REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

Evaluation of a YARP-Based Architectural Framework for Robotic Vision Applications

Stefán Freyr Stefánsson, Björn Þór Jónsson, Kristinn R. Thórisson

RUTR-CS08004 — November 2008

Reykjavík University - School of Computer Science

Technical Report

ISSN 1670-5777

REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

Evaluation of a YARP-Based Architectural Framework for Robotic Vision Applications

Stefán Freyr Stefánsson*, Björn Þór Jónsson*, Kristinn R. Thórisson*

Technical Report RUTR-CS08004, November 2008

Abstract: The complexity of robot vision architectures calls for an architectural framework with great flexibility with regards to sensory, hardware, processing, and communications requirements. We are currently working towards a system that will use time-of-flight and a regular video stream for mobile robot vision applications. We present an architectural framework based on YARP, and evaluate its efficiency. Overall, we have found YARP to be easy to use, and our experiments show that the overhead is a reasonable tradeoff for the convenience.

Keywords: YARP, computer vision, architecture, performance evaluation.

(Útdráttur: næsta síða)

* Reykjavík University, Kringlan 1, IS-103 Reykjavík, Iceland. stefan/bjorn/thorisson@ru.is

HÁSKÓLINN Í REYKJAVÍK
REYKJAVÍK UNIVERSITY

Mat á hentugleika YARP fyrir högun tölvusjónarkerfis

Stefán Freyr Stefánsson, Björn Þór Jónsson, Kristinn R. Thórisson

Tækniskýrsla RUTR-CS08004, Nóvember 2008

Útdráttur: Hátt flækjustig högunar tölvusjónarkerfa kallar á högunarramma sem er mjög sveigjanlegur gagnvart kröfum um skynjara, vélbúnað, úrvinnslu og samskipti. Við erum að þróa tölvusjónarkerfi sem mun nota dýptarmyndavélar auk vefmyndavéla til að útfæra tölvusjón. Við lýsum högun sem byggð er á YARP og metum afköst hennar. Við teljum YARP þægilegt í notkun og tilraunir okkar sýna að yfirbygging YARP kerfisins er ásættanleg miðað við þægindin við notkun þess.

Lykilorð: YARP, tölvusjón, högun, afkastamælingar.

(Abstract: previous page)

Contents

1 Introduction

One of the most important sensory mechanisms for mobile robots is a sense of vision that robustly supports movement and manipulations in a three-dimensional world. Here, we use the term "vision" broadly to encompass any visuospatial sensory inputs and processing required for an understanding of the environment. Accumulated experience has shown, however, that for such robotic vision it is necessary to employ a number of sensors and processing mechanisms, integrated in various ways—often dynamically—to support realtime action in various contexts. We are developing such a vision system, which will eventually employ a number of techniques, including (a) color video cameras, which provide shape and color information but do not easily give any depth information, (b) time-of-flight cameras, which can yield sufficient depth information to create a depth map of the environment, (c) image descriptions, such as edge maps and SIFT descriptors [Low04], which can be used for object recognition and obstacle detection, and (d) a communications infrastructure [TLPD07, TFF+05, TLPD05] which allows basic real-time processing on the robot itself while more advanced processing may take place on a dedicated off-board cluster.

In order to study the use and interactions of all these components it is clearly necessary to use an architectural framework which supports flexible manipulation of such compound, multimodal data, on diverse hardware platforms. Such a framework must allow for easy runtime configuration of the processing pipeline, while incurring limited overhead. Low-level options, such as shared memory and/or remote procedure calls, are not flexible enough, as they must be augmented with mechanisms for handling variable latency, priorities or other necessary features of complex architectures and soft-realtime response generation. What is needed is a higher-level framework that supports free selection of communication methods, including shared memory and TCP/IP, depending on the data and architectural constraints at any point in time.

