is given, if the number of required broadcasting agents has a minimum, as this provides the least bandwidth requirements. The main drawback of such an approach depends on the mobility of the robots, which can change the network topology dynamically. Therefore, the routing

protocol either has to consider the robot positions or provide a general broadcasting mechanism, which ensures that all agents in the team are reachable. In PROViDE, we can use a simplified version of the routing problem, as duplicated packets are filtered due to the appended logical timestamp. Furthermore, it is not required to guarantee full network coverage for every transmission, as PROViDE is able to handle packet loss.

The research area of distributed systems developed a wide variety of different routing protocols for message broadcasting. Here, we identified four classes of algorithms:

**Flooding** is the most basic method that is also the easiest to implement. Here, each node forwards each received message exactly once. This guarantees that all agents are reachable. However, it can result in the problem that the communication channel overloads due to the redundantly sent messages. This problem is called broadcast storm problem [114]. Due to the simplicity of flooding, many other approaches build upon it.

**Probabilistic approaches** compute the probability whether packet forwarding is necessary on each agent. The probability is usually adjusted to the network topology, e. g., in sparse networks, a high probability is required, whereas almost fully linked networks only require a small probability. Here, the probability determines a trade-off between redundant messages and network coverage [45].

**Counter-based approaches** initialize a counter for each unique message. The counter increases whenever a duplicate appears at the node. If the number of duplicated messages is still below a specific threshold after a certain time, the agent retransmits the packet [108].

**Area-based approaches** require a global map to determine a distance metric, which decides whether a message requires rebroadcasting [118]. However, since PROViDE does not have explicit information about agent positions or a global map, this approach is not applicable.

To apply PROViDE to not fully connected teams, the internal network interface allows to either implement one of these algorithms or to rely on a routing software product such as LifeNet [106], which satisfy the requirements of PROViDE. Since sufficient solutions for the problem already exist, a more detailed investigation of this area is out of the scope of this work.

## 7.7 Summary

In this chapter, we introduced the PROViDE middleware, which implements the negotiation procedure of our decision process. Therefore, we distinguish between the two concepts of variable values and proposal values. A proposal can be distributed by any agent as contribution to the current issue, whereas a variable groups all proposals of the same issue together. To negotiate proposals, PROViDE includes three steps: proposal replication, conflict resolution, and value decision. The negotiation process can be adapted to the given problem setting by choosing the method for each of the three steps. Here, the proposal distribution is implemented by one out of four distribution methods, which provide different convergence guarantees for the proposal replication process. To enable

conflict resolution, an acceptance method can be selected for each variable. This controls when an agent discards its own proposal in favor of the proposal of another agent. As final component, the value decision method allows the inference of a value from all replicated proposals.

The communication protocol of PROViDE supports unreliable network environments and mobile agents, which can engage and disengage the communication process. Thus, the communication protocol is designed connectionless. To detect lost network packets, we derived a resending mechanism to achieve convergence of the replication process. This mechanism is based on a probability model, which also allowed us to infer convergence of the proposal replica over time. We could further investigated the conditions under which conflicts are resolved during the negotiation process.

In order to save bandwidth during the absence of some team members, we interrupt the communication protocol during the resending procedure until the agents re-engage within the communication range. Furthermore, PROViDE uses scopes to define relevant variables and proposals. Hence, an agent engagement does not require a complete exchange of all missed messages, but focuses on the data that is still relevant for at least one present agent.

# 8 Software Architecture and Implementation

In this chapter, we describe the software architecture of PROViDE, which was released under the MIT license in August 2015.[1] The major goal during the development process was the creation of an extensible and lightweight software architecture. The extensibility does not only focus on the possibility to easily integrate new acceptance and decision methods but also to implement new replication mechanisms or even high-level protocols that establish consensus. In the same fashion, the implementation has to provide separation of concerns. In particular, we separated the low-level communication layers, which are responsible for the transmission, from the PROViDE protocol logic. This separation makes PROViDE independent from specific communication infrastructures. Hence, PROViDE is easy to apply within other domains or when using new communication technologies, e. g., wireless sensor networks. Reference implementations of PROViDE are available in C++ and C#. The C++ version provides additional bindings to realize the low-level network interaction, while the C# version is tightly coupled with ROS.

This chapter starts with an overview of the internal software architecture of PROViDE, given in Section 8.1. Afterwards, we describe the most important software components. In Section 8.2, we describe the implementation of the CNSMT solver and the internal automated differentiation. Afterwards, we investigate the protocol logic of PROViDE and its separation into *tasks* in Section 8.3. The communication interface allows the implementation of additional communication technologies, which is analyzed in Section 8.4. The clock synchronization and network model estimation are performed by the *time manager* component. In Section 8.5, we describe the implementation details concerning the internal timestamps and network model approximation such as the calculation of the *Lambert W function*. PROViDE stores the replicated data in an internal data store implementing persistence properties, as described in Section 8.6. The storage and transmission of complex data types requires a serialization process. Although PROViDE does not make serialization technique contributions, Section 8.7 describes the implemented techniques. Section 8.8 explains the implementation of a console monitoring tool, which acts as a PROViDE agent and can be used to distribute values or explicitly perform operations on the local data store of other agents. We conclude this chapter with analyzing the implementation of variable prototypes in Section 8.9. These support the implementation of decision-specific variables such as mutual exclusion.

[1] https://github.com/carpe-noctem-cassel/cace

## 8.1 Overview

The core of PROViDE is composed of six component types: the middleware interface, a data store, a time manager, variables, the communication protocol, and a low-level network interface. Furthermore, PROViDE supports complex data types as proposal values. To transmit such values over the network interface, a serialization component is required. PROVIDE is furthermore embedded in two applications: the PROViDE monitor and the ALICA communication interface. The software architecture of PROViDE is shown by the component diagram of Figure 8.1.

The ALICA interface is a lightweight software module realizing the communication facilities of ALICA and giving access to the PROViDE negotiation process. Thus, the ALICA messages are serialized and transmitted with the fire and forget consistency and the collection acceptance method. The ALICA interface implements proposal change subscriptions and forwards received messages to ALICA. For the separation of the different ALICA messages, we use a distinct PROViDE variable for each message type. This setup imitates a publish-subscribe-based middleware. We give a more detailed description of the interaction between ALICA and PROVIDE in Section 9.4, which mainly concerns the synchronized clock and the variable storage for the implementation of the PROViDE-based decisions. The other software components of PROViDE are described in the next sections.

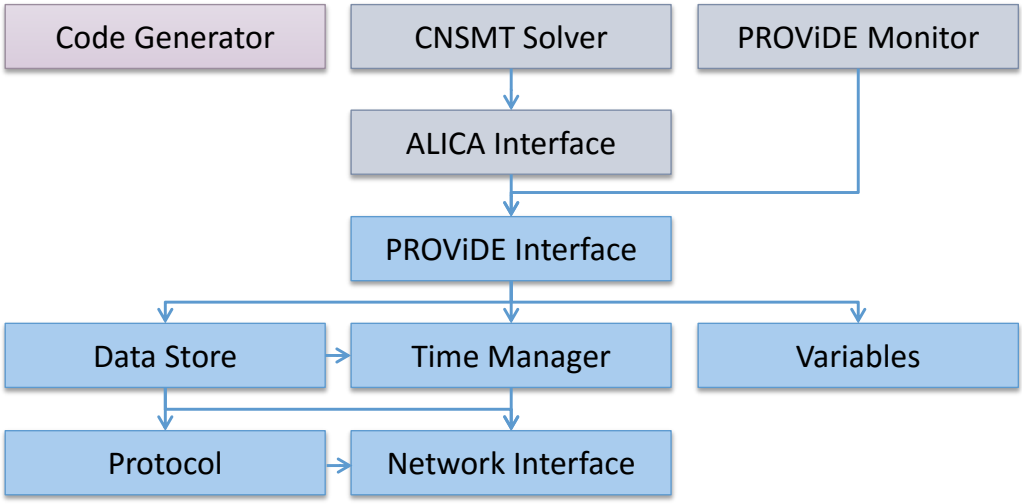



**Figure 8.1:** Component diagram of the PROViDE software architecture. The the arrows indicate relationships between the internal classes.

## 8.2 CNSMT Solver

The CNSMT solver component implements the satisfiability modulo theories solver described in Section 6.5. Our implementation consists of three components: *SAT solver,*

*local search solver,* and *interval propagation*. These work independently of each other. This means they can be used in stand-alone mode, e. g., to solve SAT problems or constraint satisfaction problems.

The SAT solver implementation is based on Minisat [52]. In contrast to the original Minisat implementation, we separated the learned clauses from *safe* and *unsafe* sources. This makes the distinction of the clause origin possible: Due to the incompleteness of the gradient-based *T solver*, learned clauses and their implications might be incorrect (from an *unsafe source*). Vice versa, clauses that are learned from interval propagation are *safe* implications from the problem description, which guarantee correctness. In case the solver cannot find a solution, we can keep the safe and clear the unsafe clauses. Afterwards, the solver restarts from the beginning. This procedure repeats until either a solution is found or to computation time is exceeded. In the latter case, the solver returns an unsatisfiability assumption.

The automatic differentiation library of Shtof [154] serves as the problem description language for our solver. Since the library was originally developed in C#, we implemented a C++ version relying on the same algorithms. For the computation of differentiations, the library resolves operator priorities by compiling them into an efficient structure called *tape*. The tape encodes a tree-like structure similar to heap data structures. After this transformation process, it is possible to apply a *visitor design pattern* [62] on the tree structure for an efficient propagation of values within the tree. This forms an ideal basis for an interval propagation *visitor class*, which performs downward or upward propagation of solution intervals, as described in Section 6.5.3.

## 8.3 Protocol Implementation

The network protocol implementation of PROViDE relies on a set of state machines, which implement the protocol logic. We call such a state machine a *task*. Each task is responsible for the transmission of a certain message type. A task terminates when it *successfully* transmitted all messages via broadcast or unicast, respectively. Thereby, the criteria for transmission success depend on the requested protocol level, i. e., transmissions that do not require an acknowledgment terminate immediately after the network stack confirms the transmission. Vice versa, a task that has to ensure the reception of its transmission terminates after an acknowledgment message has been received from all destinations.

All transmissions include a unique message identifier. This allows the unique identification of the processing task for received acknowledgments. In order to generate a unique message identifier, we perform a bit shift to the left on the current logical timestamp by $\log_2 n_{\text{max}}$ bits, where $n_{\text{max}}$ is the maximum unique identifier of all agents that can appear within the domain. Afterwards, the least significant bits are replaced with the binary-encoded identifier number of the sending agent. As the logical time grows strongly monotonically for all sent messages and the identifier numbers of all agents are unique, this method avoids collisions as long as no overrun occurs. However, the 64 bit integer value range of most today's systems is sufficient to avoid overruns.

The current implementation of PROViDE relies on the following tasks:

**The command task** transmits a command message, which is required for consistency levels 1 to 3. For consistency level 1 (*fire and forget*), the task terminates immediately after sending the command message.

**The acknowledgment task** is responsible for the transmission of acknowledgments of the distribution methods *monotonic updates* and *monotonic accepts*.

**The short acknowledgment task** is only required for consistency level 3 and always provides immediate termination. Here, the short acknowledgment message includes an identifier that corresponds to the preceding acknowledgment and command message.

**The proposal update task** is initiated when an agent re-engagement is detected and proposal updates of at least consistency level 2 were missed. The task sends a standard command message on a different unicast communication channel, which distinguishes it from a command task. For consistency level 2, it terminates immediately. Otherwise, the reception of an acknowledgment message causes termination.

**The update acknowledgment task** executes the acknowledgment to a proposal update task. It only deviates from a standard *acknowledgment task* by sending a unicast instead of a broadcast response.

**The write task** transmits a message to set a robot proposal remotely according to Section 7.3.5. These messages always expect an acknowledgment.

**The proposal request task** requests a proposal at a certain agent to realize data access for read operations.

**The write acknowledgment task** processes the acknowledgment for *write tasks* and *Proposal Request Tasks*.

**The time synchronization task** performs the time synchronization and transmission of a heartbeat message. As this is an infinite process, the task never terminates. Note that received messages are always stored within a message queue and processed by the *time manager*.

PROViDE supports two operation modes to process tasks: *event-based* or *cyclic* mode. *Event-based* task execution triggers the task state machine whenever a message is received or a distribution process is initiated. Thus, all tasks are processed immediately when an event occurs. However, since the tasks access the proposals in the data storage, each variable access requires permission by a *mutex variable* [164] to avoid concurrent operations and memory access violations. However, mutex variables internally rely on system calls, which slow down the execution. To avoid the resulting delays, tasks can operate in *cyclic* mode. Here, the processing routine of tasks is triggered iteratively by an arbiter. This arbiter operates synchronously with the robotic application avoiding concurrent memory access.

A further advantage of cyclic transmissions is the possibility to synchronize the robot transmission cycles, which helps to avoid network collisions and follows the findings of the real-time database [13]. Here, a robot is only allowed to transmit messages during its own dedicated time slot. To assign a time slot to each robot, we use a list with all active robots. This list is ordered by the unique identifier of the robots. Each robot can compute

its own time slot by dividing the cycle time $t_f$ into $n$ equally long time slots, where $n$ is the number of robots in the list. The time slot of robot $i$ in the list is computed by

$$\frac{it_f}{n} \leq t_{\text{sync}} \mod t_f < \frac{(i+1)t_f}{n}. \tag{8.1}$$

This method distributes the time slots uniformly over the cycle time. As a result, small synchronization errors do not necessarily provoke network collisions. We assume cycle time slots of $5$ ms to be sufficient with respect to the precision of the provided time synchronization. Empirical results show that a cycle time of $30$ ms is sufficient for up to 6 robots.

## 8.4 Communication Interface

The architecture of PROViDE separates the low-level communication interface from the protocol implementation. This allows the use of PROViDE with many different communication technologies. Therefore, the communication interface implements the low-level functions to realize transmissions.

For the event-based task execution, our implementation uses the same threads to process incoming messages and to execute the callbacks for *proposal change subscriptions*, i. e., these callbacks can block the reception of network messages and lead to lost messages if the message processing is too slow. We refer to this issue with the term *overloading problem*. In order not to excessively prune the computation time demanded by the user, we use a *thread pool* maintaining a certain number of threads that can execute callbacks.

The overloading problem can also occure when using cyclic task execution. Here, PROViDE asynchronously stores received messages in an internal message buffer. This buffer is processed in the main loop of PROViDE. Hence, a buffer overrun can only occur if the message processing time exceeds the cycle time of PROViDE. However, the internal message buffer provides more storage capacity than the usual buffers in network stacks.

The default communication interface relies on ordinary BSD sockets, which makes it independent from any middleware. However, in order to enable the usage of sophisticated middleware approaches, which for example realize the routing, message queuing, or security features, the communication interface allows the integration of such middleware products. Our implementation provides – besides the binding for ordinary BSD sockets – a ROS binding. ROS does not provide sophisticated inter-robot communication features, which makes it rather inadequate for our targeted problem domains. In the context of distributed applications, ROS serves as case study to show the applicability. Additionally, we used ROS for debugging purposes during the integration process of new protocols.

## 8.5 Time Manager

The *time manager* component implements the synchronized clock according to Section 7.3.6 and a local reference wall-clock time. As we use the same data for clock

synchronization and the estimation of the network model, the time manager determines packet loss, network delay, and the list of currently connected agents. This also includes the calculation for the resend time $t_r$ for the detection of lost network packets and the probability for consistency, see Section 7.3.3 and Section 7.3.4. Therefore, we need a implementation of the *Lambert W function* (the inverse function of $f(w) = we^w$), which is required to compute $t_r$. This implementation is based on the *GNU Scientific Library* (GSL)[2], which uses the *Halley's method* [124] for approximating the function values by:

$$w_{j+1} = w_j - \frac{w_j e^{w_j} - z}{e^{w_j}(w_j + 1) - \frac{(w_j+2)(w_j e^{w_j} - z)}{2w_j+2}}. \tag{8.2}$$

For further details see [67].

The time manager does its calculations based on a dictionary of time messages. One list entry consists of the data of a clock synchronization message, the local wall-clock time when the message arrived, and the synchronized timestamp from the message arrival. Here, the clock synchronization message includes the robot identifier, the local time, and the synchronized time of the origin from the time the message was sent. This allows the computation of a maximum likelihood estimate for the average network delays and divergence of the assumed underlying Laplace distribution according to Section 7.3.3. Therefore, PROViDE uses by default the most current data sets received in the last $30\,\mathrm{s}$.

## 8.6 Data Store

The data store component implements a dictionary of PROViDE variables. To avoid concurrent access on the dictionary, , as is possible with write operations, critical sections are secured by a *mutex variable*. Furthermore, the data store checks whether its validity time is exceeded and replaces outdated proposals by an empty proposal. Moreover, the data store collects all responses for remote proposal access, i. e., the acknowledgments of write operations and the received proposal values of read operations.

The current data store implementation only stores data within nonpermanent memory. This bears the risk that data is lost when an agent restarts its processes due to a crash or a repair action, which are executed by frameworks such as [82]. If these kinds of restarts are expected, the data store component can interface an external database service such as MySQL [175], which allow for persistent storage of the data on permanent memory, e. g., the hard disk. However, the persistence property comes at the price of an increased time to access the data.

## 8.7 Serialization

Similar to most communication middlewares, PROViDE has to address the problem of serializing and deserializing its data to transmit it via the network. However, PROViDE

[2] http://www.gnu.org/software/gsl/

aims for simplicity and does not focus on sophisticated serialization methods for complex data types. Therefore, the implementation of PROViDE offers methods for most common variable types of robotic applications: *integers*, *floats*, *string*, *long*, and *Vectors*, which are composed of floating point variables. Additionally, each of these elementary variable types can be used in different value lists, i. e., in a list of integers, a list of strings, and so on. Internally, PROViDE stores variable values in the serialized representation and allows application developers to serialize complex data types on their own, e. g., by using ROS messages, which are automatically generated with support for boost serialization [39].

During the integration of PROViDE as a middleware core for ALICA, we discovered the need to address more and complex data types. We therefore integrated and extended the *meta serialization* library [128], which provides a set of template functions that can recursively serialize the data containers of the C++ Standard Template Library (STL) [164]. In particular, combinations of STL tuples, lists, and vectors allow to encode the most important data structures. Furthermore, the library author gives evidence that the implementation outperforms boost serialization in computational efficiency and data size overhead.

PROViDE also supports a lightweight code generation tool, which generates classes providing serialization and deserialization methods. The generated structures again rely on the *meta serialization* library. The code generation is based on text files that describe the data structure using the ROS interface definition language (IDL)[3] [134].

## 8.8 PROViDE Monitor

The *PROViDE monitor* is a console tool estimating proposals of a single agent and serialization manipulation functions. The PROViDE monitor hosts an instance of PROViDE that responds to messages and distributes proposals according to the protocol logic. In the same way, it participates in the time synchronization process.

The user can navigate through the robot scopes similar to a UNIX console using the *CD* command. With the *LS* command, the user receives a list of all variables that have the current scope as sub-scope. The *CMD* command initiates the proposal distribution and negotiation process. By default, this is done with consistency level 3 and the *happened-after* acceptance method. Furthermore, the user can remotely set proposals of other agents with the *WRITE* command or request the current proposal of an agent by using the *REQ* command.

For testing purposes, it is also possible to manually announce an agent engagement (*ADD*) or disengagement (*REM*). This also allows for updating these agents with proposals that require replication. To check whether all tasks are finished, the command *TASKS* prints a list of all currently active tasks.

Although a PROViDE monitor instance acts as an agent, it is possible to switch it into *quiet* mode. In this mode, the program acts as a pure monitor and does not transmit messages autonomously. This means that it does not send out an acknowledgment although the data store is kept up to date according the received messages of agents. However, it is still possible for the user to actively interact with other agents.

[3] http://wiki.ros.org/msg

## 8.9 Variable Prototypes

Variable prototypes are the implementations of distributed applications for PROViDE. In particular, these implement the decision method and high-level protocols as described in Chapter 12. All these applications inherit from the base class *PROViDEVariable*. It contains all proposals that are assigned to it and provides access methods for the proposal properties. Furthermore, the *PROViDEVariable* class implements the decision method and allows change subscriptions for variables or proposals. The subscriptions are invoked by delegating function pointers.

An example for a distributed application we implemented, among others is the *PROViDE mutex* variable. This mutex variable provides the standard $P()$ and $V()$ methods. An agent can call $P()$ if it intends to enter the critical section and $V()$ in order to leave it. Since the $P()$ call requires waiting until the critical section is accessible, we implemented a blocking and non-blocking version. The non-blocking version distributes the intention to enter to all other agents. Hence, this variable type encapsulates the functionality described in Section 12.2.

## 8.10 Summary

In this chapter, we described the software architecture of PROViDE. In particular, we highlighted its decomposition into the key components: the *CNSMT solver*, *code generation*, *PROViDE monitor*, *ALICA interface*, *PROViDE interface*, *data store*, *time manager*, *variable prototypes*, *protocol interface*, and *communication interface*. This makes the implementation maintainable and extensible. Using this structure, we developed a PROViDE implementation requiring less than 30,000 lines of code including the constraint satisfaction solver and the automatic differentiation library. Hence, we are able to provide a lightweight reference implementation for our decision process.

