

Rapid Prototyping Framework for Visual Control of Autonomous Micro Aerial Vehicles

Ignacio Mellado-Bataller¹, Pascual Campoy¹, Miguel A. Olivares-Mendez¹,
and Luis Mejias²

¹ Centre for Automation and Robotics (CAR), Universidad Politécnica de Madrid
C/ Jose Gutierrez Abascal, 2. 28006 Madrid, Spain
www.vision4uav.com

² Australian Research Centre for Aerospace Automation (ARCAA),
Queensland University of Technology
GPO Box 2434, Brisbane Queensland 4001
luis.mejias@qut.edu.au

Abstract. Rapid prototyping environments can speed up the research of visual control algorithms. We have designed and implemented a software framework for fast prototyping of visual control algorithms for Micro Aerial Vehicles (MAV). We have applied a combination of a proxy-based network communication architecture and a custom Application Programming Interface. This allows multiple experimental configurations, like drone swarms or distributed processing of a drone's video stream. Currently, the framework supports a low-cost MAV: the Parrot AR.Drone. Real tests have been performed on this platform and the results show comparatively low figures of the extra communication delay introduced by the framework, while adding new functionalities and flexibility to the selected drone. This implementation is open-source and can be downloaded from www.vision4uav.com/?q=VC4MAV-FW

Keywords: MAV, UAV, communications, software framework.

1 Introduction

"Fail early, fail often" is a wise mantra. The earlier you find the mistakes in your new idea, concept or system design, the sooner you can fix them and get your path to success. This is especially applicable in the field of visual control, where dynamic systems are controlled using images from one or more cameras as feedback. Visual control algorithms that work fine on the simulator may fail catastrophically in the real world. In this paper, we propose a flexible rapid prototyping software framework for visual control of Micro Aerial Vehicles (MAV).

In the last few years, personal Micro Aerial Vehicles and their components are hitting the consumer market like [1], [2] and [3]. For the work in this paper, we are using the Parrot AR.Drone as prototyping platform [1]. This drones is sold as a toy at amateur-affordable prices and can be remotely operated from a smart phone. It has

out-of-the-box onboard cameras and Inertial Measurement Units (IMU). Whereas the overall quality is low compared to a professional MAV, it can be 20-40 times cheaper. In addition, it can be bought at several toy stores and taken straight to the lab without worrying about delivery delays. Furthermore, because of its low cost, taking risks is acceptable: if you crash and break one, you can just buy a new unit. For these reasons, it is worth to be taken into account as prototyping platform, especially when developing algorithms for MAV swarms with many units, where the total cost might be prohibitive with more professional MAVs.

The Parrot AR.Drone has a Software Development Kit (SDK) that enables third-party developers to write applications for the drone. A third-party program that is run on an external workstation is able to send commands and receive information from the sensors, the camera and the IMU through a WiFi link. The drone has internal high frequency control loops that drive the rotors to maintain the demanded attitude and altitude. However, the SDK is too limited for our research requirements, as it only supports a single point-to-point link between the program and the drone, thus, a program can only communicate with a single drone. Besides that, we would like to work with networked communication schemes like those shown in Fig. 1.

The required new functionalities are provided by the proposed software framework, while increasing the isolation between the application and the hardware platform, and opening the possibility to easily port applications to other MAVs with either onboard or off-board computing.

In section 2, other related works are explored. In section 3, we introduce some general guidelines of the framework architecture, while a specific implementation for the Parrot AR.Drone and a C++ application is discussed in section 4. In section 5, some test results of this implementation are presented and, in section 6, they are discussed. Section 7 concludes the paper.

2 Related Work

The AR.Drone SDK already offers an API for developing third-party applications [10]. Examples of research works with the AR.Drone are [7], [8] and [9]. However, by the time this paper is written, it does not support communications with multiple drones across a network. With regards to the communications between the application and the drone, reference [4] points to an existing open project by ETH PIXHAWK. It offers a communication architecture for MAVs that is based on a library for message transmission over a network [5], but it does not have native support for the Parrot AR.Drone. On the other hand, there is a driver for AR.Drone by Brown University [6] for the Robot Operating System (ROS) that was used in [9]. Nevertheless, it does not implement either access control to the drone or parameter configuration. Moreover, we would like our framework to remain lightweight, without burdening the new developer with the installation of heavy and complex packages like ROS. The framework implementation presented in this paper tries to fill the gap left by the cons of the other alternatives. It is being used to carry out experiments of a see-and-avoid fuzzy controller, after tuning the system with the robot simulator Gazebo. More information can be found at [11].

