# Real-time inter-process communication in heterogeneous programming environments

Adrian Alexandrescu, Andrei Stan, Nicolae-Alexandru Botezatu, Simona Caraiman
Faculty of Automatic Control and Computer Engineering,
"Gheorghe Asachi" Technical University of Iasi
Iasi, Romania
{aalexandrescu, andreis, nbotezatu, sarustei}@tuiasi.ro

***Abstract*—In a complex real-time application, each module can be independently developed, therefore, different processes need to communicate with each other safely and as quickly as possible. This paper proposes a flexible and efficient solution for handling the inter-process communication and for helping the programmer to quickly create modules that use the producer-consumer paradigm in order to streamline the data flow between the different processes. The communication is performed by means of a first-in first-out circular buffer which is kept in a shared memory area in order to allow quick data transfer between modules. The proposed solution allows the development of complex modularized applications suitable for real-time data processing. We describe the use of the proposed framework in a practical setup practice to as part of a software synchronization mechanism between two acquisition devices: a video capture device and an inertial measurement unit**

***Keywords—communication; inter-process; real-time; shared memory; heterogeneous programming environment***

## I. INTRODUCTION

Real-time applications that continuously process large amounts of data are used in different domains like image processing, big data or sensor networks. Such complex applications require that the different modules communicate with each other quickly and safely.

For example, a module can obtain images from one or more cameras, another module can apply different filters on those images, another one can further process those filtered images and extract some relevant information, which, in turn, can be used by another module to identify certain features from the filmed environment.

There can be different developers for the modules and each developer can code using a different programming language, therefore, the whole application runs in a heterogeneous programming environment, which must ensure proper communication between the modules. In a real-time scenario, the main goal is to process the information as quickly as possible without any potential data loss.

The research presented in this paper focuses on presenting an efficient solution for allowing developers to independently create modules that communicate efficiently with each other and which form an application used for real-time data processing. One of the goals is to use this research in practice for synchronizing the data obtained by two processes, each of them acquiring the data in real-time from a different device.

## II. PROBLEM STATEMENT AND SOLUTIONS

Complex real-time applications require an efficient communication between the different components of the application. In this case, the efficiency is defined by how fast and reliable the modules transfer information among them.

The goal is to have an application which is composed of modules created by persons with different programming backgrounds and which allows real-time processing of data in a somewhat pipeline fashion; i.e., a module can receive data from one or more modules, process that data, and then send the processed information to other modules. The application can be seen as a directed graph in which a node represents a module and the edges represent the communication between modules.

There are different approaches to this problem but the most important issue is that each module has to be developed independently from the rest of the application and not necessarily in the same programming language or even on the same type of operating system.

If the application is distributed between different computers, then the communication is performed usually by web services, remoting or by a custom protocol that uses TCP. A significant advantage of having a distributed application is that each module is independent and exposes an API with whom users can access the different functionalities. The drawback of using this approach is that the data needs to be transferred quickly from one module to another, and this is largely dependent on the network speed. In this case there are two possible approaches to the communication issue: one is to have a module send the data directly to the modules that need it, and the other one is to use a central repository or database. In the latter case, a module sends the data to the repository and other modules retrieve it when they can and require it.

This aspect poses another issue: the data producing and the data consuming speed. The considered problem is basically a grouping of multiple producer-consumer problems, which means that a module produces data and another one consumes the data, and, as a result, it produces other data, which in turn is consumed by another module and so on. This is a simple,

pipeline, scenario which has only three modules. Each module has a specific *processing rate*, which is defined as receiving a piece of data, making some computation on it and sending the result. If a module which is earlier in the pipeline has a lower processing rate than a module which is later in the pipeline, then there is no problem in the flow of the application because a piece of data can go through the pipeline without any delay. The real problem arises when a module produces pieces of data faster than another module can process them. This can lead to a bottleneck in the system which can be solved by having pieces of data skipped from processing or by storing the data somewhere. In this latter case, a solution similar to the aforementioned repository or database can be used, but the application is no longer real-time.

A high communication speed can be obtained if the application is a single process and the different modules are execution threads. In this situation, the producer thread writes data in a common buffer and the consumer thread reads from that buffer. The read and write have to be synchronized but the communication speed is significantly improved because there is no data transfer overhead as it is the case when using a protocol like TCP or, even worse, HTTP or SOAP. This approach does not simply solve the processing rate problem but significantly reduces it due to the time gained by using a common buffer to transfer data between modules. Although, the performance improvement is evident, the issue is that the modules must be able to be written in different programming languages. This means that there must be a coordinator module overseeing the data flow between the different threads, and each thread must somehow call functions written in another programming language. This can be achievable but, from the point of view of a module designer, it becomes difficult to properly test the implemented functionalities. The thread must continuously call one or more library (module) functions and make use of the data produced by those functions. Yet another issue is to have the possibility to even call those functions among the different programming languages.

