X-Copter Studio

Michal Koutný
Faculty of Mathematics and Physics
Charles University
Prague, Czech republic

Ondřej Pilát
Faculty of Mathematics and Physics
Charles University
Prague, Czech republic

Patrik Černý
Faculty of Mathematics and Physics
Charles University
Prague, Czech republic

Maroš Kasinec
Faculty of Mathematics and Physics
Charles University
Prague, Czech republic

***Abstract*—We present a project that aggregates various existing robotic software and serves as a platform to conveniently control a quadrocopter, mainly for research or educational purposes. User interface runs in a browser and other components are also made with portability in mind. We provide a common interface that unifies different quadrocopter models and we implemented it for Parrot AR.Drone 2.0. The platform is data oriented, i.e. it is based on dataflow between user objects. We implemented several such objects for: data recording and replaying, inertial and visual localization and following a given path.**

I. Motivation

Despite the fact that cheap hardware (such as Parrot AR.Drone 2.0 [1] or ready kits [2]) is available, there are not many possibilities for application programmers to develop software for these robots without need to distinguish between individual models.

Our aim is to provide a platform for development of software for quadrocopters. The target users are AI programmers or students and expected tasks are general algorithms (basic example in figure 1) for quadrocopters. The result should work with any robot compatible with our software (section IV-C). The testing should be further simplified by running the application without physical access to a quadrocopter either by using a simulator or data previously captured during live flights.

```
at(xcheckpoint.reachedCheckpoint) {
    xcheckpoint.checkpoint =
        nextCheckpoint();
},;

// x = 3 m, y = 1 m, z = 1.5 m
var cp0 = Checkpoint.new(3, 1, 1.5);
xcheckpoint.checkpoint = cp0;
```

Fig. 1. Sample script that flies through computed checkpoints. It assumes there is created a dataflow graph with node `xcheckpoint` in it.

II. Related work

A. Middleware for robotics

Probably most popular middleware for robotics is the Robot Operating System (ROS) [3]. It implements communication between objects using publish–subscribe mechanism. It is open source, mainly targeted on Linux platforms.

Similar project is Urbi SDK, developed by former private company Gostai [4]. The current maintainer is Aldebaran Robotics [5], unfortunately the community around Urbi SDK is much smaller and much less active in comparison with ROS. Reasons why we chose Urbi SDK despite this fact are in the section III-B.

B. Parrot AR.Drone 2 API

Part of our project is an API for Parrot AR.Drone 2. Various other projects are dealing with this. There is the official SDK [6] with C++ API, ControlTower [7] that provide Java interface, a ROS package ardrone_autonomy [8], UObject for Urbi SDK [9] or implementation of Czech Technical University [10].

Because none of the aforementioned fit to our requirements for OS portability, stability, functionality or documentation, we implemented our own (see the section IV-C).

C. Ground control system

There is official application for Parrot AR.Drone [11] intended for mobile phone users allowing manual control and displaying only limited data from sensors. The PC application ControlTower [7] allows controlling quadrocopter with specialized computer peripheries and has airplane-like GUI. More complex application is QGroundControl [12] that cooperates with Pixhawk project [13] that encompasses own hardware and uses visual localization.

III. Used technology

A. Architecture

Our system is divided into three components. First interacts directly with a user, second controls the robot and the last one connects the former two.

The components are separate processes that communicate with each other over network, with intention to run components on different machines.

1) Client: Client software is used to create and launch user scripts, edit them, manually control the robot and visualize data (e.g. directly from quadrocopter's sensors).

Despite the technology challenges the client is thin – running in a web browser.[1]

[1] Google Chrome is strongly recommended, though Mozilla Firefox will also get by (without visualization of video data).

2) Server: The server component conveys communication between the client and the actual control machine (further *onboard*). Its task is to control the access to the onboard and monitor quality of the connection between the client and the onboard. In the case the overall connection latency (Client–Server and Server–Onboard) exceeds preset limits, a warning message is shown to the user. If the connection is lost, onboard execution is correctly terminated and user is notified too.

3) Onboard: The onboard is the main executive component. The robot control and data processing run here because it is closest to the robot. The onboard is executing commands obtained from the client and sends back various data selected by user.

The onboard component is supposed to run under normal operating system.[2] Our implementation exploits a PC that communicates with quadrocopter via Wi-Fi. We did not test the onboard component directly on a robot.[3]

B. Urbi SDK

Urbi SDK is a C++ middleware for robotics, which we based the onboard component on. Basically it provides support for communication between user objects (`UObjects`) and schedules user jobs.

