Master's Degree in Embedded Computing Systems

FORB: A Fast Object Request Broker

Component-Based Software Design course

A.Y. 2017/2018

Gabriele Ara

Contents

1. Introduction

This document is a report for the project required for the course of Component-Based Software Design for Master's Degree in Embedded Computing Systems during A.Y. 2017/2018.

This project aims to develop an Object Request Broker (ORB) framework that will provide C++ developers a fast, efficient and lightweight automated mechanism to perform Remote Procedure Calls (RPCs) between software components, named Fast Object Request Broker (FORB).

Albeit simple, this framework will allow developers to write classes that can be accessed remotely without many restrictions, supporting the exchange of both primitive types, arrays and custom-declared types. In addition, the framework will support basic C++ features like namespaces declaration and methods overloading.

This document will first describe briefly the main features that FORB framework will have, followed by a complete specification of the language used to define interacting components within the framrwork and by the description of its implementation. Finally, a performance evaluation of the given implementation will be performed.

2. Application Context and Requirements

While FORB framework could be used in any scenario in which a simple and lightweight ORB framework is required, its main characteristic will be the enhancement of some application performances with respect to the usage of traditional ORB frameworks. In particular, the target applications of this framework are the ones that may deploy software components and services over shared infrastructures, either as available at a public cloud provider, or within private cloud data centres.

In this context, OS-level virtualization mechanisms (like Linux Containers (LXC), Docker Containers or others) are growing in demand and popularity as deployment and isolation mechanisms, thanks to their increased efficiency in resource usage, when compared with traditional machine virtualization techniques.

The best scenario in which FORB framework will enhance application performances is the one in which multiple communicating components are deployed on different containers (or other OS-level virtualized environments) on the same host. Current solutions adopted by middlewares for communicating among containers rely solely on networking protocols, often leading to excessive overhead, since network stack traversal is required even when source and destination hosts are actually the same physical machine.

To avoid such overheads, the approach of this library will be to exchange data over a shared memory area if the optimal scenario is detected, which will eliminate the need of data marshalling/unmarshalling when serialization/deserialization of exchanged data is performed. If said condition is not met, the library will resort once again to socket communication, thus performance of applications deployed on different hosts will not be affected by the framework implementation.

While right now this acceleration mechanism works only if the two programs that need to communicate share an IPC namespace, it will be further expanded in the future to adapt it to the more general scenario illustrated above.

2.1 Framework Features

This section will illustrate in details all the features that FORB framework will have, in addition to the performance enhancement provided in the scenario illustrated above.

2.1.1 Multithreading

FORB framework will support multithreaded access to remote objects. To achieve low call overhead, however, unnecessary call-multiplexing shall be eliminated, forcing clients to perform at most one call in-flight in each communication channel between two address spaces at any one time. This implies that clients which may want to perform multiple calls simultaneously to the same server will need to open several communication channels, each one being able to perform one call at the time.

Each channel will be served by a dedicated thread on the server side. This arrangement provides maximal concurrency and eliminates any thread switching in either of the address spaces to process a call.

While the library will ensure there won't be any race condition between concurrent calls performed using different channels while exchanging data and preparing the virtual call to the server implementation object, thread safety on the object itself shall be ensured by the user, like any simple object that could be accessed concurrently by multiple threads.

This arrangement has been designed to allow developers the maximum flexibility in their implementation design, with support for custom concurrency management policies, like for example using `std::atomic` variables, `std::mutex` and `std::condition_variable` and others.

2.1.2 No External Dependencies

FORB will be implemented using standard C++11, using whenever possible real C++ mechanisms, like real exceptions for example. Using standard C++ dialect will allow developers to use the framework with ease, without being forced to use a specific non-standard library (e.g. Boost), again to provide them the maximum flexibility in their design choices.

FORB will rely on native thread libraries to provide multithreading capability, in particular using C++ standard `std::thread` objects. However, communication will use POSIX-only features, such as POSIX sockets, POSIX Threads-only features like POSIX Mutexes and Condition Variables, and Linux-only API for shared memory implementations. This limits the framework to be compatible with Linux Operating System only, which however is justified by the context in which FORB is supposed to be adopted.