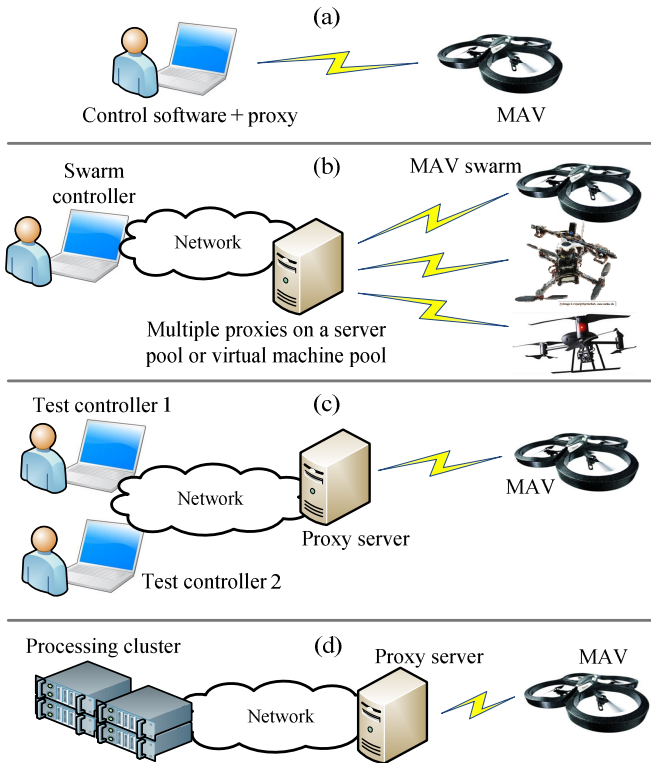


Fig. 1. New communication schemes provided by our framework. In (a), a point-to-point scheme, also allowed by the AR.Drone SDK; in (b), an application controls an MAV swarm; in (c), multiple researchers may share the same MAV resources -one at a time-; in (d), video is broadcasted over a cluster for parallel processing.

3 Framework Architecture

In this section, we define a general model for the implementation of our framework architecture. These are guidelines and requirements that are extensible to any MAV, (whose capabilities are similar to the AR.Drone's), any packet network and any application programming language. In the next section, the model will be applied to the AR.Drone and to specific network technologies and a C++ API.

To give network capabilities to the drone, a proxy-based architecture has been defined. The architecture is depicted in Fig. 2. The proxy is responsible for connecting a single drone with the network. With one proxy per drone, all drones can share the network as communication mean. At the application side, we define an Application Programming Interface (API). Thanks to this API, the application is able to communicate with the proxies of the different drones that it aims to control.

It is worth to notice that the framework components do not have fixed running locations. If the MAV processing platform has open access, the proxy can be run onboard. Then, the control application can reside either onboard, communicating locally, or off-board, through a wireless link.

