Università di Pisa
Department of Computer Science

Master Degree
Data and Knowledge: Science and Technologies Curriculum

Enabling multi-protocol support in FastFlow distributed runtime

Candidate
Federico Finocchio

Supervisors
Massimo Torquati

A.Y. 2021-2022

Alla mia famiglia.

Abstract

A shift toward distributed computing is being recorded in order to address storage and computational needs of the new generation of data-intensive applications. This new programming environment requires new tools and frameworks, as well as new abstractions, to allow programmers to provide applications that deal with real-time decisions in an accurate and actionable way. Portability, performance, and programmability are provided to the application programmer at different levels of abstraction. However, handling the heterogeneity of a distributed environment remains challenging and requires special effort to correctly and efficiently exploit parallel and distributed resources. The thesis provides an in-depth study of the communication landscape in the distributed environment, proposing a multi-protocol extension to the existing distributed runtime of FastFlow, a C++ structured parallel and distributed programming framework. Our extension targets the runtime system of FastFlow, providing a flexible and extensible way of defining multi-protocol applications. In contrast to existing frameworks, we propose true multi-protocol support, which allows remote nodes to connect using multiple transport protocols (e.g., MPI, TCP, Margo), effectively enabling heterogeneity of compute nodes. A set of tests validate the implemented extension and evaluates its performance against the existing distributed runtime of FastFlow.

Contents

List of Figures

List of Tables

Listings

Introduction

The ubiquitous increase in data production rates and the increasing computational needs of data-intensive applications are imposing, to application and system developers, to shift toward the distributed computing environment. Data generated by scientific simulations, experimental facilities, connected devices, and commercial applications surpass the storage and computational needs of single machines. The distributed programming model provides an effective way of overcoming the limitations of single computing nodes, allowing computations to be scattered across multiple machines. However, developing an application in a distributed environment can be difficult and time-consuming, and it's most of the time a privilege of domain experts. Nevertheless, as the difference between Big Data and High Performance Computing becomes more and more blurry, new frameworks, programming models and tools have surfaced to help the programmer to deal with the complexity of designing parallel and distributed applications [2, 3]. However, dealing with scalable, portable, and efficient parallel and distributed applications is not an easy task. The shift to distributed computing introduces new challenges, like resource management, data distribution, coordination of participating processes and monitoring of the application. Given that each of these challenges acts at a different abstraction layer, application developers can rely on high-level abstraction frameworks to withdraw from the distributed challenges and focus only on the application development. The abstractions provided to the user are many, covering the programmer's needs at every layer of the system. Application-specific systems provide an easy-to-use interface for the application developer, mostly in the Big Data field, providing systems to analyze stream or historical data in a straightforward way, by completely abstracting the underlying challenges of the parallel and distributed domain. Examples are Hadoop MapReduce [4], Spark [5, 6], and Flink [7]. Lower-level abstractions are available, providing more flexibility to the programmer which can define her own application topology using a structured approach. The leading paradigm is the algorithmic skeleton programming and the structured parallel programming [8, 9], implemented by GRPPI [1, 10], SkePU [11, 12], FastFlow [13, 14, 15].

However, the heterogeneity of the distributed environment brings additional challenges. HPC cluster environments often rely on high-performance network infrastructures, like InfiniBand [16], and provide vendor-specific protocols and drivers. Thus, communication frameworks specifically targeted at providing support to vendor-specific protocols have surfaced, such as OFI [17] and UCX [18]. As a result, higher-level communication frameworks can be built over these libraries to support a multitude of network infrastructures by abstracting them from the underlying implementation.

This provides programmability and portability to current and future systems, but it still lacks support for heterogeneity in communications at the execution phase. In fact, the landscape of *true* multi-protocol support at the execution phase is very limited. It is currently restricted to the selection of a specific protocol at startup, effectively cutting off computing nodes not compatible with the selected protocols. Considering a cloud environment, this can represent a limitation. A user might want to enable communications between different computing environments, like IoT nodes generating streams of data at the edge of the network and forwarding these streams to an HPC environment in order to perform computations via hardware accelerators like GPGPUs.

This thesis aims at studying the landscape of communication frameworks in the HPC environment, analyzing the existing solutions for effective multi-protocol support during the execution of a distributed application. In particular, the Mochi framework [19] was of particular interest in the preliminary phase of our thesis. However, after having witnessed very limited support for widely accepted transports like TCP and MPI, we decided to provide our implementation of multi-protocol communication functionalities. We provide a multi-protocol extension to FastFlow, a C++ structured parallel and distributed programming framework, recently extended with distributed-memory support. The thesis introduces novel communication functionalities with a modular approach, allowing extensibility to additional transport protocols with no code changes to existing code. The introduced multi-protocol support enables heterogeneity in the computing nodes, allowing the RTS programmer to extend existing applications, as well as developing new ones, connecting different nodes with multiple transport protocols. The implemented software components provide an automatized way of plugging different protocols inside the existing application without breaking the code and with little effort. The RTS programmer is provided with a set of extensible classes in order to allow the integration of existing transports or even future ones. Furthermore, to follow and maintain the LEGO style *philosophy* of the FastFlow library, also the implemented communication classes can be composed in order to address the needs of the specific application. The communication functionalities presented in this thesis allow each application partition to communicate using the protocol that is most suited for the task at hand or for the architecture-specific limitations.

The thesis is organized as follows:

- Chapter 1: provides a broad vision over the various level of abstraction which allow the application developer to create, deploy and run parallel and distributed applications. The chapter focuses on FastFlow framework, providing a description of its internal layered structure and programming model;

- Chapter 2: describes the state-of-the-art of communication frameworks, focusing on existing solutions for multi-protocol support. The chapter provides the information needed to understand the contribution of our extensions to the communication functionalities of FastFlow framework;

- Chapter 3: presents the existing distributed functionalities of the FastFlow frame-

work and its APIs which allow the application developer to easily create a distributed application. Additionally, the chapter provides description of the goals and design of the provided extension at the RTS level of FastFlow framework;

- Chapter 4: focuses on the presentation of sample applications and testing we performed to evaluate the effectiveness of the implemented communication layer. It additionally provides a case study to show the usefulness of the implemented multi-protocol functionalities.

Chapter 1

Background

Data production rates are growing much faster than processing capabilities offered by single computing nodes. An evolution toward distributed computing is being recorded, allowing to leverage a larger amount of storage and computing power by means of multiple interconnected machines [3]. New abstractions, tools, frameworks, and libraries are being specifically developed to help writing parallel and distributed programs in an efficient and structured fashion. Abstractions help the programmer to avoid dealing with the challenges of orchestrating multiple computing entities both in the parallel and distributed domain. Examples are the algorithmic skeletons [8, 9] and the structured parallel programming models. They provide well-defined components that can be composed in a structured way to express parallelism. The single components can be efficiently optimized for specific architectures, allowing the user to completely abstract from the implementation details in the target architecture, allowing portability, programmability, and scalability. Frameworks following these approaches are GRPPI [1], SkePU [11], Muesli [20], YewPar [21], and FastFlow [13, 22]. Frameworks providing even higher abstractions are available, they offer application-specific models which, however, reduce the degrees of freedom of the programmer since they provide a unique application model. Examples of these frameworks are Spark [6, 5], Hadoop MapReduce [4], and Flink [7]. Additionally, frameworks targeting the runtime system of HPC application exists, such as HPX [23] and Argobots [24], and they specifically provide functionalities to address the management of parallel work units at a fine-grained level. The HPC software stack is composed by multiple layers [3], each one of them providing different levels of abstractions in order to tackle the various parallel and distributed computing challenges, such as thread synchronization, load balancing, communication, and resource management. Essentially, the common rationale is "writing parallel programs that scale with the number of cores and are as easy to write and efficient as sequential programs" [25]. In the following we present existing frameworks and tools targeted at the parallel and distributed environment, providing a view over the different levels of abstractions in order to highlight the state-of-the-art at every level. We distinguish between three main levels of abstraction, which we present in three dedicated sections, describing application-specific engines, structured models for the creation of arbitrary application topologies, and low-level frameworks targeting the application runtime systems at a very fine-grained level.

1.1 Distributed Processing Systems

Most of the frameworks in these fields provide an application-specific approach to deal with both streaming and batch processing tasks. Stream processing applications take a (possibly) infinite stream of input elements and produce (near) real-time results. The operations are applied per single element or to a limited set of elements (mini-batch) of the stream. The (near) real-time processing requires the system to offer high throughput and low latency in its operations. Batch processing, on the other hand, handles historic and bounded sets of data and usually applies transformations to large amounts of data in order to gather statistics or insights. Usually the dimension of these datasets is prohibitive for individual node's storage, so data is distributed across multiple nodes via specialized file systems, like HDFS [26], which allows to move computation where the data is located. We describe below the main frameworks and libraries. The high-level of abstraction provided by these frameworks allow the user to be completely agnostic of the challenges brought by the parallelisation of the underlying application.

1.1.1 Hadoop MapReduce

Hadoop MapReduce [4] is a framework implementing the functionalities of the MapReduce programming model [27]. The MapReduce model targets batch processing and allows to build parallel and distributed applications via a composition of the *kernel* functions *map* and *reduce*. In particular, the *map* operation allows to define a routine to be applied to key-value pairs and produces an intermediate set of key-value pairs; the *reduce* function defines the routine which gathers the intermediate results of the *map* operation and merges values for the same keys according to a specific operation, usually producing a reduced set of elements. The underlying run-time system manages the data partitioning and the scheduling of the work between the workers for both *map* and *reduce* functions, as depicted in Figure 1.1.1. Intermediate results produced by the workers in the *map* phase are written to local disk and subsequently dispatched by the manager toward workers operating the *reduce* phase, which retrieves the data via RPC.

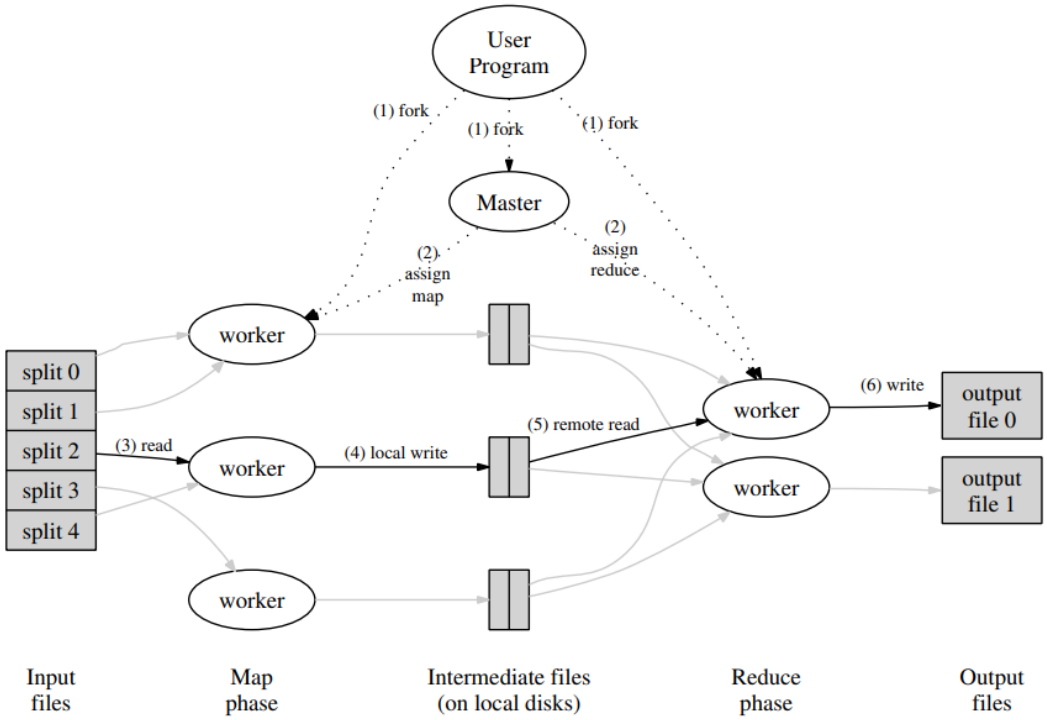


Figure 1.1.1: MapReduce execution overview. *Reprinted from [27].*

1.1.2 Apache Spark

Spark is a unified engine for distributed data processing and machine learning which allows to handle both batch and streaming workloads [5, 6]. The programming model is the same as the well-known MapReduce [27], but Spark introduces data-sharing capabilities and dynamic optimizations by expanding its model with a data structure called Resilient Distributed Datasets (RDDs), which is the main programming abstraction of the Spark framework. The RDD model builds a Directed Acyclic Graph of transformations over data and records the relationship between them, allowing the system to optimize the operations at run-time. The RDDs are *read-only* data structures which are stored in memory and created by applying transformations over data. The RDD model allows Spark to build a fault tolerance framework by keeping the graph of transformations that was used to build the RDDs, and using the graph of operations back over base data in order to reconstruct any lost partition, without having to rely on checkpointing or data replication to provide fault tolerance. Moreover, thanks to the *"in-memory computation"* feature, Spark is faster than the base MapReduce model proposed by the Hadoop framework. Spark can be operated, through the high-level libraries built on top of its main RDD programming model, in both batch and stream processing. Spark architecture is composed of a master node which is connected to a driver program that is in charge of calling the main program of an application, as depicted in Figure 1.1.2. The driver program is further connected to the cluster manager which in turn allocates resources and schedules tasks among the computing nodes, which can run multiple tasks in parallel, according to the nodes architecture.

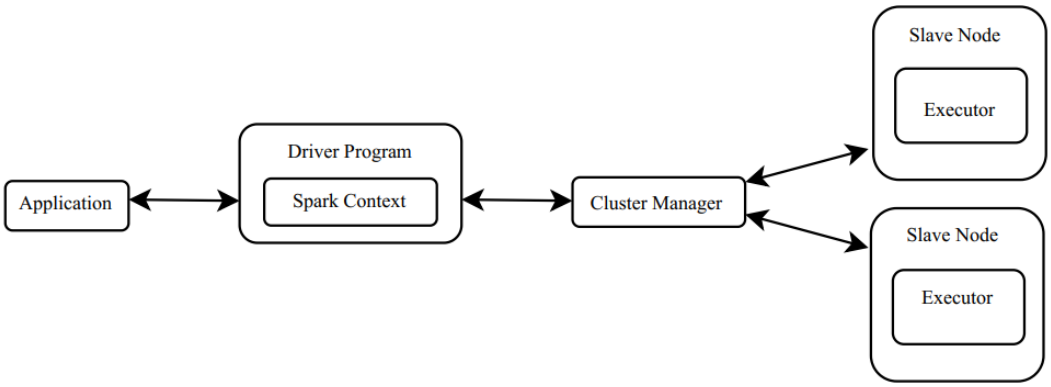


Figure 1.1.2: Apache Spark architecture. *Reprinted from [6].*

1.1.3 Apache Flink

Flink [7] is an open-source framework and distributed processing engine for stateful computations over data streams and data batches, which builds on the philosophy that many classes of data processing applications and iterative algorithms can be expressed as pipelined dataflows computations. It merges the aspects of both batch processing and continuous streams under a unified model based on the data-stream processing paradigm. The paradigm used by Flink considers batch programs as special cases of streaming programs, for this reason Flink offers a specialized API to deal with static data sets, providing dedicated data structures and algorithms to correctly fit batch processing on top of its streaming runtime. Flink provides APIs to build dataflow programs as DAG of stateful operators connected with data streams. The provided DataStream and DataSet APIs tagets, respectively, stream and batch processing. Flink's core runtime dataflow engine runs the programs constructed by the provided API. On top of the core APIs, Flink offers domain specific libraries that allows to build programs for machine learning (FlinkML), graph processing (Gelly) and SQL-like operations (Table). Flink's processing model comprises three processes, as shown in Figure 1.1.3: the client, which transforms the program code into a dataflow graph; the JobManager receives the graph and schedules it to distributed entities, tracking state, progress and coordinating checkpoints and recovery; TaskManager are the actual processors of data, they execute operators which produces streams and they report their status to the JobManager.

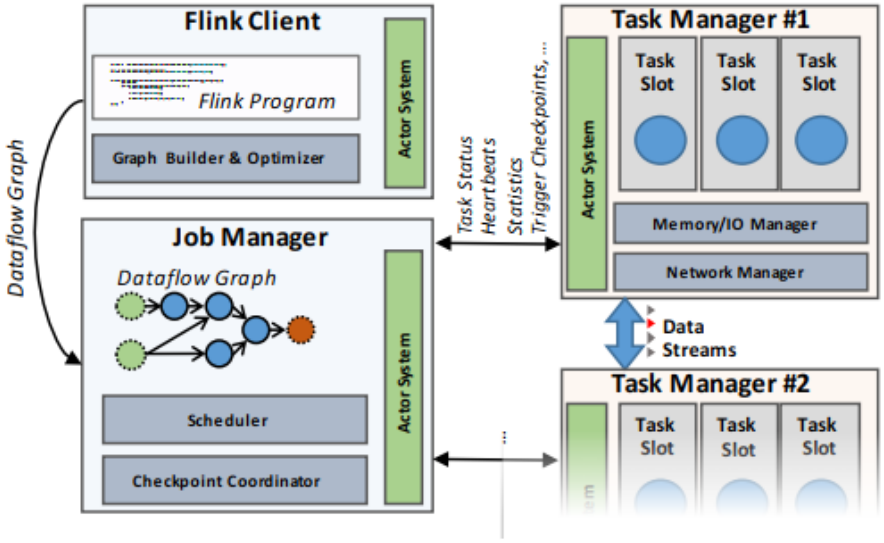


Figure 1.1.3: Flink processing model. *Reprinted from [7].*

1.2 Distributed Algorithmic Skeletons

Algorithmic skeletons provide a structured approach to build parallel applications via the composition of high-level, well-known, and recurrent parallel implementations. This approach, opposed to the application-specific approach of higher-level frameworks, provides an additional degree of freedom to the programmer, which can build her own parallel application following the compositional model provided by the specific framework. Each recurrent pattern encapsulate specific algorithmic features, making them portable and reusable as well as highly optimizable for target architectures. Moreover, algorithmic skeletons framework abstract to the user the management of non-functional aspects of a parallel decomposition like threads, communication and work scheduling. Frameworks implementing an algorithmic skeleton approach, besides offering a lower level of abstraction, enable the construction of arbitrary application topologies by the careful selection of patterns defining the decomposition for the specific parallel problem. Finally, the only aspect left to the programmer is the definition of the business logic code for each patter via a set of function in order to specify the behavior of the overall application graph. We present, in the following, examples of parallel and distributed frameworks implementing the algorithmic skeleton programming model. We particularly focus on the FastFlow framework, providing an in-depth description of its building blocks, serving as an introduction to its programming model that we extended during our thesis work. We present the main characteristics of the framework building blocks in the shared-memory programming model, delaying the description of the distributed functionalities in Chapter 3.