A better solution is to have different processes, written in different languages, communicate with each other. This way each process represents a module and it has the advantage that it can be developed independently. The main drawback is, again, the communication between processes. One possible solution is to have the standard output of one process connected to the standard input of another process. This poses two problems. Firstly, the consumer process needs to know how to interpret each piece of data (e.g., the first four bytes represent the string length, and the rest of the bytes represent a string of characters). Secondly, one process could require to send the same data to multiple processes. For the first problem, the solution is to have serialize/deserialize functions which convert a data structure to a byte array, and inverse. For the second problem, there is no easy solution; in some way the data has to be duplicated and sent to the input of the consumer processes, which is significantly inefficient.

Probably the most flexible and fairly efficient method is a compromise between having multiple threads which communicate via a shared buffer and having multiple processes which communicate by means of the standard input/output, i.e., a solution with multiple processes that communicate via shared memory buffers. The considered approach is described and discussed in the Proposed Solution section of this paper.

Taking into account the aforementioned aspects, the key elements of the considered problem are as follows:

- Multiple modules need to communicate with each other in *real-time*,
- The modules are independently developed, not necessarily in the same programming language,
- Each module has a specific data processing rate,
- Each module can be a producer and/or consumer,
- Each module must be able to easily interpret the data that is being consumed (type safety).

## III. Related Work

The inter-process communication (IPC) is an important component of all modern operating systems and provides the basic mechanisms for data exchange between communicating processes. However, the applications often need a higher level abstraction for a communication channel built on top of the basic services provided by the operating system. This is the main motivation for the developers to build various libraries that hide some of the details of the bare/raw communication infrastructure and provide a user-friendly interface that fits the applications' needs. All the designed libraries use one or more of the following approaches for effective data transfer: sockets, (named) pipes, shared memory or memory mapped files.

The most efficient approach in terms of memory footprint and transfer time uses local memory and kernel level mechanisms [1]. This approach is frequently used for the development of inter-process communication infrastructure in embedded systems which often must satisfy severe real-time requirements [2]. These systems have custom resources that can be used to implement efficient communication libraries [3].

Most of the libraries implement the socket approach. This mechanism is supported by all operating systems and allows the communication to take place between distinct computing systems (e.g., over Ethernet) or between processes on the same computer. Although this is a common and flexible approach, its throughput may be affected by the communication stack overhead (e.g., TCP/IP). Some of the most used and representative libraries are outlined below.

The Apache ActiveMQ [4] is a complex and full featured library written in Java that uses sockets. It enables the development of message oriented systems and provides APIs for working with connectors, message persistence, authentication and authorization. Apache ActiveMQ is fast and supports many cross language clients (e.g., C, C++, Python, etc.) and protocols (e.g., OpenWire, REST, etc.).

Boost.asio [5] is a library written in C++ that uses sockets for communication. Its main feature is asynchronous I/O which means that once an I/O operation is initiated it does not block the initiating process, allowing for concurrent execution of both the process and I/O operation. The library is scalable (i.e., allows multiple connections) and efficient (i.e., minimize data copying).

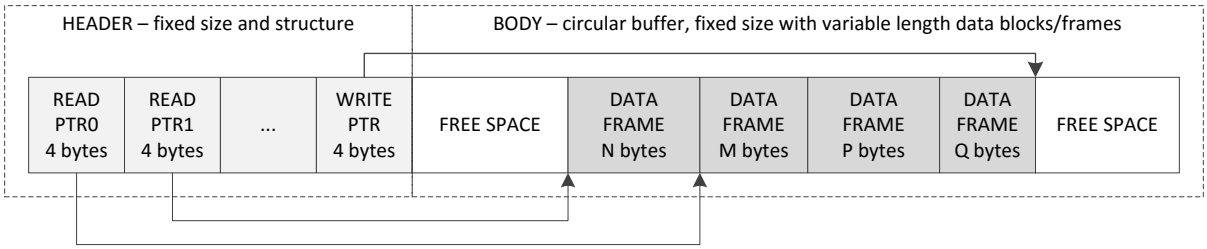


Fig. 1. Memory layout for the FIFO circular buffer used for the inter-process communication.

