Autonomous Robots (2021) 45:351–369
https://doi.org/10.1007/s10514-021-09967-8

Reliable distribution of computational load in robot teams

Ivan Valkov[1] · Phil Trinder[1] · Natalia Chechina[2]

Received: 7 April 2020 / Accepted: 5 January 2021 / Published online: 2 February 2021

Abstract
Modern multi-robot systems often need to solve computationally intensive tasks but operate with limited compute resources and in the presence of failures. Cooperating to share computational tasks between robots at the edge reduces execution time. We introduce and evaluate a new computation load management technology for teams of robots: Reliable Autonomous Mobile Programs (RAMPs). RAMPs use information about the computational resources available in the team and a cost model to decide where to execute. RAMPs are implemented in ROS on a collection of Raspberry Pi-based robots. The performance of RAMPs is evaluated using route planning, a typical computationally-intensive robotics application. A systematic study of RAMPs demonstrates a high likelihood of optimal or near-optimal distribution and hence efficient resource utilisation. RAMPs successfully complete in the presence of simultaneous, or successive, robot failures and network failures, while preserving near-optimal distribution.

Keywords Cooperative robotics · Autonomous mobility · Computational load distribution · Fault tolerance · ROS

1 Introduction

Multi-robot systems (MRS) are used in a wide range of domains, including factory automation, hazardous waste search, and rescue missions (Kurfess 2004). Many companies use robots in warehouses to carry heavy goods and assemble parcels, e.g. Amazon,[1] Ocado[2]—in fact, it is expected that the worldwide market for warehouse robotics will grow from \$1.1Bn in 2019 to \$2.2Bn in 2024.[3] However, despite rapid advances in robotic technology, many robots still operate with limited compute resources (Wang et al. 2019).

When a robot is overloaded with computational tasks, it can become a bottleneck in an MRS and lead to overall performance degradation and failures of robots (Garzón et al. 2017). Sharing the computational workload between members of a team of robots can reduce the time to complete the tasks, and is an example of *cooperative* robotics. Cooperative robotics focuses on a completion of tasks utilising available resources (e.g. robots, humans, servers) (Cao et al. 1997), rather than on an efficient and safe collaboration with humans as does collaborative robotics (Breazeal et al. 2004).

Importantly, if not taken into consideration, a failure in a single robot can be detrimental for the overall success of an MRS. Research in the area shows that robots are prone to failures, such as crashes, freezing, unexpected robot shutdown, and network failures (Crestani et al. 2015; Khalastchi and Kalech 2019).

This paper investigates a novel multi-robot computation load management technology: Reliable Autonomous Programs (RAMPs). RAMPs are inspired by Autonomous Mobile Programs (AMPs) (Deng et al. 2006), and each RAMP autonomously decides where to execute in a multi-robot system. This decision is driven by the program's computational requirements, the current resource utilisation of the robots in the system, and a cost model. RAMPs share

[1] https://roboticsandautomationnews.com/2020/01/21/amazon-now-has-200000-robots-working-in-its-warehouses/28840/
[2] https://www.chargedretail.co.uk/2020/01/24/tesco-uses-ocados-tech-partner-for-automated-warehouse-robots/
[3] https://www.marketwatch.com/press--release/warehouse-robotics-market-size-2020-competitive-situation-gross-margin-revenue-and-trends-forecast-report-2024-2019-12-09

✉ Natalia Chechina
nchechina@bournemouth.ac.uk

Ivan Valkov
iv.v.valkov@gmail.com

Phil Trinder
Phil.Trinder@glasgow.ac.uk

1 School of Computing Science, Glasgow University, Glasgow G12 8RZ, UK

2 Department of Computing and Informatics, Bournemouth University, Poole BH12 5BB, UK

only *computational* tasks (and hence work) between robots, and not *physical* tasks or actions. To demonstrate RAMPs we have implemented them on ROS, a widely used robot middleware (Quigley et al. 2009), but their design is independent of ROS and other middleware could be used.

RAMPs are specifically designed to address the challenges of computational load distribution within a MRS by addressing their distinct characteristics. MRS characteristics are edge-like, and far from typical of most distributed and multi-agent systems, such as having limited computational and energy resources, using wireless connectivity, and periodically losing connectivity due to robot mobility. Of course RAMPs can also be used for computational offload in less challenging environments, e.g. systems that offload computations from robots over a stable network to heavyweight external compute services (like a cloud) are both simpler and more common (Nimmagadda et al. 2010; Cano et al. 2018). RAMPs support heterogeneous compute resources and workloads like many other distributed load managers.

The specific features of RAMPs that support computational load distribution within a MRS are as follows. (1) The approach is *decentralised* so any one robot that requires to solve an intensive task can seek cooperation from the team to solve it. (2) RAMPs adapt to a dynamic set of compute resources where robots may join or leave the team. (3) RAMPs adapt to the failures that may occasionally be caused by hardware or software, but are commonly caused when a mobile robot has moved out of wireless communication range.

The performance of RAMPs is evaluated in *loaded* and *unloaded* MRS. *Unloaded* systems have no computational tasks other than the RAMPs, while the robots in *loaded* systems may have other computational loads in addition to any RAMPs they execute, e.g. computational tasks without RAMP wrappers. Such loaded systems emulate typical MRS where some resources may be permanently or temporarily unavailable to RAMPs. For example periodic system tasks or Unix daemons consume computational resources.

Research contributions of this paper include the following:

– The design and implementation of a Reliable Autonomous Programs (RAMP) architecture—the first fault tolerant Autonomous Mobile Program load management system. RAMPs are designed for MRS and implemented using ROS (Sect. 3).
– Demonstrating that in loaded and unloaded systems RAMPs are likely to achieve an optimal or near-optimal distribution and hence efficient resource utilisation in teams of robots. RAMPs have low overheads compared to optimal scheduling in unloaded systems (Sect. 4).
– Demonstrating that RAMPs successfully complete in the presence of simultaneous, or successive, robot failures and network failures, while preserving either optimal or near-optimal distribution (Sect. 5).

2 Background

This section presents an overview of related work on load management in both distributed systems and in teams of robots (Sect. 2.1). Autonomous Mobile Programs (AMPs) are described in more detail (Sect. 2.3), together with the Robot Operating System, ROS (Sect. 2.4) and state-of-the-art in robotics' fault tolerance (Sect. 2.5).

2.1 Load management

Load management and task distribution in robotics have a number of meanings depending on the area of research and interest. Often "load management" refers to distribution of a physical weight by a robot (or within a group of robots) to ensure stability in handling and carrying objects; while task distribution often refers to actions (work) that robots perform to complete some large task, for example, monitoring a perimeter while other robots map specific parts of the office space. In this paper we address distribution of computational load to enable robots to complete their computational tasks efficiently and utilise available computing resources.

General Distributed Systems. Load management in distributed systems has been widely researched, e.g. (Lopes and Menascé 2016; Casavant and Kuhl 1988; Rotithor 1994; Coulouris et al. 2011). The taxonomy in (Casavant and Kuhl 1988) focuses on global load management. There, the problem is deciding *where* to execute a process, in contrast to local load management which focuses on how to assign processes on a single-processor systems. According to this taxonomy, global management can be further categorised into static or dynamic.

In *static* load managers the decision where tasks will execute is taken before the execution of the program, and the information necessary for making this decision is assumed to be known beforehand. This includes task execution times, processing resources of the different nodes, and communication times. Unavailability of such information in many cases is the fundamental drawback of static load managers.

In contrast, in *dynamic* load managers the decision where tasks to run happens during program execution. System-state information is used to make the decisions. The assumption of prior knowledge about the system in this type of managers is minimal, which is one of the biggest advantages of dynamic load management. RAMPs provide dynamic load management.

The taxonomy in Rotithor (1994) further explore dynamic load management algorithms, categorizing them as centralised or decentralised. In centralised load managers, one agent is responsible for collecting state information of all the other agents in the system. This central agent then decides where each task should be allocated. The advantage of this approach is that there is low overhead during estimation where to assign tasks. However, the drawbacks are poor responsiveness in a large scale system and the fact that having a single central resource is failure-prone.

In decentralised load managers, each agent in the system is responsible for collecting state information and deciding where their tasks should be executed. This makes the system more tolerant to failures, but the overhead of maintaining accurate state information across all nodes can hinder the scalability of this approach. Despite the drawbacks of decentralized load managers, its advantages of improved fault tolerance can prove important when working with teams of robots. RAMPs provide decentralised load management.

2.2 Multi-robot systems

Lack of computing resources available to a single robot is a well known issue (Sarker et al. 2019; Lan et al. 2018). While robots become more and more computationally powerful their computation demands also grow in particular in the areas of image recognition, navigation, and data analysis. Approaches used to address the issue can be divided into two types: algorithm optimisation and computation offloading. The two types are not mutually exclusive and used simultaneously to complement each other.

Algorithm optimisation is widely used to speed-up computation and reduce power consumption (Parhi 2018), however, it is often not sufficient on its own due to sheer volume of information and the speed with which robots are expected to react to events, especially in dependable and safety critical scenarios (Gouveia et al. 2014; Chen et al. 2018).

Computation offloading explores opportunities to utilise computing resources outside of the robot (Dey and Mukherjee 2016). Depending on the distance to where computation is sent, the following types of computation offloading are distinguished: cloud robotics, offloading to local servers, edge computing.

Cloud or elastic (Hu et al. 2012) robotics approaches assume an Internet connection that enables the robots to utilise on-demand hardware and software resources. Cloud robotics is the focus of significant research effort, e.g. Du et al. (2017), Afrin et al. (2019), and faces some challenges. Offloading to the cloud resources raises security concerns, e.g. the majority of industrial robots operate behind a firewall. Moreover to be effective the internet connection must be fast and reliable. This is infeasible in many situations, e.g. in outdoor, urban street, or disaster sites. In such situations an approach like RAMPs that shares computational load at the edge is more appropriate.