Communication is possible via so called `UVars`, which are slots of `UObjects`. Sender just writes to these slots and a receiver's callback handles the change of `UVar`'s value. This allows both apparently asynchronous communication and really asynchronous when a thread pool is used to run the callbacks. Further, `UObjects` can run in different processes and Urbi SDK ensures transparent messaging via TCP or UDP sockets.

Orchestration user scripts (written in Urbiscript) can be executed by the Urbi runtime. Urbiscript is a prototype-based object-oriented language conceptually similar to JavaScript. It is possible to implement `UObject` functionality exclusively in the Urbiscript.

We chose Urbi SDK because of its portability (Linux and Windows systems are supported) and the own scripting language and runtime. Considered alternative was Robot Operating System.

C. NodeJS

NodeJS is a server-side JavaScript engine. Recently, it became quite popular among developers of interactive web applications and various modules [14] exists that extends core functionality. It suited our needs for the server component.

D. HTML5

Thanks to the standardization efforts many features that were earlier common only for desktop applications or via third party plug-ins (Flash, Java applets, native plug-ins) are now implemented directly in the browser, generally referred to as HTML5.

[2] We support Windows 7/8 and Ubuntu 14.04 systems.

[3] The hardware of Parrot AR Drone 2.0 theoretically should be able to run our onboard with limited performance.

To make client as multi-platform as possible, we decided to implement the client for the browser using aforementioned HTML5 technologies. Most importantly, we use the web socket API [15] for sending data back to the client and `<video>` tag to display streamed video.

E. C++

The executive parts of the onboard component are written in C++ (which is consequence of using Urbi SDK). We utilize features of the C++11 standard, mainly for threading and memory management.

IV. Features

A. Dataflow graph

We chose dataflow driven approach to describe the user's program. It consists of units of operation which we call *nodes*. Each node has at least one input or at least one output. Any operations are either results of changes on node's inputs or are launched by an external event hidden in a node (e.g. timer expiration, a socket received data).

Urbi SDK itself encourages dataflow control by its communication paradigm. We extended the original mechanism to ensure syntactic and semantic compatibility between nodes.

Dataflow bears following advantages for the end user:

1) Implicit parallelism: Nodes are implicit synchronized on outputs-inputs connections, thus avoiding excessive user effort. This also ensures certain level of scalability on multiple CPUs machines.

2) Separating abstraction levels: User can concentrate on high level objectives only, implementation details are hidden in particular nodes. On the other hand, a creator of a node is limited by particular node's interface.

3) Visual development: Each node defines an interface, which makes it possible to check what connections are possible (not only syntactically but also semantically with flat system of semantic types). In the end, user can assemble various dataflow graphs interactively (figure 2).

4) In-time data inspection: Selected outputs of the nodes can be connected with a special node that resends the data to the GUI where the data are visualized on the fly (figure 2).

B. Scripting

Only the dataflow graph of nodes is not enough for controlling the quadrocopter. Thus, a user can write scripts that are sent and executed on onboard. In such scripts it's possible to handle events occurring within the dataflow graph or manually affect the dataflow. The script execution can be paused and resumed (that is a feature of Urbiscript, for instance useful for intentionally infinite loops).

C. Quadrocopters unified API

The X-Copter Studio supplies a unified application programming interface (denoted as XCI) for data acquisition and control of quadrocopters. It is ready for common sensors on quadrocopters and it can be easily extended.

