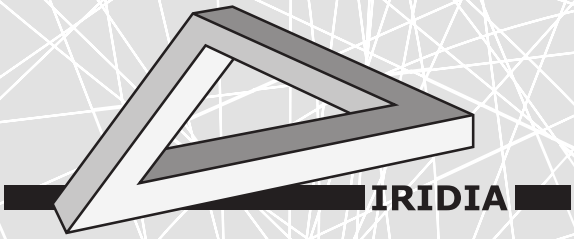

Université Libre de Bruxelles

Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Artificielle

Tycho: a robust, ROS-based tracking system for robot swarms

G. Legarda Herranz, D. Garzón Ramos, J. Kuckling, M. Kegeleirs, and M. Birattari

IRIDIA – Technical Report Series

Technical Report No.
TR/IRIDIA/2022-009

September 2022

IRIDIA – Technical Report Series
ISSN 1781-3794

Published by:
IRIDIA, *Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle*
Université Libre de Bruxelles
Av F. D. Roosevelt 50, CP 194/6
1050 Bruxelles, Belgium

Technical report number TR/IRIDIA/2022-009

Tycho: a robust, ROS-based tracking system for robot swarms

Guillermo Legarda Herranz, David Garzón Ramos,
Jonas Kuckling, Miquel Kegeleirs and Mauro Birattari

IRIDIA, Université libre de Bruxelles, Belgium.

September 2022

1 Introduction

The following document presents Tycho[1], the new tracking system designed for swarm robotics experiments carried out in IRIDIA's Robotics Arena. Compared to its predecessor [3], Tycho provides two main advantages. First, it uses an unscented Kalman filter (UKF) to fuse the detection output of an array of cameras with overlapping fields of view. It is therefore inherently robust to the loss of robots by some of the cameras. Second, the software is developed using ROS, which promotes transparency and reusability of the code.

2 Quick start

This section outlines the steps that must be followed to configure and run the Tycho software. To keep the section concise, we assume that the user has already arranged the required hardware infrastructure as described in Section 3. Thus, the user should have an array of three gigabit ethernet (GigE) cameras connected to a server through a switch. For our implementation of Tycho and throughout this report, we use Prosilica GC1600C cameras, which were already used in the original tracking system for IRIDIA's Robotics Arena [3]. Accordingly, the corresponding Vimba SDK and ROS driver should be installed by following the instructions in the forked `avt_vimba_camera` repository.

The quick start involves installation and calibration of the software, calibration of the hardware, basic usage instructions and visualisation of the output. Further details on all hardware and software components are provided in Sections 3–5.

2.1 Installation

The software has been developed and tested under Ubuntu Linux using the ROS Noetic distribution. To ensure that all dependencies are met, we recommend to install the `desktop_full` ROS metapackage. Additionally, install the `robot_localization` package in order to use state estimation through sensor fusion[2]. To download and build the software, clone the project from GitHub[3] into you catkin workspace using the `--recurse submodules` flag and run

```
$ catkin_make
```

2.2 Configuration

Before attempting to run Tycho for the first time, three components must be configured: the cameras, the `transformer` node and the `cropper` node. For more details on the cameras, refer to Section 3. For more details on the nodes and their functionalities, see Section 4.1.

[1] The name honours the astronomer Tycho Brahe, whose innovative measuring instruments allowed him to track celestial bodies with unprecedented accuracy.

[2] http://wiki.ros.org/robot_localization

[3] https://github.com/demiurge-project/demiurge-tycho.git

Cameras

Each of the `tycho_launchers/launch/mono_camera_<i>.launch` files, where `<i>` $= \{0, 1, 2\}$, need to be individually configured. For each camera, first set the `ip` and `guid` parameters to the corresponding values, which can be found by running Vimba Viewer, included in the Vimba SDK. Then, set each of the following parameters as instructed:

`camera_info_url` The `camera_info_url` parameter points to a YAML file containing all calibration parameters for camera `<i>`, which is stored in the `tycho_launchers/calibrations` directory as `camera_<i>_calibration.yaml`. To generate each YAML file, we use the `camera_calibration` package[4]. This step should only be required the first time the tracking system is used.

We follow the steps explained in the package documentation for calibration of monocular cameras[5]. For completeness of the technical report, we summarise these instructions here, with comments on the aspects to which one must pay special attention.

First, a calibration board with a checkered pattern is required. As per the instructions, we also use an 8×6 checkerboard with 108 mm squares. The board is made of a 3 mm MDF sheet reinforced with pine wood beams to prevent it from bending while manipulating it.

The calibration node is then run as

```
$ rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.108 image:=/camera_<i>/
    image_raw camera:=/camera_<i>
```

Move the checkerboard around the camera frame until the CALIBRATE button in the calibration window is highlighted. Press CALIBRATE and wait until the process is finished. Hit SAVE to generate a TAR.GZ file containing all the images that were used to obtain the calibration parameters, as well as the corresponding YAML file. Finally, press COMMIT to permanently upload the calibration to the camera. In the YAML file, set the `camera_name` parameter to `camera_<i>`.

`exposure` The exposure parameter should be adjusted whenever the lighting conditions in the arena change. To ensure consistent performance of the tracking system, set the ambient light intensity to a fixed value. Using Vimba Viewer, open the camera feed and navigate to the Brightness tab. Adjust the exposure until the contrast between black and white is clear and there are no visible reflections; then, set the parameter in the YAML file to the value obtained. Make sure the gain parameter is set to zero throughout this process.

`whitebalance` Similarly, the white balance should also be adjusted according to the lighting conditions. Using Vimba Viewer, navigate to the white balance configuration menu in the Color tab and hit "Once" to obtain an adequate value.

