Flexible Image Acquisition Service for Distributed Robotic Systems

Oleksandr Semeniuta[1] and Petter Falkman[2], *Member, IEEE*

***Abstract*— The widespread use vision systems in robotics introduces a number of challenges related to management of image acquisition and image processing tasks, as well as their coupling to the robot control function. With the proliferation of more distributed setups and flexible robotic architectures, the workflow of image acquisition needs to support a wider variety of communication styles and application scenarios. This paper presents FxIS, a flexible image acquisition service targeting distributed robotic systems with event-based communication. The principal idea a FxIS is in composition of a number of execution threads with a set of concurrent data structures, supporting acquisition from multiple cameras that is closely synchronized in time, both between the cameras and with the request timestamp.**

I. Introduction

Vision systems are pervasively used in the context of robotics and automation. They are of different levels of sophistication, including cameras connected to external computers, smart cameras performing processing locally, systems based on FPGA, systems including additional hardware such as light sources and sensors, and so on. When it comes to vision software, it ranges from off-the-shelf tools with graphical user interface to highly-customized solutions based on software development kits supplied with the cameras of choice.

A machine vision system is functionally comprised of such processes as imaging (with the optical system and an image sensor), acquisition (realizing the triggering and streaming function), and image processing, combined with high-level decision making. Naturally, computer vision researchers largely focus on development of processing algorithms (once an image is available). However, in distributed environments, inherent in robotic systems, it is important to consider the dynamic nature of image capture, communication, and acquisition. This is even more important in context of the emerging paradigm of cloud robotics and automation, which is inherently based on network-based capabilities.

There exist several camera communication standards, such as Camera Link, CoaXPress, IEEE1394, USB 3 Vision, and GigE Vision. The latter, based on the widely-used Gigabit Ethernet networking standard, provides a number of advantages, such as the ability to design distributed network topologies (as opposed to point-to-point), scalability in terms of adding more cameras into the systems, ability to work on standard hardware and software, and bandwidth suitable for real time video streaming [1].

When dealing with a GigE Vision camera driver, one distinguishes between image *capture* and image *acquisition*. The former constitutes obtaining images inside the camera, while the latter – transforming the corresponding images to the computer's memory. Industrial cameras can be subdivided into two classes: *asynchronous reset cameras*, which perform capture only when synchronized with an external event, and *free-running cameras*, which continuously capture images at a constant rate. The latter are more commonly used, and have two acquisition modes that can be realized. Depending on the design of the image acquisition module, the caller may issue a trigger event and process the acquired image sequentially (the synchronous mode), or perform triggering and processing concurrently (the asynchronous mode) [2]. The latter case is more effective, since it allows to process each individual image the camera is able to capture, and involves several threads of execution.

Regardless whether synchronous or asynchronous acquisition mode is used, the traditional approach is to perform all the necessary processing in-place, right after the image is obtained. In such a way, no additional image transport is necessary, and one is able to realize a real-time processing system. At the same time, the resulting solution is inherently monolithic, and it becomes harder to integrate numerous networked components, including other cameras and different kinds of sensors. Another extreme use case is associated with off-the-shelf vision software, where far less control is given to the system developer.

To deal with the increased complexity of distributed systems, various solutions based on service-oriented thinking has been realized. Service-oriented computing is important theme of research and development in information systems and software engineering when it comes to realization of distributed systems. Though there exist several technically and semantically different service-oriented architectures and paradigms (such as SOAP, REST, publish/subscribe, microservices), they share the common principle of decomposition of computational units and provision of services on demand. In the field of industrial automation, SOA manifested itself in such forms as OPC-UA, DPWS, the Tweeting Factory [3], and services and topics in the Robot Operating System (ROS).

Because of lack of precise understanding of timing characteristic of a vision system, as well as its interaction with other concurrent processes, the system efficiency can be far from optimal. A related downside can be caused by application of simplified APIs that waste time by performing a series of

*This work was supported by the Norwegian Research Council

[1]Oleksandr Semeniuta is with Department of Manufacturing and Civil Engineering, NTNU Norwegian University of Science and Technology, Gjøvik, Norway oleksandr.semeniuta@ntnu.no

[2]Petter Falkman is with Department of Electrical Engineering, Chalmers University of Technology, Göteborg, Sweden petter.falkman@chalmers.se

initialization and closing operations on each call.

