Over-The-Air Updates for Robotic Swarms

Vivek Shankar Varadharajan, David St Onge, Christian Guß
and Giovanni Beltrame

Abstract

With the growing number of robotic devices introduced by automation and the Internet-of-Things, comes also a growth in the interest for methods and tools to deploy new code updates to active sensor arrays and to swarms of robots. This paper presents a toolset that can perform an over-the-air code update of the robots in a swarm while the swarm is active, without interrupting the swarm's mission. Each update is generated as a patch from the currently deployed code. A consensus mechanism borrowed from swarm intelligence ensures that, at any given time, all robots in the swarm run the same code version. Simulations were conducted with thousands of units to study the scalability and bandwidth consumption of the update process. Real deployment experiments were then performed on a small swarm of commercial quadcopters.

Introduction

Unmanned teams of robots are growing in popularity for many real-world scenarios such as search and rescue, surgical nanorobots, and space exploration. Despite the recent technological advancements in these applications, numerous problems still need to be addressed to support robust multi-robot deployments. Among these is the need for Over-The-Air (OTA) behavior updates during a mission. A deployment tool that could safely update robot software OTA with minimal downtime and without redeploying the hardware would be a real mission enabler [1]. OTA software deployment tools could provide similar advantages to update sensor arrays, e.g. for traffic monitoring in smart cities, deployment in mines or construction sites to quickly enhance their adaptability to the environment.

Common practices A common method to update the software of a group of robots relies on a connection to a central node transmitting the changes to each robot within range. The commercial platform of the Kilobots [2] uses this strategy. The protocols to transmit, encrypt and test the updates are well-studied [3], but this approach comes with critical disadvantages:1. it requires a pre-deployed network infrastructure like a router mediated communication; 2. its robustness depends on a single central node; 3. it often requires the swarm to regroup in order to process the update in a certain range, before being re-deployed. Such constraints are difficult and sometimes even impossible to achieve in real-world missions such as space exploration or off-grid emergency response.

A possible workaround is to load a set of pre-test binaries before the deployment of the group and switch between them in-mission, as proposed in [4], [5] and [6]. Nevertheless, the operator needs to ensure all robots switch properly their binary and new binaries cannot be tested and uploaded in-mission.

Contributions This work presents a deployment tool to update a robotic swarm OTA, overcoming the above limitations. Our approach has four key features:

Authors' accepted manuscript. Article published in *IEEE Software*, vol. 35 no 2, p. 44-50. https://doi.org/10.1109/MS.2018.111095718

Update consensus An optimized mechanism to achieve consensus within the swarm to avoid code variability and code version conflicts. To achieve consensus in a distributed network, our approach borrows concepts from swarm intelligence [7]. It reaches an agreement on the code version using a gossip-based algorithm [8]. For the interested reader, more details are available in the supplementary material [9].

Packet exchange protocol An OTA protocol to relay updates, avoiding a need for pre-deployed network infrastructure. Our packet exchange protocol uses a gossip-based package routing for relaying the update packets (i.e. the patches) across the swarm network. The robot initiating the update broadcasts the patch to its neighbors. Whenever a robot receives an update patch, it performs a two-stage testing procedure and then re-broadcast the patch to its neighbors. This way, the update patch propagates across the network from its source to all the robots.

Patch generation A patch generation tool to share only binary deltas for minimizing the bandwidth requirements and update time. When a robot shares a new update with its peers, it generates a patch using `bsdiff` (`http://www.daemonology.net/bsdiff/`). Before broadcasting, the patch is encrypted using a stream cipher called `salsa20` [10]. The hash of the previous code versions artifact is used as the initialization vector of the stream cipher.

Standby state A safe standby state during the update process is used to avoid a perturbed state within the swarm. During the update process, the robots will switch to a standby state. In this state, the robots are brought to a safe environmental configuration while they wait for their peers to reach consensus on the update. The standby state itself is a behavioral script, pre-loaded before a flight, and implemented for a given scenario and a given hardware. For instance, the standby script for an Unmanned Aerial Vehicle (UAV) could be a hovering behavior.

An arbitrarily chosen robot within the swarm creates a new release when modifications are made to its script in the local file system. The robot creating the release generates a patch to the binary and propagates the patch to its peers. The patch recipients apply it to its current running binary to update its control software.