`Transformer` node

The `transformer` node transforms the pose of each robot detected from the camera frame to the arena frame. For each camera, the required calibration parameters are stored in YAML files in the `tycho_transformer/config` directory. To generate the YAML files for each of the three cameras, place the transformer calibration board (see Section 4.1) in the centre of the arena. Rotate the board until the x and y axes are aligned with those of the arena and run

```
$ roslaunch tycho_launchers transformer_calibration
```

The YAML files will be stored automatically in the `tycho_transformer/config` directory. This step should be repeated whenever the position or the orientation of any of the cameras change.

`Cropper` node

The cropper node allows the user to define a polygonal region of interest within the field of view of each camera, such that every point outside this region is ignored during tracking. The vertices of the polygons are stored in YAML files in the `tycho_cropper/config` directory. This is an optional feature; therefore, this step is not required if the region of interest is the entire field of the view of the cameras.

To generate the YAML files for each of the three cameras, run

```
$ roslaunch tycho_launchers cropper_calibration [ratio:=0.8]
```

[4] http://wiki.ros.org/camera_calibration
[5] https://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration

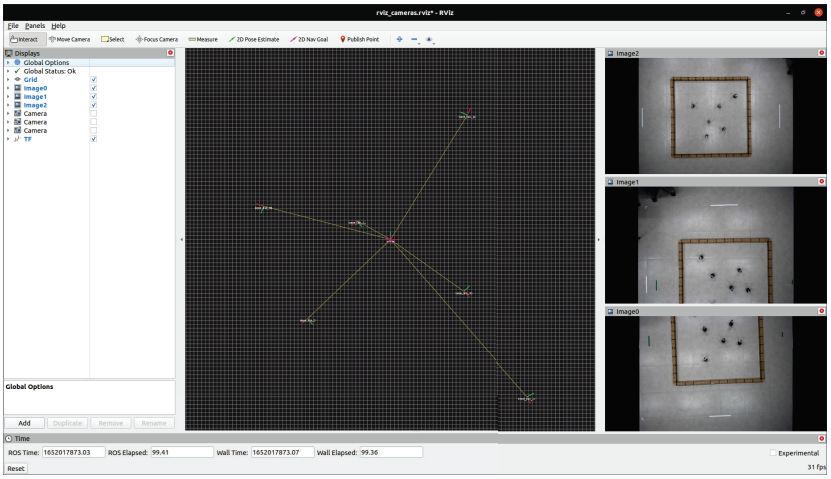

Figure 1: Visualisation of the tracking system with RViz.

Three windows will appear – one for each camera. If the windows appear either too large or too small, use the `ratio` parameter to adjust the window size.

On each window, left-click on the vertices of the region of interest in either clockwise or anticlockwise order. After the last vertex has been defined, right-click to save the YAML file and close the window. Once all three windows are closed, the calibrator will shut down automatically.

2.3 Basic usage

To launch Tycho, simply run

```
$ roslaunch tycho_launchers multi_camera.launch [crop:=false]
```

Setting the `crop` parameter to `true` will activate the cropper node.

2.4 Visualisation

The user can visualise the output of Tycho using RViz. Navigate to the `tycho_launchers/config` directory and run

```
$ rosrun rviz rviz -d rviz_cameras.rviz
```

The three camera feeds and the poses of the robots in the arena will be displayed, as shown in Figure 1.

3 Hardware

Tycho has been designed to be deployed in IRIDIA's Robotics Arena, yet the system can be replicated and adapted in other environments. IRIDIA's Robotics Arena is a rectangular room of 9.9×7.5 m. The current configuration of Tycho includes an arrangement of three cameras that provide input images to the tracking pipeline. The fields of view of the three cameras overlap and offer an effective tracking area of approximately 3×3 m. The cameras feed a video stream to a dedicated desktop computer through an Ethernet connection. Both the cameras and the dedicated computer are directly connected through a dedicated router. The dedicated computer runs all software required to track the robots out of the video stream feed by all cameras. This section summarises the configuration of the experimental arena and Tycho's hardware specifications.

3.1 Cameras

Tycho is currently deployed with an arrangement of three Prosilica GC1600C cameras, manufactured by Allied Vision Technologies[6]. These cameras are a subset of the cameras integrated in IRIDIA's original Arena Tracking System [3]. In our implementation of Tycho, we have configured the tracking pipeline to operate with the

[6] https://www.alliedvision.com/en/camera-selector/detail/prosilica-gc/1600/

Specification	Description
Interface	IEEE 802.3 1000baseT
Resolution	1620 (H) × 1220 (V)
Sensor size	Type 1/1.8
Shutter mode	Global shutter
Max. frame rate at full resolution	15 fps
Image buffer (RAM)	16 MByte
Bit depth	8/12 Bit
Monochrome pixel formats	Mono8, Mono12, Mono12Packed
Raw pixel formats	BayerRG8, BayerRG12, BayerGR12Packed
Power consumption	3.2 W at 12 VDC
Body dimensions (L × W × H in mm)	59 × 46 × 33 (including connectors)

Table 1: Relevant specifications of Prosilica GC1600 C cameras.

specification of the Prosilica GC1600C cameras; however, the system is not restricted to this specific camera hardware. Table 1 lists relevant specifications of the Prosilica GC1600C cameras. More information is available on the Allied Vision Technologies website[6].

3.2 Camera layout

The three Prosilica cameras are arranged as shown in Figure 2. A central camera is kept high enough to capture the entire arena. This serves two purposes: first, it guarantees that at least one camera will provide tracking data at any point in the arena; second, it allows the user to record videos of their runs. Two additional cameras are placed on either side of the central camera. These cameras are lowered with respect to the central camera in order to provide tracking data of a region of the arena with better resolution. To account for the scaling of the area captured by the cameras as they are lowered, the side cameras are rotated by 90 degrees with respect to the central camera, which allows them to fully capture the arena along one of its dimensions. As a result, this configuration of the Prosilica cameras guarantees that every point in the arena is captured by, at least, two cameras, thus attaining the desired redundancy for tracking purposes.