1.2.1 GRPPI

GRPPI is a Generic and Reusable Parallel Pattern Interface for both stream processing and data-intensive C++ applications [1, 10]. GRPPI programming model exposes parallel and distributed programming by leveraging parallel patterns as building blocks, providing composability, portability, and reusability. The library provides patterns for both stream processing, such as the Pipeline, Farm, and Filter, as well as data parallel patterns, such as Map, Reduce, and MapReduce. Recently, the GRPPI backend has been extended with an MPI implementation in order to allow hybrid parallelism for the Pipeline and Farm patterns. The user can select the new MPI execution policy in order to describe distributed computation of parts of her application graph. The structured approach of building parallel applications via the composition of well-defined patterns allows to ease the burden of handling challenges specific of the parallel programming model, such as communication overheads, thread synchronization and data layout. Deeply optimized building blocks, well-defined rules of composability, and higher abstraction in the base mechanisms, allow to build efficient and maintainable applications in an elegant and immediate fashion, as can be seen in the example in Listing 1.2.1. The provided example shows an application which reads integer vectors from a file, find the maximum in each vector and finally prints the value found in the standard output. Each operation is implemented as a different stage of a three-staged `Pipeline` pattern composed with a `Farm` pattern as second stage with 6 threads.

```
Pipeline( parallel_execution_omp,
  // Stage 0: read values from a file
  [&]() -> optional<vector<int>>  {
    auto r = read_list(is);
    return (r.size() == 0) ? {} : r;
  },
  // Stage 1: takes the maximum value of the vector
  Farm( parallel_execution_omp{6},
    [] (vector<int> v) {
      return (v.size() > 0) ?
        max_element(v.begin(), v.end()) :
        numeric_limits<int>::min();
    }),
  // Stage 2: prints out the result
  [&os] (int x) {
    os << x << endl;
  }
);
```

Listing 1.2.1: GRPPI pattern composition example. *Reprinted from [1].*

1.2.2 SkePU

SkePU [11] is a data-parallel skeleton programming framework targeting heterogeneous parallel and distributed architectures, which offers support for multiple backends for CPU and GPU programming. SkePU has recently been redesigned to target the HPC domain. It aims at providing programmability, scalability, and performance to the HPC

domain with a simple and unified interface for data-parallel applications. It leverages the skeleton programming model to offer highly optimized building blocks which can be composed to build scalable and portable parallel applications. The advantage of having standardized components comes from the fact that they can be easily extended to support different architectures by providing platform-specific implementations without rewriting the application code. The third version of the framework, Skepu 3 [12], introduces the possibility to leverage platform-specific SIMD instructions, new scheduling variants for the multicore CPU backend, and a new cluster backend targeting StarPU's MPI interface. Additionally, data containers have been revised to provide automatic optimizations and data sharing between connected devices memory.

1.2.3 Muesli

Muesli [20] is a template library which provides data and task parallel algorithmic skeletons, it targets heterogeneous parallel and distributed architectures, with support to multi-GPU systems. The two kinds of algorithmic skeletons are provided with different approaches: the data parallel skeletons are implemented as member functions of the library's distributed data structures; the task parallel skeletons offer, instead, a way to construct computation topologies, such as *farm*, *pipe*, and *divide and conquer*. Muesli offers high-level, portable and easy to use abstractions which enables the development of parallel and distributed applications in a structured way, allowing to map the same application to a multitude of architectures with no changes from the user point of view. The high-level interface, moreover, hides all the challenges related to synchronization, communication and resource management. Complex application topologies can be defined by the composition of multiple task-parallel skeletons, which behavior depends on a set of user-provided functions. Additionally, the distributed data structures provided by the library allow the partition of data, as well as intra/inter-node parallelism, among computing nodes in a transparent and seamless way.

1.2.4 YewPar

YewPar [21] is a domain-specific, scalable algorithmic skeleton programming framework providing a set of widely applicable algorithmic skeletons for exact combinatorial search on shared-memory and distributed-memory architectures. The framework implements general high-level components which allow, via composition with the provided algorithmic skeletons, the parallelisation of combinatorial search applications. It additionally provides low-level components to allow the creation of new skeletons. YewPar relies over the task-based parallel and distributed runtime offered by the HPX framework [23], extending its functionalities by the implementation of custom HPX schedulers in order to provide work-stealing strategies particularly targeted at handling the search paths in the skeletons it provides. Thanks to the distributed-memory support offered by the HPX runtime system, YewPar provides algorithmic skeleton to HPC and Cloud environments exploiting the portability offered by the HPX framework.

1.2.5 FastFlow

FastFlow [13, 22] is the result of a joint effort of the Parallel Programming Model Group of the University of Pisa and the Parallel Computing Research Group of the University of Turin with the main aim of providing key aspects of parallel programming via a set of abstractions and a specifically designed Run-Time System (RTS). FastFlow is a C++ structured parallel programming framework that leverages a streaming data-flow approach and originally targeted cache-coherent shared-memory architectures [13]. The FastFlow library was built with a layered design by keeping efficiency in the base mechanisms to retain it across the whole framework [13]. Each of the implemented layers provides a set of highly efficient, customizable and composable abstractions in order to help both the application programmer and the RTS developer. The top level abstractions provide well-known and high-level parallel patterns targeting the application developer. The lowest layers, instead, expose abstractions for the RTS programmer, by providing a reduced set of highly-efficient, customizable and composable parallel `Building Blocks` (BBs, from now on), that can be used to efficiently implement most of the existing parallel applications [13, 22]. The philosophy behind FastFlow lies on the idea of providing a well defined set of basic structured parallel components (BBs) and a streaming data-flow programming approach. FastFlow's BBs are efficient and reusable, and they allow the programmer to build applications by following a LEGO-style approach. This programming style gives flexibility to the programmer, which, opposed to a pure algorithmic skeleton approach, offers a lower level abstraction which allows the programmer to build and orchestrate more complex parallel structures. FastFlow was recently extended with a distributed-memory runtime in order to allow developers to build and deploy FastFlow applications over a distributed environment without disrupting the original shared-memory FastFlow streaming graph semantic. By the provided distributed-memory extension, FastFlow aspire to define a *unique programming model* for both shared and distributed-memory systems. We delay description of the existing distributed runtime to Chapter 3, presenting in this section only the basic building blocks provided by the framework. We depict in Figure 1.2.1 the layered structure of FastFlow library.

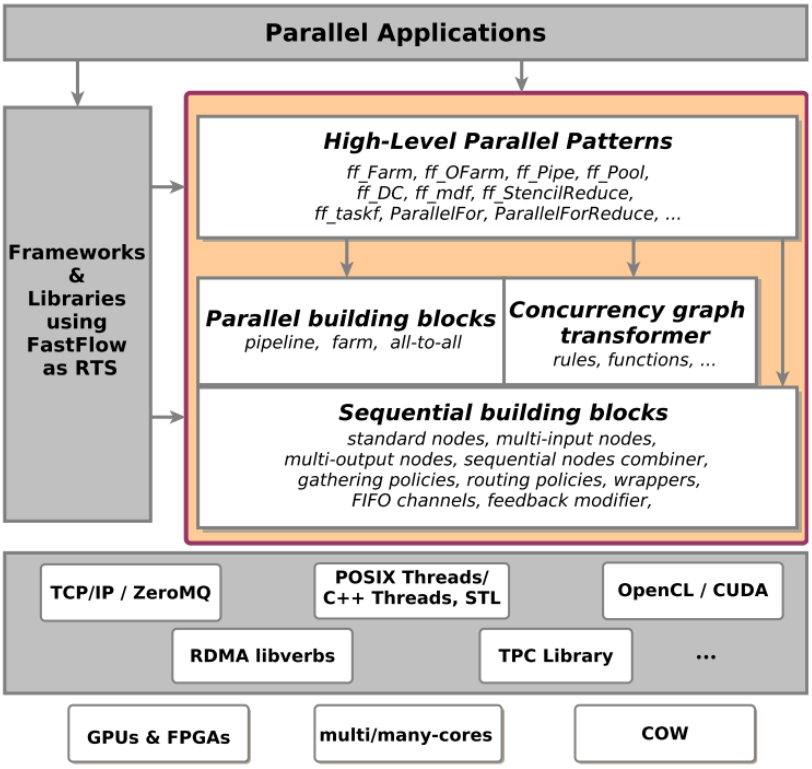


Figure 1.2.1: FastFlow's software layers. *Reprinted from [22].*

Building Blocks

`Building Blocks` (`BBs`) are the core elements of FastFlow programming. They offer, both to the application and RTS programmer, composable and well-structured abstractions of common parallel components that represent fundamental elements of any structured parallel application. They are used as the basic abstraction layer to properly build FastFlow structured streaming parallel patterns. The main characteristic of `BBs` is their composition-oriented nature, which allows the programmer to assemble them in a LEGO-style fashion in order to build complex parallel applications and to model both data and control flows. This reflects the style of the structured parallel programming methodology and enables a more organized way of building parallel applications. The composition of `BBs` forms what's called a *concurrency graph*, whose nodes and edges are, respectively, the FastFlow's concurrent entities and communication channels.

`BBs` come in two flavor: *Sequential Building Blocks* (`SBBs`), represented by the *node* and *node combiner*; *Parallel Building Blocks* (`PBBs`), defined by *pipe*, *farm* and *all-to-all*. Both `SBBs` and `PBBs` rely over Single-Producer Single-Consumer (SPSC) lock-free queues to share heap-allocated memory and allow efficient communication. SPSC channels are also used to implement Single-Producer Multi-Consumer (SPMC), Multi-Producer Single-Consumer (MPSC) and Multi-Producer Multi-Consumer (MPMC) strategies. These interactions are enabled by means of a mediator node that uses the set of SPSC channels to distribute and gather data elements from other nodes.

Figure 1.2.2 shows the communication semantics as described above. All the communications between `BBs` have this structure, with necessary channel replications for 1-to-N/N-to-1/N-to-N communication patterns.

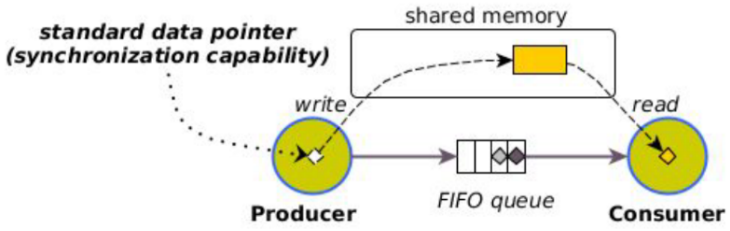


Figure 1.2.2: FastFlow's SPSC FIFO channel. *Reprinted from [22].*

In the following we present the semantic attached to each `BB`:

- **node**: represents a sequential unit of work and can encapsulate both user and RTS code. It can be further categorized as *standard*, *multi-input*, and *multi-output*, depending on the input/output channels connected to the node. The node applies a function (by means of its service method `svc()`) to every element received from the input channel(s), possibly using internal state maintained by the node itself, and finally puts a memory reference to the results into the output channel(s) according to a scheduling policy (either predefined or user-defined).

- **node combiner**: promotes code reuse by allowing the composition of two `SBBs` into one single node. The `svc()` method of the resulting composition computes the function composition of the respective `SBBs` service methods.

- **pipeline**: acts as a container of `BBs` and as a topology builder. It connects `BBs` one after the other, grouping them in a single parallel component and modeling stream data-flow. The pipline `BB` allows to build arbitrarily complex structures by concatenating both `SBBs` and `PBBs`.

- **farm**: allows functional replication of `BBs` coordinated by a broker node called *Emitter*. The standard farm `PBB` is composed of two parallel entities, the *Emitter* (a *multi-output* node) and a pool of `BBs`, the *Workers*. The *Emitter* schedules data items received from its input channel to the *Workers*, by following a given policy. A further extension of the standard farm is provided, where an additional *multi-input* `BB`, the *Collector*, is added with the aim of collecting the results coming from each *Worker*.

- **all-to-all**: defines two sets of computing entities (*R-Workers* and *L-Workers*) connected according to the *shuffle communication pattern*. Each *L-Worker* from the first set is connected to each *R-Worker* in the second set. Optional *feedback channels* can be used to connect *R-Workers* with *L-Workers*.

A simple example computing $\sum_{i=0}^{N} i^2$ is shown in Listing 1.2.2. We defined a `ff_farm BB` with a customized *Emitter* and *Collector* nodes acting, respectively, as

producer and consumer of tasks. Each *Worker* node computes the square of the input element received by the *Emitter*, forwarding to the *Collector* the result. The *Collector* keeps as internal state the partial sum of all the received elements, and prints the final sum upon termination.

```
struct Producer: ff_node_t<float> {
    Generator(const size_t length):length(length) {}
    float* svc(float *) {
        for(size_t i=0; i<=length; i++) ff_send_out(new float(i));
        return EOS; // signals end-of-stream
    }
    const size_t length;
};

struct Worker: ff_node_t<float> {
    float* svc(float* task) {
        float &t = *task;
        t = t*t;
        return task;
    }
};

struct Consumer: ff_node_t<float> {
    float* svc(float * task) { sum += *task; return GO_ON; }
    void svc_end() { std::cout << "Final sum is: " << sum << "\n"; }
    float sum = 0.0;
};

int main(int argc, char *argv[]) {
    ...
    // Define custom farm emitter and collector as simple ff_node_t instances
    Producer  P(stream_len);
    Consumer  C;

    // Create farm Worker replicas, each one of them is a new ff_node_t
    std::vector<ff_node*> W;
    for(size_t i=0; i<nworkers; ++i) W.push_back(new Worker());

    // Explicitly providing Emitter and Collector nodes
    // replaces standard ones
    ff_farm farm(P, W, C);

    // Run farm building block and wait termination
    farm.run_and_wait_end()

    ...
}
```

Listing 1.2.2: Simple example to show node composition and functional replication using the `ff_farm` building block.

All the described `BBs` can be composed to build concurrent streaming graphs of

nodes executing according to the data-flow model. The composition of `BBs` can happen regardless their type, and proceed as follows:

- Two `SBBs` can be connected into a pipeline regardless of their input/output cardinality.
- `PBBs` can be connected to `SBBs` (and viceversa) into a pipeline container by using *multi-input* (*multi-output*) sequential nodes.
- Two `PBBs` can be connected into a pipeline container either if they have the same number of nodes, or through *multi-input/multi-output* sequential nodes if they have different number of nodes at the edges.

In most cases, the above rules are enforced by the RTS, automatically, by transforming the edge nodes of two connecting `BBs` by using node wrappers or adding helper nodes by means of the *node combiner* `BB`.

Programming model

The library's programming model is an adaptation of the Data-Flow model, and it leverages the specialization and composition, in a pipeline fashion, of `BBs` and high-level parallel patterns. Each node is a stateful concurrent object, whose activation depends on the reception of data elements from its input channels. The nodes can have zero or more input channels and zero or more output channels. As soon as a message is received from (one of) its input channel(s), the service method of the node (namely, the `svc()` method of the `ff_node` class) is called by the RTS, which passes as argument a reference to the input element memory location. FastFlow nodes without any input channels (mainly represented by the first stage of a pipeline application) are activated by the RTS upon application startup by sending a `nullptr` argument to the `svc()` method. The termination of a FastFlow application is related to a special message implemented in the FastFlow namespace, that is the `EOS` message (End Of Stream). A node is terminated upon reception of the `EOS` message in **every** input channels of the considered node. Upon termination, the `EOS` message is propagated into all output channels in order to allow the rest of the application to terminate accordingly. In case an application contains cyclic graphs, the termination is more complicated and cannot be managed by the RTS, but must be implemented by means of extending the standard `eosnotify` method of the `ff_node` class.

1.3 Distributed Runtime Systems

To complete our discussion of parallel and distributed abstractions provided by the HPC software stack, we discuss about the lowest layer analyzed in this preliminary study, relative to abstractions at the runtime system layer. Frameworks targeted at the runtime system directly enable parallelism at the lowest level possible, still abstracting from the OS-specific mechanisms to implement those functionalities. The

framework we analyze are specifically focused at providing a very fine-grained management of concurrent units by providing *lightweight* threads. They are used to enable thousands of tasks to work in parallel without incurring in degradation of performance which would be caused by context switches in a traditional threading environment. The provided programming model allows to define, schedule, and manage massive amounts of execution units by means of a small set of OS-level threads, reducing costs related to context switching, synchronization and data sharing. We provide in this category the description of two frameworks, and we particularly focus on the distributed runtime system provided by the Mochi framework [19] since it has been used in the preliminary part of our thesis work to provide communication services, at a low level, to the FastFlow framework.

1.3.1 HPX

HPX [23] is a parallel runtime system which extends the C++11/14 standard to facilitate the development of user applications and easily manage time-adaptive resources, fine-grained parallelism, and distributed operations. The main aim is providing performance portability, programmability, scalability, energy efficiency, and resiliency. The HPX runtime system relies over a task-based programming model and a global address space which allow the system to support efficient communication, synchronization and load balancing, as well as autonomic management of resources. HPX defines a uniform API, conforming to the C++14 standard, which is properly extended to support remote operations implementing communications using remote method calls functionalities. Fine-grained task execution is handled by the threading subsystem, constituted of HPX-threads (namely, user-level threads) using a work-queue execution strategy. The threading subsystem is non-preemptive, but HPX-threads can decide to yield control to the scheduler in order to let other threads proceed with their execution. HPX represents a standalone distributed runtime system, which can provide both data service and parallelism management to higher-level frameworks, as we saw for YewPar [21].