The envisioned solution is a streaming service, easily integrable into a distributed robotic system based on a service-oriented architecture and a standard communication middleware. The internal dynamics of the service is well-understood and robust enough to deal with request from other networked components. External systems are able to request individual images, multi-view tuples of images (in case of multi-camera systems), or sequences of images (consecutive or interval-based). Image request can be done both synchronously, using a request-response protocol, and asynchronously, using message queues in a publish-subscribe manner.

This paper presents FxIS[1] (for *flexible image service*), an image acquisition framework, intended for use as a part of distributed robotic system with service-oriented architecture. FxIS supports robust acquisition from several connected cameras and ensures synchronized response by providing images most closely associated with the time instant of the request.

The paper is organized as follows. The technical context of FxIS development is described in section II. The principles of image capture and acquisition, along with the traditional approaches of utilizing these capabilities are presented in section III, followed by the justifications for the approach manifested by FxIS. Formulation of the FxIS architecture and functional principles are presented in section IV. A set of concurrent data structures, providing the important function of data sharing, communication and synchronization between multiple threads of an FxIS application are described in details in section V. Analysis of time measurements during an FxIS operation experiment are presented in section VI. Description of the future functionality of the framework is outlined in section VII.

II. Technical context

In the pilot implementation of FxIS, two identical Allied Vision GigE Vision cameras (Prosilica GC1350) were used for testing. Therefore, the lower-level primitives for controlling cameras are based on the Allied Vision's Vimba SDK [4], with `x86-64` Linux as the primary platform.

Vimba SDK provides a C++ object-oriented application programming interface. The recent C++ standards (C++11, C++14, C++17), have introduced numerous modern programming abstractions while preserving efficient mapping to machine instructions. FxIS is developed in C++14, which allows having an effective coupling with image processing libraries and making use of modern C++ language features. In particular, FxIS is heavily based on the unified concurrency support (`<chrono>`, `<mutex>`, `<condition_variable>`, `<future>`), high precision time measurement (`<chrono>`), as well as the randomization capabilities (`<random>`).

OpenCV [5] has become the open standard in image processing and computer vision. It includes a large number of tested open source algorithm implementations, as well as an object-oriented image/matrix data structure, `Mat`. FxIS uses OpenCV for the core image processing functionality, and, instead of dealing with raw image bytes, `Mat` objects are internally handled.

III. Image acquisition process

A camera is constrained its maximal framerate r, measured in frames per second (fps). The same performance can be expressed with the inversely proportional inter-arrival time $\tau_{ia} = r^{-1}$. For example, Prosilica GC1350 cameras, used in this work, offer $r = 20$ fps, which result in $\tau_{ia} = 50$ ms.

Once the image is acquired, it is laid-down contiguously (row-by-row) in the computer memory as an array of `unsigned chars`. An image sensor having the dimension of $w \times h$ and c color channels (for monochrome and Bayer images, $c = 1$), produces the array of size hwc. Data from this array can either be processed in-place or copied for later processing as follows:

```
memcpy(
  out_image_ptr,
  image_ptr,
  h * w * c
);
```

If the duration of image processing τ_p is shorter than inter-arrival time ($\tau_p \leq \tau_{ia}$), one processing thread is sufficient, and the acquisition mode is equivalent to the synchronous approach. Otherwise, one requires several processing threads. In general, for $n \geq 1$, n threads are required if the processing time lies in the following range:

$$(n-1)\tau_{ia} < \tau_p \leq n\tau_{ia}$$

The GigE Vision standard is based on UDP, hence allowing for higher speed of image transmission as compared to TCP. On top of it, two application-specific protocols are defined, GigE Vision Control Protocol (GVCP) and GigE Vision Streaming Protocol (GVSP). These protocols increase transmission reliability as compared to the raw UDP. Additional research has been done in improvement of packet loss during UDP-based communication in GigE Vision systems [6].

Research related to systems based on the GigE Vision standard largely tackles the problem of minimizing τ_p. Hence, it typically concerns development of FPGA-based smart cameras and other FPGA-based hardware solutions. Examples include a custom-designed smart camera with GigE Vision interface and FPGA-based processing [7], a multi-camera FPGA- and GigE Vision-based image processing system [8], and a custom FPGA-based solution for acquiring images from GigE Vision interface (by bypassing the machine's CPUs) and direct writing to RAM [9].