Several frameworks exist which partially address our needs, but very few address all of them. One of these frameworks is YARP (Yet Another Robot Platform), which is a set of libraries to cleanly decouple devices from software architecture [MFN06, FMN08]. It is an attempt to provide a foundation that makes robot software more stable and long-lasting, while allowing for frequent changes of sensors, actuators, processors and networks. As the authors themselves say: "YARP is written by and for researchers in robotics, particularly humanoid robotics, who find themselves with a complicated pile of hardware to control with an equally complicated pile of software" [MFN06, p. 1].

In this paper we report on an effort to evaluate the use of YARP for architectures that capture and manipulate data from a standard color video stream. We explore how well the platform supports our basic needs, such as for sequential processing in a pipeline architecture, how easy the platform is to use and how well it performs.

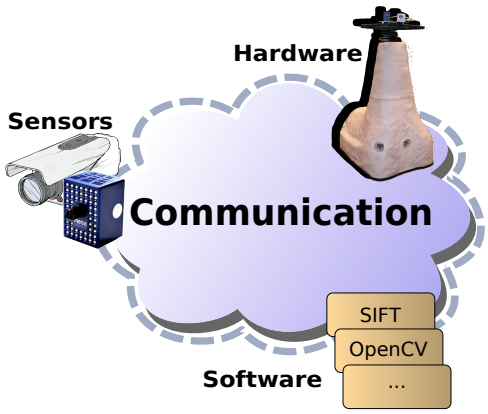

Figure 1: Architectural Requirements

2 Architectural Requirements

Any computer vision architecture has a set of requirements that must be satisfied. The main requirements for our project are shared with many other mobile robot projects and put a strong emphasis on multimodal integration. The requirements are illustrated in Figure 1, which depicts four major categories of requirements: sensory aparati, other hardware, software and communication. We now describe each of these.

2.1 Sensory Requirements

As Figure 1 shows, a robot sensory and vision system must handle input from a variety of sensors, such as:

- Color video cameras, which capture 2-D image streams.
- Depth cameras, such as time-of-flight cameras. Their output can be used to build a map of the area, be combined with data from other range sensors such as sonars, and/or be combined with the color images to yield color+depth information.
- Proximity sensors can augment a standard computer vision system, for example through collision detection, as depth cameras typically have an upper and lower limit on their range.
- Position sensors for head motion provide information about the direction that cameras and other sensors point in; the robot's head has cameras and a directional microphone, which, when combined with depth information, could be used to determine the source of environmental sounds, e.g. human speech.

Furthermore, advanced processing methods can be used to build "super sensors", such as head-motion sensors, through a combination of image analysis, face detection, outline detection, and other means. This is discussed further in the section on processing requirements.

Finally, most sensors require or allow some parameter settings and it is imperative that the infrastructure allows easy runtime access to those.

2.2 Hardware Requirements

Mobile robots typically have at least one on-board computer. The processing power of on-board computers, however, tends to be less than the average desktop machine for reasons of energy efficiency. Such low-power, slow computers can nonetheless be used for important real-time analysis such as collision detection and other critical tasks. Performing more significant processing on a mobile robot thus requires off-robot processing, e.g. on computing clusters, necessitating network communication with the robot. Ideally the developer should not need to be much concerned with whether processes are located locally or remotely; changing the configuration should be as seamless as possible. In a similar manner, the sensor/processing architecture must gracefully handle access to sensors that necessitate dedicated hardware and/or software.

2.3 Processing Requirements

We divide processing requirements roughly into three categories, based on complexity:

- Routine signal processing and pre-processing tasks. Good examples are standard routines included in OpenCV and other image processing libraries, including image resizing, sharpening, blurring, brightness adjustment, and so on.
- Ad-hoc image processing tasks, including multimodal fusion such as the merger of information from a multitude of sensors.
- Advanced image description tasks such as database searches, semantic analysis/tagging and the like.

An example of an ad-hoc image processing task is the segmentation of a color image based on distance information. As the depth camera can not be located at exactly the same spot as the color image camera, and is quite unlikely to have exactly the same focal length, the two images will always differ in some respects, especially for close objects. Determining how to project the distance information onto the image is a typical complex and ad-hoc image processing task [LK07].