1.3.2 Mochi core - Argobots

Argobots[1] [24], part of the Mochi software stack [19], is a lightweight low-level threading and tasking framework, which allows specialized runtime management to the user. Besides not representing directly a distributed runtime system, when paired with higher-level frameworks such as Margo[19], it enables a distributed communication model which can be tweeked by means of the underlying runtime system provided by Argobots. The main aim of Argobots is to provide a mapping between high-level abstractions to low level implementations, as well as offering a lightweight layer of execution. To this aim, Argobots implements lightweight parallel work units, such as user-level threads (ULT) and tasklets. Both are non-preemptable execution units and offer, respectively, different models of execution. User-level threads are independent execution units in user space, with a private stack and context-saving capabilities, which can yield control

[1] Argobots: A lightweight low-level threading framework. https://www.argobots.org/

to the scheduler and can be dynamically migrated on a different execution stream at runtime. Tasklets, on the other hand, do not incur in costs related to context saving and stack management. They should be considered as atomic units of execution, which can't yield to a different execution and run to completion without context switching or suspension. Work units are executed by Execution Streams (ESs), associated to OS-level threads. Each ES represents a sequential and independent stream of non-preemptive work units. Given the sequential nature of an ES, work units which are executed by the same ES do not require expensive synchronization mechanisms, since they do not run concurrently. The execution flow is guided by two additional building blocks, namely *Pools* and *Schedulers*. Each ES is associated to a set of *Pools*, which are containers of work units, and execute tasks in the order provided by *Scheduler* entities. *Pools* map work units and events to one (SPSC) or more ESs (SPMC/MPSC/MPMC), hence they can also provide work-stealing between execution streams when paired with shared *Schedulers*. The *Scheduler* building block provides work units to the assigned ES by a specific policy. They are themselves schedulable units, so dynamic scheduling policies can be defined. When a higher-level *Scheduler* pops another *Scheduler* unit from its pools, the new *Scheduler* starts its execution. Once the popped *Scheduler* terminates, the original *Scheduler* is resumed. In order to allow a very fine-grained management of parallelism, Argobots allows the creation of work units, user-defined schedulers and pools, as well as operations for synchronization and yielding control between work units. A representation of Argobots runtime entities is provided in Figure 1.3.1, depicting the relations between the different building blocks participating in the execution of concurrent activities.

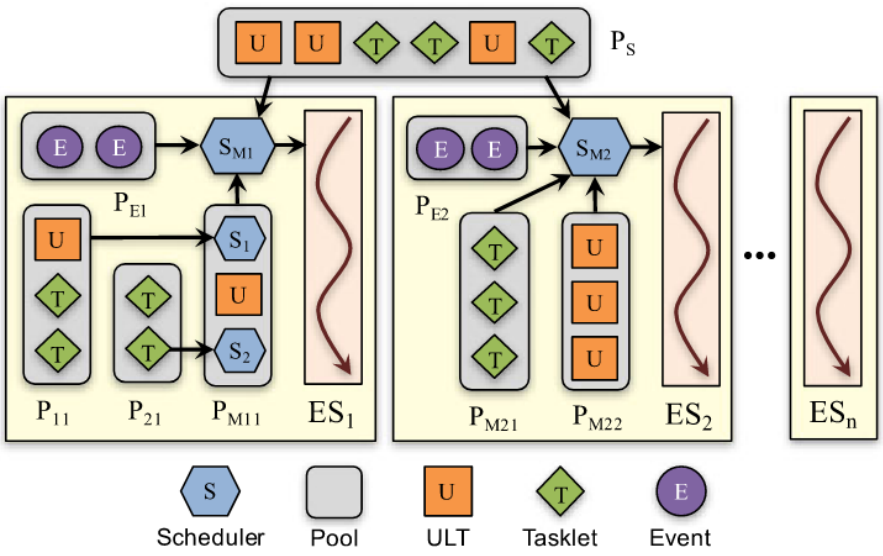


Figure 1.3.1: Argobots execution model. *Reprinted from [24].*

Chapter 2

Communication Frameworks - state of the art and related work

Communication between nodes represent one of the main challenges in the distributed programming environment [2, 3, 28]. In this chapter, we particularly focus on the frameworks and challenges related to the communication layer of the HPC software stack. We consider the *HPC communication layer* as *"any framework, library, tool, protocol or pattern that eases the communication between distributed processes"* [3]. The only purpose of the HPC communication layer is to provide a set of APIs to send and receive data in order to abstract the user from the underlying infrastructure and protocols. Given the aim of this thesis work of extending the existing communication functionalities of the FastFlow library, we particularly focus on communication-related frameworks in order to give an extensive description of the state-of-the-art for the communication layer of the HPC software stack. Understanding the common characteristics of the existing communication frameworks is of particular importance to better grasp the necessity of additional abstraction in the communication layer in order to enable the creation of flexible and portable distributed applications, which are also able to effectively address the heterogeneity of the distributed environment.

2.1 Communication models

As we described in Chapter 1, frameworks providing distributed computing capabilities require underlying functionalities in order to provide communication between nodes scattered across the network. Each distributed programming framework define its own preference for the underlying communication model, but we can distinguish the leading communication models as belonging to two main paradigms, in particular:

- **Remote Invocation**: allows the user to call functions from one node to the other across the network. We consider in this category both *Remote Method Invocation* (RMI) and *Remote Procedure Call* (RPC), with the only distinction between the two being they are used to invoke object's methods or program functions, respectively.

- **Message Oriented**: also in this category we can distinguish two paradigms, *Message Passing Interface* (MPI) and *Message Queuing* (MQ). The former being the de-facto standard in HPC, with minimal overhead, low level minimal abstraction and no fault tolerance; the latter, instead, offers queues to store intermediate messages still not received and it offers fault tolerance.

In the following we present the most common frameworks in the two categories, in order to highlight (some of) the available alternatives for implementing communication services in the distributed environment.

2.1.1 Remote Invocation frameworks

Remote Invocation (RI) frameworks operates as a client/server model which allows the execution of a procedure in a different process than the calling one [29]. They hide the complexity of calling procedures between processes, allowing parameter passing and results retrieval over a specific communication channel. RPC uses a request-reply communication model, where the client nodes send *request* messages to the server, which returns *reply* messages containing the results of the execution. The communication is performed by means of *stubs*, one for (each of) the client(s) and one for the server. The role of the *stubs* is twofold: they implement the RPC communication protocol, specifying how messages are constructed and exchanged; they abstract the networking procedures needed to connect to a server to send requests and receive responses, returning them as results of the called procedure. The RPC stubs are the interface through which the user interacts with distributed systems as if they were local, without having to deal with the low-level details of a communication service based over *sockets*. Another important aspect which is (usually) automatically addressed by the RPC frameworks is the data representation problem. Machine-independent format is needed in order to allow different machines to exchange data. Examples of libraries providing such functionalities are XDR [30], Cereal [31], Boost Serialization [32], Cap'N Proto [33], Protocol Buffers [34]. Each of these libraries allow to transform local data in a representation which can be later reconstructed, in the receiving machine, after being received via the network. Finally, the connected *stubs* communicate, by means of a common transport, in order to exchange data between remote calls. The most common transport protocols are TCP and UDP, but as we will show in the rest of the chapter, various other transports can be used in order to fulfill the needs of high-performance systems. Support for multiple protocols can be added in a RPC framework by abstracting the transport layer and offering networking functionalities via *transport-independent* components.

Java RMI - Java Remote Method Invocation[1] (Java RMI) system is the de-facto standard in remote communication between Java Virtual Machines, since it provides an efficient mechanism for method invocations on Java objects residing in different machines. To match the distributed object system, Java RMI relies over a stub which

[1]Java remote method invocation specification. https://docs.oracle.com/javase/9/docs/specs/rmi/intro.html

manages the invocation of remote object's methods. The used communication protocol is based on TCP/IP connections and is strictly integrated with the Java environment [35]. The tight integration of the distributed object model into the Java programming language makes writing distributed application simple and less error prone, retaining the semantic of a local implementation.

gRPC - is a high-performance and universal RPC framework with support for both asynchronous and synchronous processing of the sequence of messages exchanged by a client and a server. It retains the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types. The service is offered by a set of stubs in the clients nodes and a gRPC server in the server node. The communication between nodes is implemented through the use of Protocol Buffers [34], which is a Google proprietary mechanism to serialize data. It requires to define structured data in a specific format which is then automatically serialized via the Protocol Buffer mechanism.

Mercury - Mercury [19, 36] is a framework implementing RPC functionalities and specifically designed for use in HPC environments as a building block for communication services. It offers an abstracted network API whose implementation is provided by a set of *plugin*s. It allows asynchronous execution of remote procedures and also offers Remote Memory Access (RMA) whenever the underlying transport fabric supports it. The system of plugins offers compatibility with a multitude of network fabrics and protocols, and allows the upper layers to be completely agnostic of the underlying network specification. Mercury is a general framework as it allows multiple networking and communication libraries to be plugged in to offer additional functionalities. The Mercury low level plugin system is currently compatible with both UCX [18] and OFI [17] frameworks, described in Section 2.2, offering communication services over "classic" protocols, like TCP, as well as platform specific ones, as Infiniband Verbs, Intel PSM2, Cray's GNI.

2.1.2 Message Oriented Frameworks

The two paradigms inside this category of frameworks, namely *Message Passing Interface* (MPI) and *Message Queueing* (MQ), provide different functionalities while adopting the same *message-passing* communication model. The MPI frameworks attempt to incur in minimal overhead, so their abstraction is usually low-level and they do not provide any fault tolerance mechanism. On the other hand, MQ frameworks insert messages into queues which are used to (temporarily) store data in order to allow fault tolerance [3]. Both paradigms allow nodes to interact by sharing messages to one another via a communication channel, using different kinds of transports and machine-independent data representations. In the *message-passing* model, opposed to the remote invocation one, programs communicate by sharing data in messages, rather than directly calling procedures in the other process. Additionally to the functionalities provided by MPI frameworks, queues provided by the MQ frameworks allow programs to access queues and retrieve messages without the need of all the communicating en-

tities to be available simultaneously. Both point-to-point and publish-subscribe communication styles are possible.

ZeroMQ - ZeroMQ [37] is a communication library built on top of sockets providing an implementation of the message-passing model, offering fault tolerance and dynamic re-connection of communicating nodes. The communication model allows nodes to share atomic messages using in-process, inter-process, TCP and multicast transports. It allows different communication patterns, like pub-sub, request-reply, fan-out and offers an asynchronous I/O model for scalable multicore applications. The library does not impose a specific data representation, which means that a further layer or representation is needed in order to create machine-independent communication services. It was originally the framework used in the first versions of FastFlow's distributed runtime, later replaced by a plain TCP/IP implementation.

MPI - The Message Passing Interface (MPI) [38] is a *message-passing library interface specification*, as it describes requirements targeting the message passing parallel programming model by providing a set of function specification that must be implemented to provide communication functionalities between processes. The main MPI's aim is standardization of message-passing functionalities to improve scalability, portability and ease-of-use. It is the de-facto standard for communications among processes in distributed systems, and various implementation of the specification exists, like OpenMPI [39] and MPICH [40]. The OpenMPI implementation is also used by the FastFlow framework for its distributed runtime, which we present in Chapter 3. OpenMPI provides compatibility with a wide range of parallel machines by means of a set of components/drivers that offer compatibility with established communication protocols. Additionally to standard protocols like TCP/IP and shared memory, OpenMPI drivers offer compatibility with high-performance fabrics such as Infiniband [16], and Myrinet [41], by supporting HPC-targeted frameworks like OFI [17] and UCX [18]. Different communication transports can be selected automatically by the OpenMPI framework, which selects the most suitable transport for a specific system configuration; otherwise, the user can specify the preferred transport to be used in all the application's communication by forcing it via execution parameters. An important thing to note is that OpenMPI does not allow the selection of different protocols at a *per-node* granularity. Hence, it is not possible to logically partition a distributed application, running as an OpenMPI job, and specify a user-defined list of protocols to be used for each of the existing partitions.

2.2 Toward Exascale computing

Independently to the adopted communication paradigm, as illustrated in [2], the path toward exascale computing requires leveraging the high-performance network fabrics offered by the various HPC systems. System vendors are providing customized network hardware, along with personalized low-level interfaces, to meet the performance and scalability requirements of HPC applications. Hence, it becomes more and more

challenging to develop and maintain portable applications, which can run on existing and future hardware, without relying over carefully crafted abstractions targeting HPC-specific challenges. For this reason, frameworks have emerged in order to leverage the functionalities and performance requirements of customized HPC system's network hardware. To address the programming effort introduced by the multiple HPC network interface and fabrics, frameworks tend to introduce a unified model to ease the burden of programming communication services over specific fabrics with vendor-specific APIs. Some of these frameworks are UCX [18] and OFI [17]. They provide compatibility with well-known communication transports, offering an high-level interface to enable scalability, portability and programmability, allowing to select the most suitable protocols for a specific target system.

Unified Communication X (UCX) - UCX [18] is a network API framework for high throughput computing targeting modern interconnects with massive parallelism. It is designed to provide a set of interfaces for implementing high-performing and highly-scalable network stack for the next generation of systems and applications. The baseline approach followed by UCX is to unify, under a unique framework, both APIs and protocols while maintaining portability across different systems, including future ones. It uses a layered design, comprising three main standalone components, which can be composed to build higher level functionalities. The layered approach provided by the UCX framework allows to abstract the direct access to hardware network resources in order to implement communication protocols, effectively allowing connection between multiple programming models while using an heterogeneous network ecosystem. Additionally, the different layers offer support to both low-level APIs to directly deal with hardware-specific transports, as well as higher-level communication protocols often used in message passing and accelerator-specific programming models.

OpenFabrics Interface (OFI) - OFI [17] is a collection of libraries, specifically designed to target scalability requirements of HPC systems, focused on exporting communication services to applications. The main aim of the set of libraries in OFI is to expose an interface of the underlying network fabrics to application and middleware programmers, in order to fulfill HPC users needs. Implemented with a layered design, it offers independence of the underlying network protocols and implementations. The two main components are the libfabric library, specifically developed to provide the user space API of the OFI framework, and the provider layer, which provides to the libfabric library access to fabric hardware and services. Providers links well to specific network devices and Netowrk Interface Controllers (NIC), and they allow to chose at runtime the correct software objects to connect the high-level API to the underlying network fabric interface. The OFI provider system allows the user-level APIs to be completely agnostic of the underlying fabrics, allowing for compatibility on a broad set of existing systems as well as future ones.

2.3 Multi-protocol support

As we anticipated in Sections 2.1.1 and 2.1.2, existing frameworks in Remote Invocation and Message Passing communication models provide, via additional abstractions, support to a broader set of communication transports. However, a further distinction has to be made at this point. In particular, considering OpenMPI [39] and Mercury [36], we point out a substantial difference about the way these two frameworks provide support for heterogeneous protocols to the programmer: the former offers support to select *one of* multiple transports, allowing applications to choose the most suitable one at runtime; the latter, instead, is a *true* multi-protocol communication framework, as it allows the *simultaneous use* of different protocols provided by the underlying network layer. We notice that there is an intrinsic advantage in preferring a framework which offers the possibility to have multi-homed communication services in order to better adapt to the heterogeneity of the distributed environment.

It is interesting to note that Mercury is currently the only framework providing such a support "out-of-the-box". Its built-in asynchronous progress model allows the coexistence of multiple endpoints in the same application, where each endpoint can use a different protocol, selected from the underlying network libraries. Since the main aim of this work was providing a true multi-protocol functionality to the FastFlow framework in order to allow distributed partitions to communicate with different protocols, we selected Mercury as a candidate for an in-depth study and a first prototype implementation. Given the compatibility with different transports, and considered the simplified model offered by the higher-level abstractions in the Mochi framework, we decided to integrate it as a baseline for the distributed communications in FastFlow. In the following sections, we particularly focus on the Mochi software stack, however, as we show in the following, these frameworks provided very limited support to the TCP and MPI transports. This motivated us to design and implement our own extensible component-based multi-protocol communication building block, initially providing support to plain TCP/MPI communication functionalities already implemented in the FastFlow framework. Hence, the remaining part of this chapter is centered at the description of two main frameworks: Mercury, the backbone of multi-protocol functionalities of the Mochi software stack; Margo, an Argobots binding for Mercury, providing a runtime system targeted at enabling efficient multi-threaded support to RPC functionalities.

2.3.1 Mochi Software Stack

The Mochi framework [19] is a composition of data-service oriented software modules which are specifically developed for the HPC environment. The main modules we analyzed in our thesis work are internally referred to as the Mochi core. They are common building blocks of higher-level data service applications and provide all necessary functionalities to abstract the application from communication-related challenges, such as compatibility with vendor-specific protocols, serialization, remote memory access and fault-tolerance. We describe the two software modules which are targeted at provid-

ing RPC functionalities at different levels of abstraction, namely Mercury and Margo, as already introduced in the previous section. Mercury introduces networking capabilities by defining an RPC-based programming model which allows programmers to define multi-protocol RPC applications seamlessly. Margo further extends the multi-protocol functionalities, by means of Argobots-enabled wrappers, providing a transparent multi-threaded runtime system for the creation of multi-threaded multi-protocol RPC services. Moreover, the reliance over HPC-oriented communication libraries enhances RPC performances by removing the classical dependency of cloud-based RPC frameworks over the TCP transport, by efficiently combining the benefits of different HPC-targeted communication frameworks [36]. In Figure 2.3.1 we depict the usual components which are participating in a distributed application using Mochi's RPC functionalities via the Margo framework, noting that additional layers can exist between the Mochi modules and the transport implementation. For example, MPI can also rely for message transport over raw TCP functionalities or transport libraries like UCX and OFI.

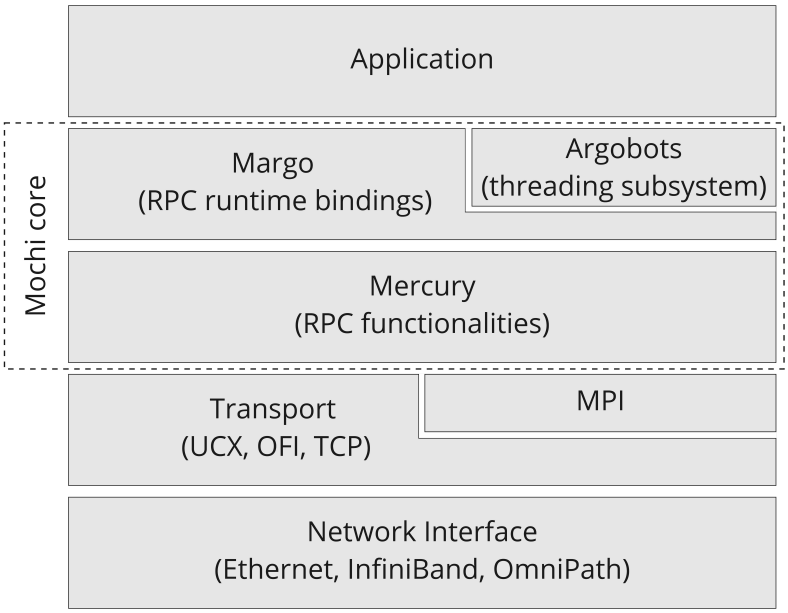


Figure 2.3.1: A Mochi application components.