# 9 ALICA Integration

Our decision process is embedded into ALICA, which supports the specification of a problem description and is able to implement the final decision. We used ALICA for four main reasons: First, its implementation allows us to assume mutual belief on the problem definition. Second, as described in Section 3.6, ALICA provides robustness against possible network failures, especially packet loss and delay. In turn, this does not make consensus to an obligatory requirement. As we described in Chapter 7, PROViDE uses this property to reduce the communication overhead for situations where temporary conflicts are acceptable. Third, ALICA provides support to implement problem-solving algorithms in a distributed setting. Fourth, a decision process requires an execution layer, which allows the implementation of the computed decisions as real world actions. As ALICA is designed for exactly this use case, it represents an ideal framework for our solution.

The combined framework of ALICA and PROVIDE abstracts the team coordination process from the robotic application. Therefore, the robotic application accesses three middleware interfaces: ALICA, which implements the execution layer and allows the team behavior description, the PROViDE interface, which injects beliefs into the believe base of ALICA, and the negotiation layer, which parameterizes the communication protocol with the consistency and fault tolerance properties. Figure 9.1 gives an overview of the communication layers of the proposed protocol structure. Note that in many cases, the low-level network transport may again rely on a communication middleware. In particular, PROViDE assumes that every instance represents an agent. Therefore, it is not adequate for intra-robot communication, e. g., to transmit commands to the actuators. For this purpose, we recommend an additional intra-robot communication middleware such as ROS.

The integration of PROViDE into ALICA requires several changes of the ALICA implementation. The first change to ALICA is required to facilitate exchanging the solution techniques. Therefore, we propose a solver-independent interface. Each compatible solver needs to be inherited from an interface and implement a solving method allowing ALICA to call it.

A second modification of ALICA needs to abstract the specification method for constraint satisfaction problems. As mentioned in Section 3.4, the original ALICA implementation requires constraints to be expressed by using the automatic differentiation library of Shtof [154]. To manage the problem specifications and their solutions, ALICA requires access to an object that represents the problem specification itself and solution references for each variable. Therefore, we rely again on inheritance. Both of these changes allow to almost arbitrarily exchange the solution techniques. Note that this modification also enables solvers to access old solution values and map them onto their local variable representations.

The third ALICA modification is the persistence of decisions. ALICA assumes non-communicating agents as malfunctioning and replaces them in the current task assignment.

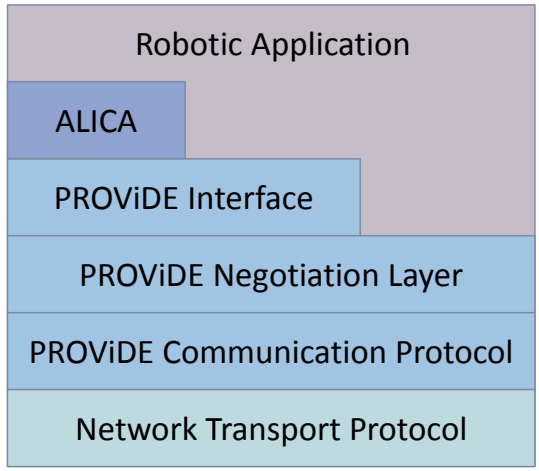



**Figure 9.1:** PROViDE protocol structure.

To address this issue, we exploit the persistence of the replicated proposals to realize persistent agent states. This allows agents to leave each other's communication while continuing their common decisions.

The rest of this chapter explains the integration of the decision process into ALICA. As mentioned before, we see this as an example framework that forms a proof of concept for the integration into other coordination middlewares. Section 9.1 and Section 9.2 describe the interface design, which realizes exchangeability of the problem solving algorithms and the concept to select an appropriate solver at runtime. Afterwards, Section 9.3 describes a variable synchronization that encapsulates the decision process, which allows querying variables or partial solver solutions. Section 9.4 explains the adapted collection of the problem statement and summarizes the integration of PROViDE into ALICA. We conclude this chapter in Section 9.5 and Section 9.6 with two applications that result from the usage of our decision process: First, persistence of task allocations, and second, auction-based task allocation.

## 9.1 Enabling Exchangeability of Problem Solvers in ALICA

To support reasoning on problem specifications and their solutions, the ALICA implementation uses four main steps to query a problem solver for solution proposals.

1. To access ALICA variable values, the developer sets up a query that starts the solution process, e. g., in behaviors, utility functions, or conditions.
2. The list of variable names and the current plan tree is processed to unfold the relevant constraints according to Section 3.4.
3. The collection of unfolded constraints is stored as a list of *ConstraintDescriptor* objects. These include the problem specification as a utility function and the constraint expression.

4. The query object calls a solution algorithm that is able to process the problem specification. Afterwards, the computed solution is returned to the calling component via the query object.

Most high-level programming languages such as C++ or C# use typed variables. As ALICA unfolds constraints to a complete problem statement and provides the interface to access the computed solution proposals, generic representations for the problem specification and the solution values are required. Therefore, we store all problem descriptions in the *SolverTerm* class by using object-oriented inheritance. The polymorphism of object-oriented programming languages allows that each solver is able to convert all terms into its own representation for the internal solution.

Many problem descriptions require references to the queried variables, e. g., to constrain them. These references are implemented through the inheritance from the *SolverVariable* class. However, during the solution process, the solver has to match its solutions to the respective variable, i. e. the list of queried variables requires the same order as the list of returned solutions. Therefore, each variable is requested and instantiated by the problem solver at runtime.

Another issue for exchangeable solving algorithms is the inhomogeneity of the solution value type, e. g., solutions for SAT problems return a list of Boolean values, while path-planning problems return a list of waypoints. Here, we assume that the developer knows at compile time which variable type the solver class returns. In conclusion, we can realize the solution query functions by using templates (C++) and generics (C# or Java), respectively.

## 9.2 Dynamic Solver Selection

One contribution of the presented decision process is the separation between the algorithm to compute proposals and the behavior to implement them. This separation offers the possibility to switch the problem solving algorithm dynamically. More precisely, we can analyze the problem structure at runtime and select the solving algorithm, which optimizes quality criteria such as computation time, robustness against sensor noise, or precision.

For a dynamic solver selection, we set up a list of all available solvers during the initialization phase of ALICA. Additionally, each query object is extended with an additional parameter referring to one of the solver instances. This enables the application developer to recommend a solver for a certain query. As the query object performs the selection dynamically, the application can use problem-specific metrics or reasoning methods to determine the most appropriate solver for a given situation. Typical solver selection methods rely on statistical properties of the problem definition. In particular, we consider the numbers of variables, constraints, conjunctions, disjunctions, and inequalities. As approach for such a selection method, we propose that each solver specifies weights for the problem statistics, which forms a utility function. Afterwards, an optimization process determines the optimal solver for a given problem. Therefore, we compute the utility value of every solver for the current problem and select the solver with the highest value.

## 9.3 Variable Synchronization Module

Many robotic applications have to repeatedly review the validity of decisions. For example, soccer robots that decide on blocking positions have to check whether the movements of their opponents require an adaptation of the targeted blocking positions. However, even in case of a movement of opponent, it is common that slight deviations of the blocking positions suffice to adapt to the new situation. Moreover, the assignment of blocking robots to opponents remains the same. In such cases, the problem solver can use old solutions as an initial seed to facilitate the computation of new solutions. Furthermore, the target points become more stable, which leads to a smoother behavior of the robots.

Our software implementation provides explicit support for the reuse of old solutions through a *variable synchronization module*. Its main function is the automatic distribution of solutions among all agents, and the operation as database for solutions received from other agents. To limit the memory requirements, the synchronization module does not store a complete solution history. Instead, equal solutions of each agent are summarized. This mechanism reduces the number of possible seeds for solvers and thereby simplifies the search procedure. Hence, the logical view of the *variable synchronization module* on old solutions is similar to a distributed tuple space such as Linda [11].

As the *variable synchronization module* internally queries the PROViDE middleware, the result list does not only contain values that are solver results. Additionally, it includes the finally negotiated values, which were selected previously. Hence, our approach puts more emphasis on the actually selected values. If a selected value is explicitly considered, the solver can realize a natural convergence to a common decision without requiring an additional negotiation of PROViDE. An experiment showing these properties was conducted in [157, p. 209ff].

## 9.4 Integration of the PROViDE Middleware Core into ALICA

One goal of this thesis is a unified decision process responsible for all decisions of ALICA agents. This requires two additional modifications of ALICA: First, a new interface object that implements the interaction between ALICA and PROViDE and allows the access to decision values is needed. Second, a unified communication interface that implements the ALICA communication with PROViDE is needed.

As described in Section 3.4, a standard ALICA query applies five processing steps:

1. Collecting requested variables.
2. Unfolding ALICA constraint.
3. Probing requested solver for a solution.
4. Injecting the solution into the variable synchronization module.
5. Returning the result to the behavior.

To start the PROViDE negotiation process for ALICA variable values, we implemented the *PROViDEQuery*. A *PROViDEQuery* requests solutions from the PROViDE middleware and returns the negotiated value decision. More precisely, the query only returns a value if the negotiation process was successfully completed. The negotiation process requires the initialization by at least one agent distributing a solver result as proposal value. An agent that requires a negotiated solution executes the query twice: first, a standard ALICA query, and second, a PROViDEQuery. If the agent does not have the knowledge to compute a solution, it can wait for negotiated solutions of other agents by only executing PROViDEQueries.

The second modification to ALICA requires a unified communication interface, which transmits executed plans, synchronizations, and authority information to PROViDE. Originally, ALICA instantiated multiple independent publisher and subscriber of a robotic middleware such as SPICA [15] or ROS [134]. These components were distributed over multiple components in the source code. This made the exchange of the underlying communication middleware to a challenging task that requires profound knowledge of the ALICA implementation and functionalities. To integrate PROViDE as communication middleware and to make ALICA to a more generic framework, we grouped the communication functionalities and provided a single interface. This interface serializes, transmits, and receives the data structures of ALICA:

**Plan tree** ALICA iteratively sends messages, which identify the current plan and state. This allows other agents to track the state of their teammates and detect conflicts.

**Solver result** As mentioned before, the variable synchronization module distributes solver solutions among the ALICA agents. This allows to replace the PROViDE negotiation by other decision middlewares.

**Synchronization** Synchronizations (see Section 3.1.3) are realized as a three-way-handshake and therefore require two messages: One message transmits the information that synchronous transition is requested and another is needed to realize the acknowledgment.

**Allocation authority** The allocation authority is part of the conflict resolution in the task allocation (see Section 3.6.2) and is sent by agents to ensure that the agent with the lowest ID resolves the conflict.

**Role switch** In case an agent changes its role during runtime, it communicates this information with a role switch message that includes the updated role. Note that this message is only required if a change of capabilities occurs, e. g., due to malfunctions.

Additionally, we transmit ALICA's debugging messages, which encode the current agent state as a string. In contrast, plan trees only include IDs of the relevant ALICA elements. An overview of how PROViDE is integrated into the ALICA components is shown in Figure 9.2 by presenting the example of the query process.

## 9.5 Persistent ALICA Agent States

The implementation of persistent decisions requires a change in ALICA's treatment of robots that do not communicate anymore. Originally, ALICA excludes such agents from

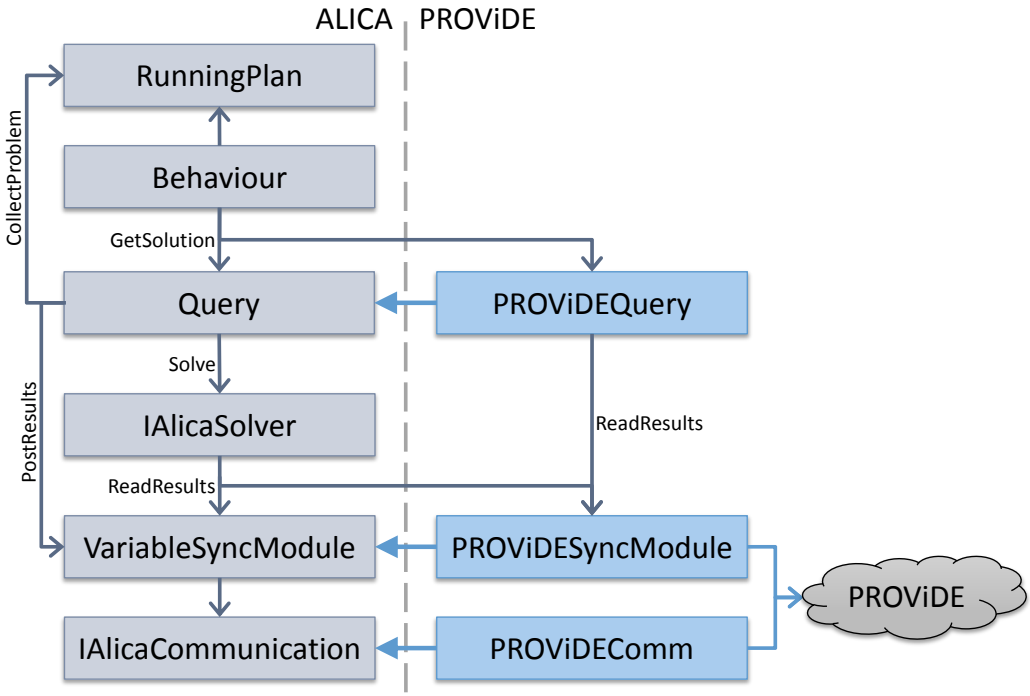



**Figure 9.2:** ALICA querying components with PROViDE middleware integration.

the team and assumes the robot to be either broken or incapacitated. As a result, the tasks that are executed by disconnected agents are released. Moreover, ALICA tries to assign other agents to the mandatory tasks. If no other robot is able to take over such tasks, ALICA detects a failure and applies its repair rules. In the worst case, this can cause the cancellation of plans. Here, ALICA agents cannot accomplish tasks, which forces them to move out of the communication range of the team. This is particularly the case because new agents are assigned to such tasks whenever the assigned agent looses its connection. In conclusion, ALICA would consecutively tasks all robots to this type of tasks.

To fix these issues, we attached a validity time to each task, which is an estimate of the required execution time. Additionally, ALICA's team-observing component is modified to consider assignments as fixed according to this timestamp. Hence, ALICA "trusts" that robots are up and running if they do not exceed the estimated execution time for their current task. As a result, we enabled ALICA agents to leave each other's communication range if it is intended by the application developer.

## 9.6 Auction-based Task Allocation

As described in Section 3.6.2, ALICA provides a sophisticated method to detect and resolve conflicts in its task allocation scheme. However, this method does not take into account

which agent has the best knowledge or confidence for the judgment on a task assignment. Instead, the decision mainly relies on the agent identifiers. More precisely, no agent can compute a task assignment that conflicts with the task assignment of an agent with a lower identifier. This approach results from the assumption that all ALICA agents act cooperatively in a team and operate on equal information. Nevertheless, in practical scenarios such as in emergency response or autonomous car driving scenarios, it is often not possible to realize common knowledge or mutual belief on all relevant facts. Vice versa, sometimes each individual agent can compute a better judgment of its own adequateness for a certain task, which is often more precise and up to date than the one of other agents. Classical methods following this idea are auction-based task allocation algorithms such as [64].

The task assignment approach of ALICA supports the implementation of auction-based task allocation in form of a utility summand. Therefore, we present a utility summand that considers the auctions of other robots. Conceptually, this utility summand has a similar implementation as the priority summand of ALICA (see Section 3.2.4):

$$\mathcal{U}(p, \mathcal{A}, T) = \begin{cases} -1 & \text{if } \exists a \in \mathcal{A} \text{ with } \mathcal{E}_{a,\tau} \leq 0 \\ 0 & \text{if } \exists a \in \mathcal{A} \text{ with } \mathcal{E}_{a,\tau} = \emptyset \\ \frac{1}{|\mathcal{A}|} \sum_{\mathcal{A}} \mathcal{E}_{a,\tau_a} & \text{otherwise} \end{cases}. \tag{9.1}$$

The summand relies on a matrix $\mathcal{E}$ of utility values, where each entry $\mathcal{E}_{a,\tau}$ describes the estimated utility of agent $a$ if assigned to a certain task $\tau$. In the standard case, the utility function returns the normalized sum of all utility values for the given assignment $T$. Additionally, it suppresses robots that are assigned to a task with a negative utility estimation by returning a value of $-1$. This avoids assignments where at least one robot provides a negative contribution to the common plan goal.

When the team starts the plan execution, the task assignment is computed for the first time. This means that $\mathcal{E}$ is initially a sparse matrix for the agents. More precisely, agents initially only know their own utility estimates and start to distribute their auction bet using PROViDE. As an optimal assignment can only be computed if the matrix is complete, the utility function returns $0$ until a "bet" for all agents exists. This indicates that the agents are ready for the execution of a task and stand by.

A further component, which is required by ALICA's task allocation algorithm, is a suitable heuristic estimating the maximum utility value a partial assignment can achieve. We therefore propose two heuristics, where the first exploits the maximum of each column and the second the maximum of each row.

1. The heuristic first estimates an upper boundary for the utility values by:

$$\vec{e} = [max(\mathcal{E}_{1,1}, \cdots, \mathcal{E}_{|\mathcal{A}|,1}), \cdots, max(\mathcal{E}_{1,|T|}, \cdots, \mathcal{E}_{|\mathcal{A}|,|T|})] \tag{9.2}$$

$$h_{max} = \sum_{\mathcal{A}} e_a \tag{9.3}$$

For $n$ *task to robot* assignments $(\tau_i, a_i)$, the upper boundary is updated with:

$$h = h_{max} - \sum_{i=1}^{n} e_{a_i} - \mathcal{E}_{a_i,\tau_i} \tag{9.4}$$

This heuristic reduces the utility estimate greedily by assigning the agent first that desires a free task the most. As a consequence, it performs best when the agents have a good self-assessment and distribute high values for tasks they are most suitable for in comparison to the others. In conclusion, if all agents are able to distribute an optimal estimate, $\mathcal{U}_{max}$ equals $h$, which is an optimal heuristic leading to an optimal search.

2. While the first approach speeds up the search by exploiting the maximum values of the columns of the matrix $\mathcal{E}$, it is also possible to adjust the approach to exploit the row maximum:

$$\vec{e} = [max(\mathcal{E}_{1,1}, \cdots, \mathcal{E}_{1,|T|}), \cdots, max(\mathcal{E}_{|\mathcal{A}|,1}, \cdots, \mathcal{E}_{|\mathcal{A}|,|T|})] \tag{9.5}$$

$$h_{max} = \sum_{T} e_{\tau} \tag{9.6}$$

The update for partial assignments follows the first approach but updates the heuristic with the difference between the chosen assignment and the row maximum:

$$h = h_{max} - \sum_{i=1}^{n} e_{\tau_i} - \mathcal{E}_{a_i,\tau_i} \tag{9.7}$$

The major advantage of this heuristic becomes apparent in scenarios where no agent estimates a high utility for some tasks. Here, the agents with the highest utility for those tasks are hold back during the assignment of other tasks.

Although both heuristics can result in fast computations of the task allocation, the problem is $NP$-hard. Hence, the agents participating in the auction-based task allocation should consider the adequateness of the others when computing their auction bets, in order to increase the performance of the heuristic.

An important benefit of this auction-based task allocation summand is revealed when considering the possibility of a conflicting allocation. In particular, a conflict can only occur if $\mathcal{E}$ is not consistently replicated. However, this can only happen when the auction starts or when agents change their auction bets. Thus, the bottleneck changes from the problem of achieving coherent knowledge bases to the data exchange to determine $\mathcal{E}$, which requires less development effort of the system developer given a sophisticated distribution and negotiation process such as PROViDE.

# Part III

# Assessment

# 10 CNSMT Solver Performance Evaluation

One performance characteristic of a decision process is the required time to reach a decision. In our decision process, the decision time depends on two characteristics: the time to compute possible proposals and the time to negotiate them with PROViDE. This chapter focuses on the performance evaluation of our proposal computation method. Therefore, we compare the CNSMT solver with other state-of-the-art solvers considering and analyzing their runtime requirements. Thereby, we conducted experiments for the following solvers:

- iSat – developer version of iSat [60],
- z3 – Microsoft's z3 solver [110],
- GSolver – the Rprop($\Sigma_\wedge$, max) solver presented in [158], which is a standalone version of the local search *T solver* described in Section 6.5.4,
- CNSMT – the approach presented in this work,
- CNSMT-no IP – the presented approach without interval propagation.

Here, the iSat solver addresses a similar problem class as the GSolver and CNSMT solver, while the z3 solver cannot solve complex nonlinear constraints, which for example appear in geometric problems. In addition, the CNSMT solver and GSolver can also handle optimization problems, which is not possible with the iSat solver or z3 solver. However, z3 solver has shown outstanding performances over years at the SMT-COMP-[16], which provides a sophisticated benchmark to compare SMT solvers. The presented experiments are conducted in single-threaded mode on an Intel i7-930 CPU ($2.8\,$GHz) with Linux 3.2.0-35 and Mono 2.10.8.1.

The chapter is organized as follows: In Section 10.1, we show the efficiency of our approach for a problem called *3-Sat-Sine problem*, which highlights its strengths in a complex nonlinear problem setting. Afterwards, we show the impact of the logical structure on the problem solving efficiency in Section 10.2. Therefore, we formulate a task allocation problem where a set of robots is assigned to target points. We present three different formulations of this problem and analyze the different results. In the following experiment presented in Section 10.3, we present the value of the interval propagation algorithm. Finally, Section 10.4 concludes the chapter with the evaluation of a constraint optimization problem, which shows the applicability of our approach to this problem class.