Computation offloading to local servers is particularly suitable for latency-sensitive applications, large-scale distributed control systems, and geo-distributed applications (Botta et al. 2019). Unlike cloud robotics local computation offloading does not require an internet connectivity and is less exposed to the "outside" world (Cano et al. 2018). However, these approaches still require additional computing resources other than robots themselves which can be expensive and technologically difficult to provide when robots work outdoors (e.g. farming field) or are deployed for a limited amount of time (e.g. disaster sites, hired (rented) robots).

The limitations of local server offloading are addressed by distributing computation load at the edge, e.g. to end devices (like robots), routers, switches, or access points (Yi et al. 2015). In the context of load distribution, teams of robots are also known as robotic clusters (Marjovi et al. 2012; Gouveia et al. 2014). Edge load distribution approaches appeal to scenarios where neither access to the internet resources is required (e.g. due to security concerns, lack of stable internet connection) nor other computing resources are available (e.g. due to security concerns, lack of infrastructure and/or expertise to set it up computation sharing, costs associated with acquiring and maintaining servers).

Conceptually RAMPs can offload to the cloud, to local servers, or to a range of edge devices. In this paper, however, we focus only on offload within MRS to explore the feasibility of the RAMPs approach for scenarios where no other computing resources other than the robots themselves are available, e.g. short-term robot deployment at manufacturing, rescue missions, farming. For RAMPs to work the robots in the MRS must be able to communicate over a network and have common software to support RAMP relocation and execution (Rizk et al. 2019). The robots in the MRS may have identical or different compute capabilities, and we term these homogeneous MRS and heterogeneous MRS respectively. There is extensive evidence that AMPs can effectively manage load on both heterogeneous and homogeneous systems (Deng et al. 2010; Chechina et al. 2010).

2.3 Autonomous mobile programs

Autonomous Mobile Programs (AMPs) have been developed to manage load of dynamic, and potentially large, networks (Deng et al. 2006). AMPs are autonomous agents that are aware of their computational resource needs. They migrate within a network to reduce program completion time and fully utilise the network resources.

AMPs aim to reduce their completion time, which makes them similar to ethological models such as ant colony optimisation algorithms (Zhang and Zhang 2010), i.e. while exploring the space in search for food, ant agents leave pheromone to communicate to other ants on the path to the place where the food has been found. The shorter path between the colony and the food, the more ants travel that path, and hence the higher the pheromone level on that track. However, while ant algorithms seek the fastest path and use other mobile agents' feedback to choose the best path, AMPs estimate the current state of the network and seek better resources.

A collection of AMPs performs decentralised dynamic load balancing. This means that there is no central agent that decides where programs should be scheduled, but rather each AMP makes the decision where and when to migrate itself. This decision happens during the execution of the program. Since AMPs are using constantly updated information about the state of the system, they can operate on a dynamic network. To decide whether and where to move, AMPs use the following cost model:

$$T_{comp\_here} > T_{comp\_there} + T_{comm}, \tag{1}$$

where T_{comp_here} is the remaining computation time on the current location, T_{comp_there} is the remaining computation time on the best available remote location, T_{comm} is the communication and coordination time required to move the AMP. If (1) is `true`, then the AMP moves. To reduce AMP coordination time each host has a *load server*. The load servers are similar to black boards; they only collect state information that is later used by the AMPs to decide where to move.

Here we use simple techniques to estimate computation and communication times using information about the structure of the tasks. For example measuring how much of the primary loop nest has been completed, and using information from the load server to estimate communication times (Sect. 3.2). More sophisticated techniques can be used. For example estimating computation time using profiling tools (Kattepur et al. 2017), such as Linux `perf`[4] or `trend prof` (Goldsmith et al. 2007). Other techniques include performance approximation approaches based on hidden Markovian models (Abeni et al. 2017), stochastic Markovian and machine learning models (Bhimani et al. 2017), and performance models (Venkataraman et al. 2016). Communication times could be estimated using techniques like regression models (Hadidi et al. 2018), or latency prediction models (Huai et al. 2019).

AMPs, like other distributed load balancing systems (Schlegel et al. 2006), can exhibit *greedy effects* as a result of non-optimal relocation. This happens due to AMPs making a *locally* optimal choice, where the information about the state of each actor in the system can be insufficient or inaccurate. To ameliorate this problem, a modification of AMPs, called cNAMPs has been proposed (Chechina et al. 2010).

cNAMPs are negotiating AMPs that use a competitive scheme to reduce two types of greedy effects: location thrashing and location blindness. Location thrashing occurs when two or more AMPs decide to relocate to one particular location based on the same information, causing overloading of the target; while location blindness occurs due to AMPs' lack of information about the remaining execution time of each other. cNAMPs address AMPs' location thrashing which is the cause of the majority their of redundant movements, i.e. rather than moving immediately to the chosen host, a cNAMPs first sends a request to make sure that the host is still a suitable option and books a space for relocation. cNAMP sequence diagram is presented in Table 1.

2.4 Robot operating system

Various robot frameworks have been developed over the last decade (Kramer and Scheutz 2007; Mohamed et al. 2009), and the Robot Operating System (ROS) (Quigley et al. 2009) is one of the most widely used. ROS is an open-source project and has a large community of contributors who have developed support for many hardware platforms and programming languages. The main philosophical goals of ROS include being tools-based, multi-lingual, thin, and peer-to-peer.

The communication infrastructure of ROS is based on message-passing (Open Source Robotic Faundation 2018). Each process in ROS is defined as a "node". These nodes form a graph that keeps track of all the processes and the communication between them. Each component in a robot that performs some sort of I/O or computation is expected to be delegated to a separate node, thus reducing the complexity of the system. ROS nodes communicate with each other using an anonymized form of the publisher-subscriber pattern (Eugster et al. 2003).

RAMPs are agnostic of the middleware, and we have chosen ROS only because it is popular robotic middleware. Although ROS has two major issues for developing scalable and reliable robotic systems, *RAMPs are not dependent on either limitation*.

ROS has a single point of failure: the *master* node (*roscore*). The master node is responsible for naming and registration of other nodes and acts as a node discovery hub. An alternative RAMP implementation could avoid having such a single point of failure, e.g. using multi-master ROS extensions (Tiderko et al. 2016; Tardioli et al. 2019), or ROS2,[5] or some completely different robotic middleware like (Sahni et al. 2019).

The ROS communication model has scalability and robustness limitations. Communication between ROS nodes can be publish/subscribe, RPC, or via a global parameter server. As ROS adopts a graph to store and manage node connections, scalability issues arise for large numbers of nodes (Lutac et al. 2016). To enhance the scalability and robust-

4 https://perf.wiki.kernel.org/index.php/Main_Page

5 https://design.ros2.org/articles/why_ros2.html

Table 1 cNAMP sequence diagram

Time	Location X	Location Y
T1	Execute a chunk of work	
T2	Request current loads from Load Server	
T3	Calculate $T_{here}, T_{there}, T_{comm}$	
T4	If NOT($T_{here} > T_{there} + T_{comm}$) then GOTO T1	
T5	Send a representative to Loc.Y	
T6	If representative has returned then GOTO T13	Representative arrives at Loc.Y
T7	Execute a chunk of work	Representative requests load from Loc.Y Load Server
T8	GOTO T6	Representative recalculates parameters $T_{@X}, T_{@Y}, T_{comm}$
T9		If ($T_{@X} > T_{@Y} + T_{comm}$) then Representative notifies local Load Server of the move
T10		Representative returns to Loc.X
T11	Representative arrives at Loc.X	
T12	Representative records the decision and terminates	
T13	If the decision is to stay	
	then GOTO T1	
	else cNAMP notifies local Load Server of the decision and moves to Loc.Y	
T14		cNAMP arrives at Loc.Y
T15		Repeat steps from T1

ness of distributed robotics systems (Erős et al. 2019) ROS2 uses the DDSI-RTPS (DDS-Interoperability Real Time Publish Subscribe) protocol[6] instead of ROS's TCPROS and UDPROS protocols. So adopting ROS2 could improve the scalability and reliability of a RAMPs implementation.

2.5 Fault tolerance in robotics

The RAMPs concept is closely related to *resilience engineering* in robotics. However, to date the main approach in resilient robotics is to use robust controllers (Zhang et al. 2017). While robust controllers have proved to be very successful in manufacturing, they are very expensive and require significant engineering resources to program and modify even by the standards of large industrial companies. As a consequence various research studies explore opportunities to distribute control and decision making between various components (Bader et al. 2017; Zhou et al. 2020). RAMPs occupy a radical position in the spectrum of distributed decision making where control over where to execute a computational task is delegated to the task.

Reliability in robotics commonly refers to the ability of the robotic system to continue performing tasks while part of the system has failed, e.g. some component or middleware element has failed. Crucially there is no attempt to automatically recover failed components (Khalastchi and Kalech 2019). Rather than adapting to system failures, RAMPs provide *computational fault-tolerance* by automatically restarting failed computations to enable them to execute to completion. This model is widely used in distributed systems.

3 RAMP design and implementation

This section presents the design and implementation of Reliable Autonomous Mobile Programs (RAMPs) an AMP-inspired load balancing mechanism for multi-robot systems operating on unstable networks and without external compute resources. Key novelties are the lightweight fault tolerance and decision making mechanisms.

3.1 RAMP design

The design of RAMPs is strongly inspired by Autonomous Mobile Programs (Sect. 2.3). Here an Autonomous Pro-

[6] https://index.ros.org/doc/ros2/Concepts/DDS-and-ROS-middleware-implementations/

gram (AP) is the main component of the solution. APs autonomously make the decision where to execute in a network. As with AMPs the choice of where to execute is based on information about hosts in the network. Moreover, the AMP cost model is used to calculate the execution time on different hosts.

There are, however, some important differences between AMPs and our current implementation of RAMPs. Firstly, RAMPs use weak mobility—only the program data is sent to the remote robot instead of the code that needs to be executed, reducing communication. The reduction in communication is program dependent: programs may have large or small code size, and independently large or small data sizes. A disadvantage of weak mobility is that the program code must be available on the remote robot. However, this drawback is minimal when the team of RAMP robots has a set of predefined tasks and the task code can be distributed to the robots in advance. Moreover, a great advantage of sending only the parameters is that communication time and bandwidth requirements are reduced.