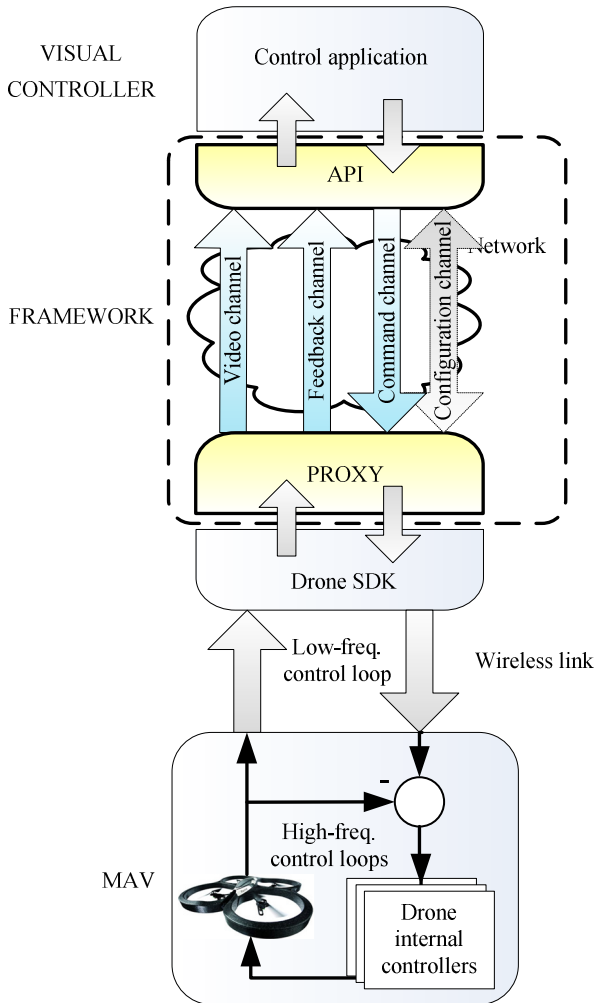


Fig. 2. System architecture. The framework interconnects the visual controller with each MAV through a network. The total system follows a cascade control structure. Usually, there are high-frequency controllers onboard the MAV, while the visual controller closes an outer loop with lower frequency. The network link for the control loop is formed by three independent channels. Configuration channel is used to read and change parameters and it is not part of the control loop.

Otherwise, if the MAV onboard processing is closed, which is the case for the AR.Drone, the proxy is run off-board, and the control application can be executed either on the same platform or on any other that is connected through a network, as seen in Fig. 1.

Regarding portability, while the proxy depends on the drone manufacturer, the Application Programming Interface (API) library is platform-independent. In other words, the proxy isolates the application from the drone specifics. In this way, there is

no need to update the control applications every time the manufacturer releases a new SDK version. Most times, updating the proxy will be enough. Another advantage of this isolation, is the possibility of porting the API to programming environments or languages not supported yet by the manufacturer's SDK. For instance, a Matlab API could be programmed, despite not existing any specific software by the manufacturer.

3.1 Communications

The communication link between the proxy and the MAV depends on the manufacturer specification and it may vary between different models. It is the manufacturer who defines the communication protocol of the drone and it will not be discussed in this paper. Our framework is responsible of the link between the proxy and the application. This link is formed by four independent communication channels, named: command, feedback, video and configuration. These channels are logical, not necessarily physical, as they are established over the network. They just represent an information flow between both network nodes. To implement the channels, no specific communication protocols are defined as mandatory; there are only recommendations.

The network between the proxy and the application may fail. A cable might break, a router might stop or a WiFi link might lose the signal. Both ends of the link must be robust to these situations and implement self-recovery mechanisms, which must be transparent to the application. The application will be notified of a failure situation but will not have to perform any actions to fix it. In case of stateless protocols –those not requiring to establish a connection– there is no extra effort to be done, as packets will continue to be transmitted after the network is recovered. Nevertheless, the application must be notified if packets do not arrive at expected times. Oppositely, connected protocols must automatically try to reconnect until the network is restored, besides informing the application of the link state.

3.1.1 Command Channel

The command channel transports all the control actions from the application to the proxy: a signature, a sequence number and drone-specific commands (required flying mode, desired attitude, etc.). The signature identifies the packet as a command channel packet and may be used as a start token.

Through this channel, data packets are transmitted periodically. Low delays are favored by allowing packet dropping, because control loops depend on this channel and delays generally harm loop stability [12]. Datagram protocols, like UDP over an IP network, are suitable for implementing this mechanism because they do not have automatic retransmission of faulty packets. At the proxy side, if a packet is lost or it arrives after a previously sent one, it is discarded. Instead of asking for a retransmission or reordering packets after a sequence error, the proxy expects that a new packet with up-to-date command information will eventually arrive. The purpose of the sequence number is to determine if a packet has arrived out of sequence.

An MAV can only be commanded by one control application at a time. Therefore, no concurrent access is allowed on this channel. When the channel is in a free state, any control application can lock it by sending an initialization packet for write access. After that, no other packets from other applications are processed until the original

application unlocks the channel or stays inactive for a time longer than a pre-configured threshold.

3.1.2 Feedback Channel

In the feedback channel, navigation information flows from the proxy to the application. The content of each feedback packet is: signature, timestamp and drone-specific information (proxy-drone link health, battery level, measured attitude, etc.). The signature identifies the packet as pertaining to this channel and may be used as start token. Like in the command channel, packet dropping at the receiver –based on the timestamp– is encouraged in order to minimize delays in control loops.