## 10.1 3-Sat-Sine Problem

The *3-Sat-Sine problem* shows the efficiency of our approach for a problem with a complex Boolean structure and the presence of complex nonlinear expressions. This benchmark was originally introduced in [157], based on [153]. In our problem instance, the number of real-valued variables $n$ is set to $25$. The variables are composed to a problem with $l = 50$ inequalities $p \in P$. Each inequality has the form:

$$k\Sigma_{i=1}^{3}\Pi_{j=1}^{3} a_{ij} \sin(2\pi x_{ij} + b_{ij}) < \theta, \tag{10.1}$$

where $k = \frac{1}{\Sigma_{i=1}^{3}\Pi_{j=1}^{3} a_{ij}}$. All $a_{ij}$ are random values, which are uniformly distributed in $[-1, 1]$. Furthermore, $b_{ij}$ is uniformly distributed within the range $[-2\pi, 2\pi]$. Finally, the variable instances $x_{ij}$ are randomly chosen among all the $n$ variables. The threshold $\theta$ is chosen to define the region of feasible values of the constraint, which covers $50\,\%$ of the domain size. This ratio is determined by random sampling. Thus, a single propositional variable controls the ratio dividing the solution space. The 3-SAT problem formula $\phi$ is composed of $m$ clauses with three literals, which are randomly drawn from $P$.

Although our approach can determine the unsatisfiability of problems, we assume that robots generally decide on satisfiable problems. In order to guarantee the satisfiability of $\phi$, we define $\vec{s}$ as random point in $\mathbb{R}^d$. The vector $\vec{s}$ represents a valuation function $v$. Furthermore, $P'$ is the set of propositional variables evaluated to *true* if $v$ is assigned. Then, for every clause in $\phi$, we pick a random literal and set its sign to positive if its propositional variable is in $P'$. Vice versa, the sign is negative if the literal is not in $P'$.

The z3 solver is not able to deal with trigonometric functions. Hence, we compared the runtime of our approach only with the iSat solver and GSolver. Therefore, we varied the constraint ratio $m/n$ to adjust the problem complexity. As shown in Figure 10.1, if the constraint ratio exceeds a value of $2$, our solver outperforms the GSolver and the iSat solver. These results are averaged over 1,000 repetitions of the experiment.

This experiment confirms the presumption stated in [153]. More precisely, we can observe a phase transition at a constraint ratio of approximately $7.8$. This value is at the same order of magnitude as the phase transition of SAT solvers for random 3-SAT problems, which is at approximately $4.2$ [20, p. 110]. From this experiment, we can conclude that our solution is able to exploit a given logical problem structure, which justifies the usage of an internal SAT solver. Since this use case is rather artificial compared to real robotic applications and provides an unusual 3-SAT problem structure, the next experiments focus on the properties of more practical problem settings.

## 10.2 Allocating Robot Positions

A typical problem in robotic applications is the task assignment problem. In Section 3.2.3, we explained the common classification into *single-rask robots* (ST) and *multi-task robots* (MT), which execute *single-robot tasks* (SR) or *multi-robot tasks* (MR). In this section, we formulate a multi-robot task assignment problem as constraint satisfaction problem. Generally, most multi-robot task assignment problems are strongly $NP$-hard [65].

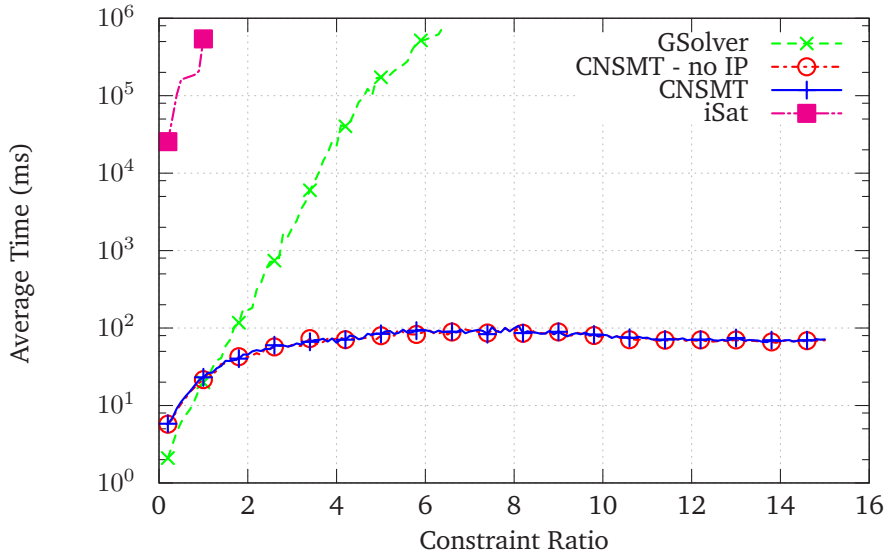



**Figure 10.1:** 3-SAT-Sine test executed by the iSat and CNSMT solver.

As our solution additionally has to deal with an arithmetical expression, we reduce the problem to an ST-SR-IA instance. Moreover, the number of tasks equals the number of robots. Hence, a valid solution assigns one robot to exactly one task. This problem can be solved with $O(n^3)$ using the Hungarian method [87].

In our scenario, we specify $n$ different locations $p_i$ within a global coordinate frame as two-dimensional points. In practice, these are given by prior knowledge, e. g., the position of disaster victims, which were detected by a search robot and have to be rescued. The task allocation problem assigns these $n$ locations as target positions $t_i$ to all robots $i \in \mathcal{A}$.

In order to ensure that the problem expression requires the assignment of each robot to a target location, the first part of the constraint expression is:

$$\bigwedge_{i=1}^{n} \bigvee_{k=1}^{n} (t_i = p_k). \tag{10.2}$$

Here, an assignment $p_i = t_i$ is present if the Euclidean distance deceeds a certain threshold: $(t_{i_x} - p_{j_x})^2 + (t_{i_y} - p_{j_y})^2 < \epsilon^2$. This induces a nonlinearity to the problem formulation, which increases the overall problem complexity again to an $NP$-hard problem. We sample the uniformly distributed target locations within an area of $10 \times 10$ km. Furthermore, a robot can detect target victims by their local sensors if they are closer than $\epsilon = 50$ m to the target. The following results are averaged over 1,000 executions.

The second part of the constraint ensures that each task is performed by exactly one robot. Therefore, we can distinguish three different problem formulations. First,

$$\bigwedge_{i=1}^{n} \bigvee_{k=1}^{n} (t_k = p_i) \tag{10.3}$$

ensures that each target location is assigned to at least one robot. Since the number of robots equals the number of target locations, the problem solution can only be valid if each robot is assigned to exactly one location. The resulting constraint is made up of $n$ clauses with $n-1$ disjunctions. This constraint requires the least memory compared to the other expressions.

Figure 10.2 shows that our solution requires more computational power than all other approaches for more than $4$ agents. This is caused by the fact that the problem formulation does not allow the exploitation of the logical problem structure. More precisely, the theorem solver requires queries for multiple sub-problems in order to detect a conflict. However, if many sub-problems do not allow to draw conclusions on the solution, it is more efficient to query for a solution of the complete expression. As a result, the GSolver shows the best overall performance.

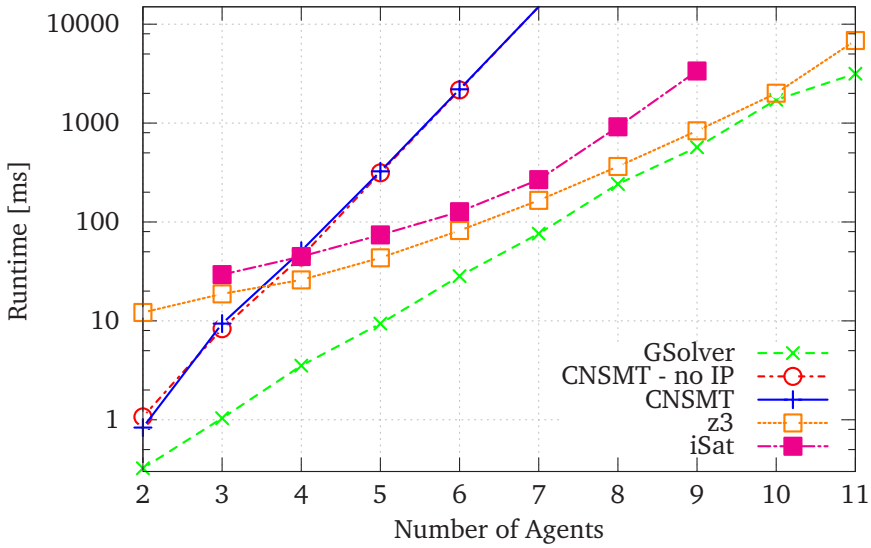



**Figure 10.2:** Required runtime to solve constraint Equation 10.3.

The second formulation is

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \bigwedge_{k=1}^{i-1} (t_i \neq p_j \vee t_k \neq p_j). \tag{10.4}$$

It ensures for all locations and all possible pairs of robots that at least one is not assigned to that location. In other words, two robots can never be assigned to the same location. Here, each clause is composed of only two literals, creating a 2-SAT problem, which is known to be $NP$-complete [55] and can be solved in polynomial time [85]. However, this comes at the price of a high memory consumption caused by the number of clauses, which is ${}^{n^3}/_{2}$.

Due to the simplicity of the 2-SAT problem, the SAT solver-based approaches outperform the GSolver if the number of agents exceeds 5, as shown in Figure 10.3. At the same time, we can identify a high runtime overhead of the iSat and z3 solvers. However, both show a

better scalability for an increasing amount of agents and tasks. In particular, the z3 solver is able to solve the problem in less than one second for 20 agents. At the same time, we claim that a performance below 100 ms is essential to allow reactive behavior. Here, the z3 solver suffers from its overhead, and both CNSMT solver solutions outperform the other approaches for less than 9 agents, i. e., the CNSMT solver is more suitable than the z3 solver for scenarios with a low amount of agents such as in robotic soccer.

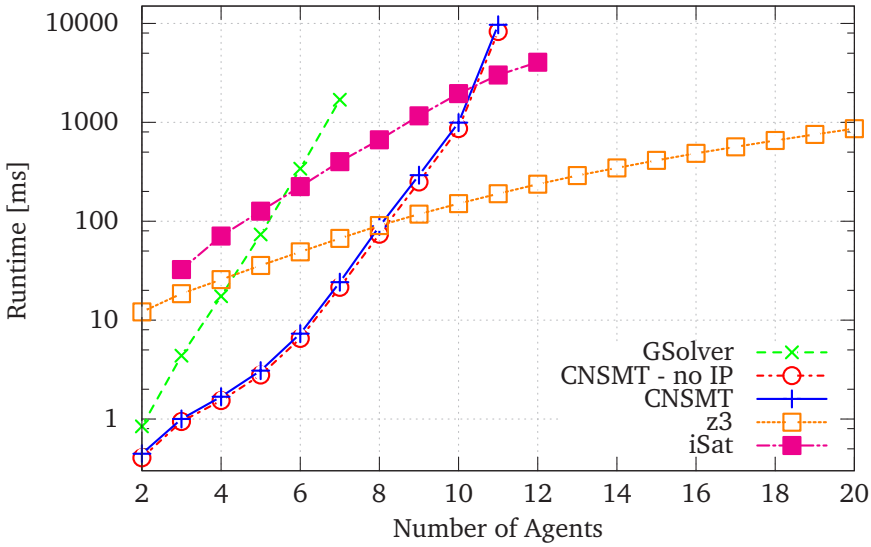



**Figure 10.3:** Required runtime to solve constraint Equation 10.4 based on a 2-SAT problem, which favors SAT-based solvers.

The last formulation is semantically equivalent to Equation 10.4:

$$\bigwedge_{i=1}^{n} \bigwedge_{k=1}^{i-1} (t_i \neq t_k). \tag{10.5}$$

Thus, all pairs of robots are required to be mutually different. We describe this pairwise inequality with an arithmetic expression, instead of using a logical structure. Hence, the number of clauses is only $n^2/2$. This causes a shift of the problem complexity from the SAT solver to the *T solver*.

The results of the final expression are shown in Figure 10.4. Here, the *T solver* provides efficient implications through expressive clauses where conflicting assignments can be identified. Therefore, the SAT-based solvers again show a better scalability than the GSolver. The comparably low overhead induced by a *T solver* call allows our CNSMT solver to provide runtimes superior to the other approaches.

Since our approach is incomplete, it is worth to mentioning that the CNSMT solver solved all 1,000 repetitions of all presented formulations within a single run. This shows that the incompleteness of our solution does not necessarily cause false positive solutions. Considering all three experiments, the z3 solver offered the best results for more than 8

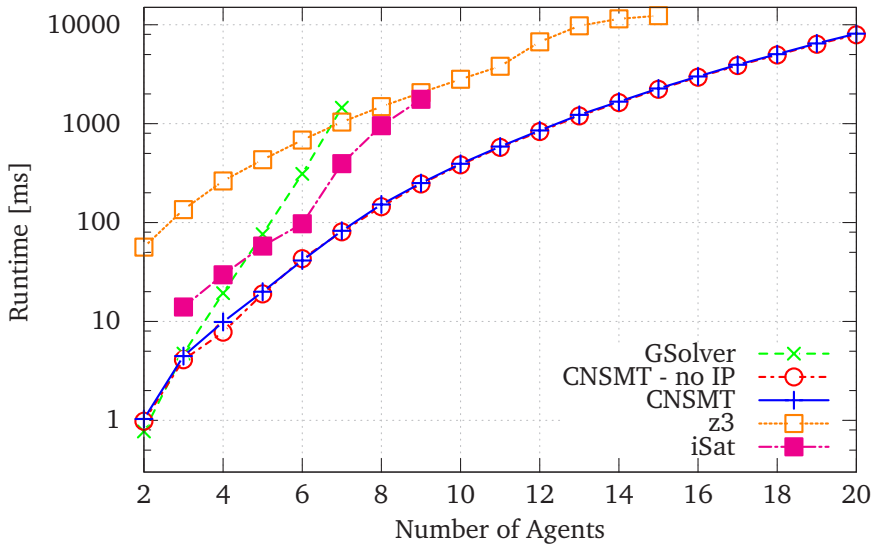



**Figure 10.4:** Required runtime to solve constraint Equation 10.5, which provides the most complex arithmetical structure.

agents with the formulation of Equation 10.4 according to the scalability. Furthermore, in all three formulations, the interval propagation algorithm did not show any impact on the runtime of the CNSMT solver. We suspect that this is caused by the usage of the Euclidean distance measure, which provides a gradient landscape with a strong indication towards the solution. In particular, the solution points only have the weak dependence to each other: inequality. Thus, the contribution of the interval propagation for the local search is almost neglectable. This experiment shows that the CNSMT solver induces low runtime overhead and benefits from a logical problem structure. This allows to outperform other approaches, especially for small problem instances. Furthermore, problem settings that allow the inference of expressive clauses by the *T solver* are beneficial for the runtime of the CNSMT solver.

## 10.3 Communication Chains

A typical problem in emergency response scenarios is the lack of a sufficient network infrastructure, e. g., caused by broken routing nodes. As a result, important network nodes may be cut off from the network connection. One solution for the establishment of the network connection is the setup of mobile network nodes, which are able to forward network packets. Therefore, we rely on a mobile robot, which can transport and install these routing nodes. However, the question question about which locations between the base station (origin) and the destination node are ideal for the placement of the routing node arises. Here, we have to respect the maximum communication range of the routing nodes and the surface of the environment, as some areas might not be accessible to the mobile robot such as mountains or valleys. The scenario setup is depicted in Figure 10.5.

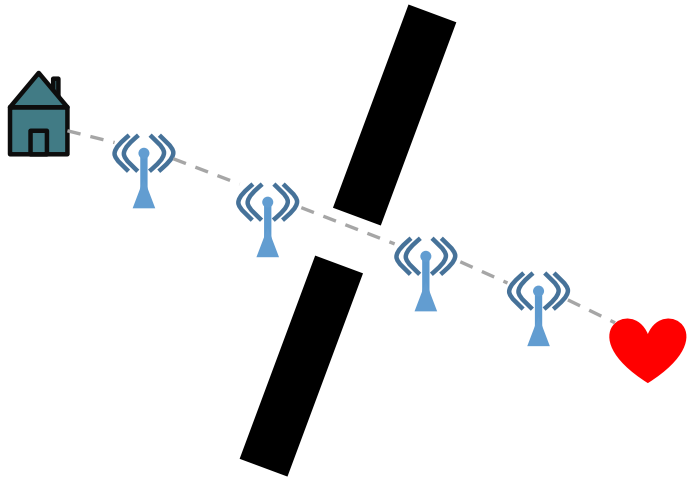


**Figure 10.5:** Setup for the communication chain scenario where the position of four routing nodes has been determined to establish a connection between a base station and a target node. The possible positions are constrained by the two inaccessible areas and the communication range limitation of the nodes.

If we formulate the node placement problem as a constraint expression, we assume the presence of a two-dimensional map with a fixed coordinate frame, e. g., through GPS positions. The expression has to describe the target points $P_n$ for $n$ nodes between a base station $b$ and the target node at a goal position $g$ inside an area of $100(n+2) \times 100(n+2)\,\mathrm{m}$ size. For each number of nodes, we repeated the experiment 1,000 times. In every run, we randomly selected the relative angle between $b$ and $g$.

We assume the communication range of all nodes to be equal and limited to $100\,\mathrm{m}$. Therefore, two consecutive nodes require a position within that distance:

$$|p_i - p_{i+1}|_2 < 100\,\mathrm{m}, \tag{10.6}$$

where the last node position has to match the destination: $p_n = g$. Since the distance between $b$ and $g$ is set to $90 \cdot n\,\mathrm{m}$, it is ensured that a feasible solution exists. At the same time, the discrepancy of only $10\,\mathrm{m}$ between the maximum communication range and the average distance a single node has to bridge shrinks the solution space. Additionally, the first node requires contact to the base station:

$$\bigvee_{i=1}^{n} |p_i - b|_2 < 100\,\mathrm{m}. \tag{10.7}$$

As already mentioned, we define areas where nodes cannot be placed, as they are not accessible for mobile agents. The shape of these areas is simplified to two rectangular shapes $\mathrm{ob}_j$, which is a simple approximation to their convex hull. This convex hull is described by surrounding straight lines $y = mx + b$. We limited $m$ to the interval $[-1000; 1000]$ to avoid numerical problems. Both areas are placed in the middle between $b$

and $g$ with a size of $50 \times 500\,\mathrm{m}$. Thus, we added the following to the constraint:

$$\bigwedge_{i=0}^{n} \mathrm{Outside}(p_i, \mathrm{ob}_1) \wedge \mathrm{Outside}(p_i, \mathrm{ob}_2), \tag{10.8}$$

where *Outside* is implemented as:

$$\begin{aligned}\mathrm{Outside}(p, \mathrm{ob}) := p_y < m_1 p_x + b_1 \vee p_y < m_1 p_x + b_2 \\ \vee p_y < m_2 p_x + b_3 \vee p_y < m_2 p_x + b_4,\end{aligned}$$

where $m$ and $b$ refer to the lines surrounding the obstacle $\mathrm{ob}$.

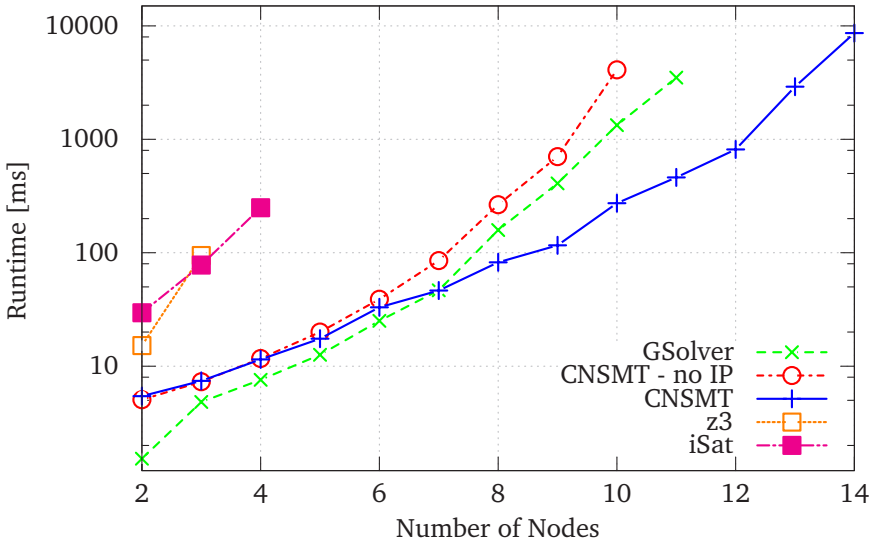



**Figure 10.6:** Required runtime to compute the target positions for network nodes to re-establish the network between a base station and a cut off network node.

Figure 10.6 shows that the interval propagation algorithm outperforms all other approaches for more than 7 nodes. In particular, the CNSMT approach without interval propagation requires a higher runtime for more than 3 nodes. We suspect that this runtime behavior originates from the strong relation between the computed target positions, which forms a more complex search space than the previous examples. In conclusion, the propagated intervals significantly improve the initial seeds for the local search solver, which causes a more efficient computation. It is also remarkable that only the z3 and iSat solvers can solve the problem within less than $10\,\mathrm{s}$ for up to 3 or 4 nodes, respectively. We assume that this is caused by the lack of an explicit search strategy for the given problem setting, which addresses the high amount and diversity of arithmetical operations.