Lab Streaming Layer (LSL) [6] is a system for the unified collection of measurement time series that uses sockets. It handles both the networking, time-synchronization, (near-) real-time access as well as optionally the centralized collection, viewing and disk recording of the data. The lab streaming layer comes with a built-in synchronized time facility for all recorded data which is designed to achieve sub-millisecond accuracy on a local network of computers.

0MQ (ZeroMQ) [7] is an embeddable networking library which acts like a concurrency framework. It uses sockets that carry atomic messages across various transports like in-process, inter-process, TCP, and multicast. It is fast and uses an asynchronous communication model. The zero component from the name comes from: zero administration, zero cost, zero waste.

D-Bus [8] is a message bus system that uses pipes to communicate between processes located on the same machine. D-Bus communications are based on the exchange of messages between processes. The messages are validated and any ill-formed messages are rejected. D-Bus uses a special daemon process that plays the bus role and to which the rest of the processes connect using any D-Bus point-to-point communications library.

Mmap [9] is a low level system call that maps files directly into local memory pages and allows the sharing of the mapping between multiple processes. This is the basic system functionality that we use to build our custom behavior on top of it. This system call is available on most operating systems with minor differences. The access to the mapping may be performed using file operations (e.g., read, write) or direct memory operations using pointers.

The data transferred between processes must be represented in some known and agreed upon format by communicating parties. The two main methods of information representation are: binary and text (e.g., ASCII). For binary representation, the developer is fully responsible for the definition of the fields, their meaning and encoding, as for text representation there are two main information interchange formats: XML and json. For these formats there are a lot of tools and libraries that can be used to manipulate and serialize data. For binary data representation, there are some tools that can assist the developer in data serialization: binn [10], protocol buffers [11], MessagePack [12]. The proposed approach presented in this paper uses binary representation of the data.

## IV. Proposed Solution

### A. General Overview

The chosen communication method is by means of shared memory. In this approach, a process writes bytes in memory and another process reads that information. The proposed shared memory component is organized as a fixed-size FIFO circular buffer with light synchronization logic. Basically, a producer writes at a location (represented by an index in the circular buffer) and a consumer reads from a previously written location, therefore, removing the need to synchronize the access. The only critical points are associated with the access to the producer/consumer offsets.

A controller component makes connections between the modules, and manages the processes and data flow between them. The controller's configuration file specifies which modules are used, what kind of data they produce/consume, how do they connect with each other and how each buffer is configured.

The main drawback is that the data being stored is an array of bytes. The proposed method adds a type safety layer using a class generator with a serialization/deserialization component. Mainly, the developer creates a json file which describes the object class structure. This file is used by the class generator to create a corresponding class which has serialization and deserialization methods. The class must be used by both the producer and the consumer processes in order to facilitate communication.

### B. Buffer Internal Structure

The used buffer is a memory area reserved and shared between processes. Each process performs read and write operations on that buffer. This area is backed up by a local memory map file (*mmap file*). When the last process finishes the work with the memory file, the contents are persisted on the local disk. The proposed buffer structure allows a single producer to periodically write pieces of data (*frames*) and multiple consumers to read that data from memory. The number of consumers and the available memory size are specified when initializing the buffer. Fig. 1. shows the memory layout for the FIFO circular buffer used for the inter-process communication. The buffer has two parts. The header size is equal to four multiplied by the number of consumers and producers; each entry contains offset values in the body part of the buffer (the offset size is equal to four bytes), and informs the producer about the memory location of the next

frame to be written and the consumer about the memory location from where to read the next unprocessed frame.

Different modules must access the same produced frame, therefore, the buffer allows multiple consumers to read and process the same data. A producer can write a frame at the current location only if all the consumers processed the frame or frames from the area where the write will occur. Frames can have variable sizes, which means that, at some point, a frame can take up an area which was previously occupied by more than one frame. In order to simplify the frame writing logic, if a frame does not fit at the end of the mmap file, then it is written from the beginning of the body section.

This approach allows very fast access to the produced frame without creating any copies. The frame resides in the reserved memory until it is overwritten. In order to permit variable produce/consume rates, the size of the buffer can be specified depending on the average frame size.

If a frame is produced and has to be written in the buffer, but there is no space available, then there are two possibilities depending on how the write is performed: either return a "full buffer" status or wait until there is space available in order to write the produced frame.