This paper, conversely, focuses on implementation of image acquisition function on a general-purpose machine, with the ability to spread-out the processing function over a number of networked nodes. Several related systems focusing on

[1]The FxIS implementation is available under the 3-clause BSD license at https://github.com/semeniuta/FxIS

video streaming function has been reported in the literature. Examples are a multi-camera system for fall detection [10], a machine learning-enhanced system for video streaming with selective resolution coding based on attention regions [11], and a real-time database and hardware/software platform for coupling real-time image processing and cognitive processing in a self-driving car [12].

IV. System description

A. Typical application scenario

An example of use case of FxIS is shown on Figure 1. It constitutes a distributed system composed of such computational units an FxIS-based image acquisition service, a robot control node, and an image processing node. Physical distribution of these components and communication modalities may differ depending on the application. More complex systems can be comprised of larger number of device and image processing nodes.

In the given example, the robot announces a request for timely image processing results, which is first handled by the acquisition service returning a pair of images most closely associated with the robot request. The resulting stereo pair is further processed by the image processing component, which communicates the results back to the robot. The communication is done via publish/subscribe connections.

The idea FxIS is to allow external systems, such as the robot control node, to request individual images, multi-view tuples of images (in case of multi-camera systems), or sequences of images (consecutive or interval-based).

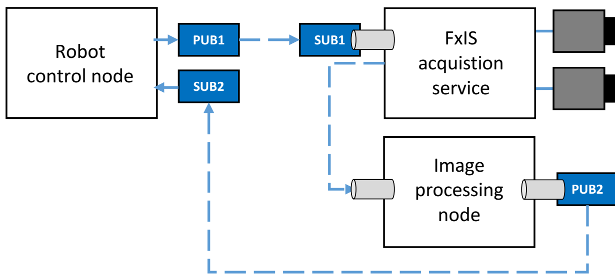

Fig. 1. Event-driven communication in a robotic cell

B. System architecture

The principle structure of an FxIS application is shown on Figure 2. The highest-level component is a `Service`. It is responsible for handling requests to the imaging system from various clients and holding the control over the image acquisition process. Per each camera connection, there exists a `Streaming` object, running in a separate thread, and an `ImageStream` concurrent data structure. As it will be further explained in Section V, an `ImageStream` can be safely accessed by multiple threads, and provides the functionality of image storage and search-based retrieval. While the camera driver writes (W) to the camera's `ImageStream`, the `Service` component may request (R) images from the stream with the current timestamp t^*.

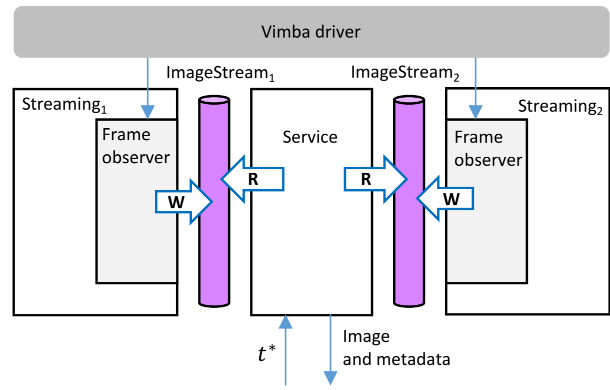

Fig. 2. FxIS application components

The programming model in Vimba C++ driver is based on the observer design pattern. A concrete frame observer class, enclosed in an instance of `Streaming`, is associated with the given camera, and the driver invokes the `FrameReceived` method once a new frame is acquired. In the free-running mode it happens every τ_{ia}.

The dynamic workflow of an FxIS application is shown on Figure 3. When started, a `Service` spawns a number of threads (each per one camera interface) based on the corresponding `Streaming` objects. When instantiated, the latter establish connections with the respective cameras. In case of using Vimba driver, this includes allocation of frames, starting capture on the camera, and queuing frames for acquisition [13]. When a new thread is spawned based on the `operator()` method of the `Streaming` object, the acquisition process is started.

When the work time of an FxIS service is finished (e.g. because of the user's request), all the threads are signaled for completion via instances of the `BlockingWait` and `EventObject` concurrent data structures $\{bw_i, eo_i\}$ (see section V).