## 10.4 Constraint Optimization

If constraints determine the goals for homogeneous robots, it is common that more than one solution exists. In many cases, a specific agent can still address a certain

target more effectively than another one, e. g., because it is located closer to the target position, which allows the agent to fulfill a task with less travel time. Although this is not the focus of our approach, the CNSMT solver is able to incorporate a cost or a utility function, respectively, in addition to the constraint expression, which determines the optimal solution, as described in Section 6.5.5. To our present state of knowledge, the only approaches that are able to solve this problem class are the GSolver and both CNSMT solver versions. Therefore, we cannot compare our results with other approaches.

If we consider the task assignment example of Section 10.2, it might be useful to minimize the average travel distance of each robot. We selected the best performing constraint formulation, Equation 10.4, and added the following utility function to create a constraint optimization problem:

$$U(\vec{t}, \vec{x}) = nc_{\text{maxDist}} - \text{dist}(\vec{t}, \vec{x}). \tag{10.9}$$

Here, the $n$ current physical positions are encoded by $\vec{x}$, and $c_{\text{maxDist}}$ denotes the maximum possible distance between objects in the $10 \times 10$ km target area, which is $14.14$ km. Thus, the value $c_{\text{maxDist}}$ ensures that the utility expression returns a value greater or equal to zero. As described in Section 6.5.5, this condition is required for our approach. Due to the negative sign of the distance between $\vec{t}$ and $\vec{x}$, a maximization of the utility function leads to a minimization of the distances.

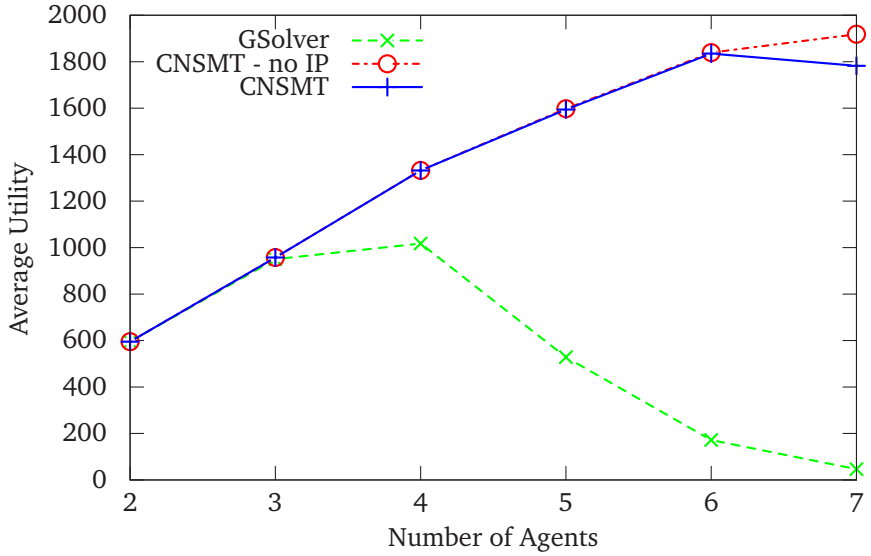



**Figure 10.7:** Average utility value when solving the constraint optimization problem according to Equation 10.9

As the solution quality of constraint optimization problems depends on the available runtime, we limited the runtime to $30$ ms. This is a typical available computation time for robotic soccer applications with dynamic environments. In our problem setting, we can measure the solution quality by the final utility value, which is shown for two to seven target positions and agents in Figure 10.7. It shows that our approach outperforms the

GSolver for more than three agents, as the CNSMT solver achieves a higher utility value. It is worth mentioning that for less than six agents the computed solution also determined the globally optimal solution.

## 10.5 Summary

This chapter analyzed the performance of our proposal computation algorithm. Thereby, we focused on the solution quality and runtime requirements for different types of problems. In the presented experiments, our approach showed a low runtime overhead, which makes it particularly useful for smaller problem instances, e. g., with less than $20$ free variables. Furthermore, at the price of incompleteness, we provide a solution that can deal with nonlinear expressions, which is an exception in the field of SMT solving. Although other solvers such as the iSat solver are also capable to handle geometric functions like sine or cosine, our approach resulted in a better runtime performance in almost every experiment we conducted.

For simple arithmetic expressions as in Section 10.2, the performance strongly depends on the logical problem structure. Due to the underlying SAT solver, the CNSMT solver is able to outperform the pure gradient search solver if a proper logical structure is present. For example, a 2-SAT problem description provides this kind of structure.

The impact of the interval propagation algorithm seems to strongly depend on the relationship between the variables. For problems with almost independent variables, the interval propagation method does not influence the runtime. In contrast, the communication chain example showed a speed-up if the number of free variables exceeded 12. Finally, we evaluated the solution quality for a fixed runtime concerning a constraint optimization problem. This experiment showed a high solution quality if the number of free variables was small.

In this chapter, we applied the CNSMT solver on in total 73,000 problem instances. Despite the use of incomplete local searches as a key component of the CNSMT solver, all problem instances were solved without producing a single false result. This result gives a strong evidence for the applicability of our solution approach. Note that we assume a dependence on the investigated problem settings. In particular, all problems were satisfiable, which we consider as the targeted use case for the CNSMT solver.

The second important part of a team decision process – besides the efficient computation of possible solutions – is the negotiation of the proposals. Therefore, our CNSMT solver is embedded in our decision process, which incorporates the PROViDE communication middleware to negotiate the solution. The performance of this negotiation part is evaluated in the next chapter.

# 11 PROViDE Negotiation Performance and Scalability

We consider the required time from the initial proposal until a value is successfully negotiated as the most important performance metric for the negotiation process. Therefore, we start the evaluation of the PROViDE middleware by investigating the two parameters that determine the negotiation time: First, we examine the latency induced by the middleware required for the information processing, e. g., to access the local data storage and apply the acceptance method, as described in Section 11.1. Second, we investigate the transmission time to apply the negotiation. This is mainly influenced by the network properties, e. g., by network delay.

An important property of PROViDE is its capability to deal with unreliable communication. To investigate PROViDE's tolerance towards network errors, we examine the influence of packet loss on the proposal replication process. In Section 11.2 and Section 11.3, we evaluate the required bandwidth and the time that is required until a proposal is successfully replicated. To control the network parameters, we rely on the Linux *NetEm* [75] network emulator. *NetEm* simulates artificial network delay and packet loss.

The bandwidth scalability of the PROViDE middleware depends on the selected distribution method and the number of agents. Section 11.4 and Section 11.5 show an analysis of these factors. Furthermore, the provided results depend on the estimation of the probability model for the network connection, which is examined in Section 11.6.

## 11.1 Latency

The PROViDE middleware implementation focuses on a lightweight design to reduce the transmission latency. In conclusion, we incrementally reworked the architecture to reduce the number of system calls and identified operations that allow parallel processing, for example the parallel processing of tasks. Additionally, we used the profiling tool *Valgrind* [152] to identify and resolve the source code parts with high runtime requirements. As a result, the latency of the final implementation mainly depends on the network stack of the operating system. As the operation speed of most network stacks depends on the packet size of the transmission, the latency of PROViDE scales with the size of the proposals. Thus, we distributed proposals of varying network sizes and measured the latency. Thereby, we considered the time between the distribution command and the call of the change subscription at the remote instance as latency. Our experiment focuses on the latency caused by the middleware and ignores the network delay. Therefore, both

PROViDE instances ran on a single host with an Intel(R) Core(TM) i7-2630QM CPU with 2 GHz. We used the Ubuntu 14.04 operating system with a 3.13.0-30-lowlatency kernel.

We compared the results with the latency of OpenSplice DDS[1] implementing the data distribution service standard to provide a state-of-the-art middleware. Since DDS is a data-centric approach, its distribution properties are similar to our approach although it does not explicitly support a negotiation process. Furthermore, we measured the latency of a vanilla UDP BSD socket, which forms the baseline of our comparison. Recall that the PROViDE implementation relies on BSD sockets to realize the low-level communication. Hence, it allows the determination of the computational offset caused by PROViDE.

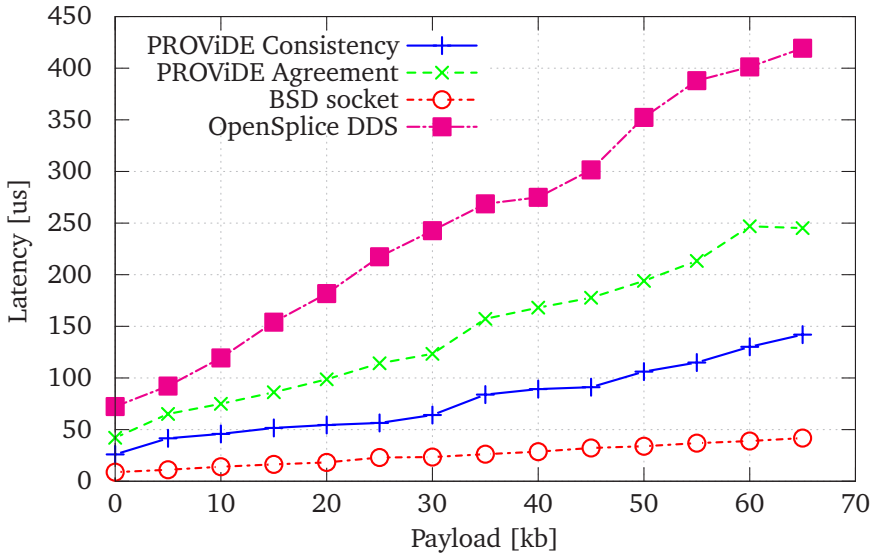



**Figure 11.1:** Latency comparison between PROViDE, OpenSplice DDS, and a BSD socket.

Figure 11.1 shows the results of the latency evaluation averaged over 1,000 trails per data point. Here, *PROViDE agreement* indicates that the proposal replication was successful and the acknowledgment was received. Thus, it includes two message transmissions in contrast to *PROViDE consistency*, which only measured the time to successfully replicate the proposal. As a result, the latency for reaching agreement approximately doubles the pure replication time of a proposal.

The comparison between PROViDE and OpenSplice DDS reveals that OpenSplice DDS is on average 3.13 times slower. Thereby, we observed the same latency for DDS independent from the usage of *reliable* or *best-effort* transmission modes. However, the BSD socket outperforms the latency of PROViDE consistency by a factor of 3.1 on average. This offset is caused by the more efficient internal variable management of PROViDE. In particular, some critical sections have to be secured by *mutex variables* to avoid errors caused by concurrent memory access. The measured latencies of BSD sockets confirm the results of [15]. This also constitutes the assumption that the computational power of the host does only have a small impact on the overall latency.

[1] OpenSplice DDS Industry Solutions (2009) Prismtech

## 11.2 Impact of Packet Loss on the Replication Process

In Section 1.4, we identified fault tolerance as key requirement for multi-robot decision processes. One of the most likely types of faults in adverse environments is packet loss. Because of the consecutive retransmissions of lost packets, PROViDE induces additional communication delay, which increases with the probability of packet loss. In order to investigate this relation, we connected three robots with a $1.6\,\mathrm{GHz}$ Intel(R) Core(TM)2 Duo CPU using a $100\,\mathrm{MBit/s}$ Ethernet connection through a network switch. There is no other network traffic in the network except for the transmissions of PROViDE. This creates a test environment, which can be controlled with *NetEm* inducing artificially drop or delay network packets. The network setup is shown in Figure 11.2.

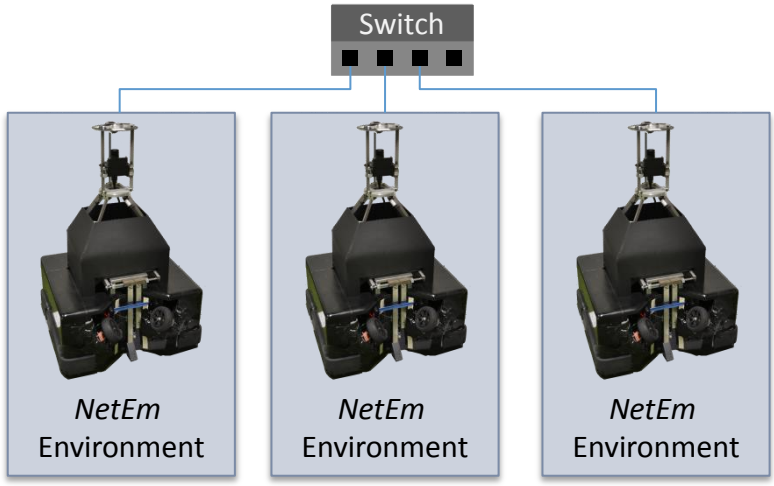



**Figure 11.2:** Experimental network setup.

In order to create network conditions that are similar to wireless networks, we simulated a network delay of $10\,\mathrm{ms}$ for all network connections, i. e., according to our assumption of Section 7.3.1, all connections are symmetric. The cable network itself induces small amounts of packet loss (less than $0.01\,\%$) and delay (less than 0.5 ms). We consider their impact as almost neglectable in this setup.

In our experiment, we evaluated the ability to consistently replicate proposals and the appropriateness of the estimated resend time. Therefore, one robot is set up as a *master*, which iteratively distributes new variable values to both *slave* robots using the *monotonic accepts* distribution method. We chose a high enough interval between two consecutive transmissions to avoid interferences. In the experiment, the number of maximum transmission retries is limited to $10$. Furthermore, we used a probability of $P(D_e \leq t_r) = 0.75$ for the calculation of the resend time $t_r$.

The time to achieve agreement for a proposal with 10 bytes and different amounts of artificial packet loss is averaged over 1,000 distribution processes per data point and depicted in Figure 11.3. Here, agreement is present after both slaves successfully transmitted their acknowledgment messages to the master. As the number of message

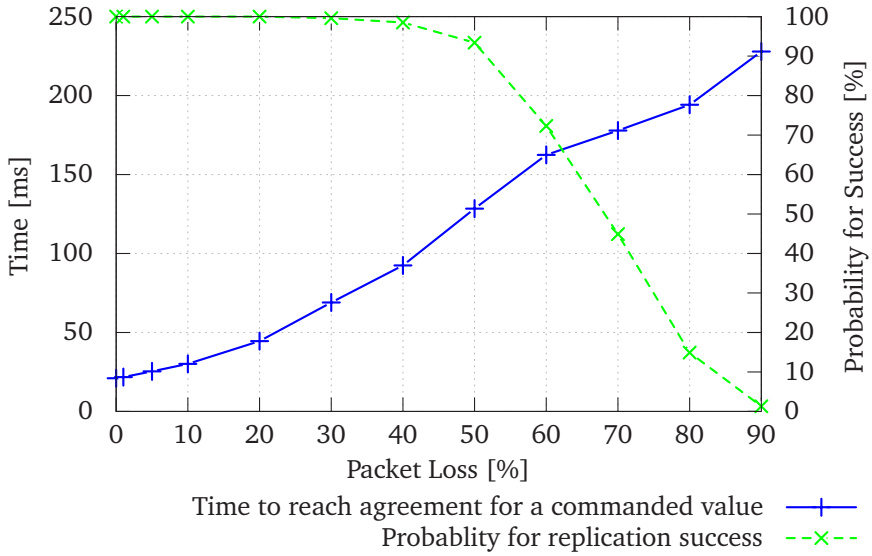



**Figure 11.3:** Influence of packet loss on the replication of PROViDE proposals.

retransmissions is limited to 10, the replication process fails more likely with an increasing packet loss rate. This relationship is shown by the probability of achieving replication success.

The results from Skubch [157] show that ALICA provides a well-coordinated behavior if the network delay does not exceed 250 ms and the packet loss rate remains below 50 %. In conclusion, our results show that the usage of PROViDE improves the usability of ALICA for networks with packet loss rates of up to 67 %, as we can still achieve a success rate above 50 % with a transmission delay below 250 ms.

## 11.3 Message Runtimes

PROViDE's opportunity to subscribe to proposal changes allows to transmit messages similar to publish-subscribe middlewares. Therefore, a message transmission is implemented as the replication of a robot proposal where the proposal name identifies the topic. Thus, the fire and forget distribution has properties similar to standard UDP transmissions. In contrast, monotonic commands distribution increases the probability of a message arrival for reliable message transmissions. The question about which transmission times can be achieved by the monotonic demands distribution method compared to TCP or sophisticated middleware approaches in networks that provide packet loss. In order to collect appropriate data, we again applied the setup of Section 11.2.

The results for the message transmission times are depicted in Figure 11.4. Each data point is averaged over 1,000 trials. In every trial, we measured the time difference from sending a message until it appears at the receiver. The results show that PROViDE requires

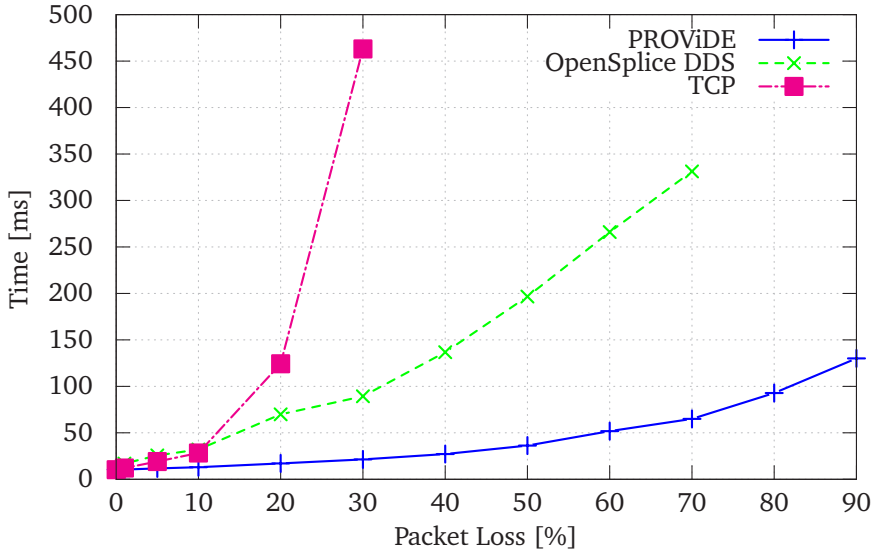



**Figure 11.4:** Comparison of the message transmission times of TCP, OpenSplice DDS, and PROViDE.

less transmission time than OpenSplice DDS and TCP. One reason for this result is the fact that our resend mechanism determines the parameters for the network connections dynamically and is therefore adjusted to the network setup. Furthermore, the focus of PROViDE is geared towards quick decisions. In contrast, TCP fosters collision avoidance. However, we assume a small number of simultaneously sending agents. Thus, collisions are unlikely in our targeted applications. In particular, TCP achieves poor results for consecutive packet losses. For packet loss rates above 30 % the network connection brakes down, prohibiting the communication. From this fact, we conclude that TCP connections are not suitable for robotic applications promoting unreliable network connections. In the same fashion, OpenSplice DDS also stops the delivery of transmissions for packet loss rates over 70 %. However, this result should suffice for most robotic scenarios.

## 11.4 Bandwidth Requirements in the Presence of Packet Loss

The distribution methods of PROViDE have different bandwidth requirements caused by the different amounts of mandatory transmissions. As each mandatory transmission is guaranteed by the resend mechanism, each distribution method scales differently in the presence of packet loss.

In order to determine the required bandwidth in relation to the induced packet loss, our experimental setup again relies on Section 11.2. Here, we additionally tracked the number of transmitted bytes for the three PROViDE instances and averaged the results over 1,000 distribution attempts. Thereby, we excluded the transmissions of the *time synchronization*

*messages*, as these only depend on their synchronization frequency and the duration of the experiment.

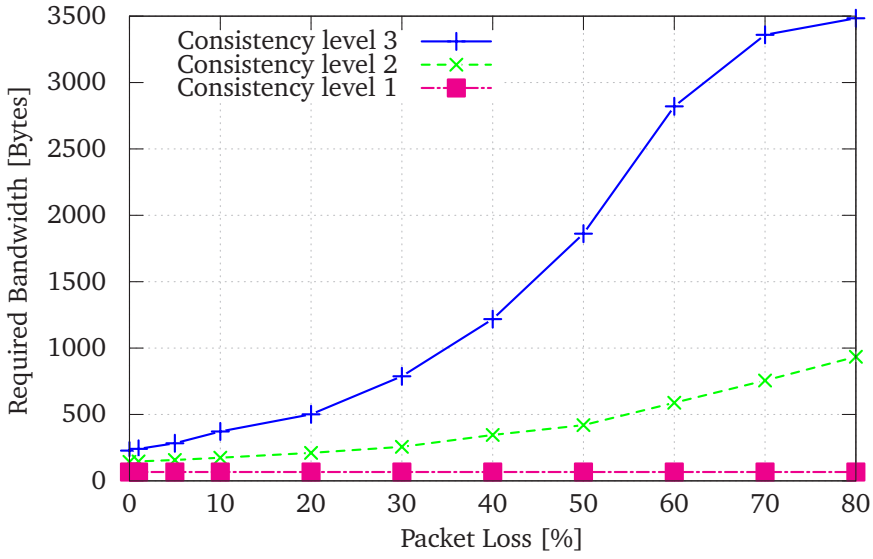



**Figure 11.5:** Required bandwidth of different distribution methods relative to the packet loss rate.

As shown in Figure 11.5, consistency level 3 causes exponentially growing bandwidth requirements for increasing packet loss rates. This exponential growth is only slowed down by the retry limit for lost messages, i. e., the success of the distribution process shrinks with packet loss rates above 50 %, which limits the bandwidth to an upper bound. In contrast, the bandwidth requirements of consistency level 2 scale linearly with the amount of packet loss, as the probability for resends also grows linearly. Finally, we can identify that consistency level 1 is not affected by packet loss, as no retransmissions occur. However, at the same time, the success rate of consistency level 1 corresponds to the packet loss rate.