The main advantage of our approach is that it scales well for large deployments, and can be applied in communication constrained and unpredictable environments. We provide an open-source implementation [11] of our approach based on the Robot Operating System (ROS) and a robot-specific implementation [12] for the Khepera IV robot. Our implementation is an add-on to the Buzz Virtual Machine [13] (BVM). Buzz is a programming language designed for heterogeneous robotic swarms. The terms *script* and *artifact* used throughout the article correspond to Buzz scripts and Buzz bytecode (compiled scripts), respectively.

Update Strategy

The current release of the control software is normally executed on all robots until the update process is initialized and interrupts the execution. Fig. 1 shows the process flow of the OTA update protocol, providing a continuous deployment [14, p. 266-267] of new releases. All the robots in the swarm are initialized with an artifact version $C_v = 0$. A new release (R_n) of the code can be created from any robot within the swarm by modifying the local script. When local script modifications occur, the new release is created, compiled into a bytecode, and tested in two steps:

1. the initialization test: loading the bytecode and verifying memory usage;
2. the step test: looping once to ensure the virtual machine stack integrity and checking the hardware calls are safe.

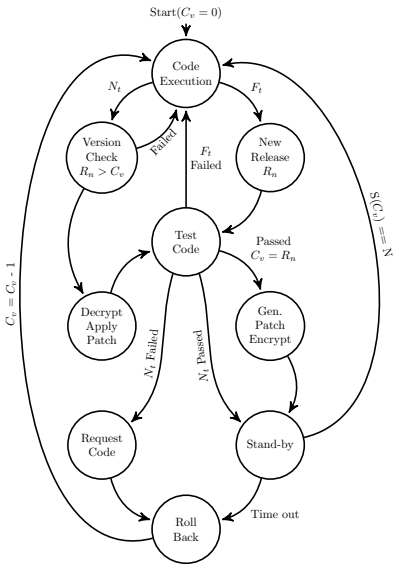


Figure 1: The process flow of the OTA update protocol.

Following successful tests, a new encrypted patch is created by comparing the old artifact with its newer version. The patch is created using the tool `bsdiff` and the encryption is performed using a stream cipher called `salsa20`. Any new release produces an encrypted patch to the bytecode, which is then broadcasted to all the neighbors to apply the new release.

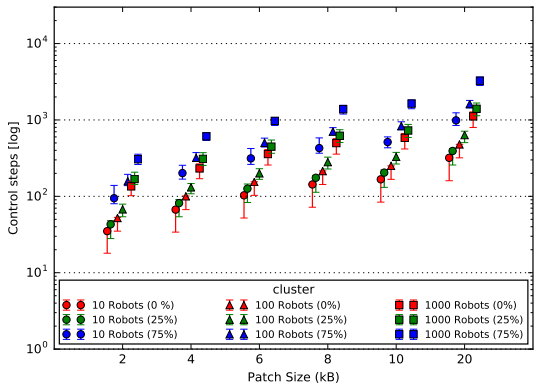
Every robot within the swarm has two triggers to start an update: either from its local file system i.e., patch generation trigger (F_t) or from a neighbor i.e., patch installation trigger (N_t).

Patch generation trigger The update is initiated by modifying the script within the local file system. If the modification passes the tests, the robot will switch to a standby state and wait for their peers to reach consensus on the update.

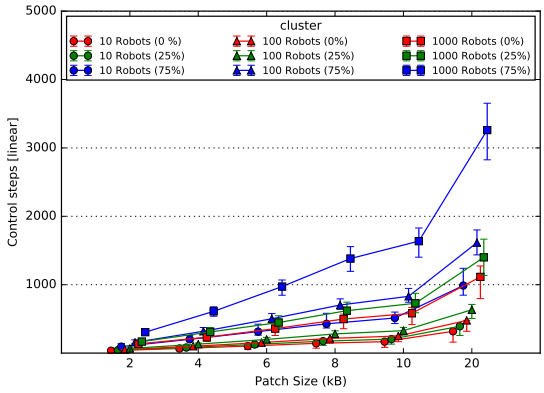
If a test fails, the update is dropped, and the operator is notified of the test results. Any robot of the swarm can start an update, but sufficient resources have to be available to compile, test and generate the patch in its embedded computer. For heterogeneous swarms, the operator is encouraged to start the update from a robot with powerful processing capabilities. However, if none is optimal, the operator can set a standard computer to generate the patch as long as it has the access to the swarm network.