The second difference is that RAMPs do not relocate after the initial distribution. AMPs, on the other hand, can also rebalance. That is, AMPs can relocate from their initial host to resolve scheduling accidents (Deng et al. 2006). Implementing similar relocation strategy for RAMPs can prove useful and is planned as a future work.

Figure 1 shows the four main components of the RAMP system, and their interaction, namely Autonomous Programs, Program Executors, Load Servers, and Load Publishers. The *Autonomous Program* describes the computation to be executed; a robot may have a number of these. Autonomous Programs periodically receive information about the load in the system from their local *Load Server* and make a decision where to execute, as in an AMP system. The Load Server is the other component that is also present in the original AMP design.

There are some subtle differences between the RAMP and classical AMP system architectures. When an Autonomous Program decides where to execute, it offloads the computation to a *Program Executor* on the selected host robot. Each local Load Server collects information about the system's load by interacting with the *Load Publishers* that are present in each robot. The Program Executor and Load Publisher are not present in a classical AMP system. The Program Executor is introduced to facilitate the weak mobility of the solution, by providing a suitable target for Autonomous Programs to offload computation. The Load Publisher exploits ROS's publish/subscribe message passing.

3.2 RAMP implementation

The implementation of the Autonomous Programs and Program Executors utilises ROS's `actionlib` package (Eitan

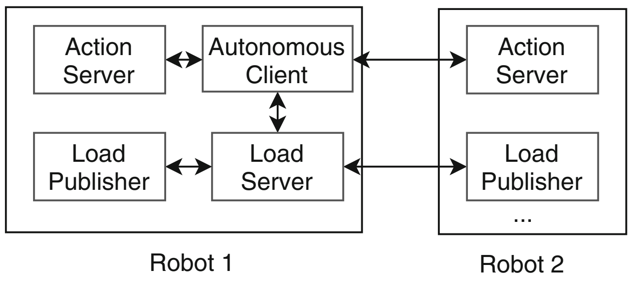


Fig. 1 RAMP system architecture

Marder-Eppstein 2018) that provides tools for creating client-server interactions for long running tasks. The RAMP code is open source and is available under the permissive MIT licence.[7]

The main components are Action Client and Action Server which communicate via a protocol built on top of ROS messages, called *"ROS Action Protocol"*. Figure 2 shows the interaction between Autonomous Programs and Program Executors. A Program Executor contains an Action Server which can accept tasks that match its specification. The Autonomous Program uses an Action Client to send computation to the Action Server in the Program Executor.

Before moving an Autonomous Program needs to decide whether to move, and where to move to. This is done by calculating the cost of executing on each available robot. The Autonomous Program first requests information about the load in the MRS from the local Load Server via an exposed ROS service. Then it calculates the cost using the AMP cost model (Eq. 1 in Sect. 2.3). The communication and computation times are specific to the computation to be executed and are obtained by profiling. For example the computation costs of the route planning program used in Sect. 4 are obtained by timing the program on an unloaded robot, and communication costs by measuring the time to communicate the parameters on an unloaded network.

The Load Server is implemented as a ROS service that returns load information about the system when prompted. The main difference between the RAMP and AMP Load Servers is the communication patterns. Instead of periodically querying robots about their load, RAMP Load Servers use the publisher/subscriber pattern. For that a new component is introduced to the system—the Load Publisher. Each robot contains a Load Publisher that periodically publishes the current load of the robot to a ROS Topic dedicated to this robot. The Load Server on another robot can subscribe to this ROS Topic and receive updates about the CPU and memory load of this robot.

The RAMP implementation includes a mechanism to minimise scheduling accidents, i.e. to address the AMP greedy effect (Sect. 2.3). A simple form of negotiation is adapted

[7] https://github.com/bbstk/offload

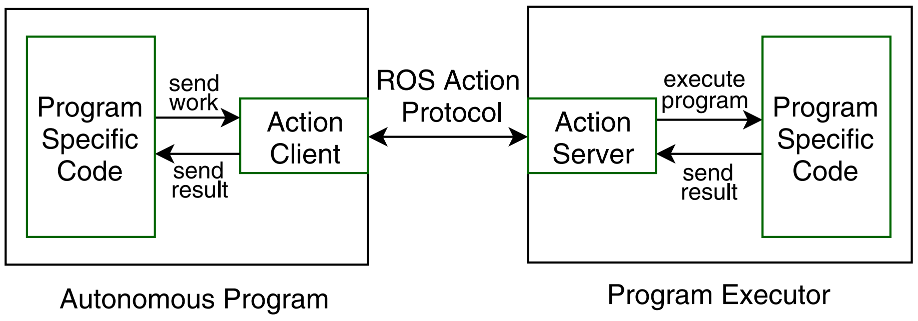


Fig. 2 Interaction between autonomous programs and program executors

to reduce this effect. Local Autonomous Programs pre-emptively notify the local Load Server of their intention to move computation to another robot. Thus, the Load Server updates the load information about the target robot with minimal delay, reducing the possibility for location thrashing. However, this solution does not completely eliminate the greedy effect of Autonomous Programs, and is restricted to Autonomous Programs originating from the same robot. A more elaborate negotiation scheme, like the one used in cNAMPs (Table 1), can be utilised in the future.

3.3 RAMP fault tolerance mechanisms

Multi-robot systems are complex with many components that may fail. Creating a load management system that can handle every possible failure is an enormous task. The RAMP design includes mechanisms to recover from some failures and the current implementation recovers from two classes of failures common in teams of mobile robots: robot and network failures (Carlson et al. 2004). Robot failures include unexpected shutdown, restart, or loss of power, e.g. due to an exhausted battery. The network failures that are tolerated consist of losing network connection between robots, e.g. due to a hardware/software malfunction or a robot getting out of range.

Table 2 summarises the failures in multi-robot systems with respect to whether they are addressed in this solution or not. In the table, the *origin* robot is where an AP is created. A *non-origin* robot is one which executes an AP that originates from another robot. The introduction of this concept of *origin* is important as the current RAMP implementation makes two main assumptions:

- An Autonomous Program is relevant only to the origin robot.
- A failure in a non-origin robot should not prevent an Autonomous Program completing.

From these two assumptions it follows that there must be a mechanism for dealing with both network connection failures and robots crashes in non-origin robots. Moreover, network connection failures in the origin robot should not prevent the completion of Autonomous Program. However, if an origin robot crashes, the completion of its Autonomous Program is not important since they are relevant only to the crashed robot.

All fault tolerance mechanisms are limited, and will not address all possible faults. Many of the faults encountered by robot teams are transient as devices and network connections work only intermittently. Likewise faults may be partial, for example, communication bandwidth may be severely reduced at times. The current RAMP implementation utilises some features of the underlying middleware, e.g. when a robot connects (manually or automatically) the load server adds the reconnected (or a newly connected) robot to the list of available resources.

Failure Detection. Two ways of detecting failures are implemented: connection timeouts and healthcheck monitoring. First, when an Autonomous Program wants to offload a computation to a remote Program Executor, a connection needs to be established between the Action Client and Action Server in the corresponding components (Fig. 2). When the Action Client tries to establish a connection, there is a time limit which ensures that an Autonomous Program will not be blocked indefinitely waiting to offload computation. Once the time limit is reached, the Autonomous Program recalculates where to send the computation, without considering the robot that failed to connect. Using this approach, both crash and network connection loss in a remote robot can be detected. Also, if the origin robot loses network connection while an Autonomous Program establishes a connection to a remote Program Executor, this will result in eventually the Autonomous Program executing in the origin robot.

Periodic healthcheck (or heartbeat) messages are the second means of detecting failures in RAMP systems. These healthcheck messages are implemented as sending simple pings between an origin and a remote robot. Once again, this approach allows for the detection of both robot crashes and network failures. If a remote robot fails during the execution of an offloaded task, the Autonomous Program can detect this failure and send the computation somewhere else.

Table 2 Robot team failures

Failure	Mitigation
Tolerated by RAMPs	
Non-origin robot failure	
Hardware/software failure, power outage, etc.	
Non-origin robot communication failure	
Origin robot communication failure	Transient failure treated as permanent
Not tolerated by RAMPs	
Origin robot permanent or transient failure	Task replication could be added
Network issues, e.g. reduced bandwidth	Could be monitored for patterns
	To get more accurate T_{comm} in (1)
Any sensors, motors, cameras, etc.	Fault escalation could be added
Fail without causing a robot crash	
ROS master node failure	Could be avoided, e.g. using ROS2

Failure Recovery. When a failure is detected the failed robot is *"blacklisted"* on the local Load Server. The Load Server removes the failed robot from the system load information, thus preventing other local Autonomous Programs from offloading computation there. The failed robot stays blacklisted until the Load Server receives a message from the ROS Topic dedicated to the load information of this robot.

Another feature related to the fault tolerance of the solution is the random back-off period after failure detection. When a failure is detected by a robot, the robot waits a random amount of time up to a configurable maximum. For computations that require several minutes to complete, a back-off period of up to 10–20s can prove beneficial, since finding an optimal robot to execute on can compensate for the additional waiting time. This approach has two benefits. Firstly, the wait time allows for other failures in the system to be detected and the unavailable robots to be removed from the local Load Server. This is particularly important when a group of robots fail at the same time, or when the origin robot loses network connection. Having this back-off period reduces the number of redundant attempts to move to an unavailable robot. The second benefit of this approach can be seen when several tasks are offloaded to a robot which then fails. In this scenario, all these tasks will try to move from the origin robot again somewhere else at the same time which may cause location thrashing. However, the random back-off period reduces the chance of this happening.