A small class library will be used to encapsulate the APIs of native C socket and shared memory libraries in C++. It's not required by users to know how to use the library to develop applications using the framework, but its functionalities are exposed in framework's public API, at developers disposal.

2.1.3 Missing Features and Limitations

While for some aspects FORB framework is heavily inspired by CORBA IDL language, its C++ language mapping and the CORBA implementation omniORB, it is not designed to be an implementation of the CORBA standard in any way.

On top of that, FORB framework cannot be used to exchange C++ objects, not even standard ones, such as `std::string` for example. It can be used to exchange only primitive types, arrays and custom-defined structures declared in FORB IDL specification files, along with interfaces that use them, see Chapter 3 for all the details.

3. C++ Language Mapping Specification

The FORB IDL language is the one used to generate C++ source and header files containing all the namespaces, classes, structures and enumerators needed to use the components specified in it within C++ applications. This section will illustrate both the syntax of said language and how it is mapped in C++ code by the FORB compiler.

3.1 Comments and White Spaces

White spaces can surround any symbol in FORB IDL files, tabs or simple spaces can be used without distinction and even vertical spaces can be put at users' discretion.

Comments within FORB IDL files are the same as C/C++ comments, in that they can be one-line comments starting with `//` or multi-line comment environments wrapped between `/* */`.

3.2 Mapping for Scoped Names

Scoped names in FORB IDL are specified by C++ scopes:

- FORB IDL modules are mapped to C++ namespaces.
- FORB IDL interfaces are mapped to C++ classes (as described in Section 3.5).
- FORB IDL structures are mapped to C++ structures (as described in Section 3.7).
- All FORB IDL constructs can be accessed via C++ scoped names.

These mappings allow the corresponding mechanisms in FORB IDL and C++ to be used to build scoped names. For instance:

```
// FORB IDL
module M {
    struct E {
        long L;
    };
};
```

is mapped into:

```cpp
// C++
namespace M {
    struct E {
        int32_t L;
    };
}
```

3.3 FORB Module

The mapping relies on some predefined types, classes, and functions that are defined in a module named `forb`. The module is automatically accessible from any C++ compilation unit that includes a header file generated from a FORB IDL specification.

The module will contain a few classes from which generated *stub* and *skeleton* classes will inherit (see Sections 3.5 and 3.9) and a registry class `forb::remote_registry` that will act as a registry of all objects available for RPC calls.

For simplicity, the first version of the framework will use a JSON configuration file as database to map all known objects with their corresponding network coordinates. The following example shows a JSON configuration file that declares an object implementing the `ns::A` interface that has been deployed on `192.0.2.1:13994`:

```json
{
  "objects": [
    {
      "name": "obj_name",
      "module": "ns",
      "class": "A",
      "ip": "192.0.2.1",
      "port": 13994
    }
  ]
}
```

References to remote objects can be requested to the registry class by using its `get` method, which returns a reference to a generic remote object that can be narrowed to a reference to the a specific interface by using either `_narrow` or `_assign` member functions. For further details about these two functions refer to Section 3.5.3.

The following example opens the JSON database defined above and retrieves a reference to the object identified by `"obj_name"` if possible:

```cpp
// C++
forb::remote_registry registry("db.json");
ns::A_var a_ref = ns::A::_assign(registry.get("obj_name"));
```

To use the functionalities provided by the FORB module, the FORB Shared Library `libforb.so` shall be available for the C++ linker in use.

3.4 Mapping for Modules

As shown in Section 3.2 a module defines a scope, and as such it is mapped to a C++ namespace with the same name:

```
// FORB IDL
module M {
    // definitions
};

// C++
namespace M {
    // definitions
}
```

3.5 Mapping for Interfaces

An interface is mapped to a couple of C++ classes that contain public definition of the operations defined in the interface.

Such classes are the *stub* class and the *skeleton* class. The stub class is the one that shall be used by clients to access remote objects, while the skeleton class is an abstract class (a pure `virtual` class in C++) that shall be extended by implementation classes defined by the user. Such implementation class will contain the code that implements the operations declared by the interface.