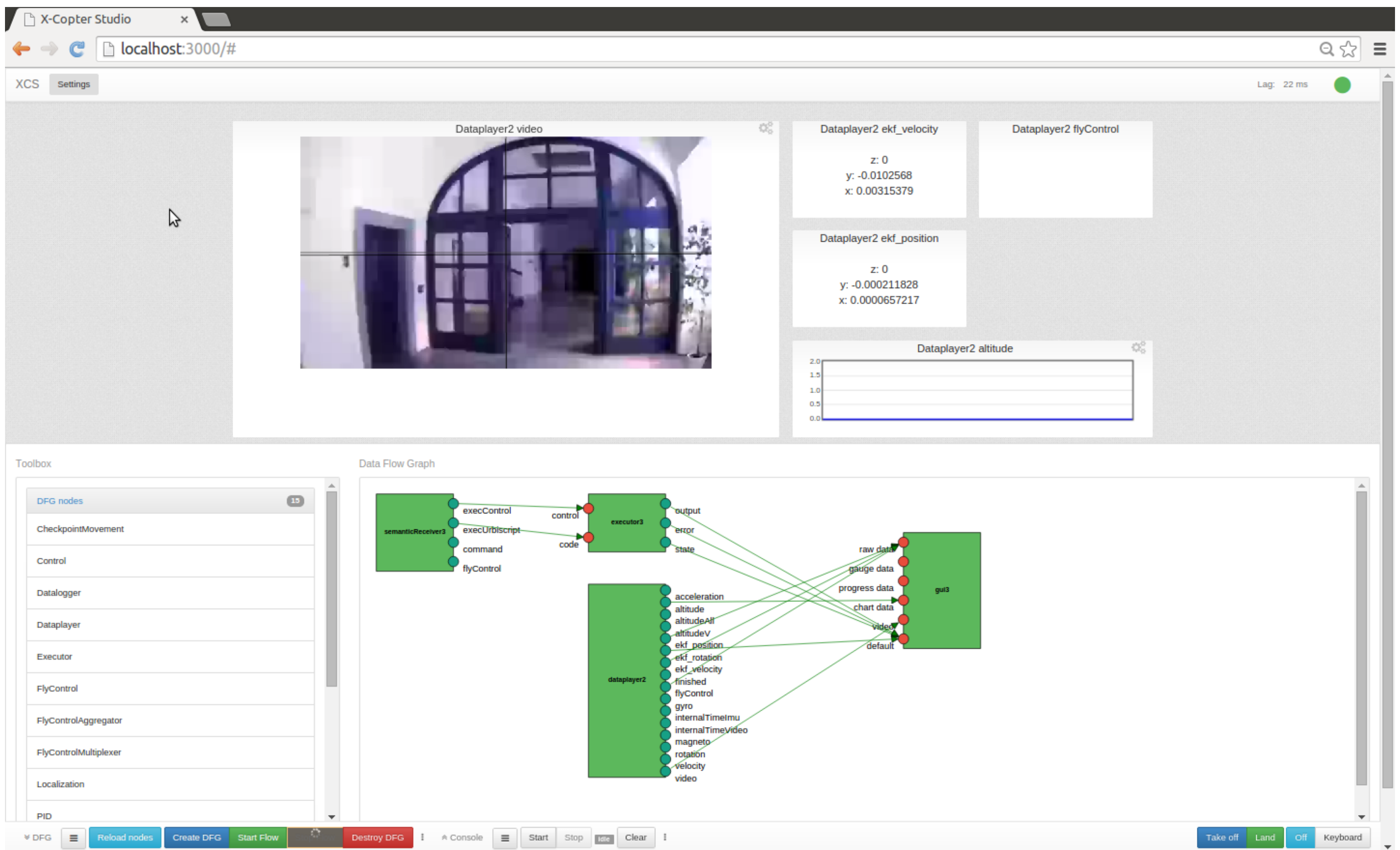

Fig. 2. X-Copter Studio GUI. Top navbar indicated connection quality, upper half is filled with widgets that display data from the onboard, lower half accommodates the editor with repository of nodes on the left, and connected nodes on the right. At the bottom there are (left to right): dataflow control buttons, scripting console control buttons and manual flight control buttons. The scripting console is hidden.

Fig. 3. Urbiscript console. On the top – script editor, middle – output of executed script, bottom left – script execution control (surrounded with other controls of X-Copter Studio).

We implemented the XCI for one of the most popular quadrocopters – Parrot AR.Drone 2.0 and also a simulated quadrocopter in the V-REP simulator [16].

Our Parrot implementation exploits asynchronous socket events handling, which allows us to detect connection failures and attempt to restore the connection with the quadrocopter. The implementation further supports full configuration of the Parrot AR.Drone 2.0 and reading data from all available sensors.

1) Hardware platform: We use Parrot for its price, robustness and popularity among research groups. The cost for this is extensibility – neither the hardware nor the software on Parrot can be expanded. We have to deal with unreliable communication over Wi-Fi and connection failures.