Patch installation trigger The update is triggered when the host receives an update patch OTA from one of its peers in the swarm. The received patch is decrypted, tested, and applied to the current code artifact. Then the host enters the standby state.

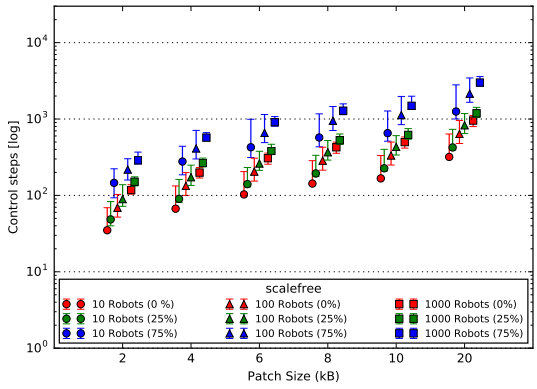
If the patch fails tests a second time, or if a robot times out waiting for a patch rebroadcast, the swarm rollbacks to the previous version. Every robot stores a copy of the previous releases' artifact used for roll backs. With heterogeneous swarms, step tests may fail when hardware calls for missing peripheral are made and hence it is advisable to perform tests on all nodes. We address device failure, communication failure, and disconnections from the network by managing a consensus over the number of robots in the network (removing unresponsive robots and adding newcomers using a heartbeat system [15]).



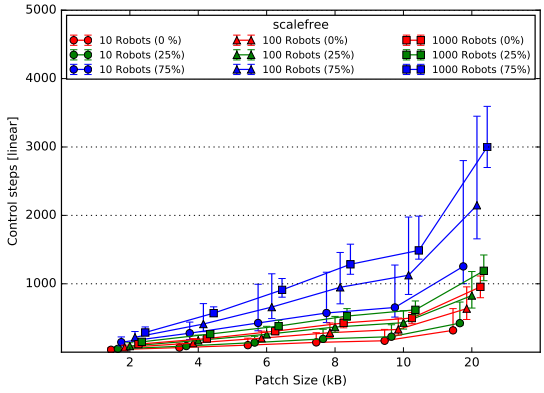
(a) cluster topology (log scale).



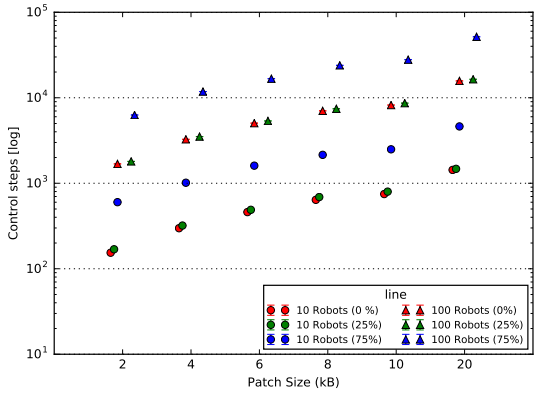
(b) cluster topology (linear scale).



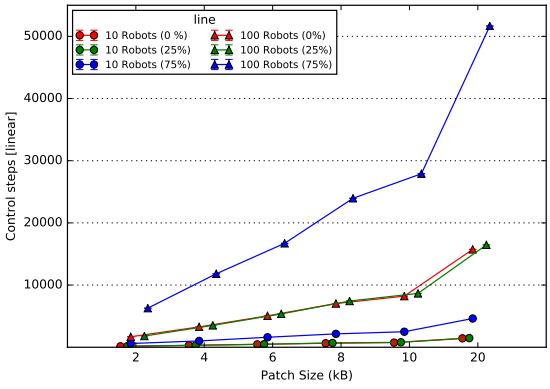
(c) scale free topology (log scale).



(d) scale free topology (linear scale).



(e) line topology (log scale).



(f) line topology (linear scale).

Figure 2: The minimum, average and maximum number of control steps required for 10, 100 and 1000 robots to apply a new patch of different sizes considering 0%, 25% and 75% packet drop. The markers within the plot are slightly offset for visual clarity. The duration of the control step is 100 ms.

Validation

This update system must operate over a wide range of use cases and must be scalable, robust, and stable. A series of simulations were performed to validate:

- its robustness in various topologies,
- its scalability up to a thousand units.

To study the impact of topology, we simulated three update topologies, inspired from [8]: `cluster`, `scale-free`, and `line`.