Skeleton classes handle all server management and communication operations, leaving to the actual implementation class only the responsibility to define the actual implementation of the declared interface.

Stub classes, despite not being abstract, cannot be declared nor referenced directly within C++ code. Some reference types are defined by the framework to handle them, refer to Section 3.5.1 for further details.

This example shows the definition of an interface:

```
// FORB IDL
interface A {
    // operations
};
```

Differently from CORBA interfaces, for the sake of simplicity FORB IDL interfaces can only define operations and nothing else (no inner types, constants, excepytions, etc.) and they cannot be extended.

3.5.1 Object Reference Types

To use object references to interfaces defined in FORB IDL within C++ code, each generated stub class will provide two reference types, a *shared* and an *unique* reference types. Their names are the name of the interface postfixed with `_ptr` or `_var` respectively and they can be obtained only by using a `forb::remote_registry` object, like shown in Section 3.3.

The following example illustrates their definition as aliases of standard C++ smart pointers:

```
// FORB IDL
interface A {
    // operations
};

// C++
class A; // Stub class name
using A_ptr = std::shared_ptr<A>; // shared reference definition
using A_var = std::unique_ptr<A>; // unique reference definition
```

An operation can be performed on an object by using the arrow operator `->` on a reference object. For example, if an interface defines an operation `op` with no parameters and `obj` is a reference to the interface type, then a call would be written `obj->op()` . This is true on both shared and unique reference types.

Both reference types will release the object reference automatically when the last reference is used, since they are implemented using `std::shared_ptr` and `std::unique_ptr` respectively. Since their implementation relies on such templates, for further details refer to Standard C++ specification available at this page: `https://en.cppreference.com/book/intro/smart_pointers`.

3.5.2 Object Reference Operations

All stub classes are derived from a common base class that is `forb::base_stub` and thus `forb::remote_registry.get` returns references to `forb::base_stub` objects. These references use the special names of `remote_ptr` and `remote_var` for shared and unique references respectively, and they can be later narrowed to the more specific reference type when obtained. An invalid reference to a remote object is equal to `nullptr` , also referred in this document as a `nil` object reference.

3.5.3 Narrowing Object References

The mapping for an interface defines a static member function named `_narrow` that returns a new object reference given an existing reference. The function returns a `nil` object reference if the given reference is `nil` .

The parameter to `_narrow` is a generic remote reference (`remote_ptr`). If the actual (runtime) type of the parameter object can be narrowed to the requested interface's type, then `_narrow` will return a valid object reference; otherwise, `_narrow` will return a `nil` object reference[1].

[1]Notice that since interfaces do not support inheritance in FORB IDL current specification, the `_narrow` function will be used only to narrow references to generic objects to references to specific interfaces.

Table 3.1: Basic Data Type Mappings

FORB IDL	C++
`void`	`void`
`char`	`int8_t`
`int`	`int32_t`
`short`	`int16_t`
`long`	`int32_t`

Another function that has been defined to narrow generic object references is `_assign`, which performs the same narrowing operation on a `forb::remote_var` reference. The returned value of the `_assign` operation is the same as the `_narrow`, but this time its argument is consumed on successful conversion, since the function handles only unique references to remote objects.

3.6 Mapping for Basic Data Types

Basic data types are mapped to fixed dimension C++ primitive types as shown in Table 3.1. Since the framework can be used to communicate between different hosts, which may have different architectures, cross-compatibility could be achieved only by adopting a common convention for the size of all basic data types. At the current version of the framework, no other basic data type is supported.

3.7 Mapping for Structured Types

A FORB IDL structure maps to a corresponding C++ structure, with each FORB IDL structure member mapped to a corresponding member of the C++ structure. Each generated structure will use default default constructor, copy constructor and assignment operators and it will have no custom member functions.

Structures can contain attributes of any type known by the compiler when the attribute declaration is reached (thus any basic type, any already-defined structure type, or even any array of a known type).