Also, the fact that the proposed solution does not allow multiple producers, is not a caveat, but rather an advantage. Because of this, there is no need for extra synchronization logic, which would otherwise reduce the communication speed, and it also offers greater flexibility due to the fact that the multiple producers problem can be solved by having one mmap file for each producer and so each consumer can process the written frames in the order they choose.

### C. Buffer Implementation - C++

The C++ implementation of the circular buffer uses memory map functions exposed by the operating system in order to facilitate quick communication between processes.

A FrameIO component allows reading from or writing to the memory map. The processes that need to communicate by means of the circular buffer, create a FrameIO object and configure it accordingly. There are three configuration modes: create, read and write. The first mode is required because the mmap file could already be created. Usually, the producer creates the file and then creates a FrameIO object configured in write mode. The consumers, in turn, create an object configured in read mode.

The critical sections are only when accessing the indices to the offset values from the header section of the buffer, i.e., the position in the header where the producer locates where the next frame is to be written or where the consumer finds the next frame to be processed. The buffer is considered full if the producer has to write in a location which has not yet been read by all the consumers. The buffer is considered empty to a consumer if that consumer has to read from the location where the producer will write the next object. These checks are performed in a synchronized manner by comparing the index value in the header section corresponding to the producer to the indices in the header section corresponding to each of the consumers.

### D. Buffer Implementation - Python

The Python design for the proposed IPC method uses the object oriented paradigm and implements two components that abstract the low level details of working with the mapped file and the circular buffer management.

The MmapIPCChannel component abstracts the mmap communication channel and provides a basic interface to access the channel. This component is responsible with the mmap file opening, closing, reading and writing byte strings from and into the mapping. Also, it has some utility functions that compute free and used space from the mapping, access individual bytes and words from the mapping in an atomic way by using a system wide semaphore or mutex. These utility functions are necessary for header data handling: in the header of the mapping are stored pointer to locations in the body of the mapping where actual data blocks are stored; writing and reading data blocks into or from the mapping implies header data update.

The most important part of the interface of the MmapIPCChannel component is represented by reading and writing string data functions which are called by clients when they need to communicate data blocks. This functionality allows the transfer of strings of bytes that are written into or read from the mapping. The write function handles the special case of full buffer by issuing an appropriate exception that must be caught by the caller and handled accordingly. It is also possible to implement a blocking behavior in this case: the caller is blocked until the buffer has enough free space to store the requested data. The blocking behavior may be implemented by using a simple approach which employs test in the loop and sleeping. A more efficient approach is by using condition variables and semaphores. Both of these behaviors for blocking on buffer full remain to be evaluated. The read function handles the special case of buffer empty by issuing an exception to the caller. The blocking behavior for this situation may be implemented in the same way as for the write operation.

In order to optimize the memory footprint of an application that uses this method for exchanging data, this component offers the possibility for an application to work directly on the data stored in the mapping. By using this feature, applications avoid the copying of large amounts of data. However, this feature must be used with care because an application may change the data in the mapping, instead of the data in its local copy. This behavior may disturb other processes that may use the same data area which now has other content.

The interface of the MmapIPCChannel has a dedicated functionality to update the pointers for reading. This function must be invoked by a caller after successfully reading or using the data block from the mapping in order to free the space allocated to the just read block.

It is possible for a process to be connected to multiple mappings in order to exchange data with many other processes. So, we designed a second component, named Communicator, with the purpose to hide all the communication details for a process that uses one or more memory mapped files for communicating data. The topology of data transfers between

multiple processes is described in a configuration file which is used at application startup and specifies what mappings are used between what processes. So, a process which communicates with multiple other processes has an array of communication channels as those described above.

### E. Class Generator

The ClassGenerator has the role of adding a type safety component to the communication between the modules. It uses a JSON file that describes one or more classes, which is then used to generate source code in different object-oriented programming languages. The generated classes are used by the module developers when working with the buffer. Each class has generated a serialization and a deserialization method. An example representing a class containing a bi-dimensional integer array and its size is shown in Fig. 2.

```
[{
 "className":"SimpleMatrix",
 "members":[
  { "name":"n", "type":"int", "comment": "" },
  {
   "name":"a", "type":"array", "size":["n*n"],
   "elem":{ "type":"int" }
  }
 ]
}]
```

Fig. 2. Example of a JSON file (used by the Class Generator) which represents a class with a two-dimensional array of int values.