As an example of an advanced image description task, the SIFT image description scheme supports well the recognition of objects [Low04]. In this scheme, each image is described through a set of high-dimensional vectors of numbers; an object is recognized through the matching of SIFT descriptors from a current image to descriptors from pre-described images in a reference collection. The architectural framework should be able to handle the

conversion of the image to such an image description, communicate with a search engine, and deliver lists of matching objects. Another example is edge detection; known edge configurations may be stored in a collection, and edges from the current image can be used to query that collection.

2.4 Communication Requirements

The communication infrastructure is a key component of any vision or multimodal sensory system. To satisfy the requirements above, the communication infrastructure must:

- Transparently allow any hardware to work together, by abstracting the communication protocol from the processing tasks. It must at least support shared memory and TCP/IP transport (other protocols may be useful as well).
- Provide support for processing tasks of any complexity, e.g. by allowing processes to communicate any data structures between themselves, whether locally or remotely over networks. For example, it must be possible to augment video streams with additional information, and it must be easy to publish such information even though it may be represented by non-standard datatypes, such as image descriptor streams or object lists. In many cases, control and tagging messages must also be transmitted.
- It must be easy to use for the developer. The chosen infrastructure should enable the developer to focus on the sensory/perception system exclusively and not draw away their attention to communication issues.
- It must have reasonable overhead. Although some overhead is acceptable—and indeed unavoidable as a trade-off for all the above benefits—any significant overhead will eventually lead to the abandonment of the communication infrastructure in favor of hard-coded, specialized solutions.

3 Communication Infrastructures

There are several potential candidates that can be chosen as underlying communication infrastructure for video data, including YARP [MFN06, FMN08], OpenAIR [TLPD07], CAVIAR [LBF+05], Psyclone [TLPD05] and others. In the remainder of this section we first give a short description of YARP, and our reasons for evaluating it, and then briefly describe some of the alternatives.

3.1 YARP

YARP (Yet Another Robot Platform) is a set of libraries to decouple devices, processing, and communication. YARP provides loose coupling between sensors, processors, and actuators, thus supporting incremental architecture evolution. The processes implemented on top of

YARP often lie relatively close to hardware devices; YARP does therefore not "take control" of the infrastructure but rather provides a set of simple abstractions for creating data paths.

A key concept in YARP is that of a communications "port". Processes can have zero, one or more input ports, and produce output on zero, one or more output ports. Ports are also not restricted to a single producer or receiver—many producers can feed a single port, and many receivers can read from a single port. To keep track of ports, YARP requires a special registry server running on the network. The data communicated over the ports may consist of arbitrary data structures, as long as the producer and receiver agree on the format. YARP provides some facility to translate common datatypes between hardware architectures and such translation can be easily implemented in user defined datatypes as well. Each port may be communicated via a host of transport mechanisms, including shared memory, TCP/IP and network multicasting. YARP is thus a fairly flexible communication protocol that leaves the programmer in control.

Our main reason for for evaluating YARP is the fact that it is unobtrusive and basic. Other reasons include the following:

- YARP abstracts the transport mechanism from the software components, allowing any software component to run on any machine. It supports shared memory for local communication, and TCP/IP, UDP, and multicast for communication over a network.
- YARP interacts well with C/C++ code, which is required for our use of the SR-3000 time-of-flight camera. YARP can be used with several other languages too.
- YARP can communicate any data structure as long as both receiver and sender agree on the format. Furthermore, it provides good built-in support for various image processing tasks and the OpenCV library.
- It is open-source software. As we wish to make our framework freely available, the communication infrastructure must also be freely available (indeed, we have already sent in a few patches for YARP).
- Finally, although this was by no means obvious from any documentation, the support given by YARP developers has been very responsive and useful.

These requirements are undoubtedly also met by alternative frameworks and libraries; we have not yet made any formal attempt to compare YARP to these other potential approaches. As described in Section 4, the overall experience of using YARP has been good. With a host of tradeoffs the choice of low-level or mid-level middleware/libraries can be quite complex. We leave it for future work to compare YARP in more detail to the approaches described next.