For example, the following is the mapping of a structure type:

```
// FORB IDL
struct A {
    short s;
    long l;
    char word[4];
};

// C++
struct A {
    int16_t s;
    int32_t l;
    int8_t word[4];
};
```

3.8 Mapping for Array Types

Arrays are mapped to the corresponding C++ array definition, which means that attributes with a statically-defined size can be defined as structure attributes and arrays can be exchanged between components as parameters for declared operations. Refer Section 3.9 for further details.

3.9 Mapping for Operations

An operation maps to a C++ function with the same name as the operation. Parameters to the operation can be either *input*, *output* or *inout* parameters. Operation overloading is supported, while there is no support to declare `const` operations.

Return types for operations can be any basic or structure type, but not arrays. Any parameter that is not an array is mapped as a parameter passed by value if it is an input parameters, by reference otherwise. Arrays are always passed by reference[2], which means that the content of any array after an operation invocation may change, even if it is an input parameter.

Notice that exchanged arrays can only have a fixed dimension, as specified in the FORB IDL method definition. Actual arguments to remote methods accepting arrays as input will require the given address to point to a memory area at least as large as the specified array size. From a logical point of view, meaningful data for the developer may occupy only part of this area, nonetheless the whole array will be transferred whenever a call request is performed.

Following are a few examples of operation definitions and their mappings:

```
// FORB IDL
interface A {
    int op1();
    int op1(in int a);
    int op2(out my_struct b);
    int op2(in char arr[100]);
};

// C++ stub class
class A : public virtual forb::base_stub {
public:
    int32_t op1();
    int32_t op1(int32_t a);
    int32_t op2(my_struct &b);
    int32_t op2(int8_t arr[100]);
```

[2]Actually they are pointers passed by value, without `const` specifier.

```cpp
    // ...
};

// C++ skeleton class
class A_skeleton : public virtual forb::base_skeleton {
public:
    virtual int32_t op1() = 0;
    virtual int32_t op1(int32_t a) = 0;
    virtual int32_t op2(my_struct &b) = 0;
    virtual int32_t op2(int8_t arr[100]) = 0;

    // ...
};
```

4. Implementation

The implementation of the FORB framework can be found on GitHub using the following url: `https://github.com/gabrieleara/forb`. The `Readme.md` file contains all useful information that allow users to build and install the framework on their platform.

This section will illustrate briefly the content of the project.

4.1 Project Structure

Project structure is shown in Figure 4.1, where some files have been omitted for a better clarity.

Its main components are represented by the FORB IDL compiler, called `forbcc` and the shared library which definitions of classes in `forb` namespace, called `libforb.so`; each is the result of the compilation of a different target of the project; for more information about building and installing the framework refer to `Readme.md` file.

Additionally, the project contains a few other targets used to test library functionality and evaluate its performances.

Targets have been grouped into a few folders, each with a different scope:

`compiler` contains source and header files for the FORB IDL language compiler.

`library` contains source and header files needed to build and install the shared library under a Linux host. Public headers (which will be installed on the host include path) are within `library/include` subfolder.

`profiler` contains a client-server application used to test framework performances when transferring various data sizes, see Chapter 5.

`evaluation` contains the results of a few performance tests performed on the framework, see Section 5.3.

`examples` contains a few examples of FORB IDL files.

docs contains input Doxygen files, used either during CMake build or during automatic documentation generation.

4.2 Documentation

The whole project is documented using Doxygen, which comprehends both compiler and library documentation. Such documentation is generated automatically whenever the project is built in *Release* mode and it is also available at `https://codedocs.xyz/gabrieleara/forb`.

5. Performance Evaluation

A simple client-server application has been developed to evaluate framework performances, in particular in its target scenario, when both client and server components are deployed on the same host.