When a module produces a frame, an instance of the generated class is created and initialized and a write frame method is called on the class working with the buffer. That method serializes the instance into a byte array, which gets stored in the buffer. When a consumer reads the frame, an instance of the generated class is created using the byte array read from the buffer. This way the consumer module does not need to know how to interpret the raw byte array.

### F. Controller Implementation - Python

A complex application that has multiple communicating processes may benefit from the presence of a master controller process that manages the creation of mappings and processes at application startup and handles some basic control information at application runtime (i.e., stop and resume commands for individual processes).

We have designed a Python module that performs the following tasks based on the content of a configuration file (i.e., in our case is a json file):

- Creates mappings with specific parameters (e.g., name, size),
- Creates system wide semaphores that may be used to avoid concurrency issues between processes on accessing shared information,
- Starts the execution of some processes, possibly with command line parameters,
- Kills the previously started processes, when controller ends its execution,
- Issues commands to the processes using their standard input and output as a communication channel

### G. Produce/Consume Rate

In an application with multiple inter-connected modules each module can process a frame with a certain rate. Basically, the lowest frame rate at which the system runs without problems is given by the module which performs the processing with the lowest frame rate. Even so, there can be situations in which a frame is sometimes processed slower by one or more modules. This situation is overcome by setting a larger buffer size in order to allow a faster producer to store a few frames until the slower consumer finishes processing its current frame. If the buffer is full when a frame is produced, then waiting for space to be available can lead to a severe slowing down of the system, which is not ideally in a real-time environment. There are two solutions to this problem: one is to allow dropped frames (if the application context permits it), and the other one is to increase the buffer size (if there is enough free memory).

## V. Experimental Results

### A. Experiment with Multiple Processes Written in Different Programming Languages

In order to evaluate the correct operation of the proposed communication mechanism, we have implemented a synthetic testing scenario as presented in Fig. 3.

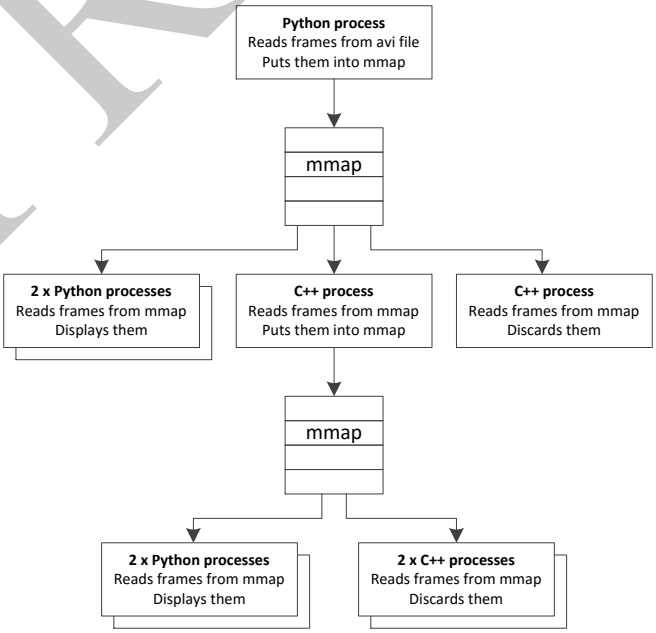


Fig. 3. Example of nine processes which communicate by means of two memory map files.

In this scenario we have used nine heterogeneously developed processes (Python and C++): a process that produces image frames by reading the frames of an AVI file and putting them into a first mapping; four other processes that consume those frames: two of them display the frames; one of these processes and writes the frames to a second channel

which is read by four other processes; again, two of them display the frames. This scenario is described into a configuration file that is used by the controller module outlined above. The controller properly creates mappings and starts all the processes. The application runs correctly without blocking as observed for long periods of time.

The size of the data blocks (i.e., frames) that are stored into mappings are 120 KB large. The write and read times have values lower than 1ms most of the time with sporadic peaks of 2 ms (which are due to OS scheduling). These values are obtained on a common laptop PC with Intel i5 microprocessor, 16GB RAM and SSD drive.

### B. *The Proposed Solution Used in a Real-Time Synchronization Mechanism*

In a more practical sense the proposed method was used as part of a synchronization mechanism between a video capture device (LeopardImaging OV580) and an IMU (i.e. Inertial Measurement Unit) (LPResearch LPMS-B). Both devices were serviced by separate acquisition processes that communicated through the use of two memory mapped files (Fig. 4). From a producer-consumer perspective, the video process was the producer for the first mmap file and generated requests each time a new frame was available. The IMU process consumed the requests and sent back the data associated with each frame through the use of the second mmap file.