3.2 Alternative Architectures

OpenAIR is "a routing and communication protocol based on a publish-subscribe architecture" [TLPD07]. It is intended to help AI researchers develop large architectures and share

code more effectively. Unlike YARP, it is based around a blackboard information exchange and optimized for publish-subscribe scenarios. It has thoroughly defined message semantics and has been used in several projects, including agent-based simulations [TLPD05] and robotics [NTHLT+07]. OpenAIR has been implemented for C++, Java and C#.

CAVIAR [TFF+05] is a system based on one global controller and a number of modules for information processing, especially geared for computer vision, providing mechanisms for self-describing module parameters, inputs and outputs, going well beyond the standard services provided by YARP and OpenAIR. The implementation contains a base module with common functionalities (interface to controller and parameter management).

Psyclone (see www.cmlabs.com) is an AI "operating system" that incorporates the OpenAIR specification. It is quite a bit higher-level than both OpenAIR and YARP and provides a number of services for distributed process management and development. Psyclone was compared to CAVIAR by List et al. [LBF+05] as a platform for computer vision. Like CAVIAR, Psyclone has mechanisms for self-describing semantics of modules and message passing. Unlike CAVIAR, however, Psyclone does not need to pre-compute the dataflow beforehand but rather manages it dynamically at runtime, optimizing based on priorities of messages and modules. Both CAVIAR and Psyclone are overkill for the relatively basic architecture we intend to accomplish at present, at least in the short term, but it is possible that with greater expansion and more architectural complexity, platforms such as Psyclone would become relevant, perhaps even necessary.

Compared to, e.g., CAVIAR and Psyclone, YARP looks like a fairly standard library—neither does it do its own message scheduling nor does it provide heavy-handed semantics for message definitions or networking. That may be its very strength.

4 Current Status

We have constructed a preliminary vision system using YARP as the communication infrastructure. We now describe the sensors, hardware, and image processing components implemented in our current setup, and discuss our experience with this setup and configuration of the system.

4.1 Sensors

In our setup, we use two cameras; a regular off the shelf web camera (Unibrain Fire-i) and the Swiss Ranger 3000 depth camera from Mesa Imaging AG.

The SR-3000 camera provides both intensity and depth information. Depth information is obtained using a phase-measuring time-of-flight principle. The camera allows adjustments of various parameters, such as integration time and amplitude threshold, which makes it suitable for a variety of applications and environments. The camera produces 176x144 pixel images with 16-bit depth resolution.

The Unibrain Fire-i RGB camera is an IEEE 1394 (FireWire) web camera with a Sony progressive scan color CCD. It can provide 640x480 pixel video resolution at 30 frames per second, although in our setup we retrieve a 320x240 pixel video stream.

While YARP does come with a device driver for FireWire cameras based on the dc1394 library, we found this driver to be unsuitable for our needs as it was only capable of producing a fixed size monochrome video stream. We therefore decided to implement a new YARP device driver for IIDC capable FireWire cameras which is able to produce color images and dynamically handle different resolutions of the stream. We plan on making this driver available and, if accepted, a part of the YARP distribution.

We also developed a small utility application for publishing the SR-3000 camera video streams on a YARP port. The code for this utility will also be made publicly available if permission is granted from the manufacturer of the SR-3000 as it depends on their proprietary software.

4.2 Hardware

Our experimental setup runs on a 2.6 GHz Pentium 4 Dell OptiPlex GX270 computer with 1.2 Gb RAM. It is equipped with an NVIDIA GeForce 6600 GT 3D accelerated graphics card. No processing is done on the GPU in our case, although YARP does provide modules and libraries for that purpose.

4.3 Image Processing Components

While YARP handles module communication, we use OpenCV for most image and signal processing. Additionally, however, we have already implemented YARP modules to combine the depth and color video streams, create IFT descriptors and perform database searches. Our future plans include research on whether depth information may be used to guide image descriptor generation, for more efficient database searches.

4.4 Experience

Overall, we have found YARP to be satisfactory and easy to use. Installing and learning to use YARP took about one man-week, while most of our time was spent on creating hardware drivers and working with the cameras. In the following section, we describe experiments to study the overhead and performance of YARP on our hardware.