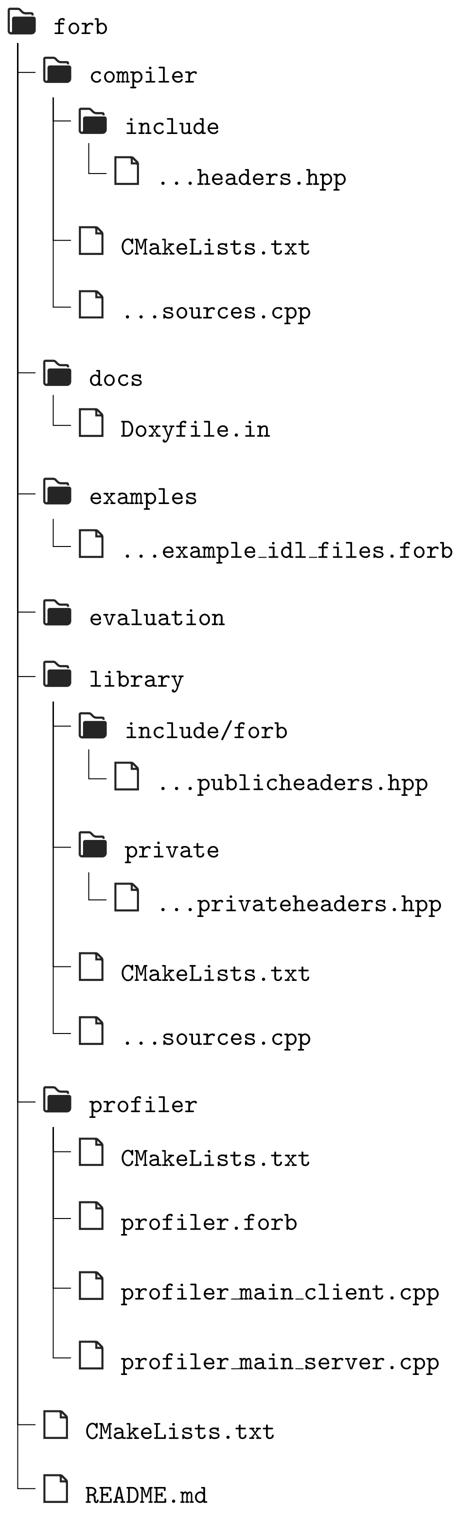
5.1 The Application

The application is composed by a client process which sends various sets of randomly-generated data to the server one, which then proceeds to calculate the checksum of received data and send it back to the client. The time needed to perform the calculation is the same for any communication channel used, hence any improvements that may be observed in round-trip-time will be due to the faster data transfer. Elapsed time is measured using microsecond precision.

Such tests have been performed on data having various sizes, growing from 4 MB up to 512 MB, either disabling/enabling the shared memory optimization performed by the framework on the local machine and varying the size of the shared memory again from 4 MB up to 512 MB. Each data transfer is repeated a certain number of times to build a statistical distribution.

The FORB IDL file used to declare the remote class is the following one:

```
module forb_profiler {
    // Simple conversion table, using the following convention:
    // 1M = 1048576

    // 4    MB = 1048576    words = 1M words
    // 8    MB = 2097152    words
    // 16   MB = 4194304    words
    // 32   MB = 8388608    words
    // 64   MB = 16777216   words
    // 128  MB = 33554432   words
    // 256  MB = 67108864   words
    // 512  MB = 134217728  words

    interface profiler {
```



```
forb
├─ compiler
│  ├─ include
│  │  └─ ...headers.hpp
│  ├─ CMakeLists.txt
│  └─ ...sources.cpp
├─ docs
│  └─ Doxyfile.in
├─ examples
│  └─ ...example_idl_files.forb
├─ evaluation
├─ library
│  ├─ include/forb
│  │  └─ ...publicheaders.hpp
│  ├─ private
│  │  └─ ...privateheaders.hpp
│  ├─ CMakeLists.txt
│  └─ ...sources.cpp
├─ profiler
│  ├─ CMakeLists.txt
│  ├─ profiler.forb
│  ├─ profiler_main_client.cpp
│  └─ profiler_main_server.cpp
├─ CMakeLists.txt
└─ README.md
```

Figure 4.1: Main project folders and files.