3.3 Network configuration and computer host

The cameras stream a video feed to a dedicated desktop computer through a 1 Gbit Ethernet connection. Communication between cameras and the computer is mediated by a Lynksys 3200 ACM MU-MIMO Gigabit Wi-Fi Router, which is manufactured by Linksys[7]. The cameras and the dedicated computer are directly connected to the router, which has an upstream connection to the internet. We selected the Lynksys 3200 ACM due to two specific features: (i) it has a highly rated performance and processing power; and (ii) it has specific features to maintain multi-device Wi-Fi connectivity at the same speed. The second feature is not particularly relevant for Tycho, but we deemed it relevant to execute experiments with the large number of robots to be operated in a robot swarm. Alternative implementations of Tycho are possible without the need of a camera network setup; for example, by using USB Cameras directly connected to the host computer. Table 2 lists relevant specifications of the Lynksys 3200 ACM router. More information is available on the Linksys website[7].

The current implementation of Tycho is hosted in a dedicated desktop computer located in IRIDIA's Robotics Arena. The specifications of the computer have been selected to ensure enough computational power to process the incoming video feed of the cameras and track the position of the robots in the video. The specifications of the required hardware might vary with respect to the number of cameras connected to Tycho and their resolution, the number of robots to be tracked, and the frequency with which the information is updated in the tracking system. Table 3 lists relevant specifications of the dedicated computer currently installed in the Robot Arena. These specifications have been determined sufficient (tested and validated) to use Tycho alongside three Prosilica GC1600 C cameras and while tracking 50 tags (see Section 3.4).

[7] https://www.linksys.com/gb/wireless-routers/wrt-wireless-routers/linksys-wrt3200acm-ac3200-mu-mimo-gigabit-wifi-router/p/p-wrt3200acm/

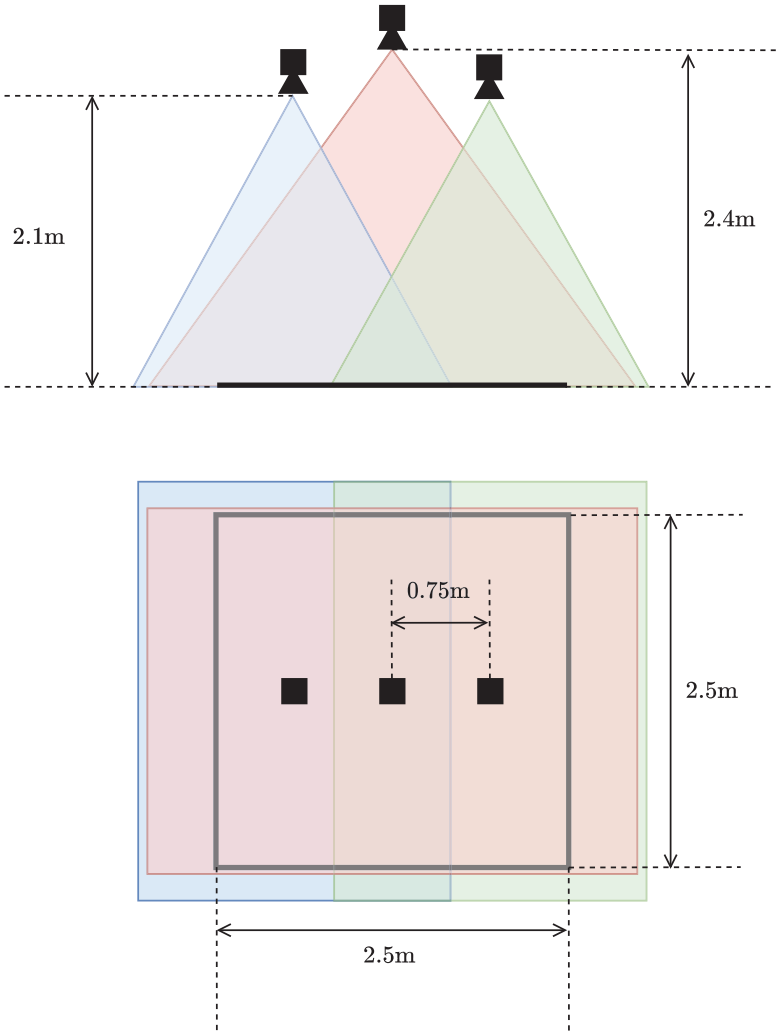

Figure 2: (top) Front view and (bottom) top view of the camera layout. The central camera covers the entire 2.5×2.5 m arena, while the side cameras are lowered and rotated by 90 degrees in order to cover smaller regions of the arena with better resolution.

Specification	Description
Wi-Fi Technology	AC3200 MU-MIMO Dual-band Gigabit, 600+2600 Mbps
Protocols	802.11a, 802.11g, 802.11n, 802.11ac
Wi-Fi Bands	2.4 and 5 GHz (simultaneous dual-band)
Wi-Fi Range	Very Large Household
Number of Ethernet Ports	1x Gigabit WAN port, 4x Gigabit LAN ports
Other Ports	1x USB 3.0 port, 1x Combo eSATA/USB 2.0 port
Processor	1.8 GHz dual-core
Operation Modes	Wireless Router, Access Point, Wired Bridge, Wireless Bridge, Wireless Repeater
Antennas	4x external, dual-band, detachable antennas
Power Supply	Input: 100-240V 50-60Hz; Output: 12V, 3.0A
Dimensions (L × W × H in mm)	245.87 x 193.80 x 51.82

Table 2: Relevant specifications of the Lynksys 3200 ACM router.