Through the feedback channel, the proxy can feed data to multiple applications simultaneously. In situations where network broadcast or multicast is not possible, applications can subscribe to a consumer list in the proxy by requesting a read-only access through the command channel. The multicasting is then emulated by the proxy by iteratively sending the information to its subscribers, at the expense of increasing processing time and delay.

3.1.3 Video Channel

In a video channel, video from a drone camera is transmitted to the control application. Like the feedback channel, multiple applications can request video channels from a proxy. While a feedback channel sample will usually fit in a network packet, a video channel sample, i.e. a frame, will need to be encoded, packetized and transmitted with some transport protocol. Like for the other channel types, the lower the transmission delay is, the higher the stability margin of a visual control loop will be. Hence, implementations with compression-ready encodings, low-delay protocols and frame-dropping mechanisms would be preferred.

The video channel transports periodic fragments with frame data that include a header with a signature (it may be used as a fragment start token), information about the video encoding and a timestamp, so the application knows how to decode the video stream and when each frame was captured. The timestamp must be as close as possible to the real capture time of a frame. If the MAV does not provide this information, the proxy will give an estimation. When possible, the timestamps of both video and feedback channels must use the same clock reference. Although this reference is unknown by the application, sample times of different channels can be compared and ordered if needed.

3.1.4 Configuration Channel

The configuration channel is used to read and write configuration parameters of the MAV from the application. It is intended for parameters that are not time-critical, such as allowed attitude ranges or video capture features, which are mainly changed at startup. In order not to disturb any other channels requiring a higher bandwidth and a lower delay, any fast changing parameters must be transferred through the command and feedback channels.

When the application writes a parameter through the configuration channel, it must have a confirmation that it has actually been changed in the MAV, as it might be safety-critical. Likewise, when reading a parameter, the application must know that it was actually read. Therefore, a connection-oriented transport protocol is required for this channel. For example, TCP on an IP network would suit these requirements.

3.2 Application Programming Interface

The API library enables the application to access the communication architecture programmatically. The control application processes the feedback information from the MAV and generates the commands to be sent in response, closing the loop. The API defines methods that are directly called to change these commands.

The application can gather the feedback information –video and navigation– in two ways. The first one consists in explicitly polling the data when needed. However, because of the asynchronous nature of the feedback channel, the data is not requested on demand to the drone, but periodically received. And, consequently, the request method returns the last sample that was received from the proxy. The second method for feedback retrieval is event-driven. The application registers a listener through the API and the listener gets a notification whenever the data is received from the proxy, so it can be processed immediately. Navigation data and video frame notifications are received independently, as they are transmitted through unrelated channels, due to their different bandwidth requirements.

4 Framework Implementation

The framework model has been implemented for a Parrot AR.Drone. The implementation is targeted to IP networks, and the API works on Linux as a C++ programming language library. A specific proxy has been built for the AR.Drone with the manufacturer's SDK examples [10], also for Linux. The proxy is a separate executable that runs off-board the MAV because the onboard computer is closed to third-party code. The manufacturer point-to-point communication with the drone is established via WiFi.

4.1 Channels

The command channel is implemented using a UDP socket. Each data packet carries the following information: sequence number, required flying mode, attitude desired values and desired altitude speed. The sequence number is required by the framework model, while the rest is payload information specific to the MAV.

The feedback channel uses a UDP socket, too. It transports the following information: timestamp, proxy-drone link health, drone state, battery level, measured attitude, measured altitude and measured velocities. The timestamp is defined as mandatory by the model. The other data is drone-specific.

As UDP is not a reliable protocol, command and feedback channels are provided with a retransmission mechanism. At the application side, as soon as commands are changed by the application, the API library transmits them to the drone through the proxy. When the application is not generating new commands, the API library keeps transmitting the last commands periodically to ensure that they eventually arrive to the other end. The proxy has the same mechanism: new sensor readings are sent immediately, but if they are not available at a predefined minimum frequency, the last readings are periodically sent through the feedback channel to ensure that they arrive to the other end. In this way, there is constant activity in the channels and both ends know that they are linked.