Table 5.1: Specifications of the two hosts used for performance evaluation

	Host A (Desktop)	**Host B (Notebook)**
CPU	Intel Core i5-4690 @3.50 GHz	Intel Core i7-2630QM @2.00 GHz
Frequency Governor	Performance	Performance
Clock Frequency	2.00 GHz	1.40 GHz
Hyper Threading	No	Yes (disabled)
Memory	8GB DDR3	6GB DDR3
OS	Ubuntu 18.04.1 (64bit)	Ubuntu 18.04.1 (64bit)

```
        // Return value will be the XOR of all the words given as input
        int method0(in int arg[1048576    ]);
        int method1(in int arg[2097152    ]);
        int method2(in int arg[4194304    ]);
        int method3(in int arg[8388608    ]);
        int method4(in int arg[16777216   ]);
        int method5(in int arg[33554432   ]);
        int method6(in int arg[67108864   ]);
        int method7(in int arg[134217728  ]);
    };
};
```

5.2 Test Environments

Tests have been performed on two different machines, whose specifications are shown in Table 5.1, along with the limitations that were adopted to achieve a better predictability of the tests. On top of that, a few other precautions have been taken:

- Server Process has been set to run always on either cores 0 and 1 with niceness -20.
- Client Process has been set to run always on either core 2 with niceness -20.
- Another application has been run on all cores with niceness +19 to maintain them at the maximum frequency available in the specified range; such application did nothing else than spawn 4 threads that repeat indefinitely the calculation of square roots of randomly generated, see `https://linux.die.net/man/1/stress`.

A few commands that may produce a similar condition to the one adopted during our performance evaluation is the following one, which however may need some adaptation if executed on a different machine:

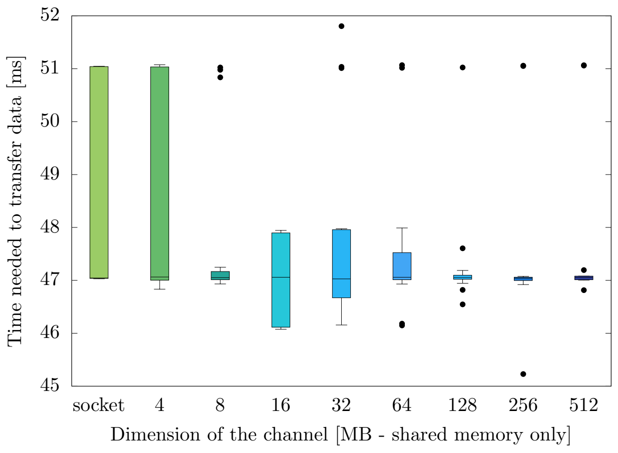
```
# Gain superuser privileges
sudo su

# Sets CPU Frequency Governor as "performance" on each CPU core
```

```
cd /sys/devices/system/cpu
for CPU_FREQ_GOVERNOR in cpu*/cpufreq/scaling_governor;
do [ -f $CPU_FREQ_GOVERNOR ] || continue;
    echo -n performance > $CPU_FREQ_GOVERNOR;
done

# Disable Turbo Boost and set a fixed desired frequency for all CPU cores
# to 50% of the maximum frequency, to avoid thermal throttling
cd /sys/devices/system/cpu/intel_pstate
echo -n 1 > no_turbo
echo -n 50 > max_perf_pct
echo -n 50 > min_perf_pct

# If Hyper Threading is enabled, disable hyper-threaded cores,
# in this example cores 0-3 have as sibling respectively cores 4-7
for i in {4..7}; do
   echo -n 0 > /sys/devices/system/cpu/cpu${i}/online;
done

# Starts a CPU hog program to maintain the clock frequency of each core
# at its maximum in the desired range
nice -n +20 stress --cpu 4 &

# Move to performance application binary folder
cd <performance_build_folder>

# Execute server process
taskset -c 0-1 nice -n -20 ./profiler_server &

# Execute client process
taskset -c 2 nice -n -20 ./profiler_client rpc_db.json
```