Specification	Description
CPU	AMD Ryzen 9 3950x 16-core processor 32 threads
Memory	32 GiB
Graphics card	NVIDIA GeForce GTX 1660 SUPER
OS	Ubuntu 20.04.4 LTS
ROS Version	ROS Noetic (ROS 1)

Table 3: Relevant specifications of the host computer.

3.4 ArUco tags

The tracking system relies on ArUco markers to detect and identify each robot [1]. However, we observe that not all markers in the same dictionary are detected with the same success rate. It is therefore desirable to determine which markers in a dictionary are detected more consistently. One can then decide which markers to use for all experimental runs accordingly.

At IRIDIA, experimental runs often include 20 e-puck robots. Each robot is equipped with a 5×5 cm tag consisting of a 4×4 cm ArUco marker and a 0.5 cm-wide white margin. We consider the 4×4 bit, 50 marker predefined dictionary of OpenCV. A 4×4 bit dictionary is used to maximise the size of each bit in a marker. Having defined the number of bits per marker, the 50 marker dictionary is selected to maximise the minimum Hamming distance between any two markers.

To determine which of the 50 markers are detected more consistently, the `tycho_aruco` package is provided. The package consists of a single python script, `aruco_analysis.py`, and a directory to store ROS bag files. In `aruco_analysis.py`, detection rate statistics are computed for every marker in the dictionary from the bag files available in the `rosbags` directory. To obtain the bag files, print the entire ArUco dictionary and place it on a movable surface at expected height of the tags during an experimental run – for e-puck robots equipped with an omni-vision module, 14 cm. Then, launch the tracking system (see Section 2.3), navigate to the `rosbags` directory and run

```
$ rosbag record --duration=60 -e "/camera_(.*)/tracker/positions_stamped" "/camera_(.*)/epuck_(.*)/
    odom"
```

Immediately start moving the dictionary around the overlapping region of the three cameras for 60 s, trying to match the maximum velocity of the markers during an experimental run. Repeat this process 10 times to ensure that the results of the analysis are statistically relevant.

Finally, run

```
$ rosrun tycho_aruco aruco_analysis.py
```

This will generate a series of files in the `rosbags` directory. First, for each bag, the detection rate of each tag with the three cameras, as well as the its mean detection rate, are shown. An instance of this result is shown in Figure 3. Second, a file containing two figures with the final results is produced, such as the one shown in Figure 4. Of the two figures, the top one shows notched box plots of the detection rates for each tag. The bottom figure shows the same box plots, but sorted in order of descending median. Thus, the tags that are detected more consistently appear on the left-hand-side of the bottom figure, while those that detected less consistently appear on the right-hand-side.

4 Software

The functionality of the Tycho software relies on four ROS packages: `tycho_tracker`, `tycho_transformer`, `tycho_cropper` and `robot_localization`. Additionally, we provide the `tycho_recorder` package, which allows users to record their runs. In this section, we go over the implementation of the first three packages as well as the launch files provided. We also review the integration of the tracking system with ARGoS3.

4.1 ROS pipeline

The ROS information flow diagram is shown in Figure 5. For each Prosilica GC1600C camera, a different namespace is generated. Each namespace is named `camera_<i>`, where `<i>` $= \{0, 1, 2\}$. Within each namespace, the `tracker` node receives the rectified images captured by the camera and calculates the pose of each of the ArUco tags – if any are visible – in the camera frame. The `transformer` node then transforms these poses to

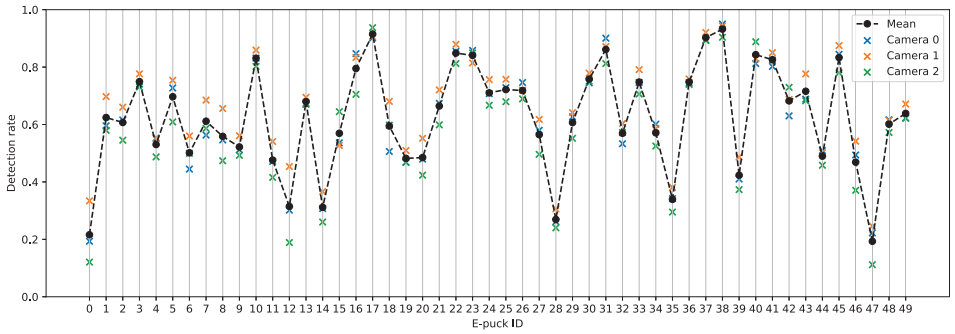

Figure 3: ArUco tag detection rates from a single bag file.