The video channel is implemented with a TCP socket. According to the model definition, this is not the most adequate protocol because it is not designed for real-time, but for reliability. Nonetheless, the transparent streaming capabilities of the protocol make the video channel implementation straight-forward. Each frame is transmitted with its own encoding. Currently, the supported encodings are JPEG and raw RGB with eight bits per plane.

To pass the received video frame to the application, the API library has a triple buffering mechanism: the reception buffer, the frame-ready buffer and the processing buffer. The first one is continuously retrieving the frames from the network, preventing the TCP buffers from overflowing, which would time out the transmission at the proxy side and would be interpreted as a connection failure. Right after a frame is received, it is copied to the frame-ready buffer, to keep it accessible by other program modules, while the reception buffer is free to receive the next frame from the network. However, the frame-ready buffer is overridden as soon as a new frame is received, therefore any operation on this buffer should last less than a frame period. For longer processing times, the processing buffer is provided. When the frame-ready buffer gets new contents, all the video channel listeners are notified. One of them is a video processor module that copies the frame-ready buffer contents to its own processing buffer only after the last processing operation has finished. Meanwhile, the frames are dropped for that video processor. Multiple video processors can be freely initiated by the application, allowing concurrent frame processing with independent frame dropping for each processor.

The configuration channel is implemented with a TCP socket, as low delays are not mandatory but reliability is. Each parameter operation is performed in a transaction consisting in a request and a response. Each request contains a signature, the request type, the parameter identifier and the parameter desired value. The desired value will only be interpreted by the proxy if it is a write request. The response is formed by a signature, a value indicating whether the last request was successful and the parameter value. The parameter value will only be meaningful if the last request was for reading.

4.2 Robustness

At both communication ends, there is code responsible for keeping communication channels synchronized. If faulty behavior occurs, the corresponding channel is restarted, so both ends are automatically synchronized back. The channel behavior can be understood as faulty when a malformed packet is received or when packets are not received as frequently as expected. Every time this happens, the application is notified so it can react accordingly. For example, it could display an alarm on a user interface. However, the channel recovery mechanism is completely transparent and all efforts for the channel restoration are performed by the framework.

At the application side, all the API errors are handled with C++ exceptions. This mechanism favors that errors show up during the development phase so they can be fixed early. In this API implementation, every thread has a last line of defense that catches all non-caught exceptions, writes the exception in a log file for debugging and prevents the thread from being terminated, so it can try to recover the normal state.

4.3 Extra Features

The API library is able to interface with a Vicon positioning system. With this system, position and attitude information of MAVs can be gathered inside a delimited space. This information can be very useful, for instance, to close control loops or as ground truth for visual pose estimation algorithms.

On the other hand, the API library provides data logging functionalities. The data logger can gather events generated by the channels and the Vicon interface. Hence, commands, navigation feedback, video feedback and Vicon data can be stored in a disk for later analysis. The data logger runs asynchronously, so the delays of the disk write operations do not bother other ongoing threads.

Finally, the API defines classes that help developing a controller by only overriding two methods. The methods are automatically called whenever navigation or visual feedback is received from the MAV. A controller may be implemented inside these methods. The received information is used as input to the controller and the controller's output is sent to the MAV directly calling the appropriate API methods. The images from the cameras are passed back with the encoding used by OpenCV. To help writing the controller code, the API also exposes matrix data types that perform common algebraic operations.

5 Experimental Results

The total communication delay between the application and the drone will be the sum of the delays introduced by the API library, the network, the proxy and the proxy-to-drone link. The drone manufacturer is accountable for the last one. The second one is given mainly by the physical network infrastructure. The first and third elements are responsibility of the framework implementation and must be measured.

In order to measure the framework contribution to the delay, the proxy is run in the same host where the application resides, so the API-proxy link is established through local sockets. Timestamps are added to channel packets at the sender and the time lapse is calculated at the receiver. As both processes run on the same computer, they share the same clock reference and time calculations can be performed without additional synchronization.

Regarding the proxy-to-application delay, the timestamps are obtained right after receiving the data from the drone, so all proxy processing time is also taken into account. The arrival time is acquired right after releasing the data to the application. For the application-to-proxy delay, the timestamps are taken right after issuing the commands to the API and the arrival time is calculated at the proxy, before sending the commands to the drone through the point-to-point link. The framework version 0.8 beta has been used in this test. The test was run on an Acer Aspire 5750G with a Intel Core i7-2630QM 2GHz processor and 8 Gbytes of DDR3 RAM. The Operating System was Linux Ubuntu 11.04. During the test, the data logging was disabled. The packet frequency for the command and feedback channels was set to 32 Hz. The video frame rate was 15 frames per second in average (this is determined by the

AR.Drone) and the video channel frames were encoded as raw RGB with eight bits per plane. The test application consists on a simple visual teleoperation interface with a waypoint-based path controller. The test duration is 5 minutes.