4 Performance evaluation

This section evaluates the performance of RAMPs as a load balancing technology and investigates the following research questions:

– How well do RAMPs balance load in unloaded teams of robots (Sect. 4.1.1), and loaded teams of robots (Sect. 4.2.1)?
– What is the performance benefit of RAMPs compared to executing on a single robot for unloaded teams of robots (Sect. 4.1.2), and loaded teams of robots (Sect. 4.2.2)?
– How close does a system of RAMPs approach the optimal load distribution in an unloaded team of robots? (Sect. 4.1.2)
– How does RAMP load distribution compare with Round Robin distribution in a loaded team of robots? (Sect. 4.2.2)

All experiments are performed on a network of homogeneous robot hardware, namely a set of five immobile SunFounder car kit robots. The intention is to conduct the evaluation on a real, if relatively simple, multiple robot system. Each robot uses Raspberry Pi 3 Model B with a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB of RAM, Ubuntu 16.04. Robot communicate wirelessly via a router. Although battery power is available, we simplify experimentation with large teams by using standard mains power, and artificially induce the failures caused by mobility.

The range of robotic hardware means that it is hard to select a "typical"' platform, and Raspberry Pi 3 with Ubuntu has the benefit of being a well studied hardware/OS platform. While the majority of modern robots have greater computational power than a Raspberry Pi 3, the load management principles for an MRS based on this modest architecture are valid for more powerful architectures. AMPs have been extensively demonstrated on heterogeneous hardware (Deng et al. 2010).

For the purpose of the experiments the RAMPs could perform any substantial computation, e.g. SLAM. Here we choose route planning, and the planner is encoded in the

MiniZinc 2.1.6 constraint programming language (Nethercote et al. 2007), and uses Gecode (Team 2018) to solve the model. The RAMPs represent a semi-realistic scenario, e.g. a warehouse robot encounters an obstruction and generates RAMPs to explore routes starting from different adjacent locations. A sufficiently large instance of the problem is selected to simulate a scenario where an offloading solution can be beneficial. Solving a single instance of the problem on one Raspberry Pi takes around 50s. Each experiment is repeated 3 times. The results of the experiments are available on github.[8]

4.1 Unloaded MRS performance

The first set of experiments investigates the load balancing capabilities of RAMPs in unloaded homogeneous MRS. Teams of size 2, 3, 4, and 5 identical robots are used to evaluate how RAMPs scale with different team sizes. In the experiments, 5, 10, 15, 20, 25, and 30 *identical* RAMPs are initiated on a single robot. The runtime is recorded together with the final distribution of RAMPs among the robots in the team.

To measure the speedups obtained by offloading work within the team, the same configuration is executed on a single robot. To compare with the optimal distribution for identical task sizes and compute resources, we compare with Round Robin scheduling. In Round Robin, the tasks are distributed equally in circular order to the robots.

4.1.1 RAMP distribution

Table 3 shows the distributions obtained for the different numbers of RAMPs and robots for the median run out of three. The results show that the distribution is most often *optimal* or *near optimal*. In an *optimal* distribution, it is not possible to improve the load balance by moving a RAMP. In a *near optimal* distribution, moving a single RAMP will produce an optimal distribution.

The results of experiments show that 46% of the distribution are optimal (33 distributions), 43% are near optimal (31 distributions), and 11% are not optimal (8 distributions). For example, in Table 3, with 5 RAMPS the distribution is optimal, i.e. the RAMPs are distributed as evenly as possible among the robots. However the distribution of 10 RAMPs with team size of 5 is non-optimal as R1 has 1 RAMP and R3 has 3 RAMPs, so moving a RAMP from R3 to R1 would produce an optimal distribution. The deviations from the optimal distribution increases with the introduction of more RAMPs to the system. The biggest discrepancy in Table 3 is when there are 25 RAMPs executing on 3 robots—R1 has 9 RAMPs, R2 has 10 RAMPs, and R3 has 6 RAMPs. Such

Table 3 RAMP distribution on an unloaded system

Robot RAMPs	Size	R1	R2	R3	R4	R5
5	5	1	1	1	1	1
	4	1	2	1	1	
	3	1	2	2		
	2	2	3			
10	5	1	2	3	2	2
	4	2	3	2	3	
	3	3	4	3		
	2	5	5			
15	5	2	3	3	3	4
	4	3	4	4	4	
	3	4	5	6		
	2	7	8			
20	5	3	4	4	5	4
	4	4	5	5	6	
	3	7	6	7		
	2	8	12			
25	5	4	6	5	5	5
	4	5	6	8	6	
	3	9	10	6		
	2	11	14			
30	5	7	5	6	6	6
	4	7	9	9	5	
	3	9	10	11		
	2	15	15			

uneven loads are common in distributed load managers, and can be resolved by sharing more information or by relocating work, as for example cNAMPs do (Chechina et al. 2010).

We conclude that *the distribution of RAMPs in unloaded MRS is very likely to be optimal or near optimal:* 89% or 64/72 distributions in the experiments above. Given that other AMP implementations also maintain good load distributions over networks of loaded/unloaded and homogeneous/heterogeneous computers (Deng et al. 2010; Chechina et al. 2010), the results in this section and the next can be viewed as demonstrating that RAMPs are correctly and effectively implemented for teams of robots.

4.1.2 RAMP performance

Figure 3 shows the RAMPs' runtime and speedup relative to executing on a single robot, and are the median of three executions. The corresponding RAMP distributions are shown in Table 3.

With 5 RAMPs the time to complete is similar in the four experiments with different team sizes. This is as expected as the Rapsberry Pis are quad core, and the route planning tasks

[8] https://github.com/bbstk/offload/blob/master/RawData.xlsx

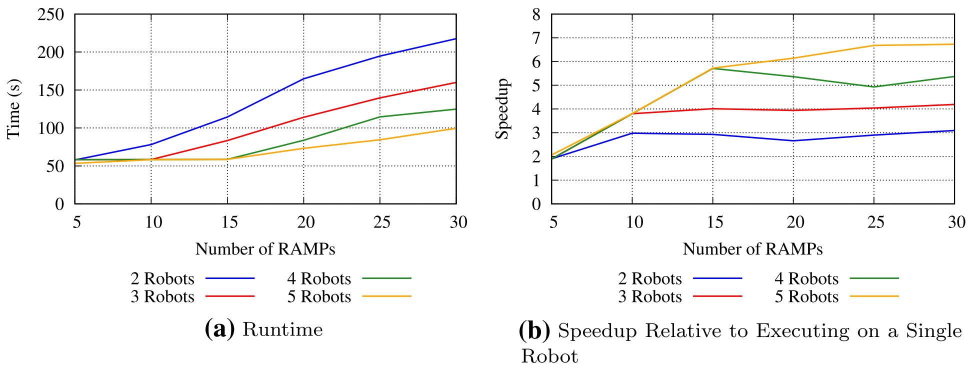


Fig. 3 RAMP speedup and runtime on an unloaded system

utilise just one core. Thus, a robot can compute four route planning tasks with similar speed to computing a single one. With 10 RAMPs, the team of two robots is outperformed by the bigger teams. Table 3 shows that at 10 RAMPs, the team of size 2 has 5 RAMPs distributed to each robot. Therefore, two RAMPs need to share a CPU core resulting in performance degradation. This also occurs with 20 RAMPs on teams of 4 and 5 robots. Predictably, increasing the number of RAMPs results in bigger performance penalties for the smaller teams. With 30 RAMPs the team of 5 robots is more than twice as fast as the 2 robot team.

Comparison with a Single Robot. The speedup relative to executing on a single robot is shown in Fig. 3b. It shows that at 5 RAMPs all team sizes outperform the single robot by a factor of 2. Increasing the number of route planning tasks results in increased speedup. The teams of two and three robots reach maximum speedups of 3 and 4 respectively. With 4 and 5 robots, there is a steady increase in speedup until 15 RAMPs, reaching almost a factor of 6. With the introduction of more RAMPs the 4-Robot team fluctuates between a speedup of 5 and 6, while the 5-Robot team shows a continuous gradual increase.

Comparison with Optimal Work Distribution. Round Robin scheduling of uniform tasks in an unloaded system is optimal. Figure 4 compares the RAMP and Round Robin runtimes for the RAMP and Robot configurations used in the previous experiments

RAMPs have slightly longer runtimes than Round Robin, at most 23% slower in our experiments. The performance difference between the two approaches is small with up to 15 tasks and increases for 20, 25, and 30 tasks.

The biggest difference is for 20 RAMPs on 2 robots, where Round Robin completes in 137s, and RAMPs in 169s. This discrepancy is caused by a sub-optimal distribution of RAMPs. Table 3 shows that for 20 RAMPs in the team of size 2, R1 has 8 RAMPs and R2 has 12 RAMPs. This is one of the biggest deviations from the optimal distribution observed in these experiments and it shows what impact sub-optimal distributions can have on the performance.

A comparison between an optimal distribution achieved by RAMPs and Round Robin can be observed in Fig. 4c, 15 Programs. There the RAMPs are distributed as (3–4–4–4). Using Round Robin leads to 57s completion time, while RAMPs take 58s—only around 2% slower. Another result from perfect distribution can be observed in Fig. 4a, 30 Programs. There RAMPs are distributed as (15–15) between two robots and are around 6% slower than using Round Robin - 216s to 204s.

We conclude that, on unloaded MRS RAMPs *(a) effectively distribute the work amongst the team, reducing the runtime (Fig.* 3a*), and hence produce speedups corresponding the additional hardware available, e.g. a maximal speedup of 6.7 on 5 robots* (Fig. 3b), and *(b) produce similar runtimes to the optimal Round Robin scheduling in most cases, and never more than 23% worse* (Fig. 4).

4.2 Loaded MRS performance

While experiments on unloaded MRS allow us to analyse performance against expectations, they are unrealistic for real robot teams. We investigate the performance of RAMPs on *loaded MRS* using 5 robots where 1, 2, 3, and 4 of the robots are loaded by running 8 CPU intensive tasks using the Linux *stress* tool, resulting in 100% CPU usage. 5, 10, 15, 20, 25, and 30 route planning RAMPs are initiated on a single robot and the distribution and time to complete are recorded.