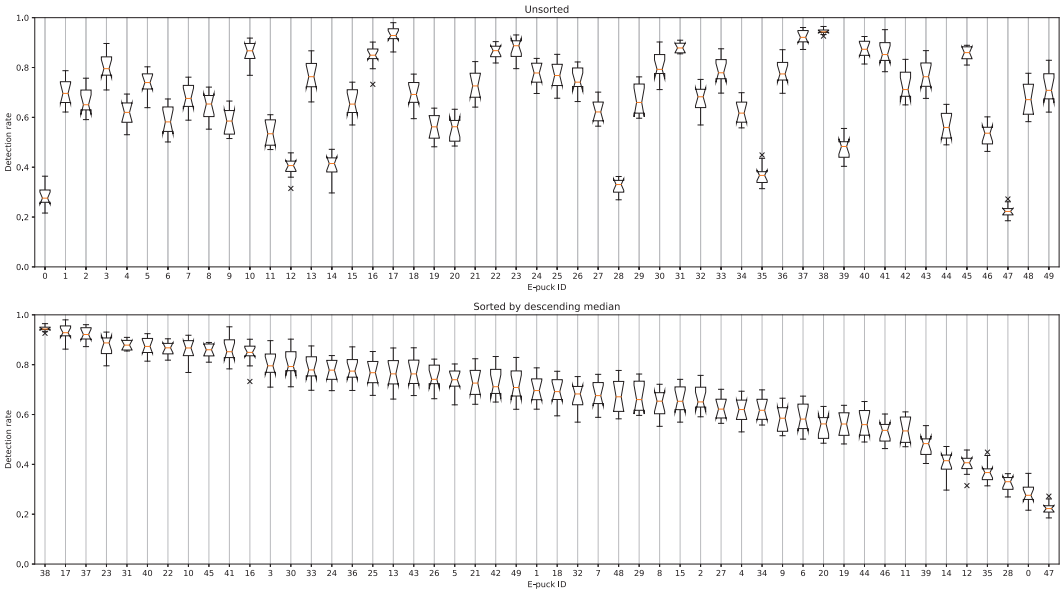

Figure 4: ArUco tag detection rates from 10 bag files.

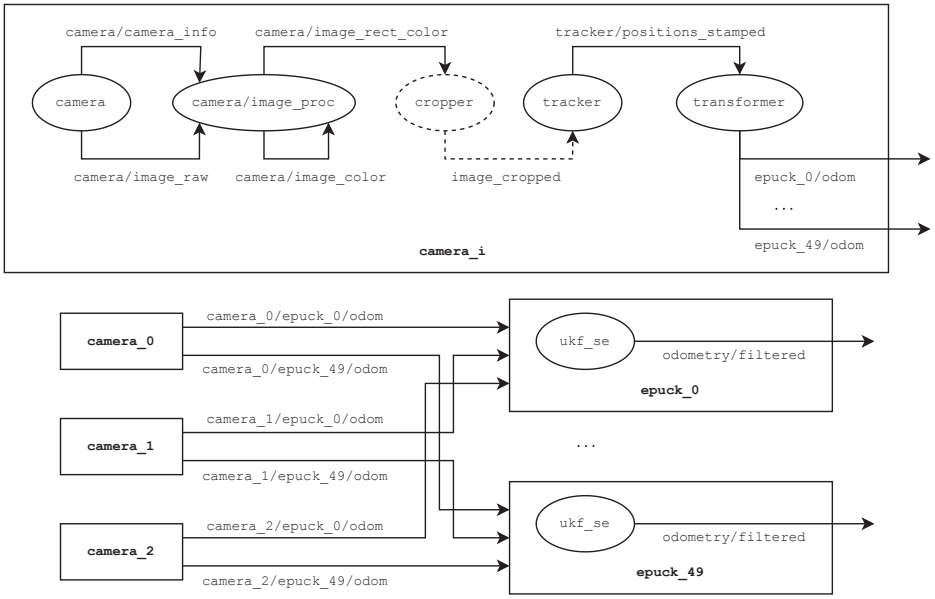

Figure 5: ROS information flow diagram. Namespaces are shown in rectangles and nodes, in ellipses. Arrows represent a publish/subscribe relationship, with the name of the topic indicated next to them. Dashed items are optional and may be activated by means of a parameter. The top diagram shows the content of the namespace of each camera. The bottom diagram shows how the three camera namespaces interact with the UKF nodes.

the arena frame. If the `cropper` node is active, it is introduced between the camera and the `tracker` node. The goal of the `cropper` node is to black out the region of the frame that is outside a user-defined region of interest.

One instance of the `ukf_localization_node` node provided by the `robot_localization` package is launched for every ArUco tag with the name `ukf_se` under the namespace `epuck_<j>`, where `<j>` $= \{0, 1, ..., 49\}$. Each node fuses the ArUco tag poses generated by the three instances of the `transformer` node to produce an estimate of the position of the tags.

The `tycho_tracker` package

The `tycho_tracker` package contains a single source file, `tracker_node.cpp`, which implements the `tracker` node. In `tracker_node.cpp`, a `Tracker` class is defined. When an object of the `Tracker` class is created, a subscriber to the `camera/image_rect_color` topic is created if the `crop` argument is `false`. Otherwise, the `tracker` node subscribes to the `image_cropped` topic published by the `cropper` node. The `image_transport` package is used to subscribe to images. Additionally, a publisher of a topic of type `std_msgs::UInt32MultiArray`, `tracker/positions_stamped`, is created. Both the subscriber and the publisher remain alive until the object is destroyed. In the class constructor, the ArUco dictionary to be used is also defined. In this case, we choose the 4×4 bit, 50 marker dictionary.

Within the subscriber callback, a ROS image is converted to an OpenCV image by means of the `cv_bridge` package. The IDs and the positions of the corners of each of the ArUco markers in the image are extracted from the OpenCV image using the `detectMarkers()` function of the OpenCV aruco module[8]. Then, a message of type `std_msgs::UInt32MultiArray` is constructed as illustrated in Figure 6. The message contains the timestamp of the image and the IDs and corner positions (in pixels) of all the markers detected. The message is then published through the `tracker/positions_stamped` topic.

The `tycho_transformer` package

The `tycho_transformer` package contains three source files: `transformer_node.cpp` implements the `transformer` node, while `transformer_calibration.cpp` and `calibration_supervisor.cpp` implement the node calibration procedure.