Mercury

The Mercury framework is platform independent, portable, generic and extendible. Thanks to its underlying plugin system, it provides compatibility with existing and future systems. Additionally, the implemented progress model allows the coexistence of multiple endpoints, listening on (possibly) different protocols, that can be operated via a set of *progress-and-trigger* calls which enable the reception of data and events, as well as the execution of remote calls. The Mochi software stack provides useful higher-level abstractions built on top of the base mechanisms offered by Mercury, including, but not limited to, automation of the *progress-and-trigger* calls and callback management.

Mercury [19, 36, 42] is an asynchronous RPC framework purposefully built to efficiently provide communication services to HPC systems with high-performance fabrics. Regarding communication services, it is the main framework in the Mochi core as it provides the basics RPC functionalities to higher-level frameworks in the stack. Mercury provides an abstracted network implementation to enable transparent support to future systems and protocols, efficient use of existing native transport mechanisms, and support of large data arguments via the RDMA enabled interface. Mercury is based on the functionalities of the *I/O Forwarding Scalability Layer* (IOFSL)[43], which allows RPC calls specifically related to file-system-specific I/O operations. By extending this layer, Mercury allows to generate RPC calls to generic functions that can be dynamically defined and registered in the application using Mercury. The library allows the asynchronous execution of arbitrary functions via function tagging and callbacks, coupled with a queue system that stores procedure calls which are pending and still not executed. The receiving process drives the progress loop in which arguments are retrieved, function calls are executed, and the results are sent back to the origin node. The asynchronous RPC interface, thanks to its execution queues, can drive progress of RPC callbacks, as well as parameter transfer, in both a blocking and non-blocking fashion.

Additionally, the library offers the possibility of being ported to various systems since the network layer is abstracted and the application interface is based on a simple set of network primitives for both point-to-point messaging and one-sided RDMA. The network functionalities are implemented on top of different plugins. The plugin system can be considered as a *compatibility layer*, which provides the communication functionalities offered by different protocols to the upper layers in the software stack. Examples of plugins are *ofi*[2], *mpi*[3], and *ucx*[4], and multiple protocols are offered by each of the plugin, ranging from vendor-specific protocols to widely accepted ones, like TCP.

RPC: a Mercury's perspective General RPC frameworks provide the possibility to serialize function parameters and ship them to a remote node that will execute the respective function call. As stated in [36], Mercury tries to address two main common problems of general-purpose RPC frameworks:

- inability to take advantage of HPC-targeted high performance communication protocols: standard frameworks are usually designed on top of TCP/IP protocols, which represent a limitation over the performances often required by HPC systems;
- inability to transfer large amount of data: standard RPC frameworks doesn't allow (or discourage) transfer of large amount of data through the implemented mechanisms.

[2]Mercury. OFI plugin. https://mercury-hpc.github.io/user/ofi/
[3]Mercury. MPI plugin. https://mercury-hpc.github.io/user/na/#deprecated-plugins
[4]Mercury. UCX plugin. https://mercury-hpc.github.io/user/na/#ucx

Mercury addresses these limitations by offering an asynchronous and flexible RPC interface specifically tailored for HPC systems. Hence, Mercury exposes an abstracted API for both RPC and large data transfer, and completely relies on the underlying network implementation provided by the plugins, in order to be independent from the used transport mechanism. In the end, Mercury's main purpose is to serve as a basis for higher-level frameworks that need to remotely exchange data in a distributed environment, by offering a flexible interface completely decoupled from the underlying protocol and system specifications.

Architecture Mercury is built over a two-layer architecture where each layer provides an abstraction for specific functionalities, as shown in Figure 2.3.2. The lowermost layer offers abstractions needed to provide networking functionalities such as point-to-point messaging, address lookup, remote memory access, progress and cancellation. The uppermost layer is further divided in two service-level components, referred to as *RPC interface* and *bulk data interface*. The RPC interface allows the programmer to remotely execute function calls, shipping function arguments and receiving results from the remote node; the bulk interface, instead, complements the RPC interface and allows large data transfer via the creation of memory descriptors, which enables the possibility to initiate raw memory transfers using remote memory access, whenever the underlying system provides it.

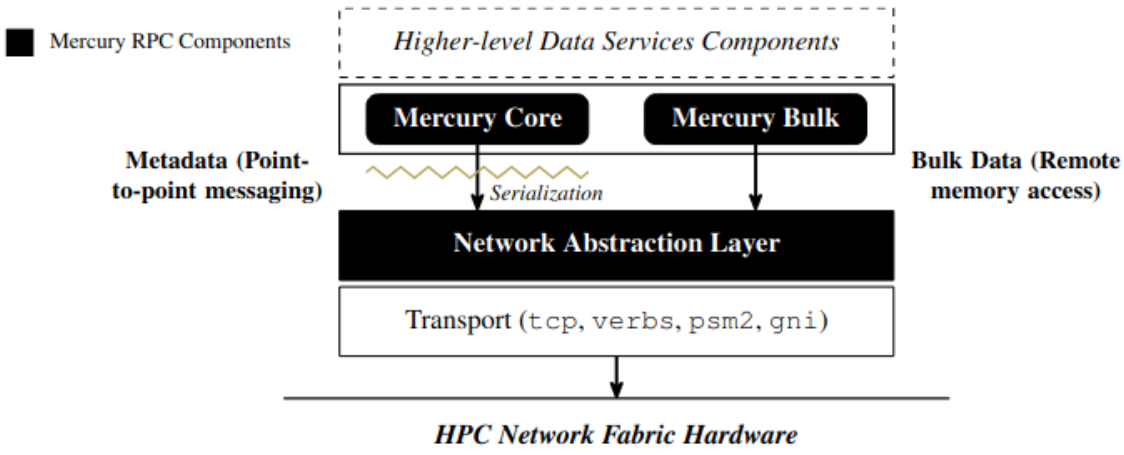


Figure 2.3.2: Mercury RPC components in the software stack. *Reprinted from [42].*

Plugins Mercury plugins are referred to as a "*support for various network protocols that can be easily added and selected at runtime*" [44]. Given that Mercury's main aim is to leverage the high performance solutions provided by HPC network fabrics, which requires specific low-level vendor APIs, it relies on plugins as an intermediate layer for network functionalities. In order to overcome the burden of implementing the network abstraction layer directly on top of those APIs, Mercury relies over various plugins for the implementation of functionalities like RDMA and point-to-point

messaging. A multitude of plugins are provided by the framework, however the vast majority of them is under testing or being deprecated, leaving as "stable" only the ones related to OFI [17], UCX [18] and shared-memory[5] for local nodes communication. We show in §2.3.1 the main limitations we have encountered during testing of these plugins.

Switching between various plugins is very simple and can be done by specifying a predefined string containing the desired plugin paired with the needed protocol to use during communications. Each plugin defines its own format, but common fields are shared among the configuration strings of various plugins, and they mostly refer to the type of plugin and the protocol to be used. Format of configuration strings is provided in Table 2.1.

Plugin	Protocol	Initialization format
ofi	tcp*	ofi+tcp[://⟨hostname,IP,interface name⟩:⟨port⟩]
	verbs	ofi+verbs[://[domain/]⟨hostname,IP,interface name⟩:⟨port⟩]
	psm2	ofi+psm2
	gni	ofi+gni[://⟨hostname,IP,interface name⟩]
ucx	all	ucx+all[://[net_device/]⟨hostname,IP,interface name⟩:⟨port⟩]
	tcp*	ucx+tcp[://[net_device/]⟨hostname,IP,interface name⟩:⟨port⟩]
	rc,ud	ucx+⟨rc,ud⟩[://[net_device/]⟨hostname,IP,interface name⟩:⟨port⟩]
na	sm	na+sm[://⟨shm_prefix⟩]
mpi*	dynamic, static	mpi+⟨dynamic, static⟩

Table 2.1: Mercury plugin's initialization format. NOTE: *Plugins or protocols marked with ∗ must be intended as faulty, with limited functionalities or end-of-life.*

Network Abstraction Layer The Network Abstraction Layer (NAL) provides an abstraction of the network infrastructure above which the communications are executed. It is a simple abstraction which only provides limited functionalities, like address lookup, point-to-point messaging, remote memory access, progress, and cancellation.

The abstractions provided by this layer allows the uppermost layers to be completely agnostic of the underlying communication protocol implemented via the plugin system. Moreover, The API is non-blocking and uses a callback mechanism to provide asynchronous execution. Progress is driven by API calls which allows user callbacks to be placed in completion queue and retrieved for execution.

Mercury refers to communicating nodes as *origin* and *target*, indicating respectively the node issuing the request and the node receiving it. Both *origin* and *target* nodes must specify the desired plugin/protocol pair at initialization phase, by providing a string as described in Section 2.3.1. Since a node can both provide and ask for services, the initialization phase is the only time a server-specific behaviour can be defined, where the user can specify if the current node will be listening for incoming RPCs.

[5]Mercury. Shared Memory plugin. https://mercury-hpc.github.io/user/sm/

The functionalities offered by the NAL refers to three main concepts, which are used by all the uppermost layers:

- expected messages: require a *receive* operation to be pre-posted by the target. Therefore, this requires the origin node to be known in advance, before the receive operation is posted. If the receive operation is not posted before the message is sent, it can be dropped;
- unexpected messages: does not require the target to post a receive operation for the message, as it can arrive from any source. The target can retrieve received messages in an asynchronous way. These messages are allowed to be dropped, but the plugin can decide to queue them anyway;
- remote memory access: allows registration of memory chunks which can be later accessed by target nodes. Abstractions are provided through API which contains operations generally provided by most RDMA protocols.

The network abstraction is designed to allow emulation of one-sided operations, such as RDMA, on top of two-sided protocols. In this way, Mercury can easily adapt the provided one-sided functionalities to protocols which only supports fixed two-sided operations, like TCP.

RPC Layer The RPC layer is based on the messaging model described in §2.3.1, and it allows nodes to issue remote calls, ship parameters and retrieve results from a remote target node. RPC requests are based on the common knowledge of *origin* and *target* nodes on the procedures which can be invoked, identified via a shared ID, as well as common procedures to encode/decode function parameters and return values.

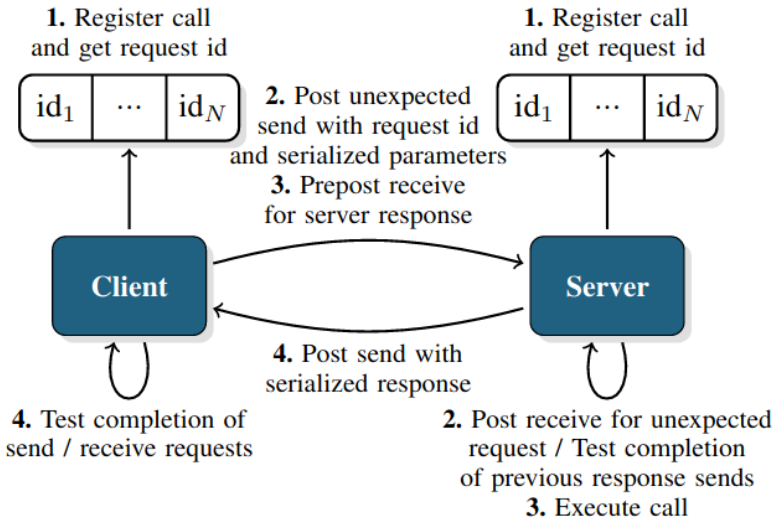


Figure 2.3.3: Execution flow of RPC call. *Reprinted from [36].*

Mercury provides mechanisms to support a set of generic function calls avoiding hard-coded routines. To allow this, *origin* and *target* nodes must register a unique function name along with encoding and decoding routines by using shared input/output types. We postpone the precise description of the registration process to §2.3.1, since Margo introduces few additional steps, which are the ones that are actually used to implement the Margo-based components of the implemented extension for FastFlow's communication classes. We show in Figure 2.3.3 a usual flow of execution needed by both *origin* and *target* nodes to execute an RPC request. The registered function is mapped to an ID which will be used in all the communications between the two nodes. A further step is needed by the *target*, which must register the callback that will be executed every time that ID is received. Once the functions are registered, the origin sends an *unexpected* message to the *target*.

After issuing an RPC request, the behavior of the two nodes depends on the following:

- the RPC call expects a response from the *target*: the *origin* node prepares its memory buffer to receive the response and uses the buffer to pre-posts an *expected receive*. At the reception of the response, the *origin* node can retrieve the response from the callback queue and proceed;
- the RPC call does not expect a response from the *target*: the *origin* node, at the moment of RPC registration, declares that this RPC does not expect a response. An RPC of this kind allows the *origin* node to proceed without posting a receive operation, progress can be made as soon as the request is sent to the *target*.

From the *target* side, receiving an *unexpected* message with a specific ID translates in the execution of the callback registered at startup by decoding the parameters sent by the *origin*, and sending the outputs by encoding them at the end of execution, if the registered RPC expects a response.

Bulk Layer This layer allows to send large data by avoiding intermediate memory copies. It allows to perform direct memory access by creating a local memory handle which points to previously registered areas of memory (not necessarily contiguous) which the *target* node can access via RDMA operations. The bulk layer is directly built on top of the RDMA interface defined in the network abstraction layer.

A typical execution flow for a bulk request is depicted in Figure 2.3.4. The *target* node manages all the bulk transfers in order to be able to control the data flow and protect its memory from concurrent accesses. This operation is one-sided, and it is started by the *origin* node which creates a bulk data descriptor, serializes it, and send the serialized descriptor to the *target* as an argument via a specific RPC request. Once the descriptor has been deserialized by the *target*, two situations can occur. The request can be related to *consumption* or *production of data*. In both cases the *target* node creates a memory handle to manage the request, allocating the necessary memory.

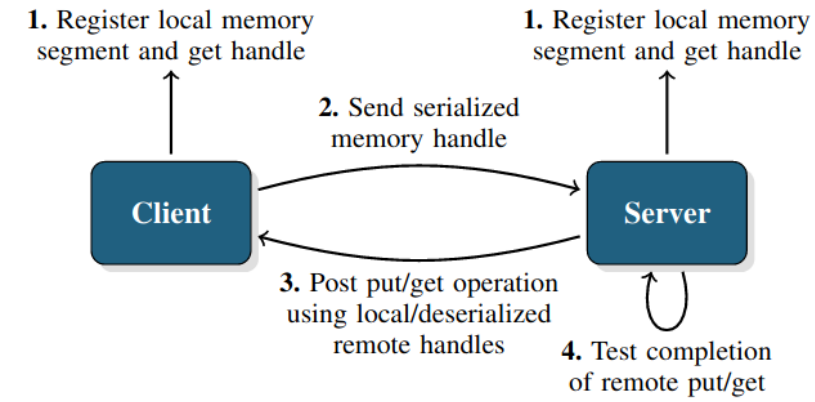


Figure 2.3.4: Execution flow of bulk request. *Reprinted from [36].*

However, in the first case the *target* initiates a remote `read` operation before performing the function call, in the second case the *target* executes the call and initiates a remote `write` operation with the produced results.

Memory handles are an important building block for memory transfer, in particular in such cases where non-contiguous memory is to be transferred. The memory of the communicating nodes is abstracted by the memory handle and allows to access memory in a transparent way without modifications to the process described above.

Progress and Cancellation Mercury adopts a set of *progress* and *trigger* calls in order to retrieve networking events and execute requests in each of the node's execution queues. Making *progress* allows the current node to send and receive data, instead performing a *trigger* operation allows the node to execute the received pending callbacks in its execution queue. The *progress-and-trigger* model is explicit and the user must properly handle a progress loop for each of the defined communication instances. For the sake of simplicity, we do not provide the explicit progress loop description here, since, as we show in the following, this is automatically managed by the runtime system of Margo and it is completely transparent to the user.

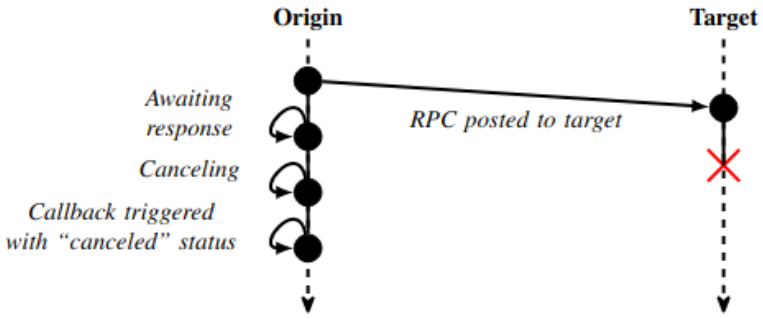


Figure 2.3.5: Cancellation of an RPC operation. *Reprinted from [42].*

Mercury also provides a fault tolerance mechanism based on cancellation of already-submitted operations. The fault tolerance mechanism is mainly provided by the possibility to interrupt calls and reclaim resources of pending operations after these have been signaled as *cancelled*. Cancellation is an asynchronous and local operation. From a user perspective, completion of offloaded operations is known only when the associated callback is placed in the local queue of pending operations. When a callback is triggered and the operation was locally canceled by the user, that operation is reported as canceled and internally aborted by the framework, as shown in Figure 2.3.5.

Margo

Margo [19] is a high-level Mercury binding that uses Argobots as a runtime library. Just like Mercury, Margo requires both origin and target nodes to register for RPC calls associated with a specific identifier (a string) and an input/output type, which must necessarily be registered using the relative macros, as we describe in the following.

By using Argobots as a runtime library, Margo hides the handling of Mercury's progress loop by delegating it to a specific Argobots ES (namely, an OS-level thread). Moreover, Margo allows to freely manage how the various RPC associated callbacks will be dispatched among internal pools and external user defined pools. When initializing Margo, the user can specify its own pools and ESs that will be responsible for both progress loop and RPC calls, otherwise the programmer can decide to completely let Margo handle the calls with its own pools and ESs.

Being a binding between Mercury and Argobots, Margo retains all the concepts we defined in the previous section, however, Margo greatly simplifies the development of RPC-based services by introducing extensions to the Mercury-based model:

- more intuitive communication: Margo defines Argobots-aware wrappers for Mercury's communicaton functionalities, in order to provide asynchronous and multi-threaded communication mechanisms as a single threaded model. Communication facility are handled by ULTs, which can be suspended and resumed as communication proceeds, they can handle multiple communication channels and multiple protocols in a transparent way for the user;

- progress loop abstraction: polling and communication events are internally managed by ULTs. Policies for handling RPC-related execution can be manually defined and tweaked to better suit the application needs. In this way, multiple providers can be easily handled and multiplexed by the same core process, without the need of handling multiple progress loops at once. Each RPC can be associated with a different Argobots execution stream in order to associate specific callbacks to different OS-level threads;

- renewed polling strategy: Margo allows both busy and idle mode, which allows to tweak Margo for both performance and resource management.