Figs. 3 and 4 show the distribution of delays introduced by the framework in the command and video channels (the feedback channel distribution is similar to the command channel one). Table 1 gives some numerical details about the delay distributions. Fig. 5 shows the time evolution of the delays. In all figures, delays are expressed as percentages of the channel period. The channel periods are 31.25 ms for command and feedback, and 66.7 ms for video.

Table 1. Characterization of channel delays

Channel	Delays (ms)			Num. samples	
	Mean ^a	Min.	Max.	Total ^b	Delay < 1% of channel period ^c
Command	0.091 (0.29%)	0.013	1.739	9,771	99.93%
Feedback	0.109 (0.35%)	0.025	1.670	9,828	99.73%
Video	1.038 (1.56%)	0.310	1.936	5,011	0.22%

a. Absolute delay in milliseconds and delay relative to channel period.

b. Total number of delay samples.

c. Delay samples lower than 1% of channel period (31.2 ms command and feedback; 66.7 ms video).

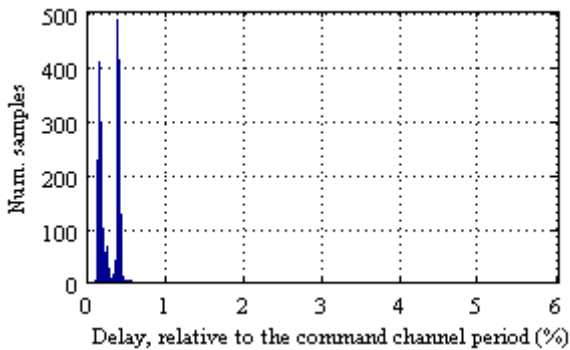


Fig. 3. Distribution of the delays introduced in the command channel by the framework. The relative delays are percentages of the command channel period, i.e. 31.25 ms. The highest sample is 5.56%.

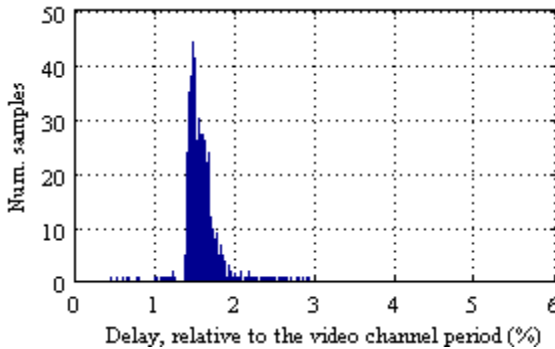


Fig. 4. Distribution of the delays introduced in the video channel by the framework. The relative delays are percentages of the average video channel period, i.e. 66.67 ms. The highest sample is 2.9%, but the horizontal scale is set as in Fig. 3 for easy comparison.

6 Discussion

As seen in table 1, the average delays introduced by the framework are considerably low, compared to frequencies of the channels. For a visual controller, the impact of the framework in the reaction time would be the result of adding the visual and command channel delays, i.e. the time it takes to see an event plus the time to react accordingly. In average, it is a contribution of 1.129 ms to the total loop delay. Assuming a visual control loop at 15 frames per seconds, this represents a 1.7% of the loop period.

In Fig. 5, there are spurious samples that might be caused by the fact that the implementation is not running on a real-time Operating System (OS). Instead, this OS has a preemptive scheduler that can interrupt a task anytime to yield some time for other tasks. Despite it might not cause problems during usual prototyping, it must be taken into account for high-frequency delay-sensitive applications.