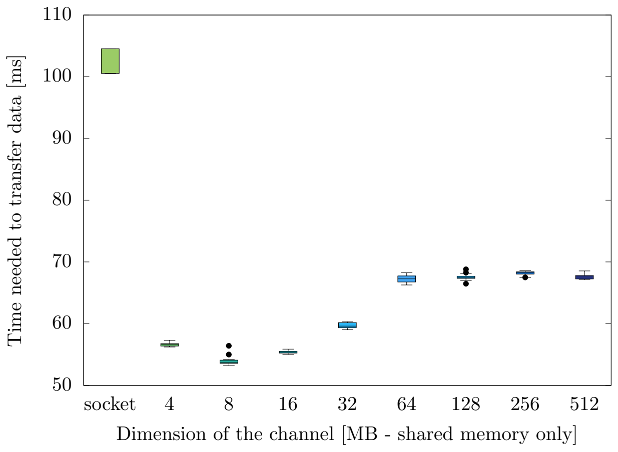
5.3 Evaluation Results

Results of said tests are shown in Figures 5.1 and 5.2, where only a few of the results have been reported for better clarity. For the complete dataset refer to `evaluation` folder. In these plots, the bars illustrate the statistical distribution of test results and dots represent outliers in the dataset.

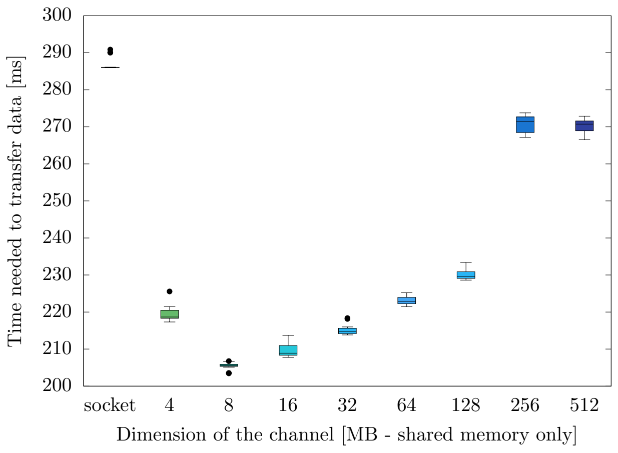
It is clear how smaller amounts of data cannot leverage too much the optimization performed by the framework, achieving results that are more or less equivalent to the ones obtained using plain socket communication. However, as data to be transferred grows in dimension improvements become clear, proving the effectiveness of the optimization performed by the framework.