Evaluating RAMPs in a system where robots have different level of available resources allows for further investigation of their load management capabilities. Moreover, the additional loads emulate heterogeneous teams of robots since the robots have different available compute capabilities. RAMPs should prefer execution on a host with more available compute resources.

4.2.1 RAMP distribution

Table 4 shows the mean distribution of RAMPs from 3 runs. When the number of RAMPs and loaded robots is small,

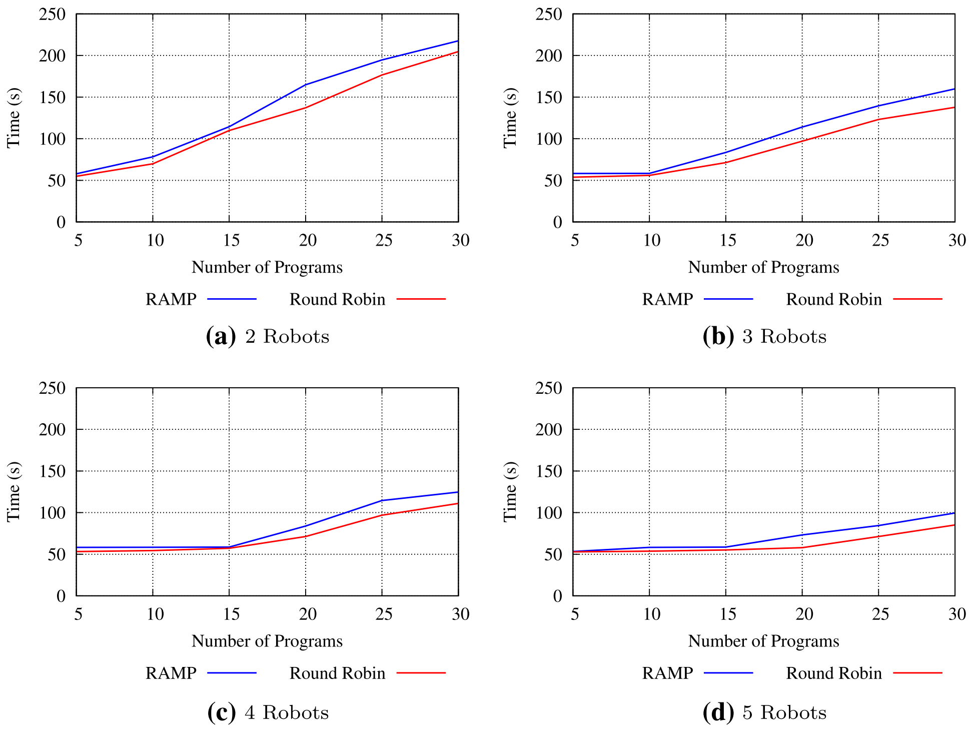


Fig. 4 RAMP and round robin runtime on an unloaded system

RAMPs successfully avoid executing on busy robots. For example, the distribution of 5 and 10 RAMPs in teams with up to 3 busy robots results in no RAMPs on the loaded robots.

Increasing the number of RAMPs and the number of loaded robots leads to some RAMPs eventually executing on the loaded robots. This can first be seen with 5 RAMPs and 4 busy robots and becomes common when there are more than 20 RAMPs. RAMPs executing on the loaded robots happens due to the initially non-busy robots reaching a high level of resource utilisation. When all available hosts for a RAMP are at maximum resource utilisation, the RAMP randomly decides where to execute. As 4 RAMPs on a single robot utilise 100% of the cores, the difference between the number of RAMPs on a loaded and unloaded robot is often close to 4, e.g. Table 4 shows 15 RAMPs 3 and 4 loaded robots.

We conclude that *RAMPs successfully avoid executing on heavily loaded robots if there are robots with more available resources*.

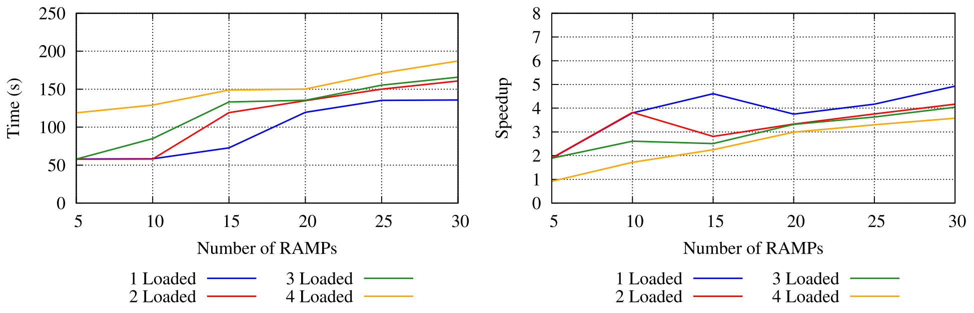
4.2.2 RAMP performance

Figure 5 shows the runtime and speedup for executing 5, 10, 15, 20, 25, and 30 RAMPs on a system where 1, 2, 3 or 4 of the robots each have 8 CPU intensive tasks as additional load. The team that has only 1 loaded robot unsurprisingly shows the best results while the team with 4 busy robots performs the worst. The difference between the teams with 2 and 3 loaded robots is small, especially when the number of RAMPs increases.

Comparison with a Single Robot. Figure 5b shows the speedup of using RAMPs on a loaded system relative to executing all tasks on a single robot with no additional load. Despite the high resource utilisation on the robots, teams with up to 4 loaded robots achieve more than three times speedup when there are more than 20 RAMPs. When there is 1 loaded robot and 30 RAMPs the speedup reaches 5.

Comparison with Round Robin. Figure 6 shows the runtime of 5, 10, 15, 20, 25, and 30 route planning tasks using RAMPs and Round Robin scheduling. RAMPs outperform Round Robin, which is unsurprising given that Round Robin does not consider resource availability. This is best observed when there are 1, 2, and 3 loaded robots (Fig. 6a–c), with RAMPs finishing more than two times faster in Fig. 6a and b at 10 programs. When there are 4 loaded robots RAMPs have only a slight advantage over Round Robin (Fig. 6d). This is due to the quick exhaustion of available resources which reduces the effect of subsequent decisions where to execute.

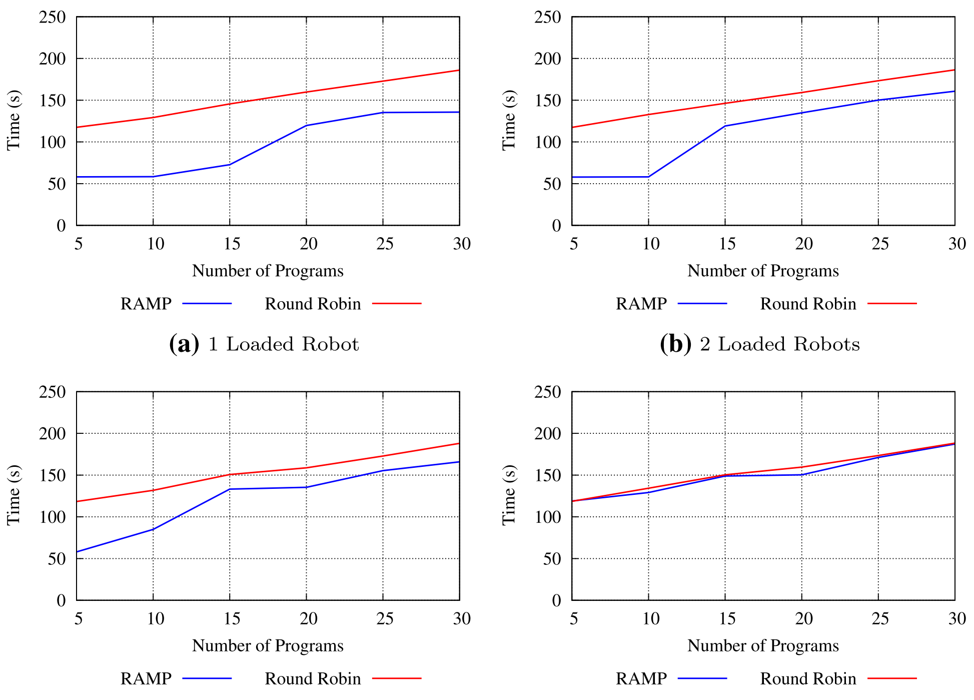
We conclude that *(a) despite having heavily loaded members, a team of five robots can achieve up to five time speedup for 30 RAMPs* and *(b) RAMPs outperform Round Robin when there is some additional load on the system*.



(a) Runtime

(b) Speedup Relative to Executing on a Single Unloaded Robot

Fig. 5 RAMP runtime and speedups on a loaded system



(a) 1 Loaded Robot

(b) 2 Loaded Robots

(c) 3 Loaded Robots

(d) 4 Loaded Robots

Fig. 6 RAMP and round robin runtime on a loaded system of 5 robots

5 Fault tolerance evaluation

This section evaluates the fault tolerance of RAMPs on a team of robots and investigates the following research questions:

– Is it possible for RAMPs executing on a non-origin robot to eventually complete in the presence of an unexpected crash or shutdown of an arbitrary number of non-origin robots? (Sect. 5.1)
– Is it possible for RAMPs executing on a non-origin robot to eventually complete in the presence of an unexpected network connection loss between the RAMPs' origin and an arbitrary number of non-origin robots? (Sect. 5.2)
– How well can RAMPs preserve a balanced load in a multi-robot system in the presence of robot or network failures affecting a subset of the robots? (Sects. 5.1, 5.2, 5.3)
– Do robot crashes and network connection failures have a uniform impact on RAMPs? (Sects. 5.1, 5.2, 5.3)

The experiments below are both simple and extreme in the sense that the robot and network failures are both total and permanent. Real robot teams more frequently encounter partial and intermittent failures. However building realistic models of intermittent and partial failures for a given application domain is non-trivial. Likewise establishing the properties of collections of RAMPs executing on a team of

Table 4 RAMP distribution on a loaded system (R*—Robot; L**—loaded robots)