RPC registration process In this section we describe the RPC registration process, which follows a standardized procedure and requires RPC input and output types, as well as routines to pack the types into network buffers, to be defined by both *origin* and *target* nodes. Additionally, the callback function to be executed must be defined by the *target* node with a predefined signature in order for Margo to encapsulate it in the internal service functions which triggers RPC execution upon receiving the request from the *origin* node.

The registration process must proceed by:

- defining input and output types: as shown in Listing 2.3.1, RPC arguments types must be defined by the means of a C-style `struct` datatype, which can internally contain all the necessary types for the RPC call;

- defining the packing routine: Listing 2.3.3 defines the routine which is internally used by Margo to correctly write to and read from the network buffers which are then used to send and receive RPC data arguments. Margo also offers a "simplified" way of defining such routines, which we omit for sake of presentation and to better explain how the internal network buffers are built and filled with data. Each packing routine is strictly tied to a specific type. Having a type `X`, the routine must be defined as `hg_proc_X`, and it must encode and decode each of the field contained in the `X` struct type. Additionally, Margo can be built with XDR capabilities which will in turn change the way data are represented in the buffer exchanged between nodes, however, since FastFlow distributed version shares already serialized streams, we do not rely upon this functionality.

- defining the actual RPC callback: this step is only required by the node which will act as a "server" for the specific RPC. In Listing 2.3.2 we provide a sample RPC callback declaration and definition, which allows the server node to define the function to be executed upon an RPC request from another node. RPC callback declaration and definition require also a call to specific macros defined by Margo, which will wrap the RPC callback defined by the user with Argobots-aware code, necessary for Margo to dispatch RPC callbacks among different ULTs.

- associating an ID to the RPC: after having registered all the types and routines, both *origin* and *target* nodes have to associate a common ID for the RPC that they intend to use. This can be done as shown in Listings 2.3.4 and 2.3.5, where we also tell the Margo runtime that we do not expect a response from the RPC call, as we can see from `margo_registered_disable_response` call at line 16. The registering process is complete, and the *origin* node can now use the returned RPC identifier to issue requests to the listening *target* node by using a `margo_forward` call. Note that, after the registration process is complete, the listening node can receive RPCs calls with the specified ID from all the *origin* nodes which registered an RPC with the same ID.

```c
typedef struct {
    hg_int64_t   id;
    hg_uint64_t  size;
    hg_bulk_t    bulk_handle;
} ff_rpc_in_t;
```

Listing 2.3.1: RPC type definition.

```c
void ff_rpc(hg_handle_t handle);
DECLARE_MARGO_RPC_HANDLER(ff_rpc);

void ff_rpc(hg_handle_t handle) {
    ff_rpc_in_t           in;
    struct hg_info* hgi;
    margo_instance_id     mid;

    // Get input data
    margo_get_input(handle, &in);

    // Retrieve RPC identifier object
    // and get registered data back
    hgi = margo_get_info(handle);
    mid = margo_hg_info_get_instance(
        hgi);

    // Here data may be retrieved
    // and used internally by RPC call

    margo_free_input(handle, &in);
    return;
}
DEFINE_MARGO_RPC_HANDLER(ff_rpc)
```

Listing 2.3.2: RPC declaration and definition. Business logic code is omitted.

```c
hg_return_t
hg_proc_ff_rpc_in_t(hg_proc_t proc,
                    void* data) {
    // Retrieve input data structure
    ff_rpc_in_t* in;
    in = (ff_rpc_in_t*)data;

    // Build Margo send/receive buffer
    hg_proc_hg_int64_t(proc,
            &in->id);

    // Specific routine for each
    // field in the RPC input struct
    hg_proc_hg_uint64_t(proc,
            &in->size);

    hg_proc_hg_bulk_t(proc,
            &in->bulk_handle);
}
```

Listing 2.3.3: Packing routine definition. Allows the Margo framework to manage data from/to the network buffer. Each of the type-specific routines allows data copies which are aware of the size of data to be packed/unpacked.

```c
int main(int argc, char** argv) {
    // Margo initialization code
    ...

    // Initialize Margo origin node
    margo_instance_id mid;
    mid = margo_init(listening_addr,
        MARGO_CLIENT_MODE, 1, 1);

    // Register RPC id and input
    // type, ignoring output
    my_rpc_id = MARGO_REGISTER(
        mid, "ff_rpc",
        ff_rpc_in_t, void,
        NULL);
    margo_registered_disable_response(
        mid, my_rpc_id, HG_TRUE);

    // Looks up for a server
    hg_addr_t svr_addr;
    margo_addr_lookup(mid,
                      svr_addr_str,
                      &svr_addr);

    // RPC handle
    hg_handle_t handle;
    margo_create(mid, svr_addr,
        my_rpc_shutdown_id, &handle);

    // Forwards RPC to target
    margo_forward(handle, NULL);
    ...
}
```

Listing 2.3.4: Finalization of the registration process in the origin node. The register macro specifies the input/output expected types as well as the packing routines as described above.

```c
int main(int argc, char** argv) {
    // Margo initialization code
    ...

    // Initialize Margo server node
    margo_instance_id mid;
    mid = margo_init(protocol,
        MARGO_SERVER_MODE, 1, 1);

    // Register RPC id, input type
    // and RPC callback function
    my_rpc_id = MARGO_REGISTER(
        mid, "ff_rpc",
        ff_rpc_in_t, void,
        ff_rpc);
    margo_registered_disable_response(
        mid, my_rpc_id, HG_TRUE);

    margo_wait_for_finalize(mid);
}
```

Listing 2.3.5: Finalization of the registration process in the target node. The listening node only needs to register the RPC types and routines, as the origin node, and additionally it needs to specify the RPC callback for the registered ID.

Mochi core review

The Mochi software stack comes with lots of functionalities and drawbacks. We list in this section all the advantages and the main limitations of Mercury and Margo frameworks. Since Mercury is the backbone of multi-protocol RPC functionalities in the Mochi stack, its functionalities and drawbacks are shared by all the higher-level libraries which depends on it. Some of the limitations we have encountered were not listed or properly documented, thus we could only build a proper assessment of each of the provided plugins and protocols after the testing phase. This forced us over a gradual trial-and-error phase for each of the provided functionalities, which eventually

led us to consider the development of our own communication abstractions.

Advantages Mercury's interface allows to easily enable communication using a system of plugin which offers compatibility with a multitude of network and vendor specific protocols. The Mercury library, once again, shows the importance of having an abstraction layer to provide efficient use of underlying protocols without having a complete expertise on how they work. Being able to address different needs and to leverage the functionalities of different systems, by the means of a general API, makes implementing communication on such systems painless and less error prone. Moreover, Mercury's capability of handling different protocols, with no code changes, allows portability of applications to new systems which may provide only vendor specific protocols or which does not offer support to common transports such as TCP or MPI. Moreover, the dependencies to use the Mercury framework are only related to the set of plugins the user intends to integrate in his application.

Margo, on its hand, introduces an incredibly easy interface to implement multi-homed RPC services, which greatly simplified the development of RPC-based communication services. It completely removes the complexity of handling progress loop for each of the listening endpoints, allowing a very easy approach to multi-protocol support.

Limitations The frameworks, however, are not exempt from problems and limitations. In this section we gathered all the main limitations that are known and which we discovered during the testing phase. As specified in §2.3.1, MPI plugin is deprecated and not maintained[6]. In fact, Mercury's developers suggest the use of the *ofi* plugin to leverage efficient HPC communication mechanisms, and *ucx* plugin as a general purpose one. As specified in [44], MPI is considered to be present in almost all systems, and the functionality offered by the NAL are only intended for prototyping and testing: this suggesting that in case an MPI implementation is needed, one shouldn't rely on Mercury's interface. However, as we discovered by testing, the MPI plugin is not working as intendend and no prototyping is possible whatsoever.

Further limitations found during testing of the *ofi* plugin are mostly related to the poor support for the TCP transport. Problems about this specific transport were already reported[7][8][9], and during testing the main that we noticed are:

- During reply phase of RPC request using `ofi+tcp`, information about the public IP of *origin* node are not used. Thus, in case the *origin* node was behind NAT, this eventually translated in a lost packet. We argue that this problem is related to the fact that the plugin internally uses information about the *origin* local IP address (forged in the exchanged data between *origin* and *target*), which may be

[6]Mercury. Issue #489. https://github.com/mercury-hpc/mercury/issues/489
[7]OFI. Tcp provider. https://ofiwg.github.io/libfabric/v1.11.1/man/fi_tcp.7.html
[8]Mercury. Issue #418. https://github.com/mercury-hpc/mercury/issues/418
[9]Mercury. Issue #332. https://github.com/mercury-hpc/mercury/issues/332

different from its public one. The same does not happen by using `ofi+sockets`, however its use is discouraged;

- Various problems related to termination of nodes which issued a request that couldn't be fulfilled.

Other plugins, such as *ucx*, presented problems in connecting nodes situated in different networks, but it is indeed able to establish a connection between endpoints situated in the same subnetwork. Moreover, the TCP implementation in the *ucx* plugin is not relyable and sometimes causes the *origin* and *target* nodes to hang indefintely. Additionally, when using Margo paired with the `"ucx+tcp"` configuration, the *ucx* plugin internal implementation does not allow to use it via the `MARGO_CLIENT_MODE`, hence a `MARGO_SERVER_MODE` is forced also for origin nodes which are not listening for incoming RPCs. This can cause the internal creation of additional concurrent entities, which may be not desirable. However, we were not able to test the rest of the provided transports for both UCX and Libfabric frameworks, since they require vendor-specific fabrics which are not at our disposal. Since the UCX and Libfabric frameworks are specifically target at the HPC environment, we believe their implementation in Mercury is much more precise and functional.

Finally, the main limitation of the internal RPC implementation are mostly related to how RPC calls are handled, in particular with reference to data copies happening between the input data and the send/receive buffers internally used to ship RPC data through the network. Each RPC call involves data copies as follows: at the sender, after a call to `margo_forward` is performed, the packing routine is called by the runtime and data copies take place internally to fill the network buffer; at the receiver of a remote call, the runtime invokes the unpacking routine to copy data from the network buffer to the user data. An alternative is provided by the bulk functionalities, which allows communicating nodes to access the origin's memory in order to bypass intermediate data copies. However, this approach introduces new challenges related to the management of memory buffers in the *origin* memory, as well as management of all the direct memory accesses request performed by the *target* node. So this is not always desirable even if it allows to completely remove data copies for large data arguments. Additionally, another problem is introduced in the utilization of bulk transfers via RDMA where no responses are expected from the sender, in fact in this particular case one must be extremely careful about freeing memory that has still not been read from the remote end.

Margo, on its hand, introduces additional dependencies which are related to the libraries it uses for its runtime system. The introduced dependencies are the following:

- Mercury: it represents the backbone of multi-protocol RPC functionalities provided by Margo. To be used with Margo it additionally requires a specific configuration during the building process. In particular, Mercury internally provides BOOST macros to automatically generate code. This is needed to enable preprocessors macro used by Margo to generate the code used for RPC callbacks and input/output packing routines;

- Argobots: it is the user-level threading library internally used as a runtime system by Margo;
- json-c: A JSON implementation for C language[45], needed internally by Margo in order to generate configurations based on json-formatted strings provided at initialization of Margo instances.

Given the requirements for the FastFlow distributed runtime, the internal limitations of the Mochi frameworks are too tight to allow its extensive utilization as FastFlow's final multi-protocol communication component. However, the provided plugins and functionalities served anyway as a prototype to develop FastFlow communication layer APIs. As we pointed out, the composition of the Mercury framework with higher-level abstractions allows to easily develop communication nodes which handle multiple endpoints. Margo represented a good starting point for the definition of a high-level set of communication APIs. The simplicity of creating communication nodes and connecting them via a high-level interface allowed us to experiment and develop generic components with a multi-protocol oriented model. Having an independent API is a natural and important step to remove strict dependencies from each of the technologies used to implement the communication functionalities, allowing portability to future systems with little to no changes in existing application code. However, given the limitations we pointed out in this section, we considered the lack of support for the TCP and MPI transports as too limiting. Hence, we decided to implement our multi-protocol networking components by initially integrating the existing FastFlow TCP and MPI communication building blocks. It was anyway conceptually interesting and very helpful to retain some of the concepts introduced by Mercury and Margo in the implementation of the final communication layer. In the next chapter we provide an overview of the currently existing distributed runtime of the FastFlow framework and the presented extension introducing multi-protocol capabilities to the distributed runtime.

Chapter 3

FastFlow distributed runtime and extensions

In this chapter we present the existing distributed runtime of the FastFlow framework, as well as the proposed multi-protocol extension which is the result of our study on existing libraries and frameworks targeted at providing multi-protocol support. Our work is based on the introduction of new software components extending communication functionalities of the FastFlow distributed runtime. The main aim is the one of enabling heterogeneity in computing nodes by providing the possibility to define logical partitions of existing distributed applications, enabling each computing node to use multiple transports implementations for each of its output connections. We initially implement support to transports like TCP, MPI and Margo, but we provide an extensible interface to allow further protocols to be added without disrupting existing applications. Additionally, we propose a proof-of-concept extension to the existing configuration file, which would provide a way to the application developer to precisely describe the configuration of her distribute application. We divide this chapter in two parts: first we describe the existing FastFlow distributed runtime, then we provide a description of the proposed extension and the requirements a developer must respect in order to further extend the implemented communication functionalities.

3.1 FastFlow existing distributed runtime

Recently, a distributed runtime has been added to the the FastFlow library which allows to deploy and execute applications on distributed systems [14, 15]. The introduced extension allows to develop distributed applications, as brand new programs or as migrations of existing ones, by means of wrapper classes and renewed `BB` components. The introduced mechanisms for the distributed runtime allow the programmer to divide an application in disjoint *distributed groups* of `BBs` (*dgroups*, henceforth) and specify the mapping of each group to available machines through a JSON configuration file. The distributed extension for the FastFlow library targets the `BB` layer in order to: provide portability of existing applications by hiding the distributed communication-related challenges; offer a set of mechanisms to build higher-level parallel and distributed

components. The programming model remains the same, since the distributed `BBs` allow to build a distributed application by following the same LEGO-style approach followed to build parallel applications on shared-memory systems, thus retaining the same data-flow streaming semantics of the original application.

3.1.1 Transition to distributed-memory

The transition from shared-memory to distributed-memory required the creation of new concepts, such as the *dgroups*, as well as new abstractions, as the classes for data serialization and communication. Existing applications can be deployed in the distributed environment by retaining the data-flow streaming semantic of the original shared-memory FastFlow application. The modifications required for a programmer to transform a shared-memory FastFlow application into a hybrid shared/distributed-memory application are related to the identification of disjoint *dgroups* and the mapping of each group to an host by means of a JSON configuration file. The *dgroups* represent logical partitions of the original FastFlow streaming concurrency graph. Each one of the *dgroups* is internally constructed as a standard shared-memory application to fully exploit the potential of the single nodes in which they are executed. Moreover, the resulting *dgroup* is internally plugged with customized sequential `BB`s and node wrappers to realize serialization/deserialization and communication between the deployed *dgroups* by retaining the original application semantic.

A *dgroup* is internally implemented via the FastFlow's farm `PBB`. The *Emitter* is the *dgroup*'s *Receiver* and the *Collector* represents the *Sender*. These nodes do not perform any operation over the received/sent data, thus the `BBs` directly communicating with the *Receiver* and *Sender* nodes are wrapped by the RTS with class wrappers that transparently perform serialization on the input/output data-types. The farm's *Worker*s are the `BBs` of the original application graph included in that particular *dgroup*. They communicate by means of shared-memory channels with the rest of the application graph.

The API to define the *dgroups* is composed of two functions:

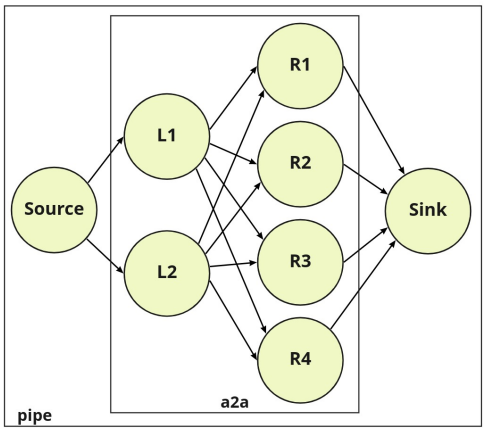
- **createGroup**: allows the creation of the individual *dgroups*. The function takes a string argument, which uniquely identifies each *dgroup*.
- C++ "`<<`" operator: allows the inclusion of `BBs` inside an existing *dgroup*. Particularly useful when the programmer wants to create multiple *dgroups* from a single `BB` and insert only a subset of its nested `BB` in each *dgroup*.

Creating a distributed group from a `BB` allows to add, via the "`<<`" operator, only direct children of the original `BB`. In this way, the granularity of the application remains coarse enough to be executed efficiently in a single multi-core node and exploit the local parallelism. Moreover, from a sequential `node` and a `farm PBB` only a *dgroup* can be created, instead the `pipeline` and `all-to-all BBs` can be split into multiple *dgroups*. The splitting of a `pipeline` can be performed only by including contiguous stages

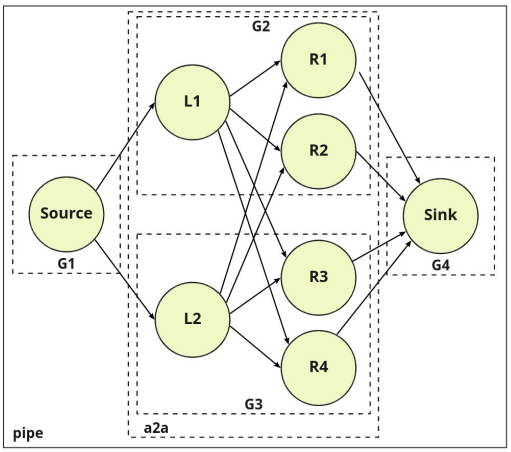
into the same *dgroup*. The `all-to-all`, instead, can be divided in different ways. A vertical or non-inter-sets horizontal cut produces *dgroups* which communicates only via distributed communications, because each *dgroup* only contains nodes either from the *L-Workers* or from the *R-Workers*. An horizontal inter-sets cut, instead, produces *dgroups* which contains both *L-Workers* and *R-Workers* nodes. This necessarily translates in some communications happening in the shared-memory domain, while others taking place in the distributed-memory one. We show, in the example at the end of this Chapter, how a `pipeline` and an `all-to-all` building blocks can be split to form multiple distributed groups.