## 11.5 Scalability

The scalability of the PROViDE middleware depends on the selected distribution method, the number of agents in the team, and the number of lost messages due to network errors. In the previous sections, we already investigated the influence of lost network packets on the required bandwidth. In the following, we analyze the impact of the distribution method and the amount of agents in the team.

If we neglect the possibility of packet loss, the proposal distribution to $n$ robots with the monotonic accepts distribution method requires

$$(c_{\mathrm{Cmd}} + (n-1)c_{\mathrm{Ack}} + (n-1)^2 c_{\mathrm{SAck}}) \text{ bytes}, \tag{11.1}$$

where $c_{\mathrm{Cmd}}$ is the size of a command message, $c_{\mathrm{Ack}}$ the size of an acknowledgment message, and $c_{\mathrm{SAck}}$ the size of a short acknowledgment message. Due to the quadratic term for the short acknowledgments, monotonic accepts distribution is only appropriate for a small number of robots (i. e. for IEEE 802.11abg networks $n \ll 20$). In particular, robot engagements represent a critical situation, as this might initiate multiple simultaneous notification tasks to restore proposal consistency.

We can similarly determine the required bandwidth for a variable update of monotonic commands by

$$(c_{\mathrm{Cmd}} + (n-1)c_{\mathrm{Ack}}). \tag{11.2}$$

This equation grows linearly with the amount of agents and packet loss. In contrast, the required bandwidth of fire and forget is constant $c_{\mathrm{Cmd}}$ due to the broadcast transmission. Note that the equation for fire and forget is independent of the probability of packet loss, which supports the results of Section 11.4. Since monotonic commands distribution suffices for most applications while providing a high success rate, we consider it as default distribution method for our decision process. We evaluated these equations by measuring the required bandwidth for the distribution of an empty proposal value.

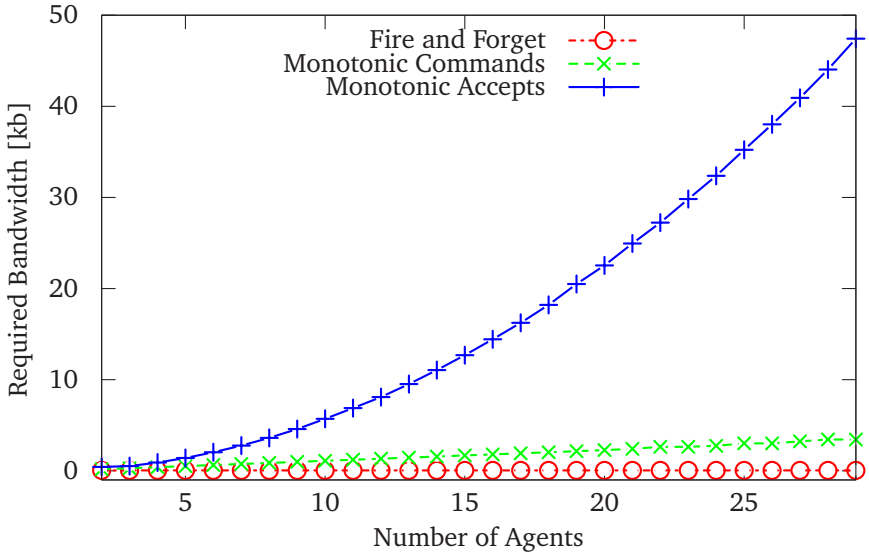



**Figure 11.6:** Comparison of the required bandwidth of three PROViDE distribution methods.

Remote write or read requests for proposals with local variable management require $(c_{\mathrm{Req}} + c_{\mathrm{Var}})$ bytes. In this case, each request with $c_{\mathrm{Req}}$ bytes is acknowledged with the current value proposal requiring $c_{\mathrm{Var}}$ bytes. This corresponds to the asymptotic requirements of TCP.

In the presence of network faults, the resend mechanism causes a growth of the required bandwidth. Here, the retransmission can either be caused by a loss of the original transmission or the acknowledgment, i. e., when the acknowledgment gets lost, two

messages require retransmission. To analyze the impact of this mechanism, we evaluated the required bandwidth with a varying number of agents in the presence of artificial uniformly distributed packet loss by distributing an empty proposal.

As depicted in Figure 11.7, packet loss has an exponential impact on the amount of required bandwidth of the monotonic commands distribution method. However, even for 30 agents and a packet loss rate of 50 %, the required bandwidth does not exceed 40 kb to successfully distribute an empty proposal. In contrast, Figure 11.8 shows that the monotonic accepts distribution method requires a bandwidth increased by an order of magnitude. These results are averaged over 100 distributions per data point.

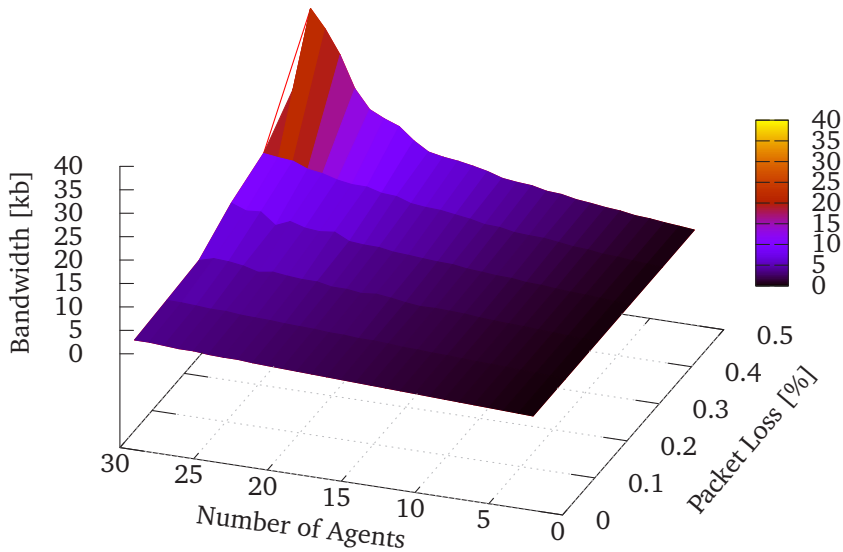



**Figure 11.7:** Required bandwidth of the monotonic commands distribution method in the presence of packet loss.

## 11.6 Communication Model Estimator

The previous experiments already included an implicit evaluation of the network estimation, as the resend time relies on the estimated network delay. Nevertheless, it is our goal to confirm the network assumptions of Section 7.3.1. In particular, we gather data that supports the hypothesis of a Laplace distribution of the network delay jitter as discovered by Zheng, Zhang, and Xu [181] and Edward J. Daniel, Christopher M. White, and Keith A. Teague. [51].

In this experiment, we set up a static network with two PROViDE instances, which are directly connected via a 100 Mbit Ethernet connection. With *NetEm*, we induced 40% artificial packet loss and a delay of 10 ms. In this setup, the master instance is distributing proposals with a frequency of 2 Hz. The proposal value contains the timestamp of the

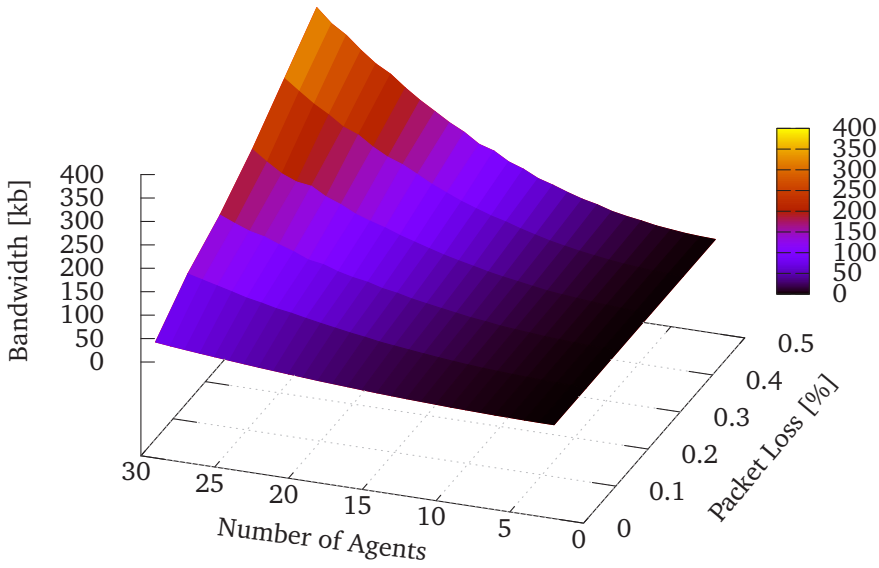



**Figure 11.8:** Required bandwidth of the monotonic accepts distribution method in the presence of packet loss.

synchronized clock indicating the time of the proposal distribution. The receiving slave instance computes the difference to its local time, resulting in the *inconsistency duration* of the proposal. For an accurate estimate of this inconsistency duration, both clocks are synchronized using the precision time protocol (PTP) [96], as PTP can achieve a precision of less than $0.5$ ms in similar network setups according to Lee and Eidson [96].

We averaged the collected data over 10,000 transmissions and computed the cumulative probability of a proposal remaining consistent with respect to the inconsistency duration. Additionally, we collected the converged consistency probability estimation, which is determined by the master instance, according to Section 7.3.4, from the master instance. Figure 11.9 compares both results including the estimation error computed by the difference between both functions. The results show that PROViDE underestimates the steepness of the rising edges of the distribution. This results from an overestimation of the diversity of the underlying Laplace distribution. Overall, the estimation error never exceeds 10 %, which gives evidence for the correctness of our assumptions and the precision of PROViDE.

## 11.7 Summary

In this chapter, we showed extensive evaluations for the PROViDE middleware framework. In particular, we found out that the induced latency is lower compared to other middleware approaches such as OpenSplice DDS. Furthermore, we evaluated the performance of PROViDE in environments with unreliable and delayed communication. Here, we

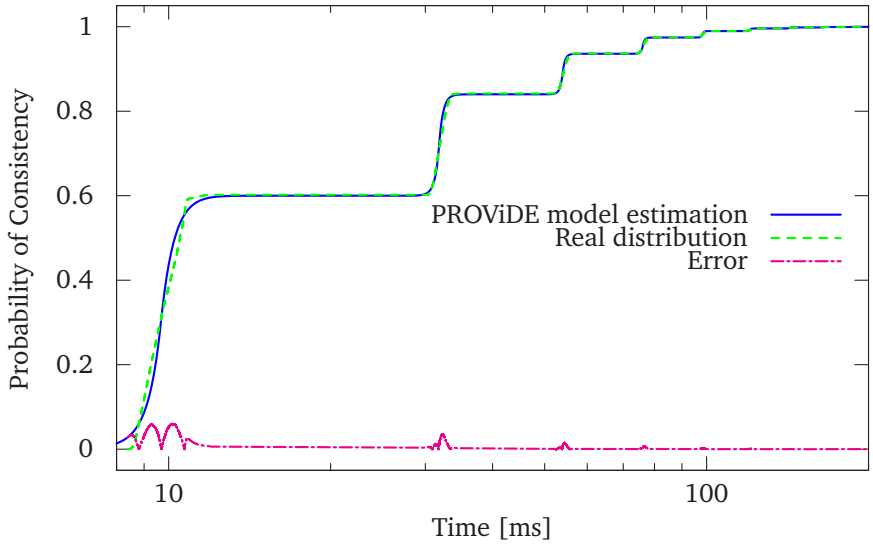



**Figure 11.9:** Real and estimated probability for consistency for a packet loss rate of 40% and 10 ms delay in a 100 Mbit Ethernet network.

investigated the success rate, bandwidth, and inconsistency duration of the replication process. From these results, we can conclude that PROViDE is suitable for scenarios with unreliable communication. Furthermore, TCP seems to be inappropriate, as network connections tend to break down if the packet loss rate exceeds 30 %. Afterwards, we provided an investigation of the built-in communication model estimation. This experiment showed that the estimation errors are neglectable and supports the *a priori* assumptions regarding the network model.

# 12 Application Examples

In this chapter, we give an overview of possible application examples of a PROViDE-based negotiation. These demonstrate the adaptability and expressiveness of our decision process and the PROViDE middleware. Each of the examples has already been solved by at least one algorithm of distributed systems research. However, the usage of PROViDE reduces the required experience of the developer by unifying the implementation of these algorithms in a single framework. Therefore, PROViDE implements variable types for the described algorithms. Note that all applications could also be implemented in a centralized fashion where one coordinating agent manages crucial values using the local variable management distribution method. Other agents can remotely request and write these values with strong consistency properties. However, we aim for a more distributed solution that does not induce a single point of failure.

## 12.1 Whiteboard

In the robot soccer domain, we experienced the need for the distribution of strategy-related knowledge among the agents. Often, this knowledge does not describe observations but relies on the intentions of the agents. For example, whenever a robot wants to execute a pass, the receiving agent requires a notification of the pass destination. This allows the receiver to adjust its location towards the destination point without having a visual perception of the ball position or its velocity vector. After the execution of the pass kick, the passing robot can transmit a refined destination position based on its local observation of the ball direction. In both cases, the pass receiver adjusts its position to intercept the pass. As a result, the knowledge improves the pass success rate and execution time. This scenario is depicted in Figure 12.1.

A property of these pass destinations is their volatileness: Due to the dynamic robot movement (with up to 5 m/s), the characteristics of the current situation change within a few seconds. In conclusion, the analysis of new sensor data usually leads to a change of the optimal destination point. This data is gathered with a frequency of 30 Hz. Hence, distribution methods such as monotonic commands would lead to an unnecessary overhead. Here, PROViDE would wait for acknowledgments referring to data that is almost outdated at the time of arrival. Furthermore, the incremental update of the destination causes new transmissions with a frequency of 30 Hz. As a result, a single packet loss delays the receiver by 33.3 ms. In conclusion, we used fire and forget consistency to replicate the pass position proposal. In this scenario, only the passing agent distributes a proposal. Therefore, the other agents simply have to accept the proposals using the happened-after acceptance method and then select their own proposal to achieve coherent behavior. Since

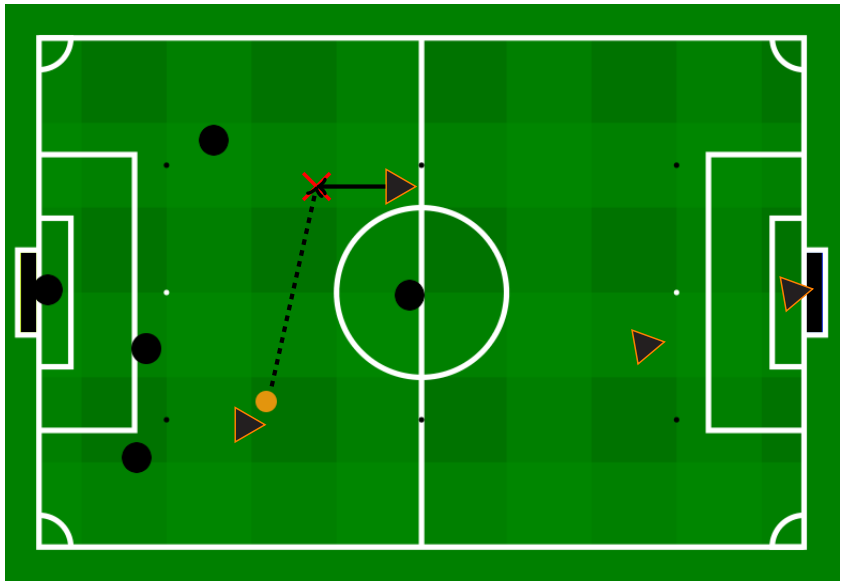


**Figure 12.1:** Pass execution in the RoboCup MSL. Observed obstacles are marked in black, the ball position in orange, and team members as triangles. The executing robots use a PROViDE whiteboard to transmit the destination point of his pass to the receiver, which uses this information to move towards this position.

the decision value is distributed by a single origin, we consider this scenario as the most elementary use case for PROViDE. However, PROViDE allows to extend this use case easily for the situation when the receiving agent can contribute reservations in form of a different destination proposal, e. g., if the receiver makes contradictory observations.

A second value, which we post on the whiteboard, is a Boolean value that indicates a ball movement during a soccer standard situation: In MSL standard situations, the team not in ball possession is only allowed to approach the ball after it started to move, e. g., by a kick. In our team *Carpe Noctem Cassel*, we can detect the ball within a distance of approximately 8 m if it is in the line of sight. However, during standard situations, many opponents try to interrupt the visibility of the ball, which complicates the detection of the ball movement. In conclusion, we defined at least one robot as a distinct ball spectator trying to maintain ball visibility. Additionally, other robots within a distance of 5 m also observe the ball while following their task to disturb the opponent's strategy. This distance is empirically determined and ensures that disturbing opponents do not cause false movement detections. Here, the whiteboard value is by default set to *false*, indicating that the ball did not move. If a single robot detects a ball movement, it switches the value to *true*. This value stays valid until the end of the standard situation, which can last 10 s at maximum. Each robot closer than 5 m can distribute a valid proposal.

In this scenario, we can again select the happened-after acceptance method. An important difference to the pass decision is the fact that the *ball-moved* value is not updated and distributed iteratively. Instead, we distribute it once when the movement has been detected. At the same time, it is important that all robots receive the information that the ball moved,

allowing them to execute their defending behavior. Otherwise, the robots do not attack the opponents, which may allow them to score uninterruptedly. For this reason, the monotonic commands distribution method is used. Note that the write operation is idempotent during the execution of the observation task, as all agents only set the value to *true*. Accordingly, the negotiation task is rather simple.

The whiteboard application is characterized by the fact that usually no conflicting proposals appear and all the decisions are designated for a specific situation. This is the most common use case for inter-robotic communication besides the distribution of shared observation data. As a consequence, it is supported by almost all modern middleware products. A standard way to tackle whiteboard applications is to apply a data-centric communication such as in DDS [125]. Here, the QoS properties can be adjusted to achieve a behavior similar to the distribution methods of PROViDE. The major contribution of PROViDE in contrast to other approaches is the dynamic broadcasting mechanism, which is able to handle packet loss and adapts to the network infrastructure. The whiteboard was road tested at the RoboCup Portuguese Open 2014, where we achieved the third place and could conduct over 30 successful passes in all eleven matches.

## 12.2 Mutual Exclusion and Election

A more complex example than whiteboard applications is mutual exclusion. Here, the middleware needs to provide a negotiation process that resolves conflicts caused by simultaneous, i. e. causally independent, proposals. Thus, we use PROViDE to avoid a concurrent behavior execution. We can identify concurrent behaviors in many practical scenarios. For example, in robotic soccer, only a single robot is allowed to attack a ball-possessing robot, in an autonomous car driving scenario, only a single lane can pass a crossing, and in emergency response scenarios, a specific victim should only be rescued by a single robot in order to save resources.

One method to implement a *mutex variable* with PROViDE is an integer-typed variable, which indicates that an agent currently operates in a critical section. Such a critical section protects potentially conflicting behavior implementations from concurrent execution. Therefore, the variable is initialized with a number that is not used as a robot identifier, in our case $-1$. When a robot wants to enter the critical section, it distributes its own value using the distribution method monotonic accepts with the happened-after acceptance method. Thus, according to Section 7.4, the conflict handling ensures the negotiation of simultaneous proposals. More precisely, the proposal with the highest logical timestamp will gain acceptance when the distribution process is completed. In case two agents have the same logical timestamp, the agent with the higher number will enter the critical section. Resulting from the convergence properties of the monotonic accepts distribution method, all agents eventually identify the winner.

As a value decision method, we use unanimous agreement to ensure that no agent enters the critical section before all agents agree on it. The value of *mutex variables* cannot be overwritten except the agent's proposal is $-1$ or invalid, i. e., no value can be inferred. To enable other agents to enter the critical section, the winner agent announces when it leaves the critical section. Accordingly, it resets the *mutex variable* to $-1$. Only the agent

inside the critical section is allowed to do so. Note that the agents can avoid a deadlock situation by using a validity time, which is adapted to the expected time inside the critical section. Once the first proposal expires, it is not possible anymore to infer the winner, which allows the other agents to again propose their intention to enter.

In order to achieve quicker decisions, a *weak synchronization variable* can be used, which is less strict regarding the access to the critical section. Therefore, the weak agreement decision method, which only requires robots within communication range to agree on a winner, is used. Additionally, the required bandwidth can be reduced by selecting the monotonic commands distribution method. In that case, the winner can identify its allowance to enter the critical section. However, all other agents are eventually not able to determine the winner, as their acknowledgment transmission might be lost.

A different application for PROViDE is a distributed election process to select a distinguished robot for a certain task, e. g., in an emergency response scenario where several equally equipped robots are available to rescue a single disaster victim. Here, it is not important which robot is sent out for the rescue, but it is important that it is exactly one of the agents who is sent out and that all robots are aware of this winner. The *election variable* is similar to a *mutex variable* except for the fact that all agents intentionally distribute proposals simultaneously at the beginning of the election. The election start is usually triggered by some external event, e. g., a team of agents entering an ALICA plan.