Physical dimensions of the AR.Drone 2 with the hull are 53 cm $\times$ 52 cm and it weights 420 g including battery. It can fly for about 15 minutes with the more powerful battery version (1500 mAh).

a) Sensors: The AR.Drone 2 provides a video stream in high definition (720p) with 30 FPS from front camera aimed forward or QVGA stream with 60 FPS from bottom camera aimed to the ground, three-axes gyroscope, three-axes accelerometer, three-axes magnetometer, pressure sensor and ultrasound altimeter. The quadrocopter sends raw and adjusted measurement from gyroscope, accelerometer, ultrasound sensor and pressure sensor with frequency of 200 Hz. All data from these sensors are accessible for user in the dataflow.

b) Control: Quadrocopter firmware uses aforementioned sensors to maintain tilt, rotation and vertical velocity of the quadrocopter according to an external fly control command. This control command $u = (\Phi, \Theta, \Psi, \dot{z}) \in [-1, 1]^4$ consists

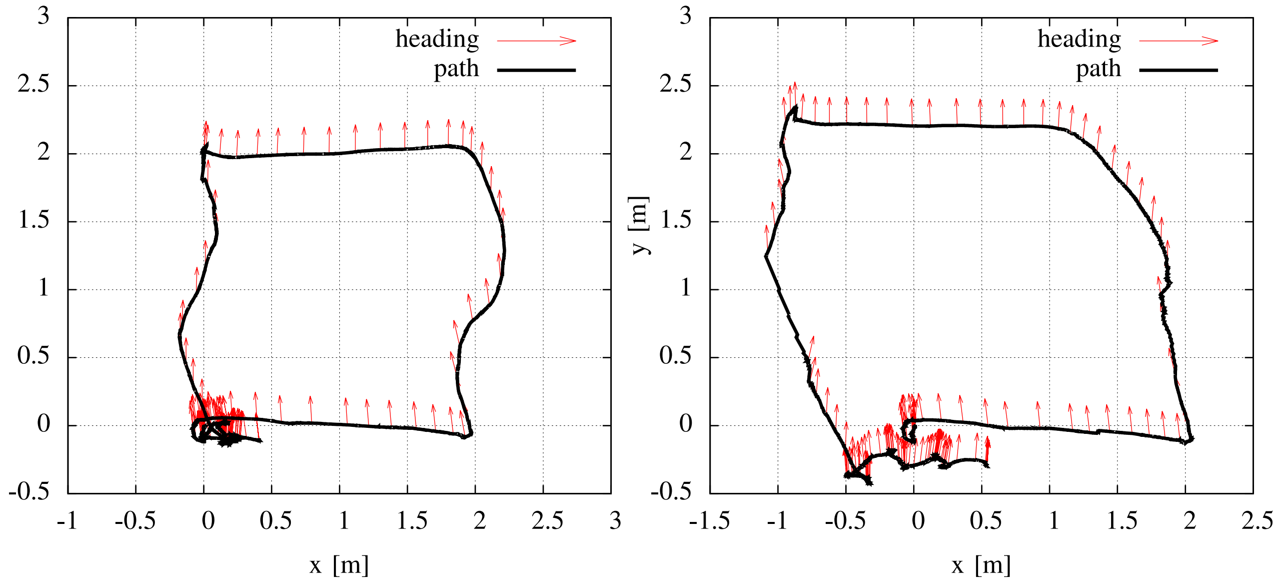

Fig. 4. Example of a path tracked by the EKF. A square 2 m × 2 m was set up and quadrocopter was navigated (with PID controller) to its vertices using inertial localization only. Supposed position is shown on the left plot. As you can see, plot on the right shows more realistic pose and position thanks to visual localization (applied to the original data). Alas, we don't have any reference data for absolute comparison.

of quadrocopter's roll Φ, pitch Θ, yaw rotation speed Ψ and vertical velocity $\dot{z}$. It should be sent every 30 ms for smooth movements.

D. Data recording and replaying

Thanks to the structured dataflow architecture, any potentially interesting data can be captured and stored in a file. We use text format which is portable and allows convenient processing with data processing tools. Data that are inherently binary aren't supported, however, special case – image data are stored with compression to a separate video file.

Recorded data are stored together with timestamps, so it's possible to "replay" them and test or debug system's response with real timing.

E. Localization in unknown environment

We use quadrocopter's onboard sensors to give the end user information about the absolute position of the aircraft. The localization is a node within the dataflow graph, thus it runs exclusively on the onboard component.

1) Inertial localization: Our implementation expects accelerometers with aggregated outputs in a form of horizontal velocity, altimeter and drone pose sensors (i.e. accelerometers for roll and pitch angles and gyroscope and/or magnetometer for yaw angle). Sensor data are filtered with extended Kalman filter (EKF) where we engage physical model based on a similar project [17].