To launch a FastFlow application in a distributed environment, an additional software module was introduced, called `dff_run`. This module, following the mapping defined in the JSON configuration file, launches the application processes on each of the specified remote nodes. The `dff_run` also takes care of defining the execution parameters for each host-group pair. In particular, the `dff_run` module creates `ssh` connections toward the remote nodes specified in the configuration file and launches the executable in each of the nodes. It also allows to gather the output of each stream coming from the `ssh` connections combined in a single file with proper annotations. This software module also supports running application using the MPI library as a communication layer, acting as a wrapper of the `mpirun` launcher. It produces a *hostfile* which will then be passed to the `mpirun` module.

To show how the creation of a distributed application proceeds, we refer to one particular example which represents our main reference example throughout this work. In the following, we omit for simplicity the definition of the business logic code of the specific nodes which are not part of the run-time system. In Listing 3.1.1 we create a shared-memory version of a three-stage pipeline. The first and the last stage are simple sequential nodes, instead the middle stage is an `a2a` `PBB` with two left workers and four right workers. The distributed version is composed of four groups: "G1" which only contains the source node, "G2" containing the top half of the `a2a`, "G3" containing the bottom half of the `a2a`, and "G4" containing the sink node. This example is rewritten in Listing 3.1.2 to show the necessary modifications to transform the previous example in a distributed application. We create the *dgroups* starting from the original `pipeline` building block, using the inclusion operator to split the `a2a` workers between different groups. Figure 3.1.1a and Figure 3.1.1b show, respectively, the application concurrency graph for shared-memory and distributed-memory version.



(a) Shared-memory version.



(b) Distributed-memory version.

Figure 3.1.1: Application concurrency graph for shared-memory and distributed-memory presented, respectively, in Listing 3.1.1 and Listing 3.1.2.

```
#include <ff/ff.hpp>
...
int main(int argc, char *argv[]){
    ...
    // Concurrency graph
    Source source;
    LNode L1, L2;
    RNode R1, R2, R3, R4;
    ff_a2a a2a
    a2a.add_firstset<LNode> (
        {&L1, &L2});
    a2a.second_set<RNode> (
        {&R1, &R2, &R3, &R4});
    Sink sink;

    ff_pipeline pipe;
    pipe.add_stage(&source);
    pipe.add_stage(&a2a);
    pipe.add_stage(&sink);

    pipe.run_and_wait_end();
    ...
}
```

Listing 3.1.1: Simple shared-memory application.

```
#include <ff/dff.hpp>
...
int main(int argc, char *argv[]) {
    DFF_Init(argc, argv);

    // Concurrency graph
    Source source;
    LNode L1, L2;
    RNode R1, R2, R3, R4;
    ff_a2a a2a
    a2a.add_firstset<LNode> (
        {&L1, &L2});
    a2a.second_set<RNode> (
        {&R1, &R2, &R3, &R4});
    Sink sink;

    ff_pipeline pipe;
    pipe.add_stage(&source);
    pipe.add_stage(&a2a);
    pipe.add_stage(&sink);

    // Distributed groups creation
    auto G1 = source.createGroup("G1");
    auto G2 = a2a.createGroup("G2");
    G2 << L1 << R1 << R2;
    auto G3 = a2a.createGroup("G3");
    G3 << L2 << R3 << R4;
    auto G4 = sink.createGroup("G4");
    return pipe.run_and_wait_end();
}
```

Listing 3.1.2: Simple distributed application.

All the remote hosts require the same executable, as well as the same JSON configuration file with group-specific parameters in order to run correctly. The call to `DFF_Init` at `line 4` in Listing 3.1.2 takes care of identifying the *dgroup* name to execute and to gather all the necessary parameters from the provided JSON configuration file.

Listing 3.1.3 provides the JSON configuration file for the example in Listing 3.1.2. The file allows to provide various information, like: the protocol (TCP or MPI) to be used for all the communications between the defined *dgroups* (defaults to TCP if not provided); the mapping of each *dgroup* to a specific host; the number of messages to be "batched" before sending; other non-functional parameters like thread mapping and the size of the logical queues between *dgroups* (`messageOTF` parameter). The JSON configuration file allows flexibility in the configuration of the various *dgroups*, since it allows to define minimal configuration if no specific parameters are needed, or additional parameters can be provided for the specific host in order to tune its behavior. All parameters have default values, thus very simple configurations of the various *dgroups* do not require much effort from the user in order to set up and run the application.

```
1  {
2      "protocol": "TCP",
3      "groups": [
4          {
5              "name" : "G1",
6              "endpoint" : "compute1:8001",
7              "batchSize": 10,
8          },
9          {
10             "name" : "G2",
11             "endpoint": "compute2:8002",
12             "messageOTF": 64
13         },
14         {
15             "name" : "G3",
16             "endpoint": "compute3:8003",
17             "batchSize": 20,
18             "threadMapping" : "0,2,4,6,8"
19         },
20         {
21             "name" : "G4",
22             "endpoint": "compute4:8004"
23         }
24     ]
25 }
```

Listing 3.1.3: JSON configuration file for the sample distributed application. Each group can be configured separately with host-specific configurations.

3.1.2 Data serialization

Working in a distributed environment requires for connected nodes to be able to transfer data in a format that can be reconstructed after being sent through the network. The (de)serialization process transforms data-structures in a suitable format, which can be stored or transmitted, for later reconstruction in a (possibly) different environment. For this reason, the distributed FastFlow RTS includes two different data serialization mechanisms: the first one employs the *Cereal serialization library* [31], which can automatically serialize base C++ types as well as compositions of C++ standard library types; the second mechanism allows the user to specify his (de)serialization routine for memory contiguous data types in order to avoid extra data copies needed by the serialization process itself. Examples for both apporaches are shown, respectively, in Listing 3.1.4 and Listing 3.1.5.

```
struct data_t {
    std::string message;
    int id, count;

    template<class Archive>
    void serialize(Archive& ar) {
        ar(key, id, count);
    }
};
```

Listing 3.1.4: Cereal serialization function for `data_t` data type.

```
struct data_t {
    char message[MAXWORD];
    int id, count;
};

template<class Pair>
void serialize(Pair& p, data_t* d) {
    p = {(char*)d, sizeof(data_t)};
}

template<class Pair>
void deserialize(const Pair& p, data_t* d) {
    d = new (p.ptr) data_t;
}
```

Listing 3.1.5: Manual serialization function for memory contiguous `data_t` data type.

3.1.3 Communication nodes

In a distributed FastFlow application, the communication nodes act as gateways for information flow between distributed groups and are the only nodes in charge of sharing data across the network. A distributed group is implemented as a farm, hence, the only nodes that communicate with other groups are the *Emitter* and *Collector* nodes. They represent, respectively, the distributed group *Receiver* and *Sender*. Communication nodes are internally initialized with information regarding the reachable internal nodes of each group they are connected to, and they use these information to build local routing tables. The routing table was originally constructed by an handshake process. In each *dgroup*, the local receiver and sender cooperated with the remote groups in order to build the routing table. The receiver built its routing table by knowing the original data-flow graph and by inspecting the attached neighbors. The sender built its routing table by merging the information received by the remote groups and the relative socket channel created upon the initial handshake. In the last version of FastFlow, the handshake process has been simplified and it is now replaced by a static procedure which

builds the exact same objects that were generated by the handshake, passing them as an argument to the *sender* nodes. In this way, the nodes only exchange the *group name* in their handshake procedure in order to associate a specific communication channel with a given *dgroup*.

The receiver starts listening for input elements over a TCP or an MPI connection at the address specified in the JSON configuration file. Both TCP and MPI receivers support multiple connections with a single-threaded approach. Thus, a receiver will perform a loop to listen for incoming messages and serve them as soon as they are received, until all nodes in the network terminate. The received messages are forwarded, following the information stored in the header file of the message, to the correct recipient which will unpack the message and perform the required operations. The sender, on the other hand, collects from internal nodes the messages addressed to another *dgroup*, which can be an external group (like those generated by vertical cuts) or an internal one (the ones generated by horizontal inter-sets cuts). By means of the message header, the sender forwards the message to the correct *dgroup* by using the routing table it built at startup.

Communication protocol

The communication protocol defines a unique low-level message for all the communications between *dgroups*. The message is composed by a header and a payload containing the serialized data, both in *Network Byte Order*. The header is structured as follows:

- Sender ID: `int`, the ID of the sender node in the original data-flow graph;
- Channel ID: `int`, the ID of the recipient in the original data-flow graph;
- Channel Type: `bool`, used to discriminate the channel type used for the communication, *external* or *feedback*;
- Size: `long`, specifies the size of the payload which follows the header.

The communication protocol also defines the necessary condition for the application termination. As for shared-memory applications, the termination is relative to the reception of EOS messages in all the input channels. Between distributed groups, EOS messages are represented by messages of size 0. Each sender waits to receive an EOS from all its input nodes (termination condition in shared-memory) and subsequently forward the distributed EOS message to all the connected *dgroups*. The receiver waits for an empty message (EOS) and forwards it in the usual way to the local nodes. Both *Sender* and *Receiver* nodes manage EOS messages by dividing them into *internal* EOS and *external* EOS, which are received by groups generated, respectively, by an horizontal inter-sets cut of the same parent `PBB` or by a vertical cut. This distinction is needed in order to correctly terminate distributed applications with mixed shared/distributed-memory communication channels at the same level of the main application pipeline (e.g. groups “G2” and “G3” in Figure 3.1.1b).

3.1.4 Intermediate representation

The functionalities described in Section 3.1.1 are part of the high-level interface offered to the application programmer to build a distributed application. Internally, the runtime system, via the `DFF_Init` call and the distributed `BB`s, translates group creation and inclusion operations into an *intermediate representation*. This representation of a distributed program contains all the RTS building blocks needed to build distributed communications functionalities, like serialization/deserialization, routing of messages from/to distributed groups, channel type definition, and networking operations, to name a few. In the following, in order to give an overview of the entities participating in a distributed group, and to better understand the extensions introduced by our thesis work to the RTS level, we describe some of the RTS nodes automatically inserted in a *dgroup* in order to allow distributed communications. For the sake of clarity, we reprint in Figure 3.1.2 the distributed graph of the sample application provided in Listing 3.1.2, which will be our case-study to present the internal structure of a *dgroup*.

In Figure 3.1.4a, we show the internal structure of *dgroup* "G2". As we can see, a *dgroup* is internally extended by the runtime with additional nodes to enable distributed communications and the correct routing of messages directed to, and coming from, the connected *dgroups*. As reported in Figure 3.1.3, the internal representation of a *dgroup* contains three categories of nodes: i) sequential nodes of the original application; ii) network-enabled nodes, implementing distributed communications and receiving/forwarding tasks from/to the network; iii) helper nodes, which are RTS nodes used to correctly route tasks internally to the *dgroup*. There are additional RTS nodes in the internal representation of the actual distributed application, but they are mostly related to forwarding and serialization/deserialization functionalities, so we ignore them in this discussion for the sake of simplicity.

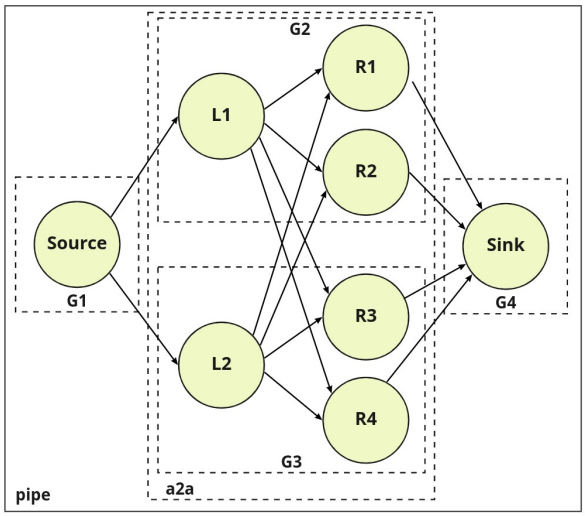


Figure 3.1.2: Distributed-memory concurrency graph for the sample application in Listing 3.1.2

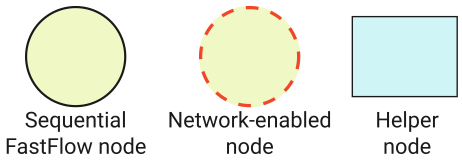
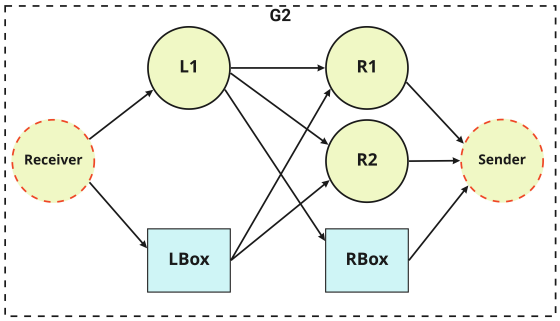
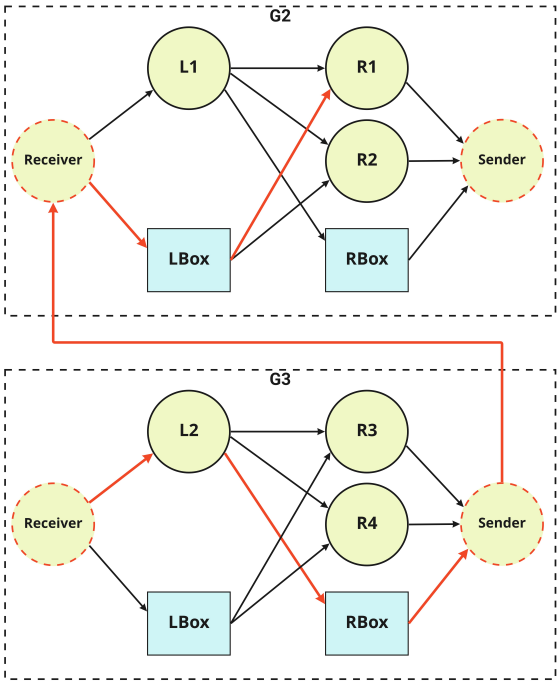


Figure 3.1.3: Nodes legend for the internal representation of a distributed group.

When talking about a *dgroup* we refer, from now on, to its internal representation containing network-enabled nodes, the `Receiver` and the `Sender`, as well as helper nodes, like the `LBox` and `RBox`. The `LBox` and the `RBox` act, respectively, as "emulators" of additional nodes in the original application at their level of depth in the top-level building block. In our example, the `LBox` in "G2" is at depth=1 in the original `a2a BB`, so it acts as a substitute for tasks coming from the `L2` node in the "G3" group. The `RBox`, respectively positioned at depth=2, emulates, for "G2", the nodes `R3` and `R4` in "G3". In this way, a distributed group is able to redirect tasks which are not following an "horizontal" direction, but they rather flow "vertically". Depending on the original application partitioning into *dgroups*, we can identify two kind of communication channels: i) *internal*, which connects *dgroups* generated via an *horizontal inter-sets cut*; ii) *external*, which connects *dgroups* created by a *vertical* or *horizontal non-inter-sets cut*. Messages flowing in these (logical) communication channels are referred to, respectively, as *internal messages* and *external messages*.



(a) Internal structure of a distributed group in the original FastFlow implementation.



(b) Internal message example, showing the role of LBox and RBox nodes.

Figure 3.1.4: Internal representation of *dgroups* with RTS nodes participating in communications.

We give an example of an internal message in Figure 3.1.4b. In this example, we depict the situation where the `L2` node forwards a message to the `R1` node. The distributed RTS node `RBox`, emulating both `R1` and `R2`, gathers the message and forwards it to the `Sender` node. A message coming from the `RBox` is necessarily an *internal message*, so the `Sender` forwards it to the correct `Receiver` using an *internal* channel. Once the `Receiver` receives the *internal message*, it directly forwards the message to the `LBox` which internally routes it to the correct recipient.

In the following sections, we describe the implemented extensions at the *intermediate representation* level for the network-enabled RTS nodes. We provide a description of the goals and design choices which guided our journey toward the introduction of multi-protocol functionalities in the distributed runtime of FastFlow.

3.2 Multi-protocol extension goals

The user may want to deploy her application in two completely different computing environments, maybe putting generic source and sink nodes outside the *dgroups* which are actually doing the computation in a cluster environment. This can represent a limitation for a communication layer with a fixed communication protocol for the entire application. This is mainly because cluster HPC environments often rely on high-performance fabrics (e.g., InfiniBand) whose protocols are not supported by generic network interface controllers. In a cloud environment, where data streams can be generated even at the edge of the network by IoT devices, there is a need for implementing support to differentiated communication protocols while retaining the ease of use of a generic programming interface. For this reason, the design of a multi-protocol interface should be targeted at providing a general, extensible and portable API for heterogeneous inter-network communications. Hence, our thesis work aims at extending the existing distributed runtime of FastFlow framework, adding support for multi-protocol communications between distributed groups and effectively enabling heterogeneity of computing nodes. As we already introduced, the main aim is to provide enough flexibility to the application developer to specify a multitude of transport protocols for each of the communication channels in the distributed application graph. A distributed FastFlow application involves communications at the edges of each *dgroup*, but in the current version of the distributed run-time of the FastFlow framework, only a unique transport protocol can be used for all the distributed communications happening in the application graph. This can be seen in Listing 3.1.3 at `line 2`, where currently allowed values are `"protocol":"TCP"` and `"protocol":"MPI"`.

Our goal is twofold: i) equip the distributed runtime with functionalities for multi-protocol communications, providing a simple mechanism for the application developer to select the most suitable configuration for her application; ii) provide an easy-to-use high-level interface to the runtime system developer in order to allow extensibility to additional protocols and communication libraries without many code changes.

Hence, in the following sections we provide a description of the design choices and the extensions we introduced in FastFlow's distributed runtime.