The concurrent access in `ImageStream` is secured by the internal mutex-based implementation. When

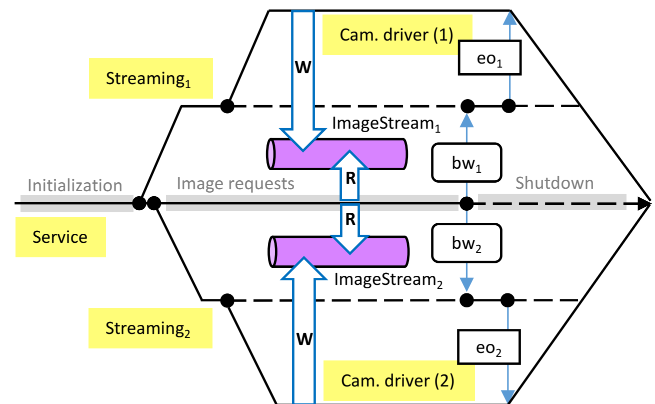

Fig. 3. Threads and concurrent data structures' interaction in an FxIS application

`ImageStream::getImage` is invoked, the thread blocks and wait until the next frame arrives from the camera driver. This ensures that, if $(t^{(\text{next image})} - t^*) < \tau_{ia}/2$, the next image will not be missed. After the thread is unblocked, the nearest timestamp search is performed, and a copy of the corresponding `Mat` image is returned to the caller.

To allow time tracking, both on writes to and reads from an `ImageStream` object, timestamps, measured with the C++ high resolution clock, are recorded. When a new image frame is acquired and processed, two timestamps are of interest: the moment when the frame becomes available (t_f) and the moment when all the processing is performed and the image is stored in `ImageStream` ($t_{p\downarrow}$). When it comes to image requests, executed as reads from `ImageStream`, the following timestamps are recorded: the moment of the image request t_{ir}, the moment of mutex request t_{mr} (after waiting for the next image to be acquired), the moment when the mutex is acquired t_{ma}, the moment when the search for the nearest image is done t_{sd}, and the moment when the image is copied to the caller's `Mat` object t_{ic}.

V. Concurrent data structures

An FxIS-based application is inherently concurrent, and is designed as a collection of multi-threaded components communication via concurrent lock-based data structures.

`ImageStream` is a concurrent data structure for storing and retrieving recent images, acquired by the camera. Internally it is based on a fixed-sized `std::vector` of `Mat` objects, which are stored in a circular manner: the array is filled sequentially, and is being overwritten once the counter reaches the maximal value. When an image stored, it is associated with the corresponding timestamp. Let the n-sized array for storing images be indexed from 0 through $n-1$. If k is the current index, the n most recent images, from the oldest to the newest, are indexed as follows:

$$\{I_k, I_{k+1}, \ldots, I_{n-1}, I_0, I_1, \ldots, I_{k-1}\} \tag{1}$$

To manage the timestamps, `ImageStream` is coupled with an instance of `CircularTimestampVector` class. When a circular vector is queried, for example during the search routine, the client assumes that it deals with sorted sequence of timestamps with indices $i_q \in \{0, n-1\}$. In the example shown on Figure 4, the actual indices of a circular vector are depicted in the lower part of the figure, while the queried indices (i_q) – in the higher part.

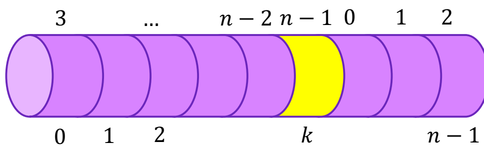

Fig. 4. Indices in a circular vector

The sequence (1) is thus corresponding to the following mapping:

$$i_q \xmapsto{\text{IndexMap}} \begin{cases} k + i_q & \text{if } i_q \le n-1-k \\ i_q - n + k & \text{otherwise} \end{cases} \tag{2}$$

A `CircularTimestampVector` stores the timestamps associated with each stored image, keeps track of the current index k, provides the nearest timestamp search functionality, and realizes IndexMap.

An `ImageStream` instance can have several threads accessing it concurrently, where thread safety is achieved with an associated mutex. The workflow of `ImageStream` is the following: when a new image frame becomes available, the associated method of the frame observer of the camera is called. It created a `Mat` object from the available image bytes and inserts it into the associated `ImageStream`. When `Service` responds to an image request event, it remembers the current timestamp t^* and queries one or several `ImageStream` objects with t^*. Because of the mutexes, the query calls may be blocked due to writing being done on the frame observers' side. However, when the mutexes are released, each `ImageStream` returns an image acquired at the time point closest to t^*.

Search of the closest timestamp in `ImageStream` is performed by the algorithm based on binary search (see Algorithm 1), with the base case of two elements, hence providing logarithmic running time.