5 Experimental Evaluation

In this section, we report on an initial performance study of the YARP transport mechanisms. In this study we focus on single processor configurations. At present, the goal is not to study the scalability of the system, but rather to compare some configuration choices of YARP for vision.

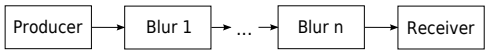

Figure 2: The basic pipeline setup.

To that end, we set up a basic processing pipeline, shown in Figure 2. The pipeline consists of 1) a producer, which produces 320x240 pixel image frames at a given frame rate; 2) a number of blur operators, which run the "simple" OpenCV blur algorithm over the frames; and 3) a receiver, which receives the frames. We change the processing pipeline length, or the number of blur operators, to study the effects of overloading the computer.

Each frame is augmented by sequence numbers and time stamps by each of these components, which are used to measure dropped frames and latency, respectively. Other metrics collected include the frame rate observed by the receiver (lower frame rate occurs when frames are dropped) and CPU load.

5.1 Experiment 1: Transport Mechanisms

In this experiment, the frame rate of the producer was set to 50 frames per second, which is similar to a high-quality video stream. The length of the processing pipeline was varied from one to five consequtive blur operators. We ran measurements using shared memory, local TCP/IP and network multicast connections, with the expectation that shared memory should be fastest. For each configuration, the experiment was run until the receiver had received 50,000 frames.

Figure 3 shows the frame rate observed by the receiver. The x-axis shows the length of the processing pipeline. Overall, two effects are visible in the figure. First, using local TCP/IP and shared memory maintains a frame rate of 50 frames per second, until the pipeline consists of four or more blur processes. At that point, the processor is overloaded and frames are dropped as a result, leading to lower frame rates observed by the receiver. Shared memory performs slightly better due to lower communication overhead.[1]

Second, turning to the performance of multicast, Figure 3 shows that the processing pipeline achieves a much lower frame rate, ranging from 25 to 8 frames per second. The reason for the lower frame rate is clearly visible in Figure 4, which shows the number of frames that are dropped for each configuration. As Figure 4 shows, even with only one blur operator, every other frame is dropped with the multicast transport mechanism. The frame rate observed by the receiver is therefore only half the frame rate of the producer. As more blur operators are added, more frames are dropped, explaining the lower frame rates seen in Figure 3.

Turning to latency, Figure 5 shows that, as expected, latency of the multicast transport mechanism is very high and constantly increasing with pipeline length as frames can be

[1] Our early experiments demonstrated a problem in the shared memory transport implementation, which has subsequently been fixed by the YARP developers.

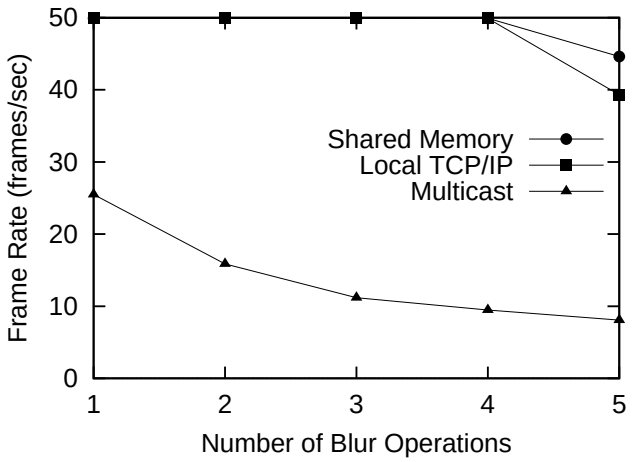

Figure 3: Exp. 1: Frame rate at receiver.