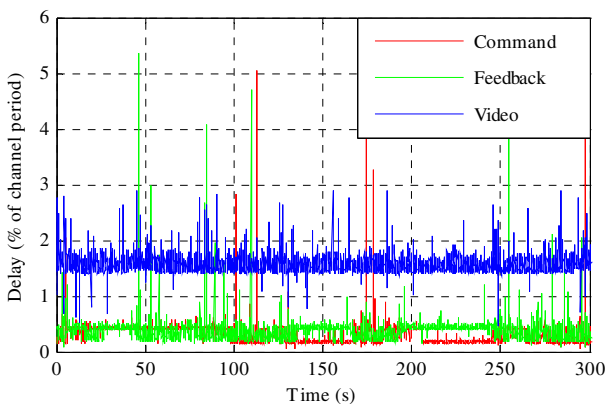


Fig. 5. Evolution of relative channel delays introduced by the framework. Most of the time, command and feedback delays are below 1% of the channel period. Highest peaks reach 5%, but are very unusual. The video channel delay is always under 3%.

7 Conclusion

We have introduced a framework for fast prototyping of visual control applications for Micro Aerial Vehicles (MAV). First, a framework model with general guidelines has been presented, without regarding specific technology details, in order to leave it open to other implementations. The framework architecture is able to transform cheap MAVs without onboard processing and networking capabilities into network nodes for off-board processing, opening the door to new prototyping configurations, like drone swarms, distributed vision processing or MAV sharing by multiple researchers. Moreover, the framework defines a common API that may be used to control MAVs with similar capabilities from different manufacturers, thus improving manufacturer independence with minimal code changes.

In order to show the framework applicability to cheap prototyping, an implementation on an amateur MAV –a Parrot AR.Drone– has been discussed. In the experimental results, the delays added by this implementation to the control loops are significantly low, compared to the loop periods. However, the timings are not deterministic because the implementation is not running on a real-time Operating System. Thus, the applicability to safety-critical controllers is disregarded. Anyhow, it does not affect most applications. On the contrary, the framework has proven to be a useful tool for rapid testing. This implementation is an open-source project. It is available at www.vision4uav.com/?q=VC4MAV-FW

Acknowledgements. The authors thank the Australian Research Centre for Aerospace Automation - Queensland University of Technologies (ARCAA-QUT), where a major part of the development and testing for this work was carried out. The authors would like to thank Caja Madrid for the mobility research grant of one of the authors. This work has been sponsored by the Spanish Science and Technology Ministry under the grant CICYT DPI2010-20751-C02-01 and by the IRSES Program Marie Curie FP7-PIRSES-GA-2009-230797 - ICPUAS (International Cooperation Program for Unmanned Aerial Systems Research and Development).

References

1. Parrot AR.Drone, <http://ardrone.parrot.com>
2. MikroKopter, <http://www.mikrokoetter.us>
3. ArduCopter Quad, http://www.udrones.com/product_p/acrtf1.htm
4. MAVCONN Aerial Middleware,
<http://pixhawk.ethz.ch/wiki/software/middleware/start>
5. MAVLINK protocol library,
<http://www.qgroundcontrol.org/mavlink/start>
6. ROS driver for the Parrot AR.Drone, http://code.google.com/p/brown-ros-pkg/wiki/ardrone_brown
7. Visser, A., Dijkshoorn, N., van der Veen, M., Jurriaans, R.: Closing the gap between simulation and reality in the sensor and motion models of an autonomous AR.Drone. In: Proc. International Micro Air Vehicle Conference and Flight Competition (IMAV 2011), pp. 40–47 (September 2011)

8. Bills, C., Chen, J., Saxena, A.: Autonomous MAV flight in indoor environments using single image perspective cues. In: Int. Conf. Robotics and Automation (ICRA), Shanghai, China, pp. 5776–5783 (May 2011)
9. Koval, M.C., Mansley, C.R., Littman, M.L.: Autonomous quadrotor control with reinforcement learning, <http://mkoval.org/projects/quadrotor/files/quadrotor-rl.pdf>
10. ARDRONE open API platform, <https://projects.ardrone.org>
11. See and avoid with a fuzzy controller, [http://vision4uav.eu/?q=research line/SeeAndAvoidCE](http://vision4uav.eu/?q=research+line/SeeAndAvoidCE)
12. Branicky, M.S., Phillips, S.M., Zhang, W.: Stability of networked control systems: explicit analysis of delay. In: Proc. American Control Conf, pp. 2352–2357 (June 2000)
13. Preemption (computing), http://en.wikipedia.org/wiki/Preemption_%28computing%29#User_mode_and_kernel_mo