## 12.3 Task Lists

Another common problem in multi-agent systems is the producer-consumer problem. In robotic applications, this problem appears in search and rescue missions if some robots are distinct searchers, while others are built to rescue a disaster victim. In this case, the search robots collect possible targets within a *task list*. Usually, an omnipresent and continuous communication channel does not exist within the team. Thus, the exchange of the list data is restricted to distinct situations, e. g., when the robots meet within communication range. As a result, the most operations are performed distributedly and concurrently.

The simplest form of a *task list* only allows agents to add items to the list. This can be realized using a PROViDE variable with list data types, monotonic commands distribution, and the collection acceptance method. Thus, it ensures that PROViDE delivers all command messages when communication is possible. Since no agent overwrites its own proposal based on the acceptance method, an explicit value decision method is not required and the inferred value is the list of all proposals. In such a model, an agent can only remove values of its own proposal list or transmit a proposal write request to the list owner. The latter is no safe operation, as this operation could overwrite local changes of the remote agent that have not yet been replicated.

To enable robots to remove or consume values from a list, we propose a second variable list, which includes all values that have been removed from the list. The drawback of this solution is that both list sizes increase monotonically. As a result, the bandwidth requirements of production and consumption grow over time.

## 12.4 Making Causally Dependent Decisions

The causal dependency of two decision variables $x$ and $y$, where $y$ is causally dependent on $x$, is given if the decision value of $x$ *happened before* the value of $y$ according to Lamport [91]. In the context of the PROViDE negotiation process, this means that a decision for $y$ cannot be inferred before the successful negotiation of $x$ has been performed. Assuming the own proposal value decision method, we can identify a sufficiency condition from Lamport's *happened before* relation:

$$Lt(\max(\mathrm{VP}(x)) < Lt(\max(\mathrm{VP}(y)) \quad \text{w.r.t.} \quad <_o, \tag{12.1}$$

i. e., the logical timestamp of the highest ranked proposal according to the ordering relation of the acceptance method of $x$ has a lower logical timestamp (*happened before*) than the one of $y$. For a general necessary condition that is not specific for a certain decision method, we have to recognize that the agent proposing $\max(\mathrm{VP}(y)$, was able to process $\max(\mathrm{VP}(x)$ before:

$$Lt(\mathrm{VP}_w(x)) < Lt(\mathrm{VP}_w(y)) \tag{12.2}$$

$$\wedge D(x) = \mathrm{VP}_w(x) \wedge D(y) = \mathrm{VP}_w(y), \tag{12.3}$$

where $D(x)$ indicates the value decision according to the decision method of $x$. We can conclude that the causal dependency of $y$ on $x$ if it holds

$$Lt(D(x)) < Lt(\mathrm{VP}_a(y)) \; \forall a \in \mathcal{A} < Lt(D(y)). \tag{12.4}$$

As a result, $D(y)$ becomes invalid if an agent proposes a new value for $x$. Note that these conditions do not ensure agreement or coherence for the decision. However, the decisions underlie the same convergence rules as described in Section 7.4. Therefore, we can eventually achieve convergence for a monotonic commands distribution method with a strict ordering relation $<_o$.

## 12.5 Consensus

PROViDE forms a framework that allows the implementation of complex network protocols such as the basic Paxos protocol [90]. Paxos is a two-phase protocol, where the first phase performs a leader election, while the second phase determines the consensus value. Each agent has at least one of the roles *client*, *proposer*, *acceptor*, *learner*, and *leader*. As setup for robotic scenarios, we assume that each robot in the team acts as *proposer*, *acceptor*, and *learner* simultaneously. The *leader* is always a *proposer*: Being the winner of the election phase, the leader is allowed to command a consensus value for the second phase. Multiple *proposers* can simultaneously believe to be the winner. The *client* role represents an agent issuing a request to the distributed system, e. g., to one or more of the *proposers*. As all robots act as *proposers*, this rule is obsolete in this setup.

As Paxos is a two-phased protocol and the PROViDE distribution methods only implement a single-phase communication, two proposal distribution steps are required: One step is needed to determine the *leader* in phase 1 and a second to distribute the consensus

value in phase 2. In the first phase, Paxos *proposers* send a unique proposal number to all *acceptors*. The *acceptors* respond with the highest proposal number they received before or, if available, with the consensus value. With the response, the *acceptor* promises to never accept a proposal with a lower identifier. After receiving a majority of responses without any rejections, the *proposer* continues with phase 2. If the *proposer* receives a rejection in form of a consensus value, it can conclude that consensus is already present for a concurrent proposal, and it adopts this value.

To implement the first phase of Paxos in PROViDE, we transmit an empty value proposal $\mathrm{VP}(x)$ with the monotonic commands distribution method. We implemented an additional acceptance method, which assumes the value type to be an $n$-tuple, where the first two values are a positive integer and represent the minimum proposal number of phase 1 or the number of the accepted proposal, respectively. Here, the proposals are ordered by the integer values:

$$\begin{aligned} <_P = \{(\mathrm{VP}_i(x), \mathrm{VP}_j(x)) \mid\ & (\mathrm{VP}_j(x)_2 \neq 0 \wedge \mathrm{VP}_i(x)_2 < \mathrm{VP}_j(x)_2) \\ & \vee (\mathrm{VP}_j(x)_2 = 0 \wedge \mathrm{VP}_i(x)_1 < \mathrm{VP}_j(x)_1) \\ & \vee (\mathrm{VP}_i(x)_1 = \mathrm{VP}_j(x)_1 \wedge Lt(\mathrm{VP}_i(x)) < Lt(\mathrm{VP}_j(x)))\}, \end{aligned} \tag{12.5}$$

where $\mathrm{VP}_j(x)_n$ is the $n$-th tuple element. If the proposal numbers are equal, the proposal with the higher logical timestamp is accepted, which allows an update by the leader in phase 2. Note that this acceptance method does only accept the highest proposal number if the agent has not accepted a consensus value before. Accordingly, a variable change callback additionally stored the maximum proposal number within the own proposal.

We ensure the uniqueness of the proposal number $\mathrm{VP}(x)_1$. It is composed of the logical timestamp of the *proposer* as the most significant bits and the robot number as the least significant bits. This acceptance method ensures that the robots will accept the highest proposal number. Afterwards, in a change subscription callback function, the *proposer* checks whether a majority of responses has been received, which implements a feedback of the decision process by initiating phase 2.

In phase 2 the Paxos *proposer* distributes its proposal number together with the value for which it requests consensus. Therefore, we assign the value to $\mathrm{VP}(x)_3$, if it has not already been set in phase 1. All *acceptors* respond with the highest proposal number they received earlier. If the *proposer* again receives a majority of responses, it once again checks for rejections, i. e., for all proposal numbers smaller than or equal to its own. If this is the case, the consensus has been established. Otherwise, the protocol restarts with phase 1 and a higher proposal number.

After receiving a majority of responses, a *proposer* $a$ can detect consensus by the fact that no response contained a proposal number greater than its own:

$$\begin{aligned} & \sum_{i \in \mathcal{A}} \mathbf{1}_{\{\mathrm{true,false}\}} \left(\mathrm{VP}_i(x)_2 > 0\right) \geq \frac{|\mathcal{A}|}{2} \\ \wedge & \sum_{i \in \mathcal{A}} \mathbf{1}_{\{\mathrm{true,false}\}} \left(\mathrm{VP}_a(x)_2 \geq \mathrm{VP}_i(x)_2\right) = 0, \end{aligned} \tag{12.6}$$

where $\mathbf{1}_{\{\mathrm{true,false}\}}$ is the indicator function, which returns 1 if its parameter is *true* and 0 otherwise.

If no consensus is present, the protocol restarts and chooses a new proposal value. As our implementation assumes that all *proposers* are also *acceptors*, any *acceptor* can perform the same check to determine the presence of consensus. Note that the only value guaranteed to satisfy consensus is $\mathrm{VP}(x)_3$ when selecting the own proposal decision method, which corresponds to a majority of $\mathrm{VP}_i(x)_3 \forall i \in \mathcal{A}$. The main algorithm is shown in Listing 12.1, which requires the callback function described in Listing 12.2.

**Algorithm** `InitPaxos(`*value*`)`
  Register($x$, `VariableChangeCallback`)
  **if** $a \in Proposers$ **then**
    proposalId := $Lt \ll$ sizeof(ID) + Id($a$)
    phase := 1
    $\mathrm{VP}(x)_a = \{$proposalId, 0, $\varnothing\}$
    DistributeProposal ($\mathrm{VP}(x)_a$, *Monotonic Commands*, $<_P$)
  **end**
  **else**
    $\mathrm{VP}(x)_a = \{0, 0, \varnothing\}$
  **end**
  **return**

**Listing 12.1:** Implementation of the basic Paxos consensus protocol using the PROViDE middleware.

```
Procedure VariableChangeCallback(x)
    VP_a(x)_1 := max(VP(x)_1 ∈ DB)
    if a ∈ Proposers then
        responsesPhase1:=0, agreeCount:=0
        for VP_i(x) ∈ DB do
            if VP_i(x)_1 ≥ proposalId then
                responsesPhase1++
            end
            if phase = 2 ∧VP_a(x)_2 < VP_i(x)_1 then
                InitPaxos(value)
            end
            if phase = 2 ∧ VP_a(x)_2 = VP_i(x)_2 ≠ 0 then
                if ++agreeCount > |A|/2 then
                    ConsensusReached()
                end
            end
        end
        if responsesPhase1 ≥ |A|/2 ∧ phase = 1 then
            forall the i ∈ A do
                if VP_i(x)_3 ≠ ∅ then
                    value := VP_i(x)_3
                    proposalId := VP_i(x)_2
                end
            end
            phase := 2
            VP(x) = {proposalId, proposalId, value}
            DistributeProposal (VP_a(x), Monotonic Commands, <_P)
        end
    end
```

**Listing 12.2:** Basic Paxos callback function for proposal changes.

# 13 Conclusion

In this work, we presented a decision process designed for teams of autonomous mobile robots. Inspired by human group decision making, we divided the decision process into five steps. Although each of these steps can appear in different forms, we proposed an implementation for each step and showed its applicability. The process starts with the identification of the problem. In the context of mobile robots, this corresponds with the definition of a proper problem description. In our solution, we propose the ALICA modeling language for this step, which embeds and executes our decision process. ALICA models cooperative behavior from a common perspective and provides conflict resolution methods for situations where agents have conflicting estimations concerning their current task. Thereby, ALICA allows to collect different partial problem statements into a single expression.

In the second step, our decision process identifies relevant solution proposals. Therefore, we proposed the CNSMT solver, which combines an incomplete solution technique for nonlinear constraint and optimization problems in an SMT-based approach. We showed that this unique combination allows for solving highly complex nonlinear problems if the underlying logical structure is suitable for SAT solving. However, even in case of arithmetical problems, our solution showed good runtime results compared with other state-of-the-art approaches. Thus, the CNSMT solver forms a universal solution, which is adequate for a variety of practical problems in the domain of autonomous robots.

The last three steps are supported by the PROViDE middleware. Therefore, the given proposals are first distributed among all available robots by using a selectable distribution method. The distribution method can be adapted according to the required distribution guarantees and decision time, i. e., the developer can chose between fast and unreliable methods for transient decisions, or slow and fault-tolerant distribution methods that guarantee convergence of the distribution process. Our protocol implementation uses a probability model to determine lost packets based on missing acknowledgments. The model is able to adapt to the network delay and packet loss rate. Thereby, our solution can reach faster transmission times then state-of-the-art middlewares if the transmission frequency makes collisions unlikely.

In the next decision step, each agent can support one of the replicated proposals. This means that the agent accepts this proposal as its own. For the selection process, the developer can choose an acceptance method that influences the convergence of the decision process to a specific proposal and the resolution of conflicts. In particular, our proposed method selects the highest ranked proposal with respect to an ordering relation. In this context, we showed that the negotiation process eventually converges if a strict ordering relation is selected.

The last step of the decision process determines the final decision value from the replicated

proposals. According to human decision-making, this allows to meet a common decision although not all agents support a certain proposal, e. g., a majority voting allows to outvote others and achieve a common decision. In the same fashion, it is possible to select the proposal that is most likely correct, e.g., selecting the most recent proposal or the proposal of an agent with the highest confidence in its observations.

Summarized, the decision process is supported by the CNSMT solver, which computes the solution proposals, and the PROViDE middleware, which implements the negotiation process to provide a profound basis to meet coherent decisions in the presence of adverse network environments. Thereby, we explicitly support network failures such as network delays and packet loss. Naturally, these environments prohibit guaranteed consensus according to the FLP proof [57]. Thus, state-of-the-art consensus algorithms such as Paxos try to maximize the probability to achieve consensus at the cost of liveness. However, many robotic domains such as autonomous driving or robotic soccer provide fast changing environments. Thus, a decision process has to enable swift decisions. Moreover, not all decisions require consensus guarantees, as the implementation of robots inherently tolerates errors. To address the trade-off between safety and efficiency, our framework allows the developer to adjust the negotiation parameters specifically to the requirements of the robotic application and decision.

## 13.1 Requirements Revisted

At the beginning of this thesis, we identified requirements for the decision process of autonomous mobile robots in Section 1.4. We inferred these from three scenarios: robotic soccer, autonomous driving, and emergency response. This allows us to collect the requirements from domains with a different degree of connectivity, safety, amount of decision incidences, and available time to meet a decision. In the following, we summarize how these requirements were solved.

**Full Distribution** The presented decision process operates without any central coordination component. Furthermore, the PROViDE middleware allows the network topology to split in arbitrary groups. Thereby, broken or separated robots are considered as additional network partition. Our decision process allows each partition to meet decisions according to the selected distribution, acceptance, and decision method.

**Incomplete Knowledge** Our solution does not imply a common knowledge base for all agents. In contrast, each agent can locally compute its own decision proposal and participate in the negotiation process. If a decision requires knowledge that cannot be gathered by a single agent, it can be requested from other agents in form of a causally dependent decision.

**Support for Dynamic Team Configurations** The disengagement of robots during the decision process is naturally covered by the selected decision method, i. e., an unanimous agreement blocks decisions for partitioned network topologies, while a majority of agents is still able to agree on a value. Hence, PROViDE allows for adjusting the decision processes to different team configurations.

**Robustness** PROViDE covers robustness against conflicting proposals by its built-in conflict resolution. In Section 7.4.3 in particular, we showed the ability of PROViDE to detect

and resolve proposal conflicts if the selected acceptance method is based on a strict ordering relation and the monotonic commands or monotonic accepts distribution method is selected.

**Fault Tolerance** The network protocol of PROViDE explicitly considers network faults. Therefore, lost network packets are detected and retransmitted based on the estimated network delay and packet loss rate. More precisely, PROViDE continuously observes the properties of the present network connections to determine a maximum waiting time for the acknowledgment of network packets. This allows the protocol to adapt to the network properties at runtime.

**Availability** Except for the local variable management distribution method, PROViDE automatically distributes all proposals within the team and keeps the replica up to date. As a result, all agents within communication range of a proposing agent can access the proposal locally after the successful replication. Due to the data replication, PROViDE fosters the availability in the local memory of the agents. Moreover, the persistent storage of proposals allows access even if the communication link to the proposal origin vanishes.

**Efficiency** In our decision process, efficient decision-making is achieved by three main factors: First, as shown in Chapter 10, the CNSMT solver provides low computation times to determine proposals and re-evaluate previous solutions for constraint satisfaction and optimization problems. Second, PROViDE by default avoids feedback loops to realize short negotiation times. Third, as shown in Section 11.1, the PROViDE implementation induces less latency than other state-of-the-art approaches such as OpenSplice DDS.

**Adaptability** Due to the support by PROViDE, the application developer can adapt the negotiation parameters to the given problem instance. This allows for fast decisions in dynamic scenarios, where decisions are rather transient, or safe decisions with longer decision times if safety has a higher priority.

## 13.2 Future Work

In this thesis, we presented a comprehensive decision process that is supported by a sophisticated middleware framework. Nevertheless, some research questions have not been addressed and are open for future investigation:

**Position-based network estimation** PROViDE uses an abstract model of network connections and does not involve all available knowledge to update the network estimates. In particular, the knowledge about the positions of all team members within a given map and the knowledge about the utilized communication devices can be used to improve the probability function that underlies our network model. Here, physical objects and the positions of the robots have an important impact on the quality of the communication link. Future research could analyze and integrate position-based models.

**Dynamic changes of negotiation parameters** The distribution, acceptance, and value decision method are currently defined by the application and have to be consistently

chosen by all participants. Thus, we assume the definition of these parameters by the multi-agent program in order to ensure consistency. However, if a scenario requires dynamic parameter changes, a by-pass mechanism is required, which allows robots to change the negotiation parameters. Here, it is especially important to consider parameter changes during the absence of agents and to elaborate a conflict resolution mechanism if such agents re-engage the team.

**Conflict resolution for constraint satisfaction problems** In our approach, we provided mechanisms for coherent decisions in all parts of the decision process except the problem statement. Thus, robot proposals can rely on conflicting problem statements. Furthermore, clauses learned by the CNSMT solver are not involved during the negotiation process. Petcu [129] presented a possible solution for this issue, which incorporates additional communication between the robots. For the integration into our decision process, the approach needs to be enriched with support for nonlinear constraints, dynamic domains, and possible network faults.

**Integration of additional theorem solvers** We optimized our CNSMT solver for general nonlinear problems. However, the incorporation of additional assumptions of the problem domain can significantly reduce the solution time for specific problem settings. Therefore, it is possible to borrow ideas from the z3 solver of Microsoft [110], which also supports empty theory, linear arithmetic, and quantifiers.

**Semantic matching of proposal values** Currently, equality of proposal values is determined on a syntactic level, i. e., proposal values are identified as equal if each bit of their serialized representation is equal. However, two syntactically different solutions could still be semantically equal, e. g., if a list of solutions contains the same values in a different order. This could be addressed with a semantic mapping and a partitioning of the solution.

# Part IV

# Appendices

# A Sum of Two Equal Laplace-Distributed Random Variables

We assume two Laplace-distributed random variables $A$ and $B$. Both distributions have the same mean $\mu$ and divergence $b$. The resulting probability distribution $P(C)$ is the convolution of their probabiltiy distributions

$$P(C) = \frac{1}{2b}\exp\left(-\frac{|t-\mu|}{b}\right) * \frac{1}{2b}\exp\left(-\frac{|t-\mu|}{b}\right). \tag{A.1}$$

For this proof, we exploit the fact that a convolution in the time domain corresponds to multiplication in the frequency domain. Therefore, we first Fourier transform $P(A)$ and $P(B)$, respectively:

$$\begin{aligned}
\mathcal{F}(P(A)) = \mathcal{F}(P(B)) &= \frac{1}{\sqrt{2\pi}}\frac{\alpha}{2}\int_{-\infty}^{\infty} e^{-\alpha|t-t_0|}e^{-i\omega t}dt && \text{(A.2)}\\
&= \frac{1}{\sqrt{2\pi}}\frac{\alpha}{2}\left(\int_{t_0}^{\infty} e^{-\alpha t+\alpha t_0-i\omega t}dt + \int_{-\infty}^{t_0} e^{\alpha t-\alpha t_0-i\omega t}dt\right) && \text{(A.3)}\\
&= \frac{1}{\sqrt{2\pi}}\frac{\alpha}{2}\left(e^{\alpha t_0}\int_{t_0}^{\infty} e^{-(\alpha+i\omega)t}dt + \right. && \text{(A.4)}\\
&\qquad\qquad \left. e^{-\alpha t_0}\int_{-\infty}^{t_0} e^{(\alpha-i\omega)t}dt\right)\\
&= \frac{1}{\sqrt{2\pi}}\frac{\alpha}{2}\left(e^{\alpha t_0}\frac{e^{-(\alpha+i\omega)t_0}}{\alpha+i\omega} + e^{-\alpha t_0}\frac{e^{(\alpha-i\omega)t_0}}{\alpha-i\omega}\right) && \text{(A.5)}\\
&= \frac{1}{\sqrt{2\pi}}\frac{\alpha}{2}\left(\frac{2\alpha e^{-it_0\omega}}{\alpha^2+\omega^2}\right) && \text{(A.6)}\\
&= \mathcal{D}\frac{\alpha^2}{(\alpha^2+\omega^2)} && \text{(A.7)}
\end{aligned}$$

with $\alpha = \frac{1}{b}$ and $\mathcal{D} = \frac{1}{\sqrt{2\pi}}e^{-i\omega t_0}$. Here, the factor $e^{-i\omega t_0}$ corresponds to a time shift in the frequency domain by $t_0$. Hence, we can compute $\mathcal{F}(P(C))$ by $\mathcal{F}(P(A))\cdot\mathcal{F}(P(B))$:

$$\Rightarrow \mathcal{F}(P(C)) = \mathcal{D}\frac{\alpha^4}{(\alpha^2+\omega^2)^2}. \tag{A.8}$$

In order to prove

$$P(C) = \frac{\alpha}{4}(1+\alpha|t-t_0|)e^{-\alpha|t-t_0|}, \tag{A.9}$$

the Fourier transformation of $P(C)$ has to correspond to Equation A.8:

$$\mathcal{F}(P(C)) = \mathcal{F}\left(\frac{\alpha}{4}(1+\alpha|t-t_0|)e^{\alpha|t-t_0|}\right) = e^{-i\omega t_0}\mathcal{F}\left(\frac{\alpha}{4}(1+\alpha|t|)e^{-\alpha|t|}\right) \tag{A.10}$$

$$= \mathcal{D}\left(\int_{-\infty}^{\infty}\left(\frac{\alpha}{4}e^{-|t|\alpha} + \frac{\alpha^2}{4}|t|e^{-|t|\alpha}\right)e^{-i\omega t}dt\right) \tag{A.11}$$

$$= \mathcal{D}\left(\int_{-\infty}^{\infty}\frac{\alpha}{4}e^{-|t|\alpha}e^{-i\omega t}dt + \right. \tag{A.12}$$

$$\left.\int_{0}^{\infty}\frac{\alpha^2}{4}te^{-t(\alpha+i\omega)}dt - \int_{-\infty}^{0}\frac{\alpha^2}{4}te^{t(\alpha-i\omega)}dt\right)$$

$$= \mathcal{D}\left(\int_{-\infty}^{\infty}\frac{\alpha}{4}e^{-|t|\alpha}e^{-i\omega t}dt + \right. \tag{A.13}$$

$$\left[\frac{\alpha^2(-t(\alpha+i\omega)-1)}{4(\alpha+i\omega)^2}e^{-t(\alpha+i\omega)}\right]\Bigg|_0^{\infty} -$$

$$\left.\left[\frac{\alpha^2(t(\alpha-i\omega)-1)}{4(\alpha-i\omega)^2}e^{t(\alpha-i\omega)}\right]\Bigg|_{-\infty}^{0}\right)$$

$$\text{with} \quad \lim_{t\to\infty} -te^{-t} = 0 \quad \text{and} \quad \lim_{t\to-\infty} te^{t} = 0:$$

$$= \mathcal{D}\left(\frac{\alpha^2}{2(\alpha^2+\omega^2)} + \frac{\alpha^2}{4(\alpha+i\omega)^2} + \frac{\alpha^2}{4(\alpha-i\omega)^2}\right) \tag{A.14}$$

$$= \mathcal{D}\frac{\alpha^4}{(\alpha^2+\omega^2)^2} \tag{A.15}$$

□

# B Publications as (Co-) Author


[1] M. Atzmüller, H. Baraki, K. Behrenbruch, D. Comes, C. Evers, A. Hoffmann, H. Hoffmann, S. Jandt, M. Kibanov, O. Kieselmann, R. Kniewel, I. König, B.-E. Macek, S. Niemczyk, C. Scholz, M. Schuldt, T. Schulz, H. Skistims, M. Söllner, C. Voigtmann, A. Witsch, and J. Zirfas. *Die VENUS-Entwicklungsmethode - Eine interdisziplinäre Methode für soziotechnische Softwaregestaltung*. Tech. rep. Zentrum für Informationstechnik Gestaltung (ITeG), 2014. URL: http://www.uni-kassel.de/upress/online/OpenAccess/978-3-86219-550-3.OpenAccess.pdf.