In the `cluster` topology, the robots are grouped into densely connected clusters. With the `scale-free` topology, the robots are grouped in clusters, but connections among the clusters are sparse. In the `line` topology, the robots are positioned on a line, one behind the other. Each robot is connected to only two of its neighbors, with the exception of the two endpoints that have only one neighbor each.

Simulation setup The simulation parameter set consists of $\langle N, \text{topology}, P, C \rangle$, in which $N = \{10, 100, 1000\}$ is the number of robots, $P = \{0, 0.25, 0.75\}$, the packet drop probability and $C = \{2, 4, 6, 8, 10, 20\}$, the patch size in kB. A simulation was run 30 times for each set of parameters to infer statistically significant results. A total of 4320 simulated experiments were performed using ARGoS multi-robot simulator (`http://www.argos-sim/info`). For the 1000 robots case, the line topology was excluded because of its long simulation time. The performance of the simulations was assessed by the time required to update N robots. The length of the control step was 100 ms.

Simulation results Figure 2 plots the simulation time taken for 10, 100, and 1000 robots to create, distribute, test, deploy, and agree on a new patch of different code sizes OTA, following the three topological distributions.

The experimental results can be summarized as follows:

1. the update time increases sub-linearly with the increase in patch size and the number of robots; and
2. the update time increases exponentially with the packet drop rate.

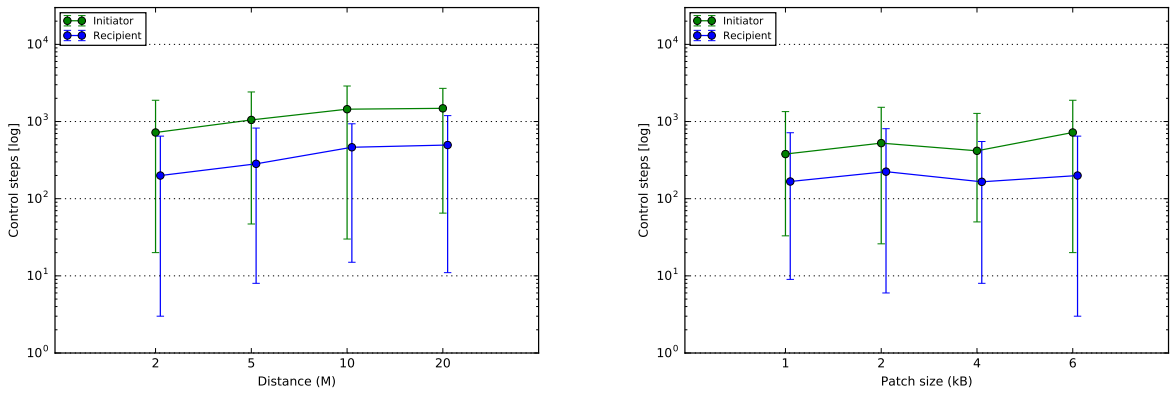
In particular, with the `cluster` and `scale-free` topology, the swarm updated in tens' of seconds for up to a thousand robots without packet drop. Whereas in `line` topology the update time varied from 10 seconds to over several minutes depending on the patch size and the number of robots.

These observations on different topologies led us to conclude that the time required to update a swarm is largely influenced by the communication paths available within the swarm.

In the interest of brevity, we present only a subset of our results; a more detailed experimental analysis, scripts, and a video are available online [9].

Field Experiments

Setup To demonstrate the OTA update protocol, we designed an experimental scenario with four quadcopters: three DJI Matrice 100 and a 3DR solo. The DJI M100's were equipped with an NVidia TK1 companion computer and the 3DR Solos, with a Raspberry Pi 3, both hosting the ROS implementation of the update tool [11]. The communication infrastructure between the robots was based on a mesh created by XBee transceivers. The robots were placed at two meters distance and artifacts of sizes {1,2,4,6} kB were used during the field experiments, with thirty repetitions for each experiment.



(a) Distance variations.

(b) Patch size Variations.

Figure 3: Both figures plot the minimum, average and maximum number of control steps needed for the thirty repetitions of the update experiments on the UAVs. The duration of the control step on the y-axis is 100 ms.

Results Figure 3a and figure 3b show the number of time steps required by the update initiator (patch generator) and recipient (patch installer) UAVs to update to a new release, over different distances and patch sizes, respectively.

A sub-linear relationship was observed in the update time growth with regard to the increase in inter-robot distances and patch sizes. Specifically, when the distances were below 10 meters, the update process took tens' of seconds and increased up to 280 seconds for distances over 10 meters. The four UAVs consumed 10 to 100 seconds on average, to update patch sizes of 1 to 6 kB.