3.3 Design choices

As we show in Section 3.6, with the provided extension an application developer can specify, via the usual JSON configuration file, the transport protocols that single partitions of the distributed application will use for their communications. Moreover, the introduced software modules provide a simple high-level interface and are extensible in order to allow the development of additional drivers for non-included networking protocol. By following a "composition-over-inheritance" approach, we separate the behaviour of FastFlow nodes from the implementation of the multi-protocol management strategy and the underlying networking functionalities. Hence, we designed two layers of abstraction providing, respectively, multi-protocol managers and network plugins implementing transport-specific functionalities.

The development of the multi-protocol support for the FastFlow distributed runtime required an initial experimentation phase with the existing frameworks, in particular with Mercury and Margo, described in Section 2.3.1. This was helpful to structure the general architecture of the software modules following a multi-protocol oriented approach, later replacing the unique Margo component with manually-crafted protocol-specific components. We list here the main design choices we made in order to ease the development and testing process of the various components.

Intermediate representation - As described in Section 3.1, the creation of a distributed application is a process which involves calls to high-level APIs provided by FastFlow. The high-level operators `createGroup` and the inclusion operator "`<<`" allow the runtime to generate a properly structured intermediate representation of the original distributed program. It includes all the necessary wrappers and RTS nodes to perform serialization, communication between *dgroups*, message (de)multiplexing and internal routing. This internal representation is completely transparent to the application developer, but it is necessary to the runtime in order to properly construct the final distributed version of the user application. In order to simplify the development process, we directly worked at this intermediate representation where RTS nodes interacts to allow the correct communication between *dgroups*. For this reason, we do not rely over high-level interface of distributed FastFlow, and we manually configure, deploy and run each part of the distributed application graph separately. This allowed us to create and test, individually, different components manually plugged in the intermediate representation of distributed applications.

Separation from Margo - During the development process, the Margo component served as a prototyping multi-protocol model. However, since the offered support for widely accepted protocols like TCP and MPI is very limited, we decided to implement our multi-protocol high-level component and only use Margo as a "guideline" for the implementation of the final multi-protocol functionalities. Moreover, since Margo internally relies on a multi-threaded model, it was very difficult to pair it, in a straightforward way, with our own transport components. For this reason, and to simplify the development of the final communication abstraction, we use the Margo component without pairing it with other transports.

FastFlow nodes, Managers and Plugins - We decided to structure the implementation of networking functionalities at two levels, completely removing the dependency of *Receiver* and *Sender* nodes over a specific protocol. In particular, our extension required the definition of: i) *Communication Managers* (*Manager*, in the following) which represent the high-level components providing multi-protocol support and allowing the orchestration of the internal transport-specific components; ii) *Transport Plugins* (*Plugin*, in the following) which provide protocol-specific functionalities to send and wait for data in a non-blocking fashion. The currently developed components offer TCP and MPI functionalities.

***Receiver* Delegation Strategy** - Just like Margo, each of the *Transport Plugins* require some data to be "registered" in order to allow the execution of the `ff_node_t` methods for each of the received tasks. The registration takes place only at the *Receiver* node and it allows the correct forwarding of tasks to the nodes in the local group. In Section 3.4.2, we describe the registration process for the Margo component.

Non-blocking components - We decided to use a single-threaded model where the main thread, running the `svc()` method, drives progress by delegating control to the high-level *Manager*. In particular, a *Receiver* delegates control to the *Manager* which polls its internal *Plugins* until the entire application terminates. The polling is performed by calling the `receive` method of a *Plugin*, which is a non-blocking implementation of the classical `select/poll` system call (for TCP) and `MPI_Iprobe` (for MPI), that returns a specific return code indicating whether data has been received or if the communication has been closed. The non-blocking nature of the `receive` operation is a strict requirement to be fulfilled if an extension with further protocol is provided. In this way, the *Manager* can drive progress by successive polling of all the internal components, effectively listening over multiple protocols. On the other hand, a *Sender* node relies over the *Manager* for each one of the received tasks in its input channels. The *Manager* forwards the task to the correct *Plugin* and returns control to the *Sender*, when the network transfer is over.

3.4 Multi-protocol software components

Our thesis work focused at extending communication functionalities between *dgroups*, and since we worked directly at the internal representation of a distributed application, the provided extensions target the RTS network-enabled nodes in order to enable multi-protocol support in a FastFlow distributed application. In this section we present the introduced software components targeting the communication classes in the original FastFlow distributed runtime. With our extension, we want to bring networking capabilities at a lower level of abstraction, refactoring existing classes at the edge of the *dgroup* in such a way that they are completely agnostic of the used protocols for the underlying communications. For this reason, we introduce a generic class for the receiver and for the sender, namely `ff_dMPreceiver` and `ff_dMPsender`, which are in-

jected, respectively, with a `ReceiverManager` and a `SenderManager`. The introduced *Manager* classes allow to orchestrate the components used for the actual communications between *dgroups*, the *Plugins*. Each of the receiver and sender class, as in the original FastFlow implementation, is also extended by *internal* versions, which are needed in order to manage internal connections taking place in a distributed channel because of inter-sets cuts.

In the following, we present the modules which have been developed to implement multi-protocol functionalities in existing FastFlow distributed applications. We provide a description of the two layer of components used to provide protocol-specific implementations and management of the multi-protocol functionalities.

3.4.1 Manager Components

The *Manager* classes, namely `ReceiverManager` and `SenderManager` are delegates of the *Receiver* and *Sender* stages respectively. This is needed in order to make FastFlow nodes, which have to deal with forwarding and termination, completely agnostic of the networking functionalities which the user may select at runtime. In the original distributed version of FastFlow there were pairs of `ff_node_t` classes for each of the provided protocols, like `ff_dreceiver` and `ff_dsender` for TCP, and `ff_dreceiverMPI` and `ff_dsenderMPI` for MPI. Since these classes both extend the `ff_node_t` class, only one receiver and one sender could be present in the concurrency graph. For this reason, in order to enable the possibility of using multiple protocols support at the edge of a *dgroup*, we had to bring networking functionalities at a lower level, refactoring the *Receiver* and *Sender* classes into generic (under a protocol point-of-view) FastFlow nodes containing a reference to a *Manager* object. Our `ff_dMPreceiver` and `ff_dMPsender` nodes are only responsible of local forwarding to nodes like business-logic nodes or RTS boxes, leaving communication responsibilities to the *Manager*s and *Plugin*s.

The *Managers* functionalities are strictly tied to the original *Receiver* and *Sender*, since they basically implement the FastFlow communication protocol. In order to simplify development, we decided to retain the logic of the FastFlow's communication protocol inside the implemented components. Building a generic multi-protocol extension would have required the implementation of basic and independent network primitives used as base calls to re-implement the original communication protocol, requiring much more time. For this reason, the *Manager*s are divided into two hierarchies, one for the *Receiver* and one for the *Sender*. They participate in the realization of the communication protocol by means of orchestrating the different *Plugins*, playing different roles in the communication protocol necessary to connect pairs of *dgroups*, as described in Section 3.1.3.

Both *Manager*s are initialized with a set of transport-specific components, described in Section 3.4.2, which are used to receive and send data by following the mapping generated by means of a statically-constructed and local routing table. We show in the following the functionalities offered by the two *Managers* and how they enable multi-

protocol support to the FastFlow nodes at the edge of the *dgroup*.

ReceiverManager - Enables multi-protocol support in the receiving phase. It is plugged into a `ff_dMPreceiver` and allows the reception of input tasks from connected *dgroups* with a polling strategy over the contained transport-specific components. The *Manager* is agnostic of the type of protocol used in each of the components, and it does not require any additional information for its operation. The polling follows a round-robin strategy, calling the *receive* method of each component whose return code specify to the manager if the listening was successful or not. In both cases, the manager continues with the polling of other plugged components, but flags terminated components to stop polling over them at the next polling pass. Each of the internal components can configure the timeout for a receive call. For example, our TCP component uses a timeout of 100ms in the `select` call in order to return control to the manager if no message is received in that amount of time. Following this strategy, the `ReceiverManager` continues polling the components which are not yet finished and only stops when all the transport components signaled a closed connection.

SenderManager - Allows the `ff_dMPsender` node to send data to other connected *dgroups*. As the `ReceiverManager`, it contains a set of transport-specific components which are used to send tasks across the network, but in contrast to the `ReceiverManager`, it follows a specific mapping between the message recipient and the component which should be used to ship it through the network. In fact, upon initialization, transport-specific components must be paired with the name of the *dgroup* they are going to communicate with. This will allow the *SenderManager* to pair each component with the information contained in its routing table. This is because receiving a task with a specific `channelID` involves only one of the components based on the user-specified mapping between *dgroups* connections. If the task has no `channelID`, the `SenderManager` uses a round-robin strategy to select the next component based on the type of task, namely *internal* or *external*. Moreover, the `SenderManager` allows the `ff_dMPsender` to forward EOS messages to both internal and external *dgroups*.

3.4.2 Transport-specific Components

Transport Plugins, just like *Manager*s, belong to two disjointed hierarchies, namely `ReceiverPlugin` and `SenderPlugin`, to separate functionalities offered to receive and send messages between *dgroups*. The *Plugins*, with the orchestration provided by their *Managers*, participate in the realization of the FastFlow communication protocol using a specific transport protocol, such as MPI, TCP or Margo. The interface for the receive functionalities must be non-blocking in order to allow polling in the *Manager*. If the *Plugin* does not provide non-blocking receive calls, like Margo, it can't be paired with other *Plugin* classes for the multi-protocol approach provided by the *Manager* objects. However, as it happens with Margo, even if a transport component is blocking, it could anyway internally provide multi-protocol functionalities. An alternative could be having the *Plugin* running on its own thread, in order to emulate the non-blocking behavior of other components, providing the main polling thread with data upon reception.

The *Transport Plugins* provide, through composition, specific networking functionalities to the upper layers, in this case the *Manager* objects. Each of the implemented *Plugin* specify its own implementation, using transport-specific functionalities, of the communication protocol defined by FastFlow distributed runtime. The choice between the transports to use to realize the FastFlow communications can be made at runtime, since *Plugin* objects are passed in a vector to the *Manager* upon initialization. The information provided to the *Manager*s, like the complete routing table, and the mapping between group names and the specific *Plugin*, allow them to correctly select the specific transport component for a given communication. Internally, each *Plugin* acts in a completely transparent way to the other transport components. In this way, the execution and termination of a distributed application is a joint operation between all the connected *Plugin*s. Each one of them participate in the execution of its own part of the network communication and when all the network-partitions are flagged as terminated, the whole application graph can gracefully shutdown and signal this situation via the *Receiver* and *Sender* nodes to the other local nodes in the *dgroup*.

Margo Plugin

The *Margo Plugin* represented the starting point of our thesis work. It provides support to a set of transports, as described in Section 2.3.1, and can therefore be used as a standalone multi-protocol component when wrapped by a *Manager* object. Unfortunately, we were not able to test the support for all the Margo-provided transports since we do not have at our disposal the underlying network fabrics for the high-performance protocols implemented in the Margo framework. Hence, we can't state anything about the support for those plugins. The only plugins which we were able to use are the `ofi+sockets` and the `ofi+tcp`. Given the internal structure of the Margo framework and its parallel nature, it was very hard to pair this component with non-blocking receive functionalities, for this reason we do not provide support for our multi-protocol strategy while using this component. Hence, when using the *Margo Plugin*, the user can only rely over the multi-protocol functionalities provided by the component itself, which are very limited.

As we described in Section 2.3.1, the process for using the Margo framework as a communication service is composed of different steps. In the following we describe the specific steps for *Margo Plugin* implementation.

RPC input/output types - We defined two different data types to pass as input parameters to the RPCs used to communicate between *dgroups*. They are `ff_rpc_in_t` and `ff_rpc_shutdown_in_t`, that are used, respectively, for the procedures of *forwarding* and *shutdown*. The `ff_rpc_in_t` allows to ship tasks between *dgroups*, carrying all the necessary information to let the application proceed as intended. It further includes an additional field to discern between internal and external messages. Since this data type is a composition of non-standard data types, its serialization must be handled manually. The `ff_rpc_shutdown_in_t`, instead, is a one-field struct type and includes only the information to specify an internal or external EOS. It is defined as a `bool`, thus the serialization is handled automatically by Margo.

RPC callbacks - Two RPCs are defined in the *Receiver* nodes, namely `ff_rpc` and `ff_rpc_shutdown`. They allow to forward tasks, logical EOS and actual EOS messages between *dgroups*. At RPC callbacks initialization, the *Receiver* node is registered as internal callback metadata. Upon callback execution, the *Receiver* is retrieved and its *forward* and *shutdown* methods are used to forward tasks and signal EOS messages to the nodes of the local *dgroup*. The RPC callbacks take as input the described data types and internally use the registered *Receiver* node to properly forward tasks in the local group.

Endpoint class extension - The original FastFlow distributed runtime relies over a `ff_endpoint` datatype to store information about distributed groups, like address and port number for the TCP protocol, and rank for the MPI one. We further extended the information provided by these objects by creating the `ff_endpoint_rpc` subclass to store the additional information required by the Margo framework. In particular, they are related to the management of the plugins configuration strings in Margo.

Execution model - As we described in Section 2.3.1, Margo is natively multi-threaded and it spawns internal Argobots threads to handle the progress loop of the generated instances. In Margo, the amount of Argobots threads and pool to be used for each RPC callback can be tweaked at initialization phase. However, our *Margo Plugin* uses only one Argobots thread to drive the internal progress loop and execute RPC callbacks.

TCP and MPI Plugins

The TCP and MPI *Plugin* classes are adaptations, with a proper component structure, of the original `ff_dsender/receiver` and `ff_dsenderMPI/receiverMPI`. We refactored the already implemented networking functionalities in the FastFlow distributed runtime and ported them in a suitable component representation. They represent the core of our multi-protocol implementation as they implement a non-blocking receive interface to provide proper polling at the *Manager* level. The *Plugin*s implement a generic interface which provide a restricted set of methods, like *initialization*, *receive*, *send* and *finalization*. Also in this case, we decided to keep the *Sender* and *Receiver* interfaces disjointed to ease the implementation and keep functionalities well separated. Separating functionalities for sender-related and receiver-related components allowed us to easily structure and distribute the various phases of the FastFlow communication protocol, described in Section 3.1.3, between *Receiver* and *Sender Plugins*.

The only changes required to implement the TCP and MPI *Plugin* classes are relative to the refactoring into proper components implementation and changing the receive model into a non-blocking one. We show, in the next section, an example of their usage to construct a distributed application at the RTS level, mixing both transports to connect different groups of the distributed application.

3.5 Multi-protocol communication model

As we described in Section 3.4.1, the *Manager* objects inside the `ff_node_t` at the edge of each *dgroup* realize the original FastFlow distributed communication protocol by orchestrating the various internal *Plugin*s. In this section we show how the communication protocol has been revised in order to not disrupt the communications between pairs of *dgroups* and to make as transparent as possible to each transport component the presence of other communication channels in the same *dgroup*.

3.5.1 Changes to routing information

The main changes are related to the routing information provided to each one of the transport-specific *Plugin*. The new fine-grained subdivision of the communication channels required a different management of the routing information in the *Plugin*s. In fact, in the original single-protocol version of the distributed runtime, the entities representing the *Sender* and *Receiver* nodes were the same entities also managing the communications in all the assigned communication channels. Instead, with the introduction of channel-specific protocols, a new way of handling the routing information has been introduced.

In a distributed application, the two information that allow proper routing of messages and termination of the application are, respectively, the routing table and the amount of input channels to a given *dgroup*. The *Receiver* keeps track of the amount of input connections, the *Sender*, instead, uses the routing table to forward the received tasks to the correct recipient in another *dgroup*. In our extension, we had to rethink the way these information are handled by the internal *Manager*s and the underlying *Plugin*s. Since in the original distributed runtime the *Receiver* and *Sender* nodes were also responsible to communicate with other *dgroups*, it was not necessary to keep track of the individual connections separately. In the provided extension, instead, each of the transport-specific *Plugin* is responsible for its own routing table and the amount of channels which it is connected to. For this reason, based on the information specified for each of the internal components, the *Manager* forwards the correct partition of the general routing information to the given *Plugin*. In this way, each *Plugin* is only aware of the channels internal to its connected *dgroups* and it's completely agnostic to the rest of the application graph. Termination can be handled locally to each of the transports and it's the *Manager*'s responsibility to correctly terminate when all the connections have been closed and forward an EOS message.

3.5.2 Construction of a multi-protocol application

In this section we provide the usual example application with runtime code showing how the internal groups construct the *Receiver* and *Sender* nodes, paired with their *Manager*s and the internal *Plugin*s. We show in Figure 3.5.1 the protocols used for each of the distributed channels and we provide in Listings 3.5.1 and 3.5.2 one possible solution to realize this specific distributed concurrency graph. We hide all the other RTS wrapper nodes, as well as `LBox` and `RBox`, in order to show the RTS classes targeted

by our extension. In particular, we provide two different applications, one for the MPI groups ("G2" and "G3") and one for the TCP-only groups ("G1" and "G4"). This could be realized also with a single application launched via the `mpirun` command, but having two versions allow us to deploy and launch each part separately, easing the situation where the TCP-only *dgroups* are not in the same cluster environment as the *dgroups* running with MPI. This particular situation will be also analyzed in Chapter 4, were we show an example with the "G1" and "G4" *dgroups* running over a personal laptop and "G2" and "G3" running in a cluster using the MPI transport.

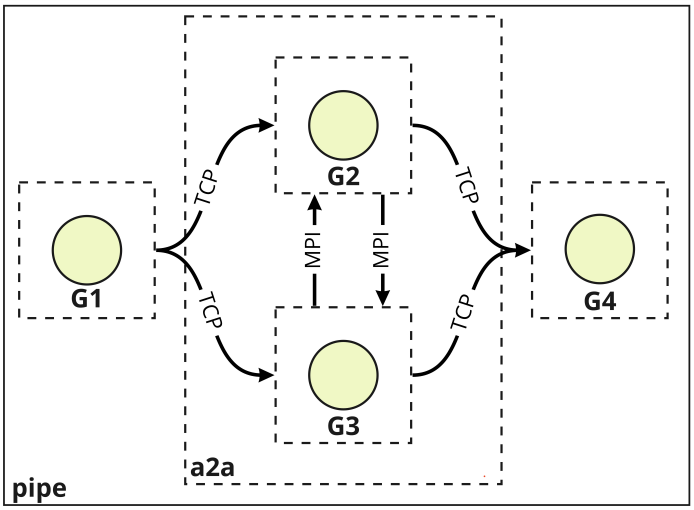


Figure 3.5.1: Example of a multi-protocol application using heterogeneous protocols for pairs of communication channels.