Algorithm 1 Nearest timestamp search algorithm

```
1:  function SearchNearestTime(t*, t, i_start, i_end)
2:      if |t| = 1 then return 0
3:      s ← i_end − i_start + 1                          ▷ Input size
4:      if s = 2 then
5:          j_start ← IndexMap(i_start)                  ▷ Actual indices
6:          j_end ← IndexMap(i_end)
7:          τ1 ← |t* − t_{j_start}|
8:          τ2 ← |t* − t_{j_end}|
9:          if τ1 ≤ τ2 then return j_start
10:         return j_end
11:     i_m ← i_start + s/2
12:     if t* = t_{i_m} then return IndexMap(i_m)
13:     if t_{i_m} < t* then
14:         return SearchNearestTime(t*, t, i_m, i_end)
15:     else
16:         return SearchNearestTime(t*, t, i_start, i_m)
```

`ThreadsafeQueue` is a class enclosing standard C++ queue object with the associated mutex and condition variable. An instance of this class blocks on invocation of `pop` if the queue is empty; the data producer, calling `push` notifies the blocked function when the new data is ready [14].

`EventObject` provides functionality for event notification between two threads. A thread performing repeating actions in a loop checks whether an event has occurred (`hasOccured` method). If so, an alternative action is taking place, e.g. termination of the loop. The notifying thread calls the `notify` method, and supplies it with the current

timestamp. An example of usage of `EventObject` is presented below:

```
while (true) {
  // ... Primary logic of the loop
  if (ready.hasOccured()) {
    break;
  }
}
```

`BlockingWait` is intended for waiting for an event while keeping the current thread blocked. It has the same underlying principle as `ThreadsafeQueue`, but encloses only a boolean variable. An instance of `BlockingWait` keeps the thread blocked until the notifying thread calls the `notify` method. A function that relies on `BlockingWait` is implemented in the following manner:

```
void func() {
  // ... Initialization
  // ... Primary logic
  bw.wait();
  // ... Shutdown
}
```

VI. Experimental results

To validate the correctness and study the timing properties on an FxIS-based application, an experiment is conducted. An FxIS application connected to two Prosilica GC1350 cameras is initialized, and the acquisition process (with writing to two `ImageStreams`) is started, concurrently running in separate threads. The main application thread then initiates image requests at random time instants, where the sleep interval between the instants is uniformly distributed as follows:

$$T_{sleep} \sim \tau_{sleep} + Uniform[-\tau_{mf}, \tau_{mf}] \quad (3)$$

where τ_{sleep} is the constant sleep interval, and τ_{mf} is the maximal time fluctuation in either positive or negative direction that defines the stochastic process.

The details of the conducted experiment are provided in table I.

TABLE I

Parameters of the experiment

Parameter	Value
Basic sleep interval, τ_{sleep}	180 ms
Maximal duration fluctuation, τ_{mf}	50 ms
Size of `ImageStream`, n	20
Number of image requests n_{req},	100

The image requests are repeated for n_{req} times, with each request $i \in \{0, n_{req} - 1\}$ being associated with such measurements as request timestamp $t^{*(i)}$, target timestamps $(t_1^{(i)}, t_2^{(i)})$, and receive timestamps $(t_{r_1}^{(i)}, t_{r_2}^{(i)})$.

As described in Section V, when images are requested with regards to timestamp $t^{*(i)}$, FxIS performs search in the respective `ImageStreams`, and chooses the image

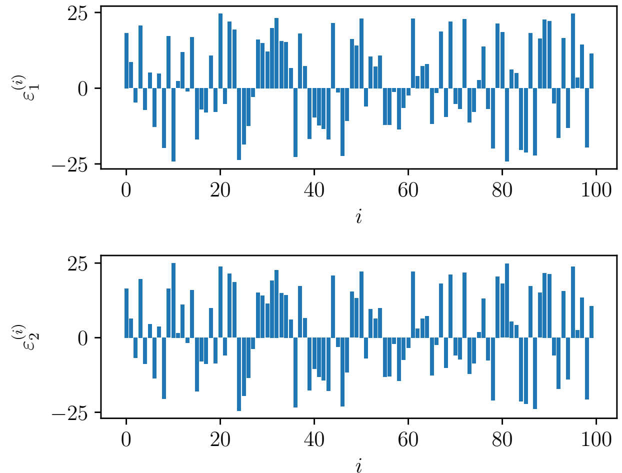

Fig. 5. Measurements of ε_1 and ε_2, ms

with the closest acquisition timestamp. For the two-camera configuration, the latter would correspond to the tuple of target timestamps $(t_1^{(i)}, t_2^{(i)})$. Difference between the request timestamp and each of the target timestamps accounts for the acquisition accuracies $\varepsilon_j^{(i)}$, $j = 1, 2$:

$$\varepsilon_j^{(i)} = t^{*(i)} - t_j^{(i)} \quad (4)$$

As shown on Figure 5, the measurements of $\varepsilon_j^{(i)}$ mirror the uniform nature of the experiment, and range between 0 and $\tau_{ia}/2 = 25$ ms.

When dealing with multiple cameras, it is of interest to acquire a collection of images from each of these cameras that most closely correspond to the same time instant. From the data obtained in this experiment, for each image request i and corresponding tuple of target times $(t_1^{(i)}, t_2^{(i)})$, the absolute difference between the latter constitutes the measure of missynchronization $\varepsilon_b^{(i)}$:

$$\varepsilon_b^{(i)} = |t_1^{(i)} - t_2^{(i)}| \quad (5)$$

Measurements of ε_b are shown on Figure 6, and one can notice that, although in most of the cases the missynchronization is rather low, two spikes occur, where the value of ε_b is close to τ_{ia}.

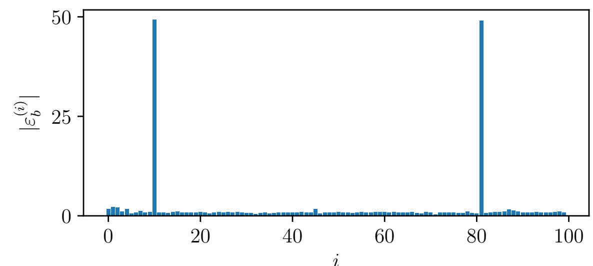

Fig. 6. Measurements of ε_b, ms

By further examining the acquisition timestamps for both cameras in the neighborhood of the request time when one of the spikes occur (Figure 7), one can see that the reason for such behavior is that the absolute difference between the request time t^* and either of two subsequent candidates for the target time $(t_{\text{before}}, t_{\text{after}})$ is close to $\tau_{ia}/2$.

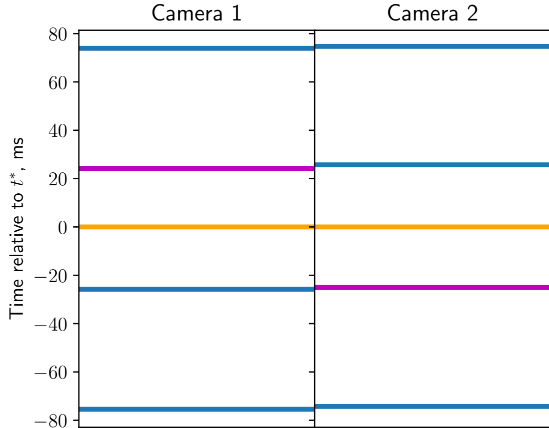

Fig. 7. Visualization of timestamps during a situation of high discrepancy between two camera's target times (purple)

Such situations with a spike in ε_b occur due to independence of the requests to each `ImageStream`. To tackle such problem and achieve more consistent behavior, FxIS architecture should be supplemented by an additional layer of logic responsible for comparing the candidates for target times from all the connected cameras.

While the metrics capturing discrepancy between time points (ε_b, ε_1, ε_2) describe the *accuracy properties* of the system, it is of major importance to also assess *timing properties* of system response. After the image request, the main thread has to wait certain period until data from each `ImageStream` becomes available. The main cause of waiting is the built-in blocking until the next frame is acquired. This can be seen in Figure 8, where the waiting times vary uniformly between 0 and $\tau_{ia} = 50$ ms.

To minimize the waiting time and still ensure the accuracy of data response, the future implementation of `ImageStream` can be supplemented with a-priory knowledge on the frame rate r of the used camera. In that case, instead of waiting for the next image every time, it can only be done when the elapsed time since the previous frame arrival is greater than $\tau_{ia}/2$.