R* RAMPs	L**	R1	R2	R3	R4	R5
5	UUUUL	0.67	1.34	1.34	1.67	**0**
	UUULL	1	2	2	**0**	**0**
	UULLL	2.33	2.67	**0**	**0**	**0**
	ULLLL	3.67	**0.67**	**0.33**	**0.33**	**0**
10	UUUUL	1.67	3.33	3	2	**0**
	UUULL	3	3.67	3.33	**0**	**0**
	UULLL	4	5.67	**0**	**0**	**0.33**
	ULLLL	6.67	**1**	**0.67**	**1**	**0.67**
15	UUUUL	3	4.33	3.33	4.33	**0**
	UUULL	3	4	6.33	**1**	**0.67**
	UULLL	5.33	5.67	**1.67**	**0.67**	**1.67**
	ULLLL	6.67	**2.33**	**2**	**2**	**2**
20	UUUUL	4	5	4.33	5.33	**1**.33
	UUULL	5.33	6.33	5.67	**1.67**	**1**
	UULLL	7.67	7.67	**1.33**	**1.33**	**2**
	ULLLL	8.67	**2.33**	**3**	**3**	**3**
25	UUUUL	5.67	6.33	6	5.33	**1.67**
	UUULL	5.33	7.33	7.33	**2.33**	**2.67**
	UULLL	8.33	9	**2.33**	**2.33**	**2.33**
	ULLLL	8.67	**4**	**4.33**	**3.67**	**4.33**
30	UUUUL	6.67	7	7	7.33	**2**
	UUULL	6.67	8	7.67	**3.67**	**4**
	UULLL	9.67	12.67	**1.67**	**2.67**	**3.33**
	ULLLL	8.33	**5**	**5.67**	**5.33**	**5.67**

robots operating in such a stochastic environment requires extensive evaluation and analysis. In short these relatively simple experiments enable us to establish the fault tolerance properties of RAMPs simply, understandably, and reproducibly.

The Autonomous Programs executing on the robots each compute a real robotic task, i.e. route planning as may be required by teams of robots operating in many environments, e.g. in a dynamic warehouse. In a real system robots would periodically encounter a barrier to their existing plan and require to re-plan. Crucially a failure in a subset of the robots in a team should not unduly disrupt the functioning of the remaining robots.

For reproducibility and ease of analysis we do not model the stochastic arrival of replanning events, and typically launch a set of APs simultaneously. The route planning computation is expressed in the MiniZinc 2.1.6 constraint programming language (Nethercote et al. 2007), and solved using Gecode (Team 2018). To ensure that the computation does not finish before the failures are introduced to the system, a significantly larger and more time consuming instance of the problem is used in this set of experiments than in Sect.ion 4.

In these experiments we use the same set of five homogeneous car kit robots with Raspberry Pi 3 Model B as in Sect. 4. In each scenario all five robots initially do not have any computation to perform.

Each experiment is repeated 3 times and consists of two distinct stages: *initial distribution* and *post-failure distribution*. All experiments start by initiating 15 RAMPs on one of the robots. The RAMPs then move to other robots, reaching a stable state. This state is recorded as the initial distribution. Due to following a weak mobility paradigm, after moving to a new location RAMPs restart their computations.

The next stage of the experiments is the introduction of failures to the system. This is done using two different approaches: *sequential failures* and *simultaneous failures*. With *sequential failures* a set delay of 20s between individual failures is introduced. This allows sufficient time for RAMPs to move to other robots. The value for the delay was chosen to *guarantee* moving of RAMPs before the next failure. In reality, failure detection and moving in RAMPs takes significantly less than 20s. With *simultaneous failures* the robot crashes or network failures occur at the same time in multiple robots.

5.1 Robot failures

The first type of failure that is examined is unexpected robot crashes and shutdowns. Being able to recover from this kind of failure is important because they are relatively common in practice. Battery exhaustion, hardware faults, and software errors may all cause a robot to shutdown unexpectedly. When this happens to a robot which is executing a RAMP, the failure should not prevent the RAMP from completing. The robot crash in this experiment is produced by switching off the power supply for a robot. Two different scenarios are examined where the robot crashes occur either sequentially (Sect. 5.1.1) or simultaneously (Sect. 5.1.2).

5.1.1 Sequential failures

Table 5 shows the RAMP distribution in a group of five robots where robots fail in succession. Each failure happens after a 20 s delay, which is long enough for the RAMPs affected by the crash to move. D1 shows initial distribution before any failures occur, and then D2–D5 show re-distributions as robots fail one by one. The distributions from three independent runs are presented.

The results show that after a robot failure the RAMPs that were running on the affected robot are successfully restarted on the remaining robots. Moreover, the RAMPs distribution after restarting computation on a new location is *near optimal*. The biggest discrepancy between the number of RAMPs

Table 5 RAMP distributions after sequential robot crashes

	Robot Dist.	R1	R2	R3	R4	R5
Run 1	D1	3	3	3	3	3
	D2	3	4	4	4	
	D3	4	6	5		
	D4	7	8			
	D5	15				
Run 2	D1	2	3	3	3	4
	D2	4	3	3	5	
	D3	5	4	6		
	D4	9	6			
	D5	15				
Run 3	D1	2	3	4	3	3
	D2	4	3	4	4	
	D3	4	6	5		
	D4	7	8			
	D5	15				

on different robots from all three runs is 3. This can be seen in Table 5, Run 2, D4, where R1 has 9 RAMPs and R2 has 6 RAMPs. However, even this distribution differs only by one RAMP from the optimal distribution, which is 8 RAMPs on one robot and 7 RAMPs on another robot.

We conclude that *(a) The RAMPs on a robot that has failed are successfully restarted on other robots, possibly repeatedly* and *(b) the new distribution is optimal or near optimal*.

5.1.2 Simultaneous failures

Table 6 shows the mean distribution of RAMPs from three experiment runs in a system that experience simultaneous robot failures. In these experiments, a number of robots are shut down together without allowing for moving of RAMPs between individual shutdowns. First, the results show that simultaneous robot crashes do not prevent RAMPS from moving to another robot. Moreover, the results are similar to the RAMP distribution when the robot crashes are sequential as shown in Table 5. The distribution of RAMPs in this experiment is also near optimal with a maximum discrepancy of 3, with 15 RAMPS on 2 robots.

We conclude that *(a) The RAMPs on a collection of robots that fail simultaneously are successfully restarted on other robots* and *(b) the new distribution is optimal or near optimal*.

5.2 Network failures

The second type of failures that is examined is unexpected network connection loss between robots. This may occur in teams of mobile robots due to hardware, e.g. network card, failures or the robot moving out of range of a WiFi signal. If the affected robot is a non-origin robot, the RAMPs should be able to recover from such failures. The network connection failures are invoked by switching off the robots' network cards. Again, two different scenarios are examined where the network connection failures occur sequentially (Sect. 5.3.1) and simultaneously (Sect. 5.2.2).

Table 6 Mean RAMP distribution after simultaneous robot crash experiment

Robot Failed	R1	R2	R3	R4	R5
none	2.67	3	3.25	3.08	3
R5	3.67	3.67	3.67	4	
R4,R5	5	5.33	4.67		
R3,R4,R5	7.33	7.67			
R2,R3,R4,R5	15				

Table 7 RAMP distributions after sequential failures of network connections

	Robot Dist.	R1	R2	R3	R4	R5
Run 1	D1	3	3	3	3	3
	D2	4	3	4	4	
	D3	5	5	5		
	D4	9	6			
	D5	15				
Run 2	D1	2	3	3	4	3
	D2	4	3	4	4	
	D3	4	5	6		
	D4	8	7			
	D5	15				
Run 3	D1	3	4	3	3	2
	D2	3	4	4	4	
	D3	5	5	5		
	D4	7	8			
	D5	15				

5.2.1 Sequential failures

Table 7 shows RAMP distribution when robots lose network connections sequentially. D1 shows distribution before the failures occur, and then D2–D5 show RAMP redistributions as network connections fail one by one. The distributions from three independent runs are presented.

The results show that RAMPs are able to successfully restart computation on a different location after their host robot loses network connection. Another similarity with the results from the robot crash experiment observed in Table 7 is

Table 8 Mean RAMP distribution in experiments with simultaneous network connection failures

Robot Failed	R1	R2	R3	R4	R5
none	2.92	3.08	3	3.08	2.92
R5	3.33	3.67	4	4	
R4,R5	4.33	5.33	5.33		
R3,R4,R5	7	8			
R2,R3,R4,R5	15				

the fact that the distributions in all three runs are near optimal. As before (9–6) distribution of 15 RAMPs on two robots is the worst case distribution.

We conclude that *(a) RAMPs successfully move from a robot that has lost a network connection by restarting computation on a different location*, *these new distributions are either optimal or near optimal*, and *(b) sequential robot crashes and network connection failures of non-origin robots have the same impact on RAMPs re-distributions*.

5.2.2 Simultaneous failures

Table 8 shows the mean distribution of RAMPs from three runs in a system that experiences simultaneous network connection failures in robots. The distribution of RAMPs in this experiment is once again near optimal. The results are similar to those shown in the previous experiments (Tables 6 and 7). The results and conclusions are are the same as in Sect. 5.2.1.

5.3 Mixed failures

Mixed failure experiments examine cases when system experiences both robot crashes and network connection failures. Again, two different scenarios are analysed here: failures occurring sequentially (Sect. 5.3.1) and simultaneously (Sect. 5.3.2).

5.3.1 Sequential failures

Table 9 shows the distribution of RAMPs in a group of five robots where robots fail sequentially. In this experiment network connection failures and robot crashes occur one after another. That is, robot R5 first loses a network connection, then robot R4 crashes, then robot R3 loses a network connection, and finally robot R2 crashes. Between each failure there is a 20-s delay for the affected RAMPs to move. The distribution from three independent runs is presented.

We conclude that, as in the previous experiments *(a) RAMPs are able to successfully restart after their host robot either fails or loses network connectivity*. Moreover, *(b) the resulting re-distributions are once again either optimal or near optimal*.