[8] https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html

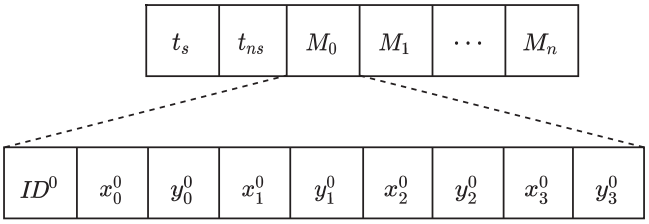

Figure 6: Message published by the `tracker` node over the `tracker/positions_stamped` topic. The first two entries contain the timestamp of the message. The remaining entries contain the ID and the coordinates of the four corners of n detected markers.

Node In `transformer_node.cpp`, a `Transformer` class is defined. When creating an object of the `Transformer` class, a subscriber to the `tracker/positions_stamped` topic published by the `tracker` node is also created. Additionally, 50 publishers – one for each marker in the dictionary – are created to broadcast the topics `epuck_<j>/odom`, where `<j>` $= \{0, ..., 49\}$. Each topic contains messages of type `nav_msgs::Odometry`, which hold pose information about a particular marker.

The subscriber callback iterates over every marker in the message. As shown in Figure 6, each message contains the ID and the x and y coordinates (in the camera frame) of the four corners of each marker. Thus, for each marker, each corner coordinate is first transformed to the arena frame. This requires three distinct operations: rotation, translation and scaling.

Two sequential rotations are applied to obtain the final orientation. First, the point is rotated about the y-axis by π rad. Then, a rotation about the z-axis by `angular_offset` returns the orientation of the arena frame. The point is obtained with respect to the origin of the camera frame through a translation by `x_offset` and `y_offset`. Finally, the point is multiplied by `scale_factor` to convert it from pixels to meters.

The position of each marker is computed as the centre of its four corners. Its orientation is obtained using the two-argument arctangent with the two top corners of the marker. The covariance of each measurement is then set to arbitrary low values in x, y and yaw. In this case, values of 1×10^{-6} were found to result in sufficiently accurate estimation of the marker poses through trial and error.

A `nav_msgs::Odometry` message is composed with the timestamp, pose and pose covariance of a detected marker. At the end of the subscriber callback, all 50 messages – including those corresponding to undetected markers – are published. To prevent the UKF from fusing data from undetected markers, all message timestamps are set to zero after publication. Since, in order to speed up calculations, the list of messages is not reset after every callback, undetected markers are published in the next callback with a tiemstamp of zero.

Calibration The `transformer_calibration.cpp` file implements the `calibrator` node of the package. The node produces a YAML file containing the `x_offset`, `y_offset`, `angular_offset` and `scale_factor` parameters for a given camera. A `TransformerCalibration` object creates a subscriber to the `tracker/positions_stamped` topic published by the `tracker` node. Moreover, a publisher broadcasts in an `alive` topic a boolean message that indicates if the calibration is running (`true`) or finished (`false`).

In the subscriber callback, the node expects to detect nine ArUco markers, arranged in an octagonal pattern with a marker in the centre. The `TransformerCalibrator` attributes `calibration_tags` and `grid_dim` hold the ID of the nine markers and the separation between the central marker and any of the other eight markers, in meters. A calibration board like the one shown in Figure 7 provides precisely this arrangement. The platform sits on a 14 cm-tall column, which is the height of the e-puck robots equipped with an omnidirectional camera. The column itself rests on a notched board for alignment with lines on the floor (e.g., gaps between tiles). The nine markers are selected through the procedure described in Section 3.4. The design files of the calibration board are given in the `assets` directory.

Since the z-axes of the camera frame and the arena frame point in opposite directions, the x-coordinates of all the points are first multiplied by -1, which effectively corresponds to a rotation of the arena frame about its y-axis by π rad. Then, the positions of the markers are calculated as the centre of their four corners. The values of `x_offset` and `y_offset` are simply the position of the central marker. To obtain `angular_offset` and `scale_factor`, imaginary lines are drawn between every pair of diametrically opposite markers. The orientation of each line is compared with their expected orientation which, given the octagonal arrangement, is $n\pi/4$, where $n = \{0, 1, 2, 3\}$. The value of `angular_offset` is then obtained as the mean difference in orientation. The value of `scale_factor` is computed as the ratio of the diameter of the grid, given as $2 \cdot$ `grid_dim`, and the mean line length in pixels. Figure 8 shows a schematic representation of the quantities involved in the calibration

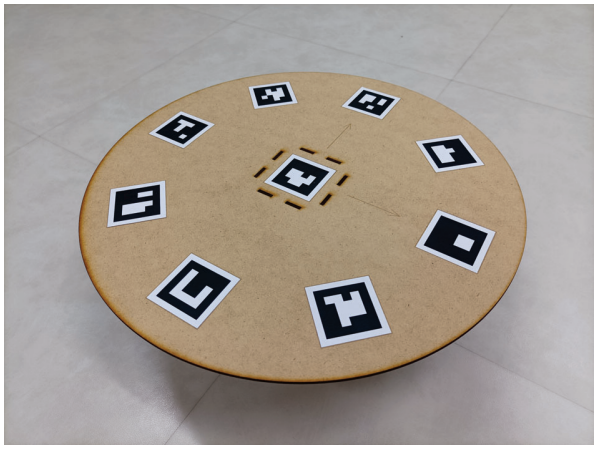

Figure 7: Calibration board for the `transformer` node.