Before we implemented this strategy within our experimental field procedures, a simple tuning of key controller gains on four UAVs required up to 10 minutes: 1. regroup all UAV to a home location, 2. manually connect to each UAV, 3. update each script and 4. resume the mission.This strategy represented for our team a significant improvement in efficiency in the field.

Conclusions

We presented an OTA update protocol for robotic swarms, providing a swift method to integrate new behaviors into a swarm with minimal human intervention and interruption time. This approach is a robust distributed method to integrate new behaviors after deployment by ensuring consensus on the updates among the swarm. Our patch generation approach minimizes the packet sizes and increases efficiency. Large set of simulation experiments with different topologies and code sizes prove its scalability and flexibility. Evaluation on a small swarm of commercial UAVs demonstrated the usability of the approach. The update strategy and the open source tools [11] developed in this work greatly increase the productivity of our team's field experiments and is believed to be a milestone in realizing swarm deployment on real mission scenarios, such as interplanetary explorations and emergency response with unmanned vehicles. Future work will focus on more sophisticated security measures, improved consensus algorithms, and better packet management.

References

[1] S. Brown and C. Sreenan, *Software Updating in Wireless Sensor Networks: A Survey and Lacunae*, 4. 2013, vol. 2, pp. 717–760.

[2] M. Rubenstein, C. Ahler, and R. Nagpal, "Kilobot: A low cost scalable robot system for collective behaviors," in *International Conference on Robotics and Automation (ICRA)*, IEEE, 2012, pp. 3293–3298.

[3] E. Laukkanen, J. Itkonen, and C. Lassenius, "Problems, causes and solutions when adopting continuous delivery???A systematic literature review," *Information and Software Technology*, vol. 82, pp. 55–79, 2017.

[4] D. T. Davis, T. H. Chung, M. R. Clement, and M. A. Day, "Consensus-Based Data Sharing for Large-Scale Aerial Swarm Coordination in Lossy Communications Environments," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 3801–3808.

[5] T. H. Chung, M. R. Clement, M. A. Day, K. D. Jones, D. Davis, and M. Jones, "Live-fly, large-scale field experimentation for large numbers of fixed-wing UAVs," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2016-June, pp. 1255–1262, 2016.

[6] T. H. Chung, K. D. Jones, M. A. Day, M. Jones, and M. Clement, "50 vs. 50 by 2015: Swarm vs. swarm uav live-fly competition at the naval postgraduate school," 2013.

[7] M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo, "Swarm robotics: A review from the swarm engineering perspective," *Swarm Intelligence*, vol. 7, no. 1, pp. 1–41, 2013.

[8] C. Pinciroli, A. Lee-Brown, and G. Beltrame, "A Tuple Space for Data Sharing in Robot Swarms," *EAI International Conference on Bio-inspired Information and Communications Technologies*, pp. 287–294, 2016.

[9] V. S. Varadharajan, D. St-Onge, C. Guß, and G. Beltrame. (2017). Over-the-air updates for robotic swarms - supplementary material, [Online]. Available: `http://www.mistlab.ca/papers/IEEESoftware/2017/`.

[10] C. Paar and J. Pelzl, *Understanding cryptography: A textbook for students and practitioners*. Springer Science & Business Media, 2009.

[11] V. S. Varadharajan, D. St-Onge, and G. Beltrame. (2016). ROSBuzz GitHub page, [Online]. Available: `https://github.com/MISTLab/ROSBuzz/tree/dev`.

[12] D. St-Onge, V. S. Varadharajan, and G. Beltrame. (2016). BuzzKH4 GitHub page, [Online]. Available: `https://github.com/MISTLab/BuzzKH4/tree/update`.

[13] C. Pinciroli and G. Beltrame, "Buzz: An extensible programming language for heterogeneous swarm robotics," in *International Conference on Intelligent Robots and Systems*, IEEE, Oct. 2016, pp. 3794–3800.

[14] J. Humble and D. Farley, *Continuous delivery: Reliable software releases through build, test, and deployment automation*, 1st. Addison-Wesley Professional, 2010.

[15] C. Pinciroli and G. Beltrame, "Swarm-oriented programming of distributed robot networks," *IEEE Computer*, vol. 49, no. 12, pp. 32–41, Dec. 2016.