[2] D. Comes, C. Evers, K. Geihs, A. Hoffmann, R. Kniewel, J. M. Leimeister, S. Niemczyk, A. Roßnagel, L. Schmidt, T. Schulz, M. Söllner, and A. Witsch. „Designing Socio-Technical Applications for Ubiquitous Computing - Results from a Multidisciplinary Case Study“. In: *Distributed Applications and Interoperable Systems (Stockholm 2012)*. Ed. by K. M. Göschka and S. Haridi. Vol. 7272. Lecture Notes in Computer Science. Berlin: Springer, 2012, pp. 194–201.

[3] D. Comes, C. Evers, K. Geihs, D. Saur, A. Witsch, and M. Zapf. „Adaptive Applications are Smart Applications“. In: *International Workshop on Smart Mobile Applications*. San Francisco, June 2011.

[4] K. Geihs, S. Niemczyk, A. Roßnagel, and A. Witsch. „On the socially aware development of self-adaptive ubiquitous computing applications“. In: *it-it* 56.1 (2014), pp. 33–41. ISSN: 21967032. URL: http://www.degruyter.com/view/j/itit.2014.56.issue-1/itit-2014-1028/itit-2014-1028.xml.

[5] R. Hoyer, A. Bartetzki, D. Kirchner, and A. Witsch. „Giving Robots a Voice: A Kineto-Acoustic Project“. In: *Third International Conference on Arts and Technology*. Milano (Italy), Mar. 2013. ISBN: 978-3-642-37981-9. URL: http://www.springer.com/computer/information+systems+and+applications/book/978-3-642-37981-9.

[6] S. Niemczyk, D. Kirchner, A. Witsch, S. Opfer, and K. Geihs. „Distributed Sensing in a Robotic Soccer Team“. In: *CPSWeek 2014 - International Workshop on Robotic Sensor Networks*. Berlin, 2014.

[7] A. Witsch and K. Geihs. „An adaptive middleware core for a multi-agent coordination language“. In: *Networked Systems (NetSys), 2015 International Conference and Workshops on*. IEEE. 2015, pp. 1–8.

[8] A. Witsch, S. Opfer, and K. Geihs. „A Formal Multi-Agent Language for Cooperative Autonomous Driving Scenarios“. In: *2014 International Conference on Connected Vehicles and Expo (ICCVE 2014)*. Vienna, Austria, Nov. 2014.

[9] A. Witsch, R. Reichle, S. Lange, M. Riedmiller, and K. Geihs. „Enhancing the Episodic Natural Actor-Critic Algorithm by a Regularisation Term to Stabilize Learning of Control Structures“. In: *IEEE Symposium Series on Computational Intelligence*. 2011.

[10] A. Witsch, H. Skubch, S. Niemczyk, and K. Geihs. „Using Incomplete Satisfiability Modulo Theories to Determine Robotic Tasks“. In: *IEEE International Workshop on Intelligent Robots and Systems (IROS)*. 2013.

# C Bibliography


[11] S. Ahuja, N. Carriero, and D. Gelernter. „Linda and Friends“. In: *Computer* 19.8 (Aug. 1986), pp. 26–34. ISSN: 0018-9162. DOI: 10.1109/MC.1986.1663305.

[12] P. Alho and J. Mattila. „Real-Time Service-Oriented Architectures: A Data-Centric Implementation for Distributed and Heterogeneous Robotic System“. English. In: *Embedded Systems: Design, Analysis and Verification*. Ed. by G. Schirner, M. Götz, A. Rettberg, M. Zanella, and F. Rammig. Vol. 403. IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2013, pp. 262–271. ISBN: 978-3-642-38852-1. DOI: 10.1007/978-3-642-38853-8_24.

[13] L. Almeida, F. Santos, T. Facchinetti, P. Pedreiras, V. Silva, and L. S. Lopes. „Coordinating Distributed Autonomous Agents with a Real-Time Database: The CAMBADA Project.“ In: *ISCIS*. Ed. by C. Aykanat, T. Dayar, and I. Korpeoglu. Vol. 3280. Lecture Notes in Computer Science. Springer, 2004, pp. 876–886. ISBN: 3-540-23526-4.

[14] F. Arrichiello. „Coordination control of multiple mobile robots“. PhD thesis. PhD dissertation, Universita Degli Studi Di Cassino, Cassino, Italy, 2006.

[15] P. A. Baer. „Platform-Independent Development of Robot Communication Software“. PhD thesis in computer science. Kassel: University of Kassel, 2008. ISBN: 978-3-89958-644-2.

[16] C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump. „6 Years of SMT-COMP“. In: *Journal of Automated Reasoning* 50.3 (2013), pp. 243–277. DOI: 10.1007/s10817-012-9246-5.

[17] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. „Satisfiability Modulo Theories“. In: *Handbook of Satisfiability*. Ed. by A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, Feb. 2009. Chap. 26, pp. 825–885.

[18] M. Bengel, K. Pfeiffer, B. Graf, A. Bubeck, and A. Verl. „Mobile robots for offshore inspection and manipulation“. In: *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*. Oct. 2009, pp. 3317–3322. DOI: 10.1109/IROS.2009.5353885.

[19] E. Bianchi and L. Dozio. „Some Experiences in fast hard realtime control in user space with RTAI-LXRT“. In: *Real time Linux workshop*. 2000.

[20] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009. ISBN: 1586039296, 9781586039295.

[21] B. Bilgin and C. Gungor. „Performance Comparison of IEEE 802.11p and IEEE 802.11b for Vehicle-to-Vehicle Communications in Highway, Rural, and Urban Areas“. In: *International Journal of Vehicular Technology* 2013 (2013), p. 10.

[22] C. Boutilier. „Sequential optimality and coordination in multiagent systems“. In: *In International Joint Conference on Artificial Intelligence*. 1999, pp. 478–485.

[23] M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Nov. 1987. ISBN: 9780674458192.

[24] L. Braubach, A. Pokahr, and W. Lamersdorf. „Jadex: A BDI-agent system combining middleware and reasoning“. In: *Software agent-based applications, platforms and development kits*. Springer, 2005, pp. 143–168.

[25] T. Bressen. „Consensus decision making“. In: *The Change Handbook: The Definitive Resource on Today's Best Methods for Engaging Whole Systems 2nd ed. San Francisco, CA: Berrett-Koehler Publishers* (2007), pp. 212–7.

[26] S. S. Brilliant and T. R. Wiseman. „The First Programming Paradigm and Language Dilemma“. In: *SIGCSE Bull.* 28.1 (Mar. 1996), pp. 338–342. ISSN: 0097-8418. DOI: 10.1145/236462.236572.

[27] M. Broxvall, B.-S. Seo, and W. Kwon. „The peis kernel: A middleware for ubiquitous robotics“. In: *In Proc. of the IROS-07 Workshop on Ubiquitous Robotic Space Design and Applications*. 2007, pp. 212–218.

[28] A. Brutschy, A. Scheidler, E. Ferrante, M. Dorigo, and M. Birattari. „Can ants inspire robots? Self-organized decision making in robotic swarms“. In: *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. Oct. 2012, pp. 4272–4273. DOI: 10.1109/IROS.2012.6386273.

[29] H. Bruyninckx, P. Soetens, and B. Koninckx. „The Real-Time Motion Control Core of the Orocos Project“. In: *IEEE International Conference on Robotics and Automation*. 2003, pp. 2766–2771.

[30] M. Buehler, K. Iagnemma, and S. Singh. *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 3642039901, 9783642039904.

[31] P. Caloud, W. Choi, J.-C. Latombe, C. Le Pape, and M. Yim. „Indoor automation with many mobile robots“. In: *Intelligent Robots and Systems' 90.'Towards a New Frontier of Applications', Proceedings. IROS'90. IEEE International Workshop on*. IEEE. 1990, pp. 67–72.

[32] A. Campo, S. Garnier, O. Dédriche, M. Zekkri, and M. Dorigo. „Self-Organized Discrimination of Resources“. In: *PLoS ONE* 6.5 (2010), e19888. DOI: http://dx.plos.org/10.1371/journal.pone.0019888.

[33] K. M. Carley and L. Gasser. „Computational organization theory“. In: *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. Ed. by G. Weiss. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-23203-0.

[34] T. D. Chandra, R. Griesemer, and J. Redstone. „Paxos Made Live: An Engineering Perspective“. In: *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*. PODC '07. Portland, Oregon, USA: ACM, 2007, pp. 398–407. ISBN: 978-1-59593-616-5. DOI: 10.1145/1281100.1281103.

[35] T. D. Chandra and S. Toueg. „Unreliable Failure Detectors for Reliable Distributed Systems“. In: *JACM* 43.2 (Mar. 1996), pp. 225–267. ISSN: 0004-5411. DOI: 10.1145/226643.226647.

[36] S. Chen, W. Nai, D. Dong, W. Zheng, and W. Jing. „Key Indices Analysis of (IEEE) 802.11p based Vehicle to Infrastructure System in Highway Environment“. In: *Procedia - Social and Behavioral Sciences* 96 (2013). Intelligent and Integrated Sustainable Multimodal Transportation Systems Proceedings from the 13th (COTA) International Conference of Transportation Professionals (CICTP2013), pp. 188–195. ISSN: 1877-0428. DOI: 10.1016/j.sbspro.2013.08.025.

[37] C.-F. Cheng and K.-T. Tsai. „Eventual strong consensus with fault detection in the presence of dual failure mode on processors under dynamic networks“. In: *Journal of Network and Computer Applications* 35.4 (2012). Intelligent Algorithms for Data-Centric Sensor Networks, pp. 1260–1276. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2012.01.011.

[38] E. Clarke, W. Klieber, M. Nováček, and P. Zuliani. „Model Checking and the State Explosion Problem“. English. In: *Tools for Practical Software Verification*. Ed. by B. Meyer and M. Nordio. Vol. 7682. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1–30. ISBN: 978-3-642-35745-9. DOI: 10.1007/978-3-642-35746-6_1.

[39] J. Cogswell. „Adding an Easy File Save and File Load Mechanism to Your C++ Program“. In: *InformIT* (July 2005).

[40] P. Cohen and H. Levesque. „Teamwork“. In: *Noûs* 25.4 (1991), pp. 487–512.

[41] P. R. Cohen and H. J. Levesque. „Confirmations and Joint Action.“ In: *IJCAI*. Ed. by J. Mylopoulos and R. Reiter. Morgan Kaufmann, 1991, pp. 951–959. ISBN: 1-55860-160-0.

[42] S. A. Cook. „The Complexity of Theorem-proving Procedures“. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: ACM, 1971, pp. 151–158. DOI: 10.1145/800157.805047.

[43] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. „Programming wireless sensor networks with the TeenyLime middleware“. In: *Middleware 2007*. Springer, 2007, pp. 429–449.

[44] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. „TeenyLIME: transiently shared tuple space middleware for wireless sensor networks“. In: *Proceedings of the international workshop on Middleware for sensor networks*. ACM. 2006, pp. 43–48.

[45] S. Crisóstomo, U. S. Schilcher, C. B. Bettstetter, and J. Barros. „Probabilistic Flooding in Stochastic Networks: Analysis of Global Information Outreach“. In: *Computer Networks* 56.1 (Jan. 2012), pp. 142–156. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2011.08.014.

[46] M. Dastani, D. Hobo, and J.-J. C. Meyer. „Practical extensions in agent programming languages.“ In: *AAMAS*. Ed. by E. H. Durfee, M. Yokoo, M. N. Huhns, and O. Shehory. IFAAMAS, 2007, p. 138. ISBN: 978-81-904262-7-5.

[47] M. Davis and H. Putnam. „A Computing Procedure for Quantification Theory“. In: *JACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034.

[48] Defense Advanced Research Projects Agency (DARPA). „DARPA Robotics Challenge“. Apr. 10, 2012.

[49] M. Dorigo and C. Blum. „Ant colony optimization theory: A survey“. In: *Theoretical computer science* 344.2 (2005), pp. 243–278.

[50] M. Dubois, C. Scheurich, and F. Briggs. „Memory Access Buffering in Multiprocessors“. In: *25 Years of the International Symposia on Computer Architecture (Selected Papers)*. ISCA '98. Barcelona, Spain: ACM, 1998, pp. 320–328. ISBN: 1-58113-058-9. DOI: 10.1145/285930.285991.

[51] Edward J. Daniel, Christopher M. White, and Keith A. Teague. „An Inter-arrival Delay Jitter Model using Multi-Structure network delay characteristics for packet networks“. In: *The ThritySeventh Asilomar Conference on Signals Systems Computers 2003* 2 (2003).

[52] N. Een and N. Sörensson. *MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization, System description for the SAT competition*. 2005.

[53] A. Elkady and T. Sobh. „Robotics middleware: A comprehensive literature survey and attribute-based bibliography“. In: *Journal of Robotics* 2012 (2012).

[54] C. Erbas, S. Sarkeshik, and M. M. Tanik. „Different Perspectives of the N-Queens Problem“. In: *Proceedings of the 1992 ACM Annual Conference on Communications*. CSC '92. Kansas City, Missouri, USA: ACM, 1992, pp. 99–108. ISBN: 0-89791-472-4. DOI: 10.1145/131214.131227.

[55] S. Even, A. Itai, and A. Shamir. „On the Complexity of Time Table and Multi-commodity Flow Problems“. In: *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*. SFCS '75. Washington, DC, USA: IEEE Computer Society, 1975, pp. 184–193. DOI: 10.1109/SFCS.1975.21.

[56] R. E. Fikes and N. J. Nilsson. „Strips: A new approach to the application of theorem proving to problem solving“. In: *Artificial Intelligence* 2.3–4 (1971), pp. 189–208. ISSN: 0004-3702. DOI: 10.1016/0004-3702(71)90010-5.

[57] M. J. Fischer, N. A. Lynch, and M. S. Paterson. „Impossibility of distributed consensus with one faulty process“. In: *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*. PODS '83. Atlanta, Georgia: ACM, 1983, pp. 1–7. ISBN: 0-89791-097-4. DOI: 10.1145/588058.588060.

[58] A. Fisher. „Interact System Model of Decision Emergence“. In: *Communication Theory* (2002).

[59] N. R. Franks, F.-X. Dechaume-Moncharmont, E. Hanmore, and J. K. Reynolds. „Speed versus accuracy in decision-making ants: expediting politics and policy implementation“. In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 364.1518 (Mar. 27, 2009), pp. 845–852. DOI: 10.1098/rstb.2008.0224.

[60] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. „Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure“. In: *Journal on Satisfiability, Boolean Modeling and Computation* 1 (2007), pp. 209–236.

[61] E. Freeman, K. Arnold, and S. Hupfer. *JavaSpaces Principles, Patterns, and Practice*. 1st. Essex, UK, UK: Addison-Wesley Longman Ltd., 1999. ISBN: 0201309556.

[62] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[63] H. Garcia-Molina. „Elections in a Distributed Computing System.“ In: *IEEE Trans. Computers* 31.1 (1982), pp. 48–59.

[64] B. Gerkey and M. Mataric. „Sold!: auction methods for multirobot coordination“. In: *Robotics and Automation, IEEE Transactions on* 18.5 (Oct. 2002), pp. 758–768. ISSN: 1042-296X. DOI: 10.1109/TRA.2002.803462.

[65] B. P. Gerkey and M. J. Mataric. „A Formal Analysis and Taxonomy of Task Allocation in Multi-Robot Systems.“ In: *I. J. Robotic Res.* 23.9 (2004), pp. 939–954.

[66] G. Goetsch and M. Campbell. „Experiments with the Null-Move Heuristic“. English. In: *Computers, Chess, and Cognition*. Ed. by T. Marsland and J. Schaeffer. Springer New York, 1990, pp. 159–168. ISBN: 978-1-4613-9082-4. DOI: 10.1007/978-1-4613-9080-0_9.

[67] B. Gough. *GNU Scientific Library Reference Manual - Third Edition*. 3rd. Network Theory Ltd., 2009. ISBN: 0954612078, 9780954612078.

[68] B. J. Grosz. „AAAI-94 Presidential Address: Collaborative Systems“. In: *AI Magazine* 17.2 (1996), pp. 67–85.

[69] B. J. Grosz and S. Kraus. „Collaborative plans for complex group action“. In: *Artificial Intelligence* 86.2 (1996), pp. 269–357. ISSN: 0004-3702. DOI: 10.1016/0004-3702(95)00103-4.

[70] D. Guha-Sapir, P. Hoyois, and R. Below. *Annual Disaster Statistical Review 2013: The Numbers and Trends*. Brussels: CRED, 2014.

[71] E. Guizzo. „How Google's Self-Driving Car Works“. In: *IEEE Spectrum* (Oct. 2011). Ed. by E. Guizzo. ISSN: 0018-9235.

[72] A. Habib, G. Bonow, A. Kroll, J. Hegenberg, L. Schmidt, T. Barz, and D. Schulz. „Forschungsprojekt RoboGasInspector: Gaslecksuche mit autonomen mobilen Robotern“. In: *Technische Sicherheit* 3.5 (2013), pp. 10–15.

[73] E. A. Hansen, D. S. Bernstein, and S. Zilberstein. „Dynamic programming for partially observable stochastic games“. In: *Proceedings of the National Conference on Artificial Intelligence*. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999. 2004, pp. 709–715.

[74] P. Hart, N. Nilsson, and B. Raphael. „A Formal Basis for the Heuristic Determination of Minimum Cost Paths“. In: *Systems Science and Cybernetics, IEEE Transactions on* 4.2 (July 1968), pp. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136.

[75] S. Hemminger. „Network Emulation with NetEm“. In: *LCA 2005, Australia's 6th national Linux conference (linux.conf.au)*. Ed. by M. Pool. Linux Australia. Canberra, ACT, Australia: Linux Australia, Apr. 18, 2005.

[76] K. V. Hindriks, F. S. De Boer, W. Van Der Hoek, and J.-J. Ch. Meyer. „Agent Programming in 3APL“. In: *Autonomous Agents and Multi-Agent Systems* 2.4 (1999). ISSN: 1387-2532. DOI: 10.1023/A:1010084620690.

[77] B. Horling and V. Lesser. „A Survey of Multi-agent Organizational Paradigms“. In: *Knowl. Eng. Rev.* 19.4 (Dec. 2004), pp. 281–316. ISSN: 0269-8889. DOI: 10.1017/S0269888905000317.

[78] M. C. Huebscher and J. A. McCann. „A Survey of Autonomic Computing&Mdash;Degrees, Models, and Applications“. In: *ACM Comput. Surv.* 40.3 (Aug. 2008), 7:1–7:28. ISSN: 0360-0300. DOI: 10.1145/1380584.1380585.