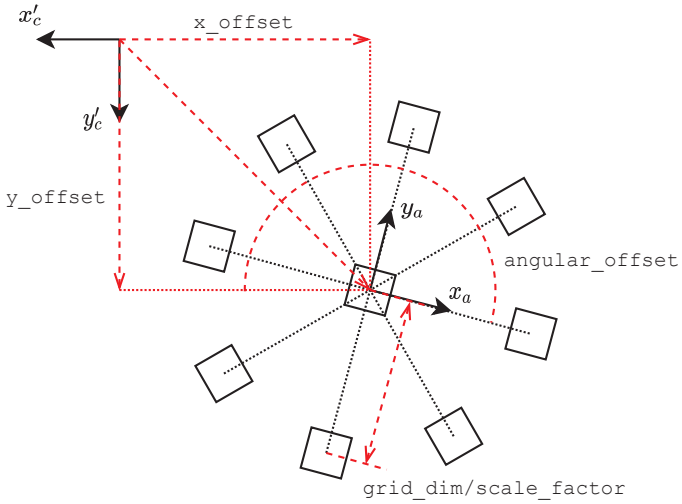

Figure 8: Quantities involved in the `transformer` node calibration procedure. The axes $\{x_a, y_a\}$ represent the origin of the arena frame. The axes $\{x'_c, y'_c\}$ represent the origin of the camera frame, rotated by π rad about the original y-axis of the camera frame (y_c). All lengths are given in pixels.

procedure.

The `calibration_supervisor.cpp` file implements the `supervisor` node of the package. An object of the `TransformerSupervisor` class creates a subscriber to the `alive` topic published by the `calibrator` node, which regularly checks the status of the calibration procedure. The node then creates three instances of `TransformerSupervisor`; one for each camera. When the calibration of all three cameras is complete, the node shuts down.

The `tycho_cropper` package

The `tycho_cropper` package also contains three source files: `cropper_node.cpp` implements the `cropper` node, while `cropper_calibration.cpp` and `calibration_supervisor.cpp` implement the node calibration procedure.

Node The `cropper_node.cpp` file implements the `Cropper` class. An object of the `Cropper` class subscribes to the `camera/image_rect_color` published by a camera, and publishes the `image_cropped` topic.

Within the subscriber callback, the images received as `sensor_msgs::Image` are first converted to OpenCV objects using the `cv_bridge` package. During the first call, the dimensions of the image and the vertices of the

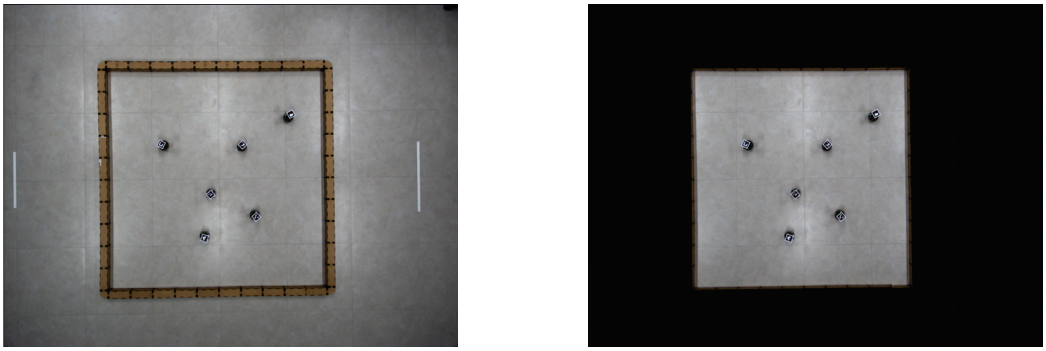

Figure 9: Effect of the `cropper` node. A square mask is applied to the camera feed (left) to black out the region of the frame that is outside the the arena bounds (right).

region of interest defined in an external YAML file are used to define a mask. In all subsequent calls to the callback, the mask is applied to the OpenCV object to black out all pixels outside the region of interest, as shown in Figure 9. The object is then converted back to a `sensor_msgs::Image` object and published.

Calibration The `cropper_calibration.cpp` file implements the `calibrator` node of the `cropper` package. An object of the `CropperCalibrator` class will produce a YAML file containing the vertices of the polygon that defines the region of interest of a given camera. To do so, the object subscribes to the `camera/image_rect_color` published by a camera. It also publishes and `alive` topic, where messages containing a boolean entry indicate if the cropping procedure is ongoing (`true`) or finished (`false`).

Within the subscriber callback, a window is created to display the camera feed to the user[9]. We make use of OpenCV tools to register left and right mouse clicks from the user, which they use to define the region of interest. Every left click defines a new point, while a right click closes the polygon and triggers the termination of the node, provided that a minimum of three points have been defined.

The implementation of the `supervisor` node in the `calibrator_supervisor.cpp` file is identical to that of the `supervisor` node of the `tycho_transformer` package. Three objects of the `CropperSupervisor` class subscribe to the `alive` topic published by the `calibrator` nodes of each of the three cameras. When the calibration procedure of all three cameras terminates, the node shuts down.

The `tycho_recorder` package

The `tycho_recorder` package only contains a single source file: `recorder_node.py` which implements the `recorder` node.

Node The `recorder_node.cpp` file implements the `RosbagRecorder` class. An object of the `RosbagRecorder` class subscribes to the `/argos3/status` topic, published by the corresponding ARGoS3 plugin (see Section 5).

When the node receives the status `Started`, it will start recording two rosbag files labelled `tycho-experiment-YYYYmmdd-HHMMSS-summary.bag` and `tycho-experiment-YYYYmmdd-HHMMSS-video.bag`, where `YYYYmmdd` is the current date and `HHMMSS` is the time of the start of the recording. The `summary` rosbag file will contain all published messages of the topics `odometry/filtered` published by each e-puck, `camera/image_rect_color/compressed` and `camera/camera/camera_info` published by all cameras. The `video` rosbag file will contain all published messages of the topic `/camera_2/camera/image_rect_color` which corresponds to the full resolution images captured by the central camera. Since the video rosbag file contains full resolution images, it can be used to generate a video of the experiment or to train and calibrate other software on real experimental footage. However, the full resolution images require large a mounts of storage capacity (approximately 6 GB per minute of footage). Therefore, a summary of the data (extracted positions as well as compressed images of the cameras) is recorded in a separate bag that can be stored and shared more easily. The summary rosbag file consumes approximately 0.4 GB of storage per minute of footage.