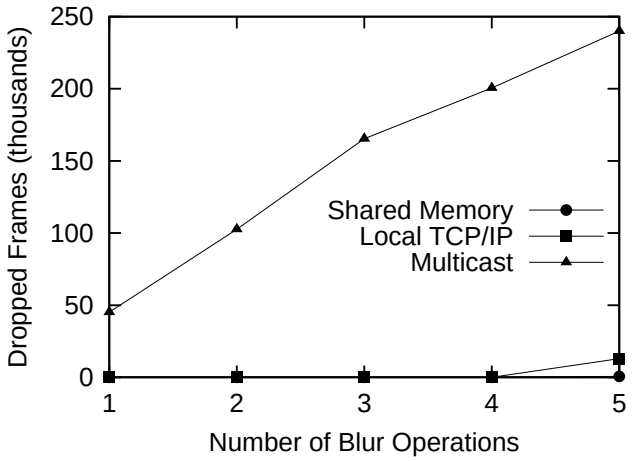

Figure 4: Exp. 1: Frame drops in pipeline.

dropped anywhere in the pipeline. For the other two transport mechanisms, latency is relatively low until the pipeline consists of five blur operators. At that point, the CPU is saturated and scheduling conflicts occur. Again, latency is significantly lower using shared memory than TCP/IP due to the lower communication overhead.

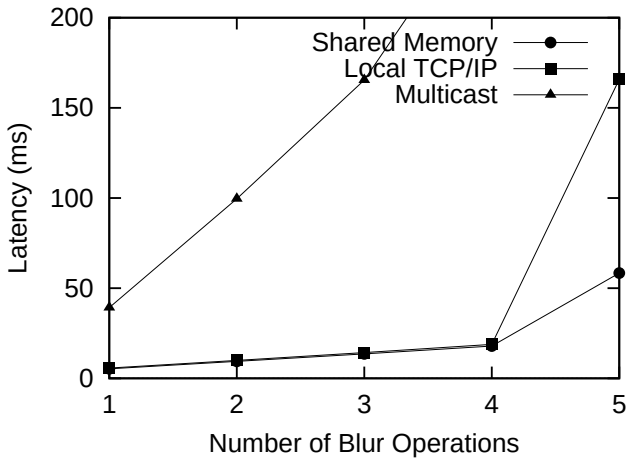

Figure 5: Exp. 1: Latency of received frames.

5.2 Experiment 2: Multiple Pipelines

The previous experiment showed that while the shared memory and TCP/IP transport mechanisms have similar performance, the multicast mechanism performs much worse. Since the pipeline was linear, however, the experiment did not exercise the potential benefit of the multicast mechanism. To achieve this we set up an experiment with multiple parallel processing pipelines each consisting of a chain of a single blur operator and a receiver which logs the same information as in our previous experiment.

A single producer still provides a stream of images at 50 frames per second. This stream then gets published to all the independent pipelines, using one of the three transport mechanisms (we use the shared memory transport mechanism between each blur operator and the corresponding receiver). In this experiment we thus increase the number of pipelines that the producer sends the video stream to as opposed to increasing the number of blur operators within a linear pipeline. Figure 6 shows this setup.

Figure 7 shows the number of frames dropped by each of the transport mechanisms. As before, the x-axis shows the number of blur operators in the configuration, but in contrast to the previous experiment each blur operator is now part of a separate pipeline. The figure shows that while dealing with one or two blur operators, frame drops are virtually non-existent for all of the transport mechanisms. When the third blur operator is added, however, the shared memory and TCP/IP transport mechanisms still have negligible frame drops, while the multicast transport mechanism suddenly starts dropping about half of the frames that are produced. Close examination of the log files revealed that roughly every other frame that is produced gets dropped before it reaches any of the blur operators. The reason is that as before, the cost of the multicast transport mechanism means that full CPU

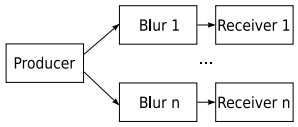

Figure 6: The multiple pipelines setup.

utilization can not be achieved. Since frames are being broadcast they either reach all blur operators or none.

The shared memory and TCP/IP transport mechanisms start experiencing frame drops with four concurrent blur operators and the drop rate increases slightly more than twofold once the fifth blur operator is added. These frame drops are explained by the fact that once four blur operators are started, the CPU is fully utilized and processing each frame takes too long for the operators to be able to keep up with the frame rate of the producer.