Table 9 RAMP distributions after sequential mix failures (NF—network failure, RC—robot crash)

	Robot Dist.	R1	R2	R3	R4	R5
Run 1	D1	2	4	3	3	3
	D2	3	4	4	4	NF
	D3	6	5	4	RC	NF
	D4	7	8	NF	RC	NF
	D5	15	RC	NF	RC	NF
Run 2	D1	3	3	3	3	3
	D2	3	4	4	4	NF
	D3	5	4	6	RC	NF
	D4	6	9	NF	RC	NF
	D5	15	RC	NF	RC	NF
Run 3	D1	3	3	3	3	3
	D2	4	4	4	3	NF
	D3	5	5	5	RC	NF
	D4	8	7	NF	RC	NF
	D5	15	RC	NF	RC	NF

5.3.2 Simultaneous failures

Table 10 shows the mean distribution of RAMPs from three runs of the experiment in a system that encounters mixed failures: network connection failures and robot crashes. In total four different scenarios are examined. Each scenario begins with initial distribution where no failures are present. Then failures are introduced in one of the following ways. First, robot R5 loses network connection. In the second scenario simultaneously robot R5 loses network connection and robot R4 crashes. In the third scenario simultaneously robots R5 and R3 lose network connection, and robot R4 crashes. In the last scenario simultaneously robots R5 and R3 lose network connection, and robots R4 and R2 crash.

We conclude that *(a) RAMPs successfully tolerate simultaneous failures in a system that encounters both robot crashes and network connection failures*, and *(b) the resulting distributions are either optimal or near optimal*.

6 Related multi-robot load managers

To the best of our knowledge no previous MRS load management system has been designed for fault-tolerance. That is, RAMPs are the first mechanism for distributing load within robot teams that adapts to hardware and network failures. RAMPs do so by recording tasks offloaded, and redistribut-

Table 10 Mean RAMP distribution in experiments with simultaneous mixed failures (NF—network failure, RC—robot crash)

Robot Failed	R1	R2	R3	R4	R5
none	3	2.92	2.92	3.25	2.92
R5	3.67	4	4	3.33	NF
R4,R5	4.33	5.67	5	RC	NF
R3,R4,R5	7.33	7.67	NF	RC	NF
R2,R3,R4,R5	15	RC	NF	RC	NF

ing them after a failure. Below we discuss related robot load managers.

In many MRS load managers robots offload to servers or to the cloud (Nimmagadda et al. 2010; Cano et al. 2018), whereas RAMPs share load within the MRS, i.e. at the edge. A server-based system that is similar to RAMPS is (Nimmagadda et al. 2010): it exploits weak mobility and relocation decisions are based on a similar cost model. However unlike AMPs that will distribute any computational task that can be costed, the cost model is specialised for motion detection tasks, e.g. it uses the number of frames and the number of pixels to calculate communication and computation time.

In contrast to server/cloud based systems the crowdsourcing manager presented in Huai et al. (2019) exploits idle compute capacity only within an MRS. Like RAMPs the communication latency and computational resources of the remote robots are checked before distributing a computation. In contrast to RAMPs this is another specialised load manager designed for deep-learning computations, i.e. the computation is partitioned into deep-learning layers and latency prediction is for individual layers.

Where RAMPs primarily use dynamic cost and load information, some MRS load managers use far more static information. For example by assuming that computation speed remains constant on a given device, the results of offline profiling can be used to predict task execution time. For example the Deep Neural Network (DNN) system (Hadidi et al. 2018) takes this approach: profiling every hardware system and DNN layer. The optimal distribution of layers can then be precomputed for a given set of devices. Systems exploiting such static information are very effective, and sometimes optimal, for applications with static and fixed task sizes. However RAMPs can effectively distribute tasks with dynamic and varying sizes, albeit with higher runtime overheads e.g. to collect current MRS load information.

7 Conclusion

We propose Reliable Autonomous Mobile Programs (RAMPs)—a novel computation offloading technology for multi-robot systems. RAMPs decide on which robots to execute using a cost model that considers the program's computational requirements and the load in the team. RAMPs are the first ever fault tolerant Autonomous Mobile Programs. Failures are detected using connection timeouts and health-check monitoring; RAMPs recover from the failures by restarting on other robots. Failure prevention is done by "blacklisting" the failed robots; that is, removing the failed robots from the pool of candidates, until they function again (Sect. 3).

The runtime and distribution of a route planning RAMP implementation have been evaluated on loaded and unloaded MRS. The route planning RAMP represents a semi-realistic scenario of a warehouse robot encountering obstructions and generating RAMPs to explore routes with different first moves.

The evaluation on unloaded MRS shows that RAMPs are likely to achieve optimal or near optimal distributions (e.g. in 89% of the 72 distributions measured), and hence achieve good speedups relative to the available hardware, e.g. a maximal speedup of 6.7 on 5 robots. Moreover, in most cases RAMPs exhibit similar runtime to the optimal Round Robin scheduling (Sect. 4.1).

On loaded MRS, RAMPs successfully avoid allocating work to busy robots if there are robots with more available resources. RAMPs on 5 robots achieve variable speedups of 5 with 1 loaded robot, and up to 3.5 with 4 loaded robots. Unsurprisingly RAMPs have better runtime than Round Robin scheduling when any additional load is present (Sect. 4.2). As other AMP implementations also maintain good load distributions over networks of loaded/unloaded and homogeneous/heterogeneous computers (Deng et al. 2010; Chechina et al. 2010), these results demonstrate that RAMPs are correctly and effectively implemented for teams of robots.

Fault tolerance is a key novelty in RAMPs, and they are able to successfully detect and recover from robot and network failures in non-origin robots. In a series of experiments exploring 15 route planning RAMPs, they preserve near-optimal or optimal distribution after either simultaneous or successive failures. There is no difference in RAMPs' toleration of robot failures, network failures, or both (Sect. 5).

Although most of the RAMP configurations in Sect. 4 achieve optimal load distribution, 11% of the distributions are non-optimal. This is a common issue arising from the incomplete information available in distributed load balancing systems. Some negotiation between RAMPs, as in cNAMPs, could alleviate this problem. That is, before relocating, cNAMPs send a request to ensure that the host is still a suitable option and books a space for relocation (Table 1). cNAMPs are shown to be less prone to this location thrashing than AMPs (Chechina et al. 2010).

A feature in the original AMP design, but not currently implemented in RAMPs, is the ability of programs to relocate after the initial distribution. Implementing this for RAMPs would allow them to better handle dynamic system load and recover from non-optimal initial distributions. That is, if a RAMP executes on a robot that becomes overloaded and there are other more suitable robots, the RAMP could relocate to reduce its completion time. We are currently working on adding this feature in RAMPs.

While this paper shows that RAMPs are likely to achieve optimal or near-optimal distribution and have low overhead in small to medium sized teams, RAMPs have yet to be evaluated in large teams of robots, and we are currently seeking to do so. Some other interesting future research directions include exploring the potential of RAMPs to allocate physical actions within a robot team and adopting software containers like Docker (Ismail et al. 2015) for RAMP execution.

References

Abeni, L., Fontanelli, D., Palopoli, L., & Frías, B. V. (2017). A Markovian model for the computation time of real-time applications. In *2017 IEEE international instrumentation and measurement technology conference (I2MTC)* (pp. 1–6). IEEE.

Afrin, M., Jin, J., Rahman, A., Tian, Y. C., & Kulkarni, A. (2019). Multi-objective resource allocation for edge cloud based robotic workflow in smart factory. *Future Generation Computer Systems*, *97*, 119–130.

Bader, K., Lussier, B., & Schön, W. (2017). A fault tolerant architecture for data fusion: A real application of Kalman filters for mobile robot localization. *Robotics and Autonomous Systems*, *88*, 11–23.

Bhimani, J., Mi, N., Leeser, M., & Yang, Z. (2017). Fim: Performance prediction for parallel computation in iterative data processing applications. In *2017 IEEE 10th international conference on cloud computing (CLOUD)* (pp. 359–366). IEEE.

Botta, A., Gallo, L., & Ventre, G. (2019). Cloud, fog, and dew robotics: Architectures for next generation applications. In *2019 7th IEEE international conference on mobile cloud computing, services, and engineering (MobileCloud)* (pp. 16–23). IEEE.

Breazeal, C., Brooks, A., Gray, J., Hoffman, G., Kidd, C., Lee, H., et al. (2004). Tutelage and collaboration for humanoid robots. *International Journal of Humanoid Robotics*, *1*(02), 315–348.

Cano, J., White, D. R., Bordallo, A., McCreesh, C., Michala, A. L., Singer, J., et al. (2018). Solving the task variant allocation problem in distributed robotics. *Autonomous Robots*, *42*(7), 1477–1495.

Cao, Y. U., Fukunaga, A. S., & Kahng, A. (1997). Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, *4*(1), 7–27.

Carlson, J., Murphy, R. R., & Nelson, A. (2004). Follow-up analysis of mobile robot failures. In *Proceedings of the IEEE international conference on robotics and automation (ICRA)* (Vol. 5, pp. 4987–4994). IEEE.

Casavant, T. L., & Kuhl, J. G. (1988). A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, *14*(2), 141–154.

Chechina, N., King, P., & Trinder, P. (2010). Using negotiation to reduce redundant autonomous mobile program movements. In *Proceedings of the IEEE/WIC/ACM international conference on web intelligence and intelligent agent technology (WI-IAT)* (Vol. 2, pp. 343–346). IEEE.

Chen, W., Yaguchi, Y., Naruse, K., Watanobe, Y., & Nakamura, K. (2018). Qos-aware robotic streaming workflow allocation in cloud robotics systems. *IEEE Transactions on Services Computing*.

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). *Distributed systems: Concepts and design* (5th ed.). Boston: Addison-Wesley.

Crestani, D., Godary-Dejean, K., & Lapierre, L. (2015). Enhancing fault tolerance of autonomous mobile robots. *Robotics and Autonomous Systems*, *68*, 140–155.