[9] The implementation is based on an answer to a question posted on Stack Overflow: https://stackoverflow.com/questions/42190687/opencv-how-can-i-select-a-region-of-image-irregularly-with-mouse-event-c-c

4.2 Launch files

All launch files are provided in the `launch` directory of the `tycho_launchers` package.

The `multi_camera.launch` file launches the entire tracking system software (see Section 2.3). All the nodes required to process the input of each camera are launched within a `camera_<i>` namespace, where `<i>` $= \{0, 1, 2\}$. These nodes include, as detailed in Section 4.1, the `tracker`, the `transformer` and, optionally, the `cropper` nodes. Additionally, all camera parameters are included through the `mono_camera_<i>.launch` file. Each of these files follows the template provided by the release 1.1.0 of the `avt_vimba_camera` repository[10]. The modifications required for them to fit within our tracking system framework are detailed in Section 2.2.

The fusion of the three cameras is achieved through the `ukf_localization_node` node of the `robot_localization` package. A namespace `epuck_<j>`, where `<j>` $= \{0, ..., 49\}$, is created for each node. The parameters of the node are given in the `ukf_tycho.yaml` file in the `config` directory of the package. In the file, the output frequency of the filter is set to 30 Hz, which is twice the frame rate of the Prosilica GC1600C cameras. Therefore, the filter alternates between estimates and pure predictions. Since the e-puck robots can only move on the surface of the arena, the `two_d_mode` is set to `true`. The `odom_differential` parameters are set to `false` for camera 2 (the central camera) and `true` for cameras 0 and 1. Thus, we consider the pose estimates from camera 2, which can see the entire arena within its field of view, and use cameras 0 and 1 for velocity estimation. This prevents the filter from generating rapid jumps in the pose of the robots.

The `cropper_calibration.launch` and `transformer_calibration.launch` files execute the calibration procedure for the `cropper` and `transformer` nodes, respectively. In both files, the corresponding `supervisor` node is run as a required node, and the `calibrator` nodes are run within the individual camera namespaces. As soon as all `calibrator` nodes terminate, so does the `supervisor`. As a consequence, all remaining nodes, that is, the cameras, will be shut down, and the calibration procedure will be complete.

5 Integration with ARGoS3

The ROS pipeline can be connected to ARGoS [2] through the use of the argos3-tycho plugin [11]. argos3-tycho is a modified version of the ITS (IRIDIA Tracking System) plugin [3]. However, it connects to the ROS pipeline instead of a server.

5.1 Build process

1. Download the source code from the repository:

```
$ git clone git@github.com:demiurge-project/argos3-tycho.git
```

2. Move into the repository:

```
$ cd argos3-tycho
```

3. Create a catkin workspace:

```
$ catkin_make
```

4. Build and install the tracking sytem:

```
$ catkin_make -DCMAKE_INSTALL_PREFIX=$ARGOS_INSTALL_PATH/argos3-dist \
  -DCMAKE_BUILD_TYPE=Release install
```

5.2 Using the plugin

In order to receive the positions of the robots in ARGoS, the Tycho physics engine needs to be included in the .argos files:

```
<iridia_tracking_system id="its" translate_x="0" translate_y="0" rotate_phi="0"
 topic="odometry/filtered"/>
```

[10] https://github.com/astuff/avt_vimba_camera.git
[11] https://github.com/demiurge-project/argos3-tycho

The physics engine offers four parameters to customise. `rotate_phi` is the angle (in radians) that the arena should be rotated around the center. `translate_x` and `translate_y` are the translations applied to the world space (after rotation). They are measured in meters. `topic` is the name of the ROS topic where the position of the robots will be published. The actual topics subscribed are `/epuck<XY>/<topic>` where `<XY>` stands for the tag of the robot and `<topic>` is the value supplied to this parameter.

6 Future work

We identify two potential improvements for the current version of Tycho. First, we would like to increase the number of cameras of the system. This would allow for even lower placement of the off-centre cameras, which would further increase the resolution of their respective feeds, while also increasing redundancy of the tracking system. Second, we could further increase the integration of Tycho with ROS by using one of the many ROS packages for ArUco detection and also by using the `tf2` library to keep track of all the transformations computed by the `transformer` node.

Acknowledgements

The project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (DEMIURGE Project, grant agreement No 681872) and from Belgium's Wallonia-Brussels Federation through the ARC Advanced project GbO-Guaranteed by Optimization. MB, JK, and MK acknowledge support from the Belgian Fonds de la Recherche Scientifique – FNRS. DGR acknowledges support from the Colombian Ministry of Science, Technology and Innovation – Minciencias.

References

[1] Sergio Garrido Jurado, Rafael Muñoz Salinas, Francisco José Madrid Cuevas, and Manuel Jesús Marín Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280–2292, 2014.

[2] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni A. Di Caro, Frederick Ducatelle, Mauro Birattari, Luca Maria Gambardella, and Marco Dorigo. ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intelligence*, 6(4):271–295, 2012.

[3] Alessandro Stranieri, Ali Emre Turgut, Mattia Salvaro, Lorenzo Garattoni, Gianpiero Francesca, Andreagiovanni Reina, Marco Dorigo, and Mauro Birattari. IRIDIA's arena tracking system. Technical Report TR/IRIDIA/2013-013, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, 2013.