Figure 8 shows the latency of frames as the number of blur operators increases. As the figure shows the latency increases steadily for the multicast transport mechanism, while the latency for shared memory and TCP/IP jumps once there are four blur operators. The different behavior is due to the different ways that frames get dropped depending on the transport mechanism used.

With the multicast transport mechanism, every other frame is not being received by the blur operators and so no processing is wasted on them. The blur operators therefore have enough CPU power to keep up with the frames that they receive and the only increase in latency is because the frames are being processed at nearly the same time by all the blur operators. This results in longer blurring times and increased latency due to saturation of the CPU.

The shared memory and TCP/IP transports display the same gradual increase for up to three concurrent blur operators but once the fourth is added a jump in latency is observed. The reason for this jump is that at four blur operators, the CPU is saturated. This means that the blur operators cannot keep up with the frame rate and frames that are sent from the producer do not get picked up instantly and wait until either the blur operator finishes or until the next frame is produced. In the former case the wait results in increased latency while in the latter case the frame that was waiting will get dropped.

5.3 Experiment 3: YARP Overhead

The conclusion that can be drawn from the previous two experiments is that the multicast transport mechanism is not suitable for local processing, and that using shared memory is slightly more efficient than using local TCP/IP. The goal of our final experiment is to measure the overhead of the shared memory transport mechanism, compared to a stand-alone process running the entire pipeline.

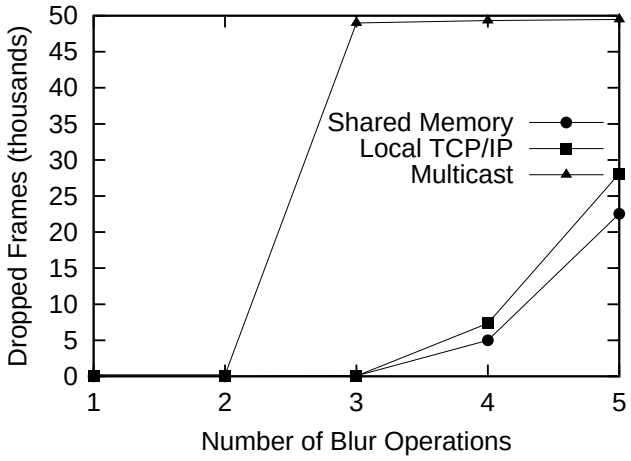

Figure 7: Exp. 2: Average frame drops per receiver.

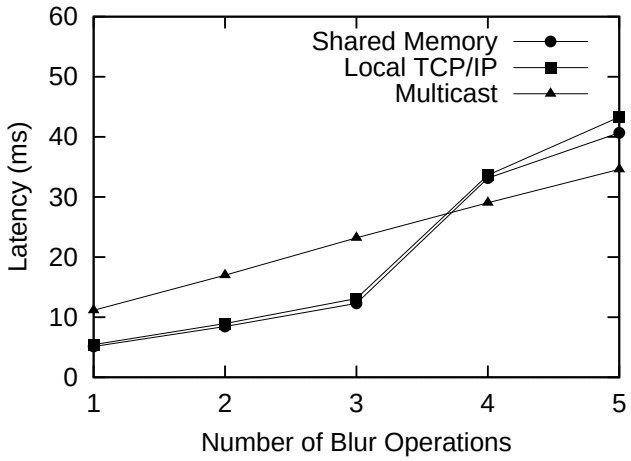

Figure 8: Exp. 2: Average latency of received frames.

With a stand-alone process, there is no inter-process communication, and all the CPU power is spent on the processing pipeline. In order to measure the overhead accurately, we modified the basic processing pipeline of Figure 2 slightly. The producer now produces a single frame and waits until it is received by the receiver (the receiver sends a notification to