[79] M. Isard. „Autopilot: Automatic Data Center Management“. In: *SIGOPS Oper. Syst. Rev.* 41.2 (Apr. 2007), pp. 60–67. ISSN: 0163-5980. DOI: 10.1145/1243418.1243426.

[80] G. A. Kaminka and I. Frenkel. „Flexible Teamwork in Behavior-Based Robots.“ In: *AAAI*. Ed. by M. M. Veloso and S. Kambhampati. AAAI Press / The MIT Press, 2005, pp. 108–113. ISBN: 1-57735-236-X.

[81] S. Kaner. *Facilitator's guide to participatory decision-making*. The Jossey-Bass Business & Management Series. San Francisco: Wiley Jossey-Bass, 2007. ISBN: 978-0-7879-8266-9.

[82] D. Kirchner and K. Geihs. „Qualitative Bayesian Failure Diagnosis for Robot Systems“. In: *International Conference on Intelligent Robots and Systems, AI and Robotics*. Ed. by W. Burgard. Chicago: IEEE, 2014, pp. 1–6.

[83] D. Kirchner, S. Niemczyk, and K. Geihs. „RoSHA: A Multi-Robot Self-Healing Architecture“. In: *17th Annual RoboCup International Symposium*. July 2013.

[84] M. Kriegleder, R. Oung, and R. D'Andrea. „Asynchronous implementation of a distributed average consensus algorithm“. In: *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*. Nov. 2013, pp. 1836–1841. DOI: 10.1109/IROS.2013.6696598.

[85] M. R. Krom. „The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary“. In: *Mathematical Logic Quarterly* 13.1-2 (1967), pp. 15–20. ISSN: 1521-3870. DOI: 10.1002/malq.19670130104.

[86] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. 1st ed. New York, NY, USA: Cambridge University Press, 2008. ISBN: 0521876346.

[87] H. W. Kuhn. „The Hungarian Method for the Assignment Problem“. In: *Naval Research Logistics Quarterly* 2.1–2 (Mar. 1955), pp. 83–97. DOI: 10.1002/nav.3800020109.

[88] L. Lamport. *Generalized Consensus and Paxos*. Tech. rep. MSR-TR-2005-33. Microsoft Research, Mar. 2005, p. 60.

[89] L. Lamport. „Paxos Made Simple“. In: *SIGACT News* 32.4 (Dec. 2001), pp. 51–58. ISSN: 0163-5700. DOI: 10.1145/568425.568433.

[90] L. Lamport. „The Part-time Parliament“. In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229.

[91] L. Lamport. „Time, Clocks, and the Ordering of Events in a Distributed System“. In: *Communication* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563.

[92] L. Lamport. „Time, clocks, and the ordering of events in a distributed system“. In: *Communication ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563.

[93] L. Lamport, R. Shostak, and M. Pease. „The Byzantine Generals Problem“. In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176.

[94] A. Land and A. Doig. „An Automatic Method of Solving Discrete Programming Problems“. In: *Econometrica* 28.3 (1960), pp. 497–520. ISSN: 00129682, 14680262.

[95] J.-C. Latombe. *Robot Motion Planning*. Norwell, MA, USA: Kluwer Academic Publishers, 1991. ISBN: 079239206X.

[96] K. Lee and J. Eidson. „IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems“. In: *In 34 th Annual Precise Time and Time Interval (PTTI) Meeting*. 2002, pp. 98–105.

[97] R. de Lemos et al. „Software Engineering for Self-Adaptive Systems: A second Research Roadmap“. In: *Software Engineering for Self-Adaptive Systems*. Ed. by R. de Lemos, H. Giese, H. Müller, and M. Shaw. Dagstuhl Seminar Proceedings 10431. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011.

[98] T. Lenz. *Konsum und Modernisierung: Die Debatte um das Warenhaus als Diskurs um die Moderne*. Kulturen der Gesellschaft. 2014. ISBN: 9783839413821.

[99] H. J. Levesque, P. R. Cohen, and J. H. T. Nunes. „On Acting Together“. In: *Proceedings of the Eighth National Conference on Artificial Intelligence*. Vol. 1. AAAI'90. Boston, Massachusetts: AAAI Press, 1990, pp. 94–99. ISBN: 0-262-51057-X.

[100] K. Li and P. Hudak. „Memory Coherence in Shared Virtual Memory Systems“. In: *ACM Trans. Comput. Syst.* 7.4 (Nov. 1989), pp. 321–359. ISSN: 0734-2071. DOI: 10.1145/75104.75105.

[101] R. Luna and K. E. Bekris. „Efficient and complete centralized multi-robot path planning.“ In: *IROS*. IEEE, 2011, pp. 3268–3275. ISBN: 978-1-61284-454-1.

[102] E. Mallada, X. Meng, M. Hack, L. Zhang, and A. Tang. „Skewless Network Clock Synchronization“. In: *CoRR* abs/1208.5703 (2012).

[103] E. Mallada and A. Tang. „Distributed clock synchronization: Joint frequency and phase consensus.“ In: *CDC-ECE*. IEEE, 2011, pp. 6742–6747. ISBN: 978-1-61284-800-6.

[104] D. W. Marquardt. „An algorithm for least-squares estimation of nonlinear parameters“. In: *SIAM Journal on Applied Mathematics* 11.2 (1963), pp. 431–441. DOI: 10.1137/0111030.

[105] F. Mattern. „Virtual Time and Global States of Distributed Systems“. In: *Parallel and Distributed Algorithms*. North-Holland, 1989, pp. 215–226.

[106] H. Mehendale, A. Paranjpe, and S. Vempala. „LifeNet: a flexible ad hoc networking solution for transient environments“. In: *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*. 2011, pp. 446–447. DOI: 10.1145/2018436.2018513.

[107] K. Michels, F. Klawonn, R. Kruse, and A. Nürnberger. *Fuzzy-Regelung: Grundlagen, Entwurf, Analyse*. Berlin: Springer-Verlag, 2002. ISBN: 3-540-43548-4.

[108] A. Mohammed, M. Ould-Khaoua, and L. Mackenzie. „Improvement to Efficient Counter-Based Broadcast Scheme through Random Assessment Delay Adaptation for MANETS“. In: *Computer Modeling and Simulation, 2008. EMS '08. Second UKSIM European Symposium on*. Sept. 2008, pp. 536–541. DOI: 10.1109/EMS.2008.69.

[109] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. „Chaff: Engineering an Efficient SAT Solver“. In: *Proceedings of the 38th Annual Design Automation Conference*. DAC '01. Las Vegas, Nevada, USA: ACM, 2001, pp. 530–535. ISBN: 1-58113-297-2. DOI: 10.1145/378239.379017.

[110] L. de Moura and N. Bjørner. „Z3: An Efficient SMT Solver“. In: *Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[111] A. L. Murphy, G. P. Picco, and G.-C. Roman. „LIME: A coordination model and middleware supporting mobility of hosts and agents“. In: *ACM Trans. Softw. Eng. Methodol.* 15.3 (July 2006), pp. 279–328. ISSN: 1049-331X. DOI: 10.1145/1151695.1151698.

[112] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. „SHOP2: An HTN Planning System“. In: *J. Artif. Int. Res.* 20.1 (Dec. 2003), pp. 379–404. ISSN: 1076-9757.

[113] A. Newell. *Unified Theories of Cognition*. Cambridge, MA, USA: Harvard University Press, 1990. ISBN: 0-674-92099-6.

[114] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. „The Broadcast Storm Problem in a Mobile Ad Hoc Network“. In: *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*. MobiCom '99. Seattle, Washington, USA: ACM, 1999, pp. 151–162. ISBN: 1-58113-142-9. DOI: 10.1145/313451.313525.

[115] S. Oaks and H. Wong. *Jini in a Nutshell*. Beijing: O'Reilly, 2001. ISBN: 978-3-89721-194-0.

[116] *Object Management Group (OMG)*. http://www.omg.org/ (accessed 18.01.2015). Object Management Group.

[117] Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification (CORBA 2.3.1 specification)*. Tech. rep. Object Management Group, Oct. 1999.

[118] K. Obraczka, K. Viswanath, and G. Tsudik. „Flooding for Reliable Multicast in Multi-hop Ad Hoc Networks“. In: *Wirel. Netw.* 7.6 (Nov. 2001), pp. 627–634. ISSN: 1022-0038. DOI: 10.1023/A:1012323519059.

[119] A. Omicini. „On the Semantics of Tuple-based Coordination Models“. In: *Proceedings of the 1999 ACM Symposium on Applied Computing*. SAC '99. San Antonio, Texas, USA: ACM, 1999, pp. 175–182. ISBN: 1-58113-086-4. DOI: 10.1145/298151.298229.

[120] A. Omicini and E. Denti. „From tuple spaces to tuple centres“. In: *Science of Computer Programming* 41.3 (2001), pp. 277–294. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(01)00011-9.

[121] A. Omicini and F. Zambonelli. „TuCSoN: a Coordination model for Mobile Information Agents“. In: *1st International Workshop on Innovative Internet Information Systems (IIIS'98)*. Ed. by D. G. Schwartz, M. Divitini, and T. Brasethvik. Pisa, Italy: IDI – NTNU, Trondheim (Norway), Aug. 1998, pp. 177–187.

[122] D. Ongaro. „Consensus: Bridging theory and practice“. PhD thesis. Stanford University, 2014.

[123] D. Ongaro and J. Ousterhout. „In Search of an Understandable Consensus Algorithm“. In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, 2014, pp. 305–320. ISBN: 978-1-931971-10-2.

[124] J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000. ISBN: 0-89871-461-3.

[125] G. Pardo-Castellote. „OMG Data-Distribution Service: Architectural Overview“. In: *23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03)*. Los Alamitos, CA, USA: IEEE Computer Society, May 22, 2003, pp. 200–206. ISBN: 0-7695-1921-0. DOI: 10.1109/icdcsw.2003.1203555.

[126] G. Pardo-Castellote, B. Farabaugh, and R. Warren. *An Introduction to DDS and Data-Centric Communications*. Tech. rep. 2005.

[127] K. Passino and T. Seeley. „Modeling and analysis of nest-site selection by honeybee swarms: the speed and accuracy trade-off“. English. In: *Behavioral Ecology and Sociobiology* 59.3 (2006), pp. 427–442. ISSN: 0340-5443. DOI: 10.1007/s00265-005-0067-y.

[128] S. Pellegrini. *Meta-Serialization Library*. https://github.com/motonacciu/meta-serialization (accessed 2015-08-20).

[129] A. Petcu. *A Class of Algorithms for Distributed Constraint Optimization: Volume 194 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009. ISBN: 158603989X, 9781586039899.

[130] S. Petters, D. Thomas, and O. von Stryk. „RoboFrame - A Modular Software Framework for Lightweight Autonomous Robots“. In: *Proc. Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware of the 2007 IEEE/RSJ IROS*. San Diego, CA, USA, Oct. 2007.

[131] M. Presburger. „Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt“. In: *Comptes-rendus du I Congrés des Mathématiciens des Pays Slaves*. Warsaw, 1929, pp. 92–101.

[132] D. V. Pynadath and M. Tambe. „An Automated Teamwork Infrastructure for Heterogeneous Software Agents and Humans.“ In: *Autonomous Agents and Multi-Agent Systems* 7.1-2 (2003), pp. 71–100.

[133] D. V. Pynadath and M. Tambe. „The Communicative Multiagent Team Decision Problem: Analyzing Teamwork Theories and Models“. In: *CoRR* abs/1106.4569 (2011).

[134] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. „ROS: an open-source Robot Operating System“. In: *ICRA Workshop on Open Source Software*. 2009.

[135] A. S. Rao and M. P. Georgeff. „Modeling rational agents within a BDI-architecture“. In: *Principles of Knowledge Representation and Reasoning. Proceedings of the second International Conference*. San Mateo: Morgan Kaufmann, 1991, pp. 473–484.

[136] A. S. Rao and M. P. George. „BDI agents: From theory to practice“. In: *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*. 1995, pp. 312–319.

[137] R. Reichle. „Information Exchange and Fusion in Dynamic and Heterogeneous Distributed Environments“. PhD thesis. Wilhelmshöher Allee 73, 34121 Kassel, Germany: University of Kassel, Fachbereich 16: Elektrotechnik/Informatik, Distributed Systems Group, Dec. 2010.

[138] W. Ren, R. W. Beard, and E. M. Atkins. „A survey of consensus problems in multi-agent coordination“. In: *American Control Conference, 2005. Proceedings of the 2005*. Portland, OR, USA: IEEE, June 2005, pp. 1859–1864. ISBN: 0-7803-9098-9. DOI: 10.1109/acc.2005.1470239. URL: http://dx.doi.org/10.1109/acc.2005.1470239.

[139] D. Mills, J. Martin, J. Burbank, and W. Kasch. *Network Time Protocol Version 4: Protocol and Algorithms Specification*. RFC 5905. Fremont, CA, USA: RFC Editor, June 2010.

[140] D. Richardson. „Some Unsolvable Problems Involving Elementary Functions of a Real Variable“. In: *Journal of Symbolic Logic* (1968).

[141] M. Riedmiller and H. Braun. „RPROP - A Fast Adaptive Learning Algorithm“. In: *International Symposium on Computer and Information Sciences - ISCIS* (1992).

[142] *RoboCup Foundation*. http://www.robocup.org/ (accessed 2015-06-29). RoboCup Organization.

[143] *ROS on DDS*. http://design.ros2.org/articles/ros_on_dds.html (accessed 2015-02-22). Open Source Robotics Foundation, Inc.

[144] M. Roth, R. Simmons, and M. Veloso. „Reasoning About Joint Beliefs for Execution-time Communication Decisions“. In: *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS '05. The Netherlands: ACM, 2005, pp. 786–793. ISBN: 1-59593-093-0. DOI: 10.1145/1082473.1082593.

[145] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Pearson Education, 2003. ISBN: 0137903952.

[146] A. Saffiotti and M. Broxvall. „PEIS Ecologies: Ambient intelligence meets autonomous robotics“. In: *Proc of the Int Conf on Smart Objects and Ambient Intelligence (sOc-EUSAI*. 2005, pp. 275–280. DOI: 10.1.1.65.833.

[147] Y. Saito and M. Shapiro. „Optimistic Replication“. In: *ACM Comput. Surv.* 37.1 (Mar. 2005), pp. 42–81. ISSN: 0360-0300. DOI: 10.1145/1057977.1057980.

[148] T. Sandholm, K. Larson, M. Andersson, O. Shehory, and F. Tohmé. „Coalition structure generation with worst case guarantees“. In: *Artificial Intelligence* 111.1–2 (1999), pp. 209–238. ISSN: 0004-3702. DOI: 10.1016/S0004-3702(99)00036-3.

[149] D. Saur, T. R. Haque, R. Herzog, and K. Geihs. „MAGiC : Multi-Agent Planning using Grid Computing concepts“. In: *12th International Symposium on Artificial Intelligence, Robotics and Automation in Space*. Montreal, 2014.

[150] D. Schmidt and H. van't Hag. „Addressing the challenges of mission-critical information management in next-generation net-centric pub/sub systems with OpenSplice DDS“. In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. Apr. 2008, pp. 1–8. DOI: 10.1109/IPDPS.2008.4536567.

[151] P. Seibt. *Algorithmic Information Theory: Mathematics of Digital Information Processing*. Signals and Communication Technology. Springer Berlin Heidelberg, 2007. ISBN: 9783540332190.

[152] J. Seward, N. Nethercote, and J. Weidendorfer. *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications*. Network Theory Ltd., 2008. ISBN: 0954612051, 9780954612054.

[153] Y. Shang, M. P. Fromherz, and L. Crawford. „A new constraint test-case generator and the importance of hybrid optimizers“. In: *European Journal of Operational Research* 173.2 (2006), pp. 419–443. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2005.02.073.

[154] A. Shtof. *AutoDiff – High-performance and high-accuracy automatic function-differentiation library suitable for optimization and numeric computing*. http://autodiff.codeplex.com/ (accessed 2012-05-21).

[155] S. Shumko. „Ice Middleware in the New Solar Telescope's Telescope Control System“. In: *Astronomical Data Analysis Software and Systems XVIII*. Vol. 411. 2009, p. 482.

[156] J. P. M. Silva and K. A. Sakallah. „GRASP&Mdash;a New Search Algorithm for Satisfiability“. In: *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '96. San Jose, California, USA: IEEE Computer Society, 1996, pp. 220–227. ISBN: 0-8186-7597-7.

[157] H. Skubch. *Modelling and Controlling Behaviour of Cooperative Autonomous Mobile Robots*. Springer Vieweg, 2012. ISBN: 978-3-658-00810-9.

[158] H. Skubch. „Solving Non-Linear Arithmetic Constraints in Soft Realtime Environments“. In: *27th Symposium On Applied Computing*. Vol. 1. ACM SAC. ACM. Riva del Garda, Italy: ACM, 2012, pp. 67–75.

[159] H. Skubch, D. Saur, and K. Geihs. „Resolving Conflicts in Highly Reactive Teams“. In: *Kommunikation in Verteilten Systemen 2011*. Open Access Series in Informatics. Open Access Series in Informatics, 2011.

[160] H. Skubch, M. Wagner, R. Reichle, and K. Geihs. „A Modelling Language for Cooperative Plans in Highly Dynamic Domains“. In: *Mechatronics* 21.2 (2011). issn 0957-4158, pp. 423–433.

[161] H. Skubch, M. Wagner, R. Reichle, S. Triller, and K. Geihs. „Towards a Comprehensive Teamwork Model for Highly Dynamic Domains“. In: *Proceedings of the 2nd International Conference on Agents and Artificial Intelligence*. Ed. by J. Filipe, A. Fred, and B. Sharp. Vol. 2. INSTICC Press. INSTICC Press, Jan. 2010, pp. 121–127.

[162] R. Soetens, R. van de Molengraft, and B. Cunha. „RoboCup MSL - History, Accomplishments, Current Status and Challenges Ahead“. In: *RoboCup Symposium*. Special Track on the Advancement of the RoboCup Leagues. 2014.

[163] E. Strickland. „24 Hours at Fukushima“. In: *IEEE Spectrum* (Oct. 2011).

[164] B. Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley Professional, 2013. ISBN: 0321563840, 9780321563842.

[165] L. Surhone, M. Tennoe, and S. Henssonow. *Apache Zookeeper*. Betascript Publishing, 2010. ISBN: 9786134617925.

[166] L. E. Susskind, S. McKearnen, and J. Thomas-Lamar. *The Consensus Building Handbook: A Comprehensive Guide to Reaching Agreement*. SAGE Publications, 1999. ISBN: 9780761908449.

[167] M. Tambe. „Towards Flexible Teamwork“. In: *Journal of Artificial Intelligence Research* 7 (1997), pp. 83–124.

[168] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN: 0132392275.

[169] A. Tate. „Generating Project Networks“. In: *Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'77. Cambridge, USA: Morgan Kaufmann Publishers Inc., 1977, pp. 888–893.

[170] D. Terry. „Replicated Data Consistency Explained Through Baseball“. In: *Communication ACM* 56.12 (Dec. 2013), pp. 82–89. ISSN: 0001-0782. DOI: 10.1145/2500500.

[171] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar. „Miro - middleware for mobile robot applications“. In: *Robotics and Automation, IEEE Transactions on* 18.4 (2002), pp. 493–497. ISSN: 1042-296X. DOI: 10.1109/TRA.2002.802930.

[172] G. Valentini, H. Hamann, and M. Dorigo. „Efficient decision-making in a self-organizing robot swarm: On the speed versus accuracy trade-off“. In: *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2015, pp. 1305–1314.

[173] D. Vassis, G. Kormentzas, A. Rouskas, and I. Maglogiannis. „The IEEE 802.11g standard for high data rate WLANs“. In: *IEEE Network*. Vol. 19. May 2005, pp. 21–26.

[174] R. Volpe, I. A. D. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. *CLARAty: Coupled Layer Architecture for Robotic Autonomy*. Tech. rep. NASA - JET PROPULSION LABORATORY, 2000.

[175] M. Widenius and D. Axmark. *Mysql Reference Manual*. Ed. by P. DuBois. 1st. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002. ISBN: 0596002653.

[176] A. Witsch and K. Geihs. „An adaptive middleware core for a multi-agent coordination language“. In: *Networked Systems (NetSys), 2015 International Conference and Workshops on*. IEEE. 2015, pp. 1–8.

[177] A. Witsch, S. Opfer, and K. Geihs. „A Formal Multi-Agent Language for Cooperative Autonomous Driving Scenarios“. In: *2014 International Conference on Connected Vehicles and Expo (ICCVE 2014)*. Vienna, Austria, Nov. 2014.

[178] A. Witsch, H. Skubch, S. Niemczyk, and K. Geihs. „Using Incomplete Satisfiability Modulo Theories to Determine Robotic Tasks“. In: *International Conference on Intelligent Robots and Systems*. IEEE, 2013.

[179] A. Witsch, H. Skubch, S. Niemczyk, and K. Geihs. „Using incomplete satisfiability modulo theories to determine robotic tasks“. In: *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*. Tokyo: IEEE, Nov. 2013.

[180] M. Woolridge. *Introduction to Multiagent Systems*. New York, NY, USA: John Wiley & Sons, Inc., 2001. ISBN: 047149691X.

[181] L. Zheng, L. Zhang, and D. Xu. „Characteristics of network delay and delay jitter and its effect on voice over IP (VoIP)“. In: *Communications, 2001. ICC 2001. IEEE International Conference on*. Vol. 1. 2001.