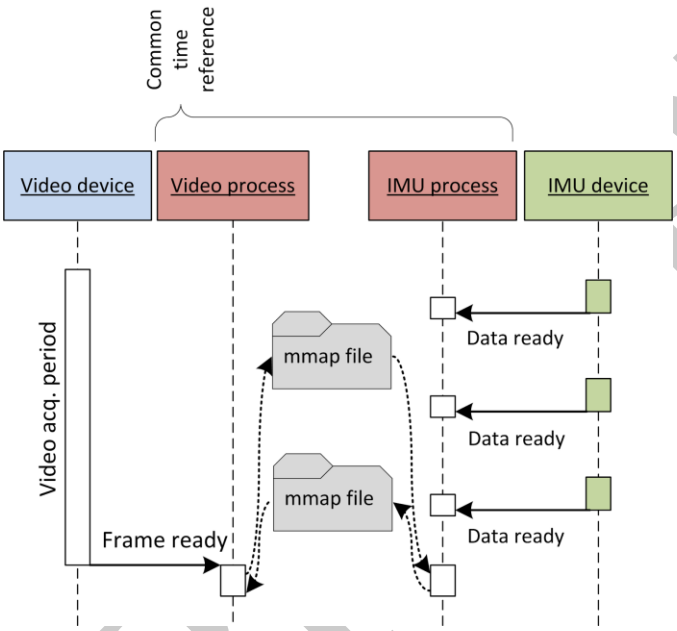


Fig. 4. The proposed solution used in a synchronization mechanism between a video capture device and an Inertial Measurement Unit.

For the requests, the video process produced fixed size frames that contained a timestamp, a sequence number and computed fps for the video capture; for the responses the IMU process produced variable size frames function of the global system time and frame rates of the two acquisition devices.

Before developing this approach based on separate processes, the synchronization mechanism was implemented in a single multithreaded application. Unfortunately, the synchronization could not be achieved due to several reasons that were combined with the unpredictability of scheduling: the size of the image frames (1280 by 480 pixels) that had to be written to disk; high acquisition rate from the IMU device (100 fps); limited input queue size provided by the IMU API. The solution proposed in this paper efficiently managed to solve the synchronization problem by ensuring a fast communication between the two data acquisition processes.

## VI. Conclusion

The proposed solution is an efficient method for handling the inter-module communication given the considered specifications and restrictions. It uses shared memory to facilitate communication between processes written in different programming languages. Moreover, it adds a type safety layer by allowing developers to easily specify the type of data that is being accessed via the memory file. From a performance standpoint, the processes access the information almost as quickly as the situation where there are two threads that communicate with each other.

The main advantage of the proposed solution is its flexibility as shown by the two experiments. In the first one multiple processes written in C++ and Python transfer data between them and in the second, more practical, experiment two processes that acquire real-time data use the proposed solution in order to ensure a software synchronization between the acquired frames.

## Acknowledgment

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 643636 "Sound of Vision".

## References

[1] X. Cheng, and L. Zhang, "A research of inter-process communication based on shared memory and address-mapping", International Conference on Computer Science and Network Technology (ICCSNT), Harbin, 2011, pp. 111 - 114

[2] H. Marzi, L. Hughes, and Y. Lin, "Optimizing interprocess communication for best performance in real-time systems", 24th Canadian Conference on Electrical and Computer Engineering (CCECE), Niagara Falls, 2011, pp. 1383 - 1386

[3] F. Fischer, A. Muth, and G. Farber, "Towards interprocess communication and interface synthesis for a heterogeneous real-time rapid prototyping environment", Proceedings of the Sixth International Workshop on Hardware/Software Codesign, 1998. (CODES/CASHE '98), Seattle, pp. 35 - 39

[4] Apache ActiveMQ, http://activemq.apache.org/

[5] Boost.asio, http://www.boost.org/doc/libs/1_61_0/doc/html/boost_asio.html

[6] Lab Streaming Layer (LSL), https://github.com/sccn/labstreaminglayer

[7] 0MQ (ZeroMQ), http://zeromq.org/

[8] D-Bus, https://www.freedesktop.org/wiki/Software/dbus/

[9] The GNU C Library. Memory-mapped I/O, http://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002fO.html

[10] binn - Binary Serialization, https://github.com/liteserver/binn

[11] Protocol buffers, https://developers.google.com/protocol-buffers

[12] MessagePack, http://msgpack.org/index.html