Deng, X. Y., Michaelson, G., & Trinder, P. (2010). Cost-driven autonomous mobility. *Computer Languages, Systems and Structures*, *36*(1), 34–59.

Deng, X. Y., Trinder, P., & Michaelson, G. (2006). Autonomous mobile programs. In *Proceedings of the IEEE/WIC/ACM international conference on intelligent agent technology* (pp. 177–186). IEEE Computer Society.

Dey, S.,& Mukherjee, A. (2016). Robotic slam: A review from fog computing and mobile edge computing perspective. In: *Adjunct proceedings of the 13th international conference on mobile and ubiquitous systems: Computing networking and services, MOBIQUITOUS 2016* (p. 153158). Association for Computing Machinery, New York, NY, USA.

Du, Z., He, L., Chen, Y., Xiao, Y., Gao, P., & Wang, T. (2017). Robot cloud: Bridging the power of robotics and cloud computing. *Future Generation Computer Systems*, *74*, 337–348.

Eitan Marder-Eppstein, V. P. (2018). *actionlib: Package summary*. http://wiki.ros.org/actionlib/. [Online: 2021/01/08 10:15:02].

Erős, E., Dahl, M., Bengtsson, K., Hanna, A., & Falkman, P. (2019). A ros2 based communication architecture for control in collaborative and intelligent automation systems. *Procedia Manufacturing*, *38*, 349–357.

Eugster, P. T., Felber, P. A., Guerraoui, R., & Kermarrec, A. M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, *35*(2), 114–131.

Garzón, M., Valente, J., Roldán, J. J., Garzón-Ramos, D., de León, J., Barrientos, A., et al. (2017). Using ros in multi-robot systems: Experiences and lessons learned from real-world field tests. In A. Koubâa (Ed.), *Robot operating system (ROS)* (pp. 449–483). Berlin: Springer.

Goldsmith, S. F., Aiken, A. S., & Wilkerson, D. S. (2007). Measuring empirical computational complexity. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (pp. 395–404).

Gouveia, B. D., Portugal, D., Silva, D. C., & Marques, L. (2014). Computation sharing in distributed robotic systems: A case study on slam. *IEEE Transactions on Automation Science and Engineering*, *12*(2), 410–422.

Hadidi, R., Cao, J., Woodward, M., Ryoo, M. S., & Kim, H. (2018). Distributed perception by collaborative robots. *IEEE Robotics and Automation Letters*, *3*(4), 3709–3716.

Hu, G., Tay, W. P., & Wen, Y. (2012). Cloud robotics: Architecture, challenges and applications. *IEEE Network*, *26*(3), 21–28.

Huai, Z., Ding, B., Wang, H., Geng, M., & Zhang, L. (2019). Towards deep learning on resource-constrained robots: A crowdsourcing approach with model partition. In *2019 IEEE SmartWorld, ubiquitous intelligence and computing, advanced and trusted computing, scalable computing and communications, cloud and big data computing, internet of people and smart city innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)* (pp. 989–994). IEEE.

Ismail, B. I., Goortani, E. M., Ab Karim, M. B., Tat, W. M., Setapa, S., Luke, J. Y., & Hoe, O. H. (2015). Evaluation of Docker as edge computing platform. In *2015 IEEE conference on open systems (ICOS)* (pp. 130–135). IEEE.

Kattepur, A., Rath, H. K., & Simha, A. (2017). A-priori estimation of computation times in fog networked robotics. In *2017 IEEE international conference on edge computing (EDGE)* (pp. 9–16). IEEE.

Khalastchi, E., & Kalech, M. (2019). Fault detection and diagnosis in multi-robot systems: A survey. *Sensors*, *19*(18), 4019.

Kramer, J., & Scheutz, M. (2007). Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, *22*(2), 101–132.

Kurfess, T. R. (2004). *Robotics and automation handbook*. Boca Raton: CRC Press.

Lan, G., Benito-Picazo, J., Roijers, D. M., Domínguez, E., & Eiben, A. (2018). Real-time robot vision on low-performance computing hardware. In *2018 15th international conference on control, automation, robotics and vision (ICARCV)* (pp. 1959–1965). IEEE.

Lopes, R. V., & Menascé, D. (2016). A taxonomy of job scheduling on distributed computing systems. *IEEE Transactions on Parallel and Distributed Systems*, *27*(12), 3412–3428.

Lutac, A., Chechina, N., Aragon-Camarasa, G., & Trinder, P. (2016). Towards reliable and scalable robot communication. In *Proceedings of the international workshop on Erlang* (pp. 12–23). ACM.

Marjovi, A., Choobdar, S., & Marques, L. (2012). Robotic clusters: Multi-robot systems as computer clusters: A topological map merging demonstration. *Robotics and Autonomous Systems*, *60*(9), 1191–1204.

Mohamed, N., Al-Jaroodi, J., & Jawhar, I. (2009). A review of middleware for networked robots. *International Journal of Computer Science and Network Security*, *9*(5), 139–148.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., & Tack, G. (2007). MiniZinc: Towards a standard CP modelling language. In *Proceedings of the international conference on principles and practice of constraint programming* (pp. 529–543). Springer.

Nimmagadda, Y., Kumar, K., Lu, Y. H., & Lee, C. G. (2010). Real-time moving object recognition and tracking using computation offloading. In *2010 IEEE/RSJ international conference on intelligent robots and systems* (pp. 2449–2455). IEEE.

Open Source Robotic Faundation. (2018). ROS documentation. http://wiki.ros.org/. [Online: 2021/01/08 10:15:02].

Parhi, D. (2018). Advancement in navigational path planning of robots using various artificial and computing techniques. *Int Rob Auto J*, *4*(2), 133–136.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., & Ng, A. Y. (2009). ROS: An open-source robot operating system. In *Proceedings of the ICRA workshop on open source software* (Vol. 3.2, p. 5). Kobe.

Rizk, Y., Awad, M., & Tunstel, E. W. (2019). Cooperative heterogeneous multi-robot systems: A survey. *ACM Computing Surveys (CSUR)*, *52*(2), 1–31.

Rotithor, H. G. (1994). Taxonomy of dynamic task scheduling schemes in distributed computing systems. *IEE Proceedings-Computers and Digital Techniques*, *141*(1), 1–10.

Sahni, Y., Cao, J., & Jiang, S. (2019). Middleware for multi-robot systems. In H. M. Ammari (Ed.), *Mission-oriented sensor networks and systems: Art and science* (pp. 633–673). Berlin: Springer.

Sarker, V., Queralta, J. P., Gia, T., Tenhunen, H., & Westerlund, T. (2019) Offloading slam for indoor mobile robots with edge-fog-cloud computing. In *2019 1st international conference on advances in science, engineering and robotics technology (ICASERT)* (pp. 1–6). IEEE.

Schlegel, T., Braun, P., & Kowalczyk, R. (2006). Towards autonomous mobile agents with emergent migration behaviour. In *Proceedings of the international joint conference on autonomous agents and multiagent systems* (pp. 585–592). ACM.

Tardioli, D., Parasuraman, R., & Ögren, P. (2019). Pound: A multi-master ros node for reducing delay and jitter in wireless multi-robot networks. *Robotics and Autonomous Systems*, *111*, 73–87.

Team, G. (2018). *Gecode: Generic constraint development environment*. http://www.gecode.org/. [Online: 2021/01/08 10:15:02].

Tiderko, A., Hoeller, F., & Röhling, T. (2016). The ros multimaster extension for simplified deployment of multi-robot systems. In A. Koubáa (Ed.), *Robot operating system (ROS)* (pp. 629–650). Berlin: Springer.

Venkataraman, S., Yang, Z., Franklin, M., Recht, B., & Stoica, I. (2016). Ernest: Efficient performance prediction for large-scale advanced analytics. In 13th {USENIX} *symposium on networked systems design and implementation*({NSDI} 16) (pp. 363–378).

Wang, S., Liu, X., Zhao, J., & Christensen, H. I. (2019). Rorg: Service robot software management with linux containers. In *2019 international conference on robotics and automation (ICRA)* (pp. 584–590). IEEE.

Yi, S., Li, C., & Li, Q. (2015). A survey of fog computing: Concepts, applications and issues. In *Proceedings of the 2015 workshop on mobile big data* (pp. 37–42).

Zhang, T., Zhang, W., & Gupta, M. M. (2017). Resilient robots: Concept, review, and future directions. *Robotics*, *6*(4), 22.

Zhang, Z., & Zhang, X. (2010). A load balancing mechanism based on ant colony and complex network theory in open cloud computing federation. In: *Proceedings of the international conference on industrial mechatronics and automation (ICIMA)* (Vol. 2, pp. 240–243). IEEE.

Zhou, F., Liu, K., Li, Y., & Liu, G. (2020). Distributed fault-tolerant control of modular and reconfigurable robots with consideration of actuator saturation. *Neural Computing and Applications*, *32*, 1–14.

Ivan Valkov is a software engineer at Improbable, UK. His research interests include distributed systems, programming languages, cooperative robotics, and cloud computing. Ivan received his Masters in Software Engineering from University of Glasgow, UK in 2018.

Phil Trinder is a Professor of Computing Science at the University of Glasgow and a leading researcher in parallel and distributed programming languages. He has been an investigator on around 20 major research projects including coordinating the very successful EU RELEASE project to improve the scaling of Erlang; and co-leading the UK MaRIONet network of manycore research excellence. He has over 100 publications in journals, books, or refereed conferences, attracting over 3000 citations.

Natalia Chechina is a Lecturer in Computing in the Department of Computing and Informatics at Bournemouth University. Her research investigates approaches and techniques to enable scaling and efficient performance on commodity hardware where components are loosely coupled, communication is significant, and any of the components may fail or disconnect at any time. Natalia's research interests include scaling distributed computation, cooperative robotics, concurrent programming, stochastic modelling, and overlay networks. Natalia received her PhD from Heriot-Watt University, UK in 2011. She then worked as a Research Associate and Research Fellow at Heriot-Watt University (2011–2013) and Glasgow University (2013–2017).