The previously discussed metrics are further summarized in Table II. It also includes statistics ε_b', which are computed as follows. Let T be the set of all target times recorded in the experiment:

$$T = \{(t_1^{(i)}, t_2^{(i)}) \ \forall i \in \{0, n_{req} - 1\}\} \quad (6)$$

If one excludes from T those target times that result in spikes of ε_b, set T' is obtained:

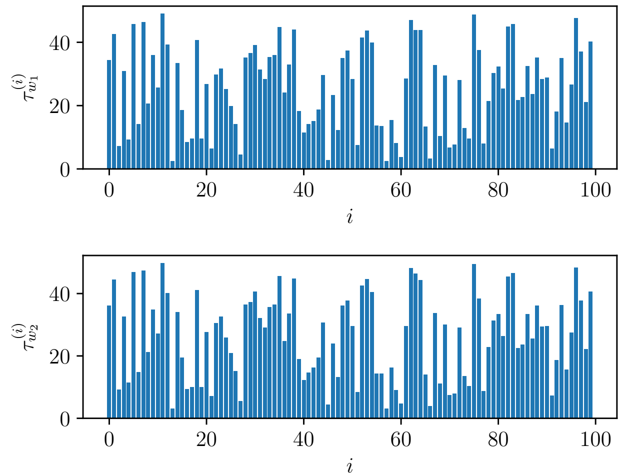

Fig. 8. Measurements of τ_{w_1} and τ_{w_2}, ms

$$T' = \{(t_1^{(i)}, t_2^{(i)}) \ \forall i \in \{0, n_{req} - 1\} \text{ s.t. } \varepsilon_b^{(i)} < g(T)\} \quad (7)$$

where $g(T)$ is a function computing some statistic on T. In the computation used in this paper, $g(T) = mean(T)$. The previous formulations allows for estimating statistics of ε_b', i.e. missynchronization between two cameras when no spikes occur.

TABLE II

STATISTICS OF TIME MEASUREMENTS, MS

Metric	min	mean	max	std
ε_b	0.36	1.85	49.27	6.79
ε_b'	0.36	0.89	2.18	0.28
$\lvert\varepsilon_1\rvert$	1.03	13.35	24.66	6.89
$\lvert\varepsilon_1\rvert$	1.51	13.33	25.06	6.78
τ_{w_1}	2.49	25.56	48.98	13.47
τ_{w_2}	3.13	26.48	49.72	13.48

VII. CONCLUSION AND FURTHER WORK

This paper presented FxIS, a flexible image acquisition framework, designed for use as a part of distributed robotic systems. The core idea of the FxIS design is composition of various functional components with a set of concurrent data structures: `ImageStream`, `ThreadsafeQueue`, `BlockingWait`, and `EventObject`. Used in conjunction with OpenCV, an FxIS application is able to keep track of the recently acquired frames and respond with correctly synchronized images. The presented system is easily scalable to a greater number of connected cameras, where the necessary synchronization is done on reads from the concurrent `ImageStream` data structures.

The future development and evaluation of FxIS shall be done in the following directions: (1) extension of the frame observer with custom-defined image processing, (3) implementation of the additional synchronization layer to tackle possibilities for spikes in ε_b, (3) incorporation of

the tracking functionality over a temporal image stream, (4) evaluation of FxIS in networked environment, and (5) integration of other camera platforms with the presented framework. A brief discussion of the above aspects follows below.

An FxIS application described in this paper was not associated with a built-in image processing routine. Each acquired image was only stored in the `ImageStream` container. In the future iterations of FxIS, an instance of `Streaming` shall be extended with the ability to accept an application-specific image processing routine, so that on each frame arrival, the image would undergo processing locally, before being stored in `ImageStream`. Such an arrangement shall be evaluated with respect to how system performance is affected by processing algorithms of varying complexity.

The most basic application of a `Streaming` with custom processing is image enhancement or other pre-processing, with the resulting image being stored instead of the original one. In more complex scenarios, the incoming image can be supplied to a vision algorithm performing, for example, segmentation or feature detection/description. Because computer vision algorithms are intrinsically computationally heavy, it is important not to discredit the system performance. One shall balance out the camera frame rate and the availability of true concurrency on the machine running FxIS.

Many vision algorithms, such as tracking, make an explicit use of image stream, by aggregating a sequence of temporal features, extracted from each subsequent image. A distinctive characteristic of such a system is preservation of historical data. The vision algorithm parameterization of a `Streaming` object shall support different modes of querying in respect to the stored data.