2) Visual localization: To enhance absolute position estimate, we are processing the video stream from the onboard camera. We use monocular SLAM framework PTAM [18] that estimates camera pose and position for each frame and also builds a map of observed feature points. First, the map is initialized from a pair of images where point correspondence is based on optical flow and the initial scale is estimated thanks to inertial localization. After the initialization, the scale estimate is continuously refined using method described in [17].

Data both from inertial and visual localization are merged together in the extended Kalman filter (see figure 4) in separate update steps. The result from PTAM (absolute position and quadrocopter's pose) is used to update the EKF, taking into account a delay of visual localization and then recalculating prediction of the current state.

F. Following given path

Path is represented as a list of checkpoints that drone strictly follows. A checkpoint is a composite of world coordinates and a tangential vector.[4]

Checkpoint attainment: Quadrocopter achieves a checkpoint when it is in a sphere around the checkpoint with 10 cm diameter (according to the localization) and immediately continues to the next checkpoint in the queue. Flight to the checkpoint is controlled by 200 Hz PID regulator.[5] Regulator uses data from the extended Kalman filter in order to eliminate data delay and predict quadrocopter's current position in the world. Output from the regulator is sent directly to the quadrocopter.

G. Runtime configuration

Any configuration (parameters, calibration constants, settings, etc.) is stored in human readable and editable text files

[4]This vector specifies direction of quadrocopter's heading at the given checkpoint. This is intended to be used when checkpoints should be followed as a spline curve.

[5]The frequency is induced by the dataflow, in this case it is refresh frequency of data from quadrocopter.

and loaded during runtime, thus avoiding necessity of recompilation to apply changes.

The same configuration can be accessed both from C++ and Urbiscript API.

V. Conclusion

We hope the X-Copter Studio will became used (at least) for educational purposes since it provides sufficiently high-level interface, for instance to prototype planning algorithms.

If it is proved as useful and practical, it is possible to extend the software for other types of robots, not only quadrocopters.

References

[1] "AR.Drone 2.0. Parrot new wi-fi quadricopter," 2014. [Online]: http://ardrone2.parrot.com/

[2] "The Crazyflie Nano Quadcopter," 2014. [Online]: http://www.bitcraze.se/crazyflie/

[3] "ROS.org," 2014. [Online]: http://www.ros.org/

[4] "Urbi," 2014. [Online]: http://www.gostai.com/products/jazz/urbi/index.html

[5] "aldebaran/urbi," 2014. [Online]: https://github.com/aldebaran/urbi

[6] "ARDRONE open API platform," 2014. [Online]: https://projects.ardrone.org/

[7] "javadrone – AR.Drone Java API – Google Project Hosting," 2014. [Online]: https://code.google.com/p/javadrone/

[8] "ardrone_autonomy – ROS Wiki," 2014. [Online]: http://wiki.ros.org/ardrone_autonomy

[9] "UrbiForge projects/Urbi 4 AR Drone," 2014. [Online]: http://www.urbiforge.org/index.php/Projects/Urbi4ARDrone

[10] Tomáš Krajník and Vojtěch Vonásek and Daniel Fišer and Jan Faigl, "AR-Drone as a Platform for Robotic Research and Education," in *Research and Education in Robotics: EUROBOT 2011*, 2011.

[11] "AR.Freeflight," 2014. [Online]: http://ardrone2.parrot.com/#freeflight

[12] "QGroundControl GCS," 2014. [Online]: http://www.qgroundcontrol.org/

[13] Meier, Lorenz and Tanskanen, Petri and Heng, Lionel and Lee, Gim Hee and Fraundorfer, Friedrich and Pollefeys, Marc, "PIXHAWK: A micro aerial vehicle design for autonomous flight using onboard computer vision," in *Autonomous Robots (AURO)*, 2012.

[14] "Node Packaged Modules," 2014. [Online]: https://www.npmjs.org/

[15] "RFC 6455 – The WebSocket Protocol," 2014. [Online]: http://tools.ietf.org/html/rfc6455/

[16] "Coppelia Robotics v-rep: Create. Compose. Simulate. Any Robot," 2014. [Online]: http://www.coppeliarobotics.com/

[17] J. Engel, J. Sturm and D. Cremers, "Camera-Based Navigation of a Low-Cost Quadrocopter," in *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, 2012.

[18] G. Klein and D. Murray, "Parallel tracking and mapping for small AR workspaces," in *Proc. IEEE Intl. Symposium on Mixed and Augmented Reality (ISMAR)*, 2007.