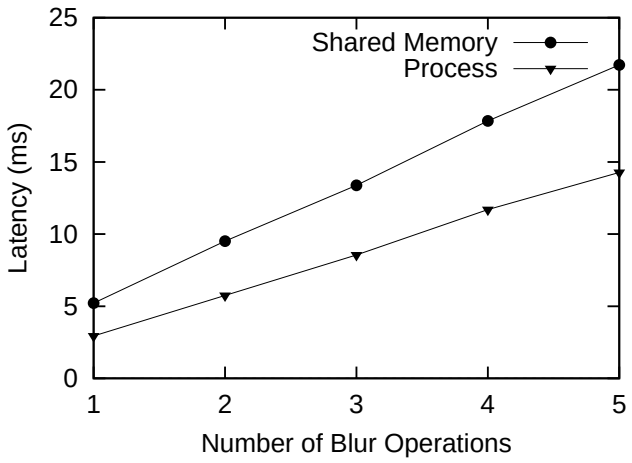

Figure 9: Exp. 3: Latency of YARP and stand-alone process.

the producer). In order to guarantee delivery, we used a synchronization feature of YARP. This experiment therefore gives a strict upper bound on the overhead of YARP.

Due to the simple configuration, the overhead is identical whether measured in terms of CPU cost, latency, or observed frame rate. Figure 9 shows the latency of YARP compared to the stand-alone process. The overhead is most significant for a short processing pipeline, about 77%, but is quickly reduced to 50–60%. The reason for the high overhead for shorter pipelines is that for a pipeline of length n there are $n + 1$ communication ports; as there are more blur operators the effect of the additional port are less pronounced. Based on this experiment, we conjecture that for a complex processing pipeline, we could expect about 50% overhead compared to a well-tuned, handwritten code. We believe that to be a great tradeoff for all the convenience that YARP has to offer.

6 Conclusions

In this paper, we have described our efforts towards a flexible computer vision infrastructure based on the YARP toolkit. YARP greatly simplifies making the infrastructure flexible towards sensors, hardware, processing, and communication requirements, compared to starting from scratch. We have found YARP easy to use, and our experiments show that the overhead is a reasonable tradeoff for the convenience gained.

References

[FMN08] Paul Fitzpatrick, Giorgio Metta, and Lorenzo Natale. Towards long-lived robot genes. *Robot. Auton. Syst.*, 56(1):29–45, 2008.

[LBF+05] T. List, J. Bins, R. B. Fisher, D. Tweed, and K. R. Thórisson. Two approaches to a plug-and-play vision architecture - CAVIAR and Psyclone. In *Workshop on Modular Construction of Human-Like Intelligence*, Pittsburgh, PA, USA, 2005.

[LK07] M. Lindner and A. Kolb. Data-fusion of PMD-based distance-information and high-resolution RGB-images. In *Proc. of the Int. IEEE Symp. on Signals, Circuits & Systems (ISSCS)*, Iasi, Romania, 2007.

[Low04] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. of Computer Vision*, 60(2):91–110, 2004.

[MFN06] G. Metta, P. Fitzpatrick, and L. Natale. YARP: Yet another robot platform. *Int. J. Adv. Robotic Systems*, 3(1):43–48, 2006.

[NTHLT+07] V. Ng-Thow-Hing, T. List, K. R. Thórisson, J. Lim, and J. Wormer. Design and evaluation of communication middleware in a distributed humanoid robot architecture. In *Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware*, San Diego, CA, USA, 2007.

[TFF+05] D. Tweed, W. Fang, R. Fisher, J. Bins, and T. List. Exploring techniques for behaviour recognition via the CAVIAR modular vision framework. In *Proc. HAREM Workshop*, Oxford, England, 2005.

[TLPD05] K. R. Thórisson, T. List, C. Pennock, and J. DiPirro. Whiteboards: Scheduling blackboards for semantic routing of messages & streams. In *Workshop on Modular Construction of Human-Like Intelligence*, Pittsburgh, PA, USA, 2005.

[TLPD07] K. R. Thórisson, T. List, C. Pennock, and J. DiPirro. OpenAIR 1.0 specification. Technical Report RUTR-CS07005, Reykjavik University School of Computer Science, 2007.

REYKJAVÍK UNIVERSITY

HÁSKÓLINN Í REYKJAVÍK

School of Computer Science
Reykjavík University
Kringlan 1, IS-103 Reykjavík, Iceland
Tel: +354 599 6200
Fax: +354 599 6301
http://www.ru.is