Image processing constitutes a highly computationally-intensive task, and it is a common knowledge that one should prefer C/C++, an even FPGA-based solutions, to implement the particular algorithms. However, it is a common practice to prototype computer vision solutions in Python while relying to OpenCV bindings and Cython-based implementations such as routines from Scikit-image [15]. In context of FxIS it is of interest to develop a Python interface, which would allow to get images from `ImageStream` as NumPy arrays and process them as a part of the available Python workflow.

The described implementation has been evaluated in local environment. However, since the primary purpose of FxIS is to be deployed as a part of a distributed system, the future work shall include a detailed performance evaluation with communication over the network. As a part of this work, several communication scenarios should be considered, in particular event-based communication over a message queue, and communication with travel time estimation for more accurate response. In context of robotic system, it is particular interest to integrate FxIS with the Robot Operating System (ROS). Another candidate for communication middleware is ZeroMQ, which supports a variety of messaging patterns, including the event-based publish/subscribe.

In relation to attachment of image processing routines to `Streaming` objects and aggregating temporal data associated with the recent images in the stream, it is of interest to develop and compare different modes of decomposition of local and remote processing, particularly those running continuously.

The current implementation of FxIS supports only Allied Vision GigE Vision cameras via the Vimba driver. However, the overall idea of the framework is to support a variety of cameras. The next step of FxIS development is to support the Raspberry Pi camera via the Video4Linux standard and the `raspicam` library.

Acknowledgments

This paper was written in association with the MultiMat project and SFI Manufacturing, funded by the Norwegian Research Council.

References

[1] J. Phillips, "Choosing the right video interface for military vision systems," in *Proceedings Volume 9481, Image Sensing Technologies: Materials, Devices, Systems, and Applications II*, N. K. Dhar and A. K. Dutta, Eds., may 2015.

[2] C. Steger, M. Ulrich, and C. Wiedemann, *Machine Vision Algorithms and Applications*, ser. Wiley-VCH Textbook. Wiley-VCH, 2007.

[3] A. Theorin, K. Bengtsson, J. Provost, M. Lieder, C. Johnsson, T. Lundholm, and B. Lennartson, "An event-driven manufacturing information system architecture for industry 4.0," *International Journal of Production Research*, vol. 48, no. 3, pp. 1–15, jul 2016.

[4] Allied Vision, "Vimba, the sdk for allied vision cameras," 2017, accessed: 2017-10-13. [Online]. Available: https://www.alliedvision.com/en/products/software.html

[5] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.

[6] K. Taouil, T. Jellad, and Z. Chtourou, "Enhanced packet loss recovery for real time pc-based gige vision avi systems," *International Journal of Communication Networks and Distributed Systems*, vol. 14, no. 4, p. 433, 2015.

[7] E. Norouznezhad, A. Bigdeli, A. Postula, and B. C. Lovell, "A high resolution smart camera with gige vision extension for surveillance applications," *2008 2nd ACM/IEEE International Conference on Distributed Smart Cameras, ICDSC 2008*, 2008.

[8] O. W. Ibraheem, A. Irwansyah, J. Hagemeyer, M. Porrmann, and U. Rueckert, "A resource-efficient multi-camera gige vision ip core for embedded vision processing platforms," *2015 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2015*, 2016.

[9] L. Ye, K. Yao, J. Hang, P. Tu, and Y. Cui, "A hardware solution for real-time image acquisition systems based on gige camera," *Journal of Real-Time Image Processing*, vol. 12, no. 4, pp. 827–834, 2016.

[10] R. Cucchiara, A. Prati, and R. Vezzani, "A multi-camera vision system for fall detection and alarm generation," *Expert Systems*, vol. 24, no. 5, pp. 334–345, nov 2007.

[11] Ç. Dikici and H. Işıl Bozma, "Attention-based video streaming," *Signal Processing: Image Communication*, vol. 25, no. 10, pp. 745–760, nov 2010.

[12] M. Goebl and G. Färber, "A real-time-capable hard-and software architecture for joint image and knowledge processing in cognitive automobiles," *Intelligent Vehicles Symposium, 2007 IEEE*, pp. 734–740, 2007.

[13] Allied Vision, "Vimba c++ manual 1.6," Report, 2017.

[14] A. Williams, *C++ Concurrency in Action: Practical Multithreading*. Manning, 2012.

[15] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu, "scikit-image: image processing in python." *PeerJ*, vol. 2, p. e453, 2014.