(a) Time required to transfer 8 MB of data

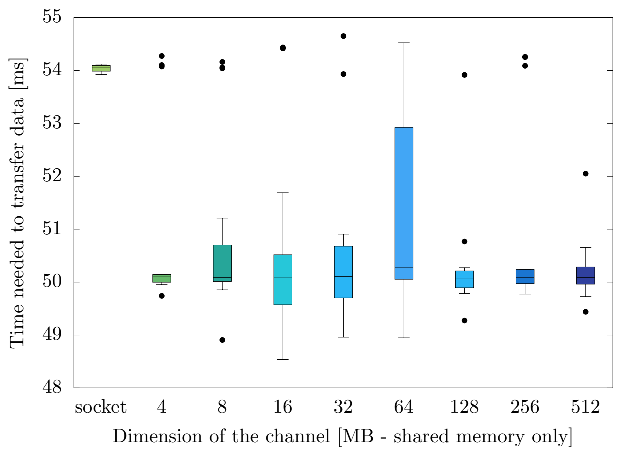


(b) Time required to transfer 64 MB of data

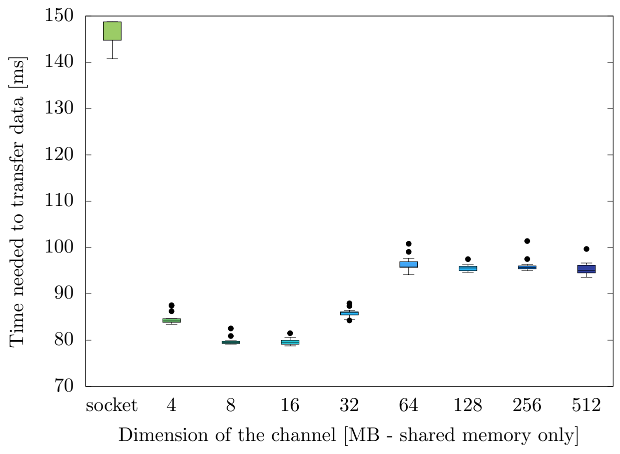


(c) Time required to transfer 256 MB of data

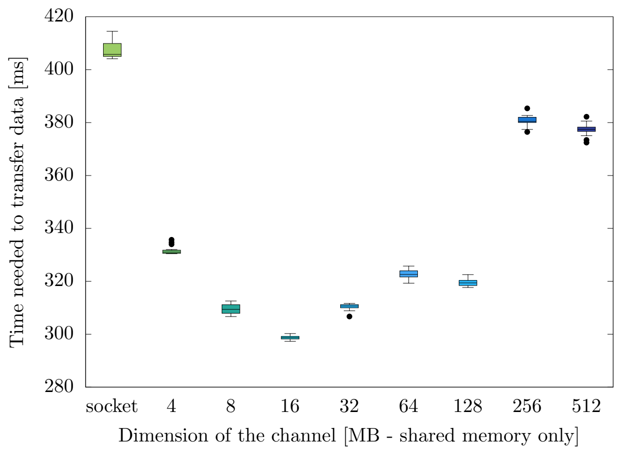
Figure 5.1: Results of performance evaluations on Host A.



(a) Time required to transfer 8 MB of data



(b) Time required to transfer 64 MB of data



(c) Time required to transfer 256 MB of data

Figure 5.2: Results of performance evaluations on Host B.

6. Conclusion

As shown by results of FORB framework performance evaluation, its effectiveness on the performance of applications that adopt it as communication mechanism is influenced by the amount of data that such application is likely to exchange between its components.

This should be no surprise, since while it is true that using the shared memory optimization provided by the framework we can avoid network stack traversal to exchange data, synchronization mechanisms necessary for concurrent access to shared resources such as the communication channel require a certain number of system calls. Therefore their impact on the time needed to move data from a process to another depends strongly on the amount of data that needs to be moved.

An interesting outcome of the performance evaluation is that the dimension of the memory area allocated as shared memory channel does not influence very much exchange time. Contrary to expectations, bigger shared memory areas proved to be less effective than smaller ones with respect to performance improvements. Of course this means that the impact of the framework on the environment in which it is used can be reduced without penalizing application performances, which is a positive result. However, this characteristic may require further investigation in the future.

6.1 Future Works

While FORB framework has been proven effective by performance evaluations when adopted in its peculiar use-case scenario, this doesn't mean that there is no room for further improvements.

At the current state, whenever an application wants to transfer data between components the following operations are required:

- Data needs to be produced/prepared on a private memory area of the producer component.
- The remote method is invoked with a pointer to the private memory area: the framework will copy such data from producer private memory to shared memory; concurrently, it will also copy said data from the shared memory into consumer process private memory. Since this is a concurrent operation it requires synchronization.
- Finally the consumer can operate on data residing in its own private memory.

This means that even using the shared memory optimization provided by the framework data needs to be copied twice (from component A to shared memory and from shared memory to component B) and synchronization mechanisms are necessary to allow concurrent access to the shared channel. The positive note is that the shared memory area does not need to be as large as the data to be exchanged, because it acts only as a concurrent buffer.

One feature that may drastically improve framework performance is the possibility to use *handled buffers* to exchange data. This alternative mechanism consists on the ability to produce/prepare data to be exchanged *directly on a shared memory area.* This would eliminate unnecessary copies and synchronization calls, thus reducing communication time to the minimum. In that scenario, the steps that an application would perform are the following ones:

- The producer requests an *handled buffer* from the framework, which will be a memory area allocated in a shared space.
- It will then produce/prepare data to be exchanged directly on the *handled buffer* area, which can only be accessed by one process at any time, thus it does not require any synchronization mechanism to access.
- When done it will invoke the remote method; the framework will then move ownership of the *handled buffer* to the consumer component by simply sending it a pointer to the buffer location.
- The consumer will then operate on data residing on the *handled buffer* like it would operate on data residing in its own private memory.
- Finally, the consumer will release the *handled buffer* when done.

This mechanism exchanges only a pointer (or any identifier adopted by *handled buffers* API) from a process to another, thus reducing sthe time needed to exchange data to the minimum. The only downside of this mechanism is the fact that a shared memory location able to contain the whole data to be exchanged must be allocated, since there is no more concurrent access to the shared memory channel.