As we can see from Listing 3.5.1 at `lines 17-23`, the creation of a *Sender* node with a single protocol functionalities is very simple, as it only requires the specification of the list of endpoints to connect to and their channels type. Since the "G1" group is only connected to *external* groups, the `ChannelType` is of type `FWD` for both the connections. The construction of the "G4" receiver at `lines 28-31`, using only the TCP protocol is simple as well, we only need to specify the listening endpoint together with the number of input channels and add it into the group emitter via a *Receiver* node. Note that the *Sender* needs to specify the endpoints of all the nodes it connects to and its *Manager* also needs the routing table with the information of the local nodes for each of the connected *dgroups*, instead the *Receiver* only needs to specify how many groups will connect to it during the whole execution.

```
1 int main() {
2   ...
3   // TCP endpoints, specifying both address and port
4   ff_endpoint g2("ip-of-G2", 42000);
5   g2.groupName = "G2";
6   ff_endpoint g3("ip-of-G3", 42001);
7   g3.groupName = "G3";
8   ff_endpoint g4("ip-of-G4", 42002);
9   g4.groupName = "G4";
```

```
10
11  ff_farm gFarm;
12  std::map<std::pair<std::string, ChannelType>, std::vector<int>> rt;
13  if (myID == 0){ //Group G1
14    rt[std::make_pair(g1.groupName, ChannelType::FWD)] = std::vector({0});
15    rt[std::make_pair(g2.groupName, ChannelType::FWD)] = std::vector({1});
16
17    SenderManager* sendMaster = new SenderManager(
18      { {{g2.groupName, g3.groupName},
19          new SenderPluginTCP(
20            {{ChannelType::FWD, g2}, {ChannelType::FWD, g3}}, "G1")}
21      }, &rt);
22
23    gFarm.add_collector(new ff_dMPsender(sendMaster));
24
25    ...
26  }
27  else if (myID == 1){ //Group G4
28    ReceiverManager *recMaster = new ReceiverManager(
29      {new ReceiverPluginTCP(g4, 2)});
30
31    gFarm.add_emitter(new ff_dMPreceiver(recMaster, 2));
32
33    ...
34  }
35 }
```

Listing 3.5.1: RTS code for the TCP nodes G1 and G4.

Listing 3.5.2, instead, provides the runtime code for the creation of groups "G2" and "G3" which use multiple protocols for their communication. In this case, we need to plug multiple *Transport Plugin*s in the *Manager*s and specify the endpoints for each of the communication pairs. We only refer to the "G2" group, since the procedure is the exact same also for "G3". We can see at `lines 28-29` the creation of a multi-protocol `ReceiverManager`, which receives two `ReceiverPlugin`s for both TCP and MPI protocols. In this case, opposed to the single-protocol version, each component only specifies as parameter a single input channel (first parameter of `ReceiverPluginMPI`, second parameter of `ReceiverPluginTCP`). For the `SenderManager` object, we specify a vector of pairs where each element gives information about a given communication channel. For example, at `line 32` we specify that a connection toward the group "G4" will be established using the TCP component using a `FWD` channel. The last parameter specifies the name of the local dgroup, needed for the handshake procedure. Since "G2" (and "G3") is an internal group, they use the `ff_dMP<receiver/sender>H` objects. Moreover, at `line 37` we can see how the *receiver* still specifies that he has two input channels. This is needed for the correct termination of the whole application graph.

```
int main() {
  // TCP endpoints, specifying both address and port
  ff_endpoint g2_tcp("ip-of-G2", 42000);
  g2_tcp.groupName = "G2";
  ff_endpoint g3_tcp("ip-of-G3", 42001);
  g3_tcp.groupName = "G3";
  ff_endpoint g4("ip-of-G4", 42002);
  g4.groupName = "G4";

  // MPI endpoints, only specifying rank
  ff_endpoint g2(0);
  g2.groupName = "G2";
  ff_endpoint g3(1);
  g3.groupName = "G3";

  ff_farm gFarm;  // Top level farm with network nodes at the edge
  ff_a2a a2a;     // Internal a2a BB containing application nodes + RTS helpers

  // Routing table with information about local nodes of the connected dgroup
  std::map<std::pair<std::string, ChannelType>, std::vector<int>> rt;
  if (rank == 0){ //Group G2
    // Routing table for G3 local nodes
    rt[std::make_pair(g3.groupName, ChannelType::INT)] = std::vector({1});
    rt[std::make_pair(g4.groupName, ChannelType::FWD)] = std::vector({0});

    // Building sender and receiver managers via set of transports
    ReceiverManager *recMaster = new ReceiverManager(
      {new ReceiverPluginMPI(1), new ReceiverPluginTCP(g2_tcp, 1)}, {{0, 0}});

    SenderManager* sendMaster = new SenderManager(
      { {{g4.groupName}, new SenderPluginTCP({{ChannelType::FWD, g4}}, "G2")},
        {{g3.groupName}, new SenderPluginMPI({{ChannelType::INT, g3}}, "G2")}
      }, &rt);

    // Adding FF receiver/sender at the edge of the top farm BB
    gFarm.add_emitter(new ff_dMPreceiverH(recMaster, 2));
    gFarm.add_collector(new ff_dMPsenderH(sendMaster));

    // Populating internal a2a with business logic nodes + RTS helpers
    ...
  } else if (rank == 1){ //Group G3
    rt[std::make_pair(g2.groupName, ChannelType::INT)] = std::vector({0});
    rt[std::make_pair(g4.groupName, ChannelType::FWD)] = std::vector({0});

    ReceiverManager *recMaster = new ReceiverManager(
      {new ReceiverPluginMPI(1), new ReceiverPluginTCP(g3_tcp, 1)}, {{1, 0}});

    SenderManager* sendMaster = new SenderManager(
      { {{g4.groupName}, new SenderPluginTCP({{ChannelType::FWD, g4}}, "G3")},
        {{g2.groupName}, new SenderPluginMPI({{ChannelType::INT, g2}}, "G3")}
      }, &rt);

    gFarm.add_emitter(new ff_dMPreceiverH(recMaster, 2));
    gFarm.add_collector(new ff_dMPsenderH(sendMaster));
```

```
55      ...
56    }
57  }
```

Listing 3.5.2: RTS code for the nodes G2 and G3 using both TCP and MPI transports.

The launch phase is then performed manually by running the "G1" and "G4" groups on a laptop, specifying for the first group how many tasks to send, like:

```
$ ./test_tcp 1 10000 &
$ ./test_tcp 4
```

Instead, the "G2" and "G3" groups are launched together with the `mpirun` command on the same cluster, like:

```
$ mpirun -n 2 test_mpi
```

3.6 User Interaction for multi-protocol selection

With the new extension for the FastFlow communication classes, we have to necessarily extend the JSON configuration file description which will be used by application developers in order to provide mappings between *dgroups* and hosts, but also to specify the communication protocols which have to be used during the execution of the distributed application. However, this is only a prototyping of what could be the final JSON configuration file. We propose in the following a suggestion for the restructuring of the configuration file in order to allow, with the simplest syntax possible, the specification of host-specific configuration as well as communication-specific ones. In particular, the main aspects that should be considered are related to two main specifications:

- **input connections**: a user must be able to specify which protocols a specific *dgroup* is willing to use for its input connections. In particular, this provides a way to specify, for each host, which are the supported protocol in this particular *dgroup*;
- **output connections**: additionally, a user might want to specify the specific mapping of input-output connections without relying on the runtime system mapping for the provided protocols.

We point out that the *input connections* must be mandatory when a specific *dgroup* wants to use multiple protocols for its inputs. However, the *output connections* can be relaxed by providing a fixed preference list of protocols which are automatically selected at runtime if the user do not specify a preferred mapping. The runtime system can select the most appropriate transport component for the specific configuration, for example two nodes on the same cluster might communicate more efficiently if they both support HPC fabrics like InfiniBand, instead of using the common TCP protocol.

```
{
  "protocol": "TCP",
  "groups": [{
      "name" : "G1",
      "endpoint" : "localhost:8001"
    }, {
      "name" : "G2",
      "endpoint": "localhost:8002"
    }, {
      "name" : "G3",
      "endpoint": "localhost:8003"
    }, {
      "name" : "G4",
      "endpoint": "localhost:8004"
    }
  ]
}
```

Listing 3.6.1: Original configuration file with omitted host-specific non-functional configurations.

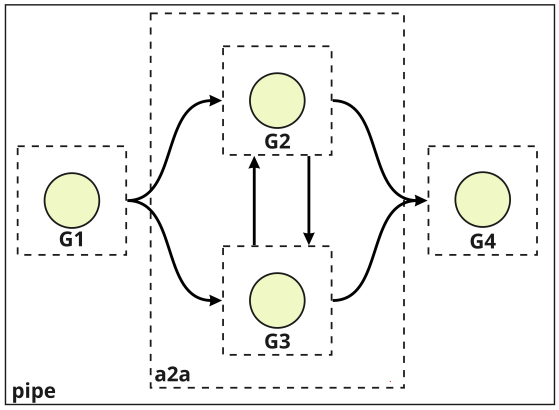


Figure 3.6.1: Simplified distributed application graph for example in Listing 3.1.2.

To give an example for the JSON configuration prototype, we make use of the example in Listing 3.1.2, and we present again its distributed application graph in Figure 3.6.1 where we only highlight the distributed groups and their communication channels. In Listing 3.6.1 we present once again the most basic configuration file which would map the whole application to use a unique protocol, the `TCP` one, as specified at the top of the configuration. This reflects the simplicity of the original configuration file for very basic configuration of a distributed application.

In Listing 3.6.2, instead, we propose a prototype configuration file to specify channel-specific protocols and which introduces a new syntax as well as a new field:

- `"endpoint"` renewed syntax: the `endpoint` field, in case no default `protocol` field is provided, is specified as a JSON string containing a comma-separated list of `protocol:info` specifications, followed by the `://` separator and finally by the `host` node where the group should be launched;
- new field `"out"`: defined as a list of JSON strings, specifies a mapping in order to define the specific protocol to use for a communication channel. Each element follows the format `<group_name:protocol>` .

```
{
  "groups": [
  {
    "name" : "G1",
    "endpoint" : "localhost",
    "out": ["G2:tcp"]
  }, {
    "name" : "G2",
    "endpoint": "tcp:8001,mpi://localhost",
    "out": ["G3:mpi", "G4:tcp"]
  }, {
    "name" : "G3",
    "endpoint": "mpi, margo:ofi+sockets:8002://localhost"
  }, {
    "name" : "G4",
    "endpoint": "mpi, margo:ofi+sockets:8004://localhost"
  }]
}
```

Listing 3.6.2: Proposed extended version of the JSON configuration file for the implemented multi-protocol classes. Host-specific configurations are omitted.

An additional note that can be made about the semantics behind each of the fields is that a multi-protocol-enabled group should list the accepted protocols in the `endpoint` field only if it is going to receive data from a distributed channel. In fact, as we see from Listing 3.6.2, the group **G1** does not need to specify which protocol it is going to listen to, but only which ones it wants to use as an output connection. Otherwise, even the output connections can be left blank and the user can rely completely over the runtime system selecting the best one, given the configuration of connected nodes. Finally, the runtime will parse the provided configuration file and, depending on the specified fields, can instantiate the relative classes for each of the distributed communication channels. Note that if no user-defined rules are provided, the runtime can freely select the most suitable between the available protocols in each node.

As a final remark, having a generalized way of building multi-protocol applications not only simplifies their construction, but also their deployment and execution. In particular, as we also show in Chapter 4, running a multi-protocol application can be a tedious work that can be greatly simplified by a `dff_run`-like module for the multi-protocol extension. Extending the `dff_run` for the multi-protocol functionalities requires, however, additional effort to correctly map the information contained in the JSON configuration file and host-specific launch commands. In fact, having mixed transports like MPI and TCP would require two different ways of executing the application in the different *dgroups*. This represents an important future extension of the work presented in this thesis, since it is necessary if we want to provide an easy way to the user to run multi-protocol FastFlow applications.

Chapter 4

Evaluation

In this Chapter we show the tests we performed to asses the provided extension of FastFlow communication functionalities. All the tests were performed in a single cluster environment hosted by the Green Data Center of the University of Pisa, called *hpc8000*, and composed of 16 nodes interconnected via a 10 Gigabit Ethernet connection. Each of the cluster nodes uses an Intel Xeon E5-2650v3 equipped with two CPU sockets with 10 cores each and two-way Hyper-Threading, for a total of 20 cores and 40 hardware contexts. However, we only used 4 nodes within the cluster to test our applications. We provide, in the following, results obtained by the tests we performed to compare the existing distributed communication classes with the ones we implemented. We aim at showing that the provided extensions do not introduce overhead in existing applications when only one protocol is selected. Moreover, we show a case study where the multi-protocol communications is effectively used to connect a personal laptop and a publicly reachable Virtual Private Server (VPS) with the *hpc8000* machine, using both TCP and MPI protocols.

The testing approach we followed is the same for all the performed tests. Since we worked at the *intermediate representation* of a distributed application, we do not rely over the FastFlow `dff_run` module. In fact, we manually launch each of the distributed groups in different cluster nodes. Each application part selects, via proper parameters, the *dgroup* to be run and additional configurations like amount of tasks to generate size of each of the tasks.

4.1 Performance assessment

All the tests were performed by using the same application. We tested various communication protocols by maintaining the application structure fixed. In Figure 4.1.1 we depict once again the concurrency graph for the application we used as a case study throughout this thesis work. The tests do not use expensive computations in the nodes and no emulation of work is performed whatsoever, hence all the nodes run at full speed to completion. The only parameters we used to assess the overhead introduced by the extension are the number of generated messages and the size of each message,

ranging from 32 Bytes to 64KB. The *Source* node generates "*dummy*" tasks which will trigger the generation of the true tasks by the nodes *L1* and *L2*. Each left node in the `a2a` building block generates, for each message received from *Source*, an amount of tasks equal to the number of right nodes, in this case 4.

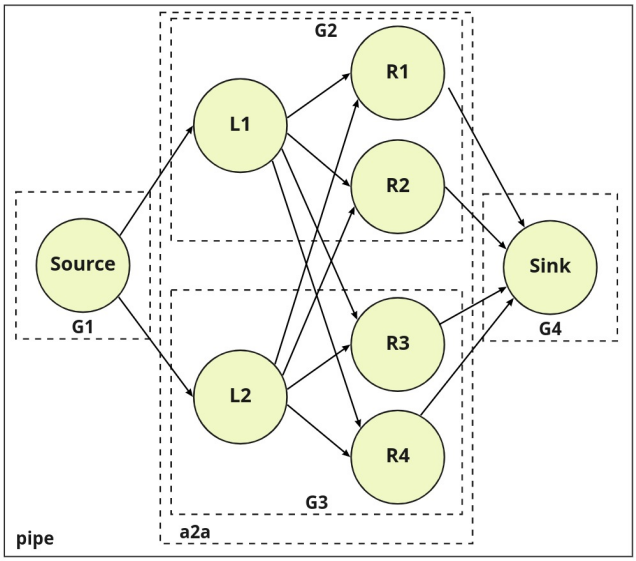


Figure 4.1.1: Sample application used for the testing process.

In the following we aim at showing that no significant overhead is introduced in distributed applications that relies on a single communication protocol. We execute this test both for TCP and MPI connections by manually running each of the *dgroups* in the cluster nodes and selecting the proper transport type. The comparison is performed, for each protocol, between the original single protocol implementation in FastFlow distributed runtime and our multi-protocol extension. Additionally we show the results with the Margo transport component in order to evaluate its performances when using the TCP protocol. Finally, we present a case-study example of the utilization of multi-protocol functionalities between different machines.

4.1.1 Overheads evaluation

In this section we present the tests we performed in order to assess if overheads are introduced by the multi-protocol extension. To do this, we use the implemented transport-specific components selecting only one protocol for all the communications and we used two versions of the same application. The original one, with single-protocol functionalities, is used as a baseline. We then run our multi-protocol version by selecting the same protocol for all the communications. In this way, we can study overheads and performance losses introduced by the additional abstraction layer our implementation relies on.

In order to run the TCP version, we manually launched via `ssh` each of the application partitions in the different cluster nodes. As we can see in Figure 4.1.1, the application is composed of 4 *dgroups*, hence we only used 4 cluster nodes. The MPI version, instead, can be launched by means of the `mpirun` command by providing a `rankfile`, specifying the mapping between different ranks and machines used to run the application.

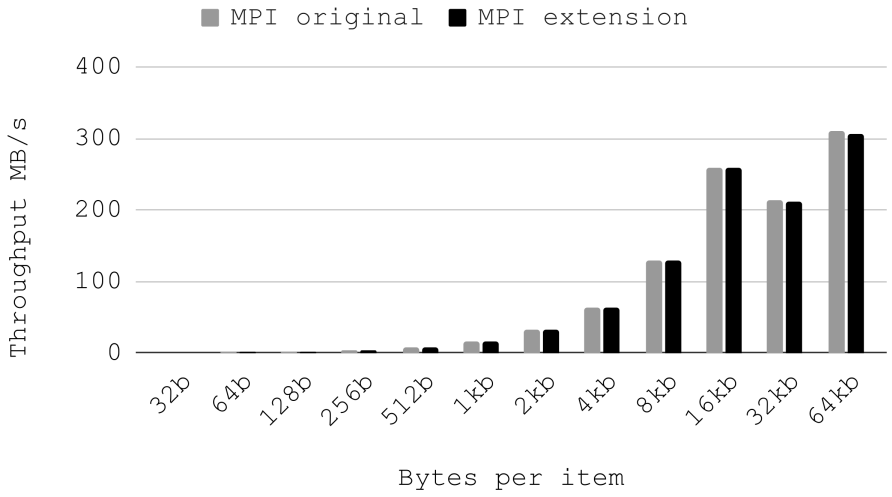
In Table 4.1 we present the execution times of the different runs using both TCP and MPI communications. This test relies on fixed-size messages of 1KB, running up to a total of 400k messages generated by the internal nodes of the `a2a BB`. As we can see, all the executions are comparable, and in particular the multi-protocol extension does not introduce any kind of overhead with this configuration.

Tasks	TCP original	TCP extension	MPI original	MPI extension
40000	2.5325	2.5449	2.5053	2.5458
50000	3.1738	3.1739	3.1749	3.1679
100000	6.3261	6.3413	6.3301	6.3158
200000	12.6835	12.6666	12.6108	12.6066
400000	25.3502	25.4031	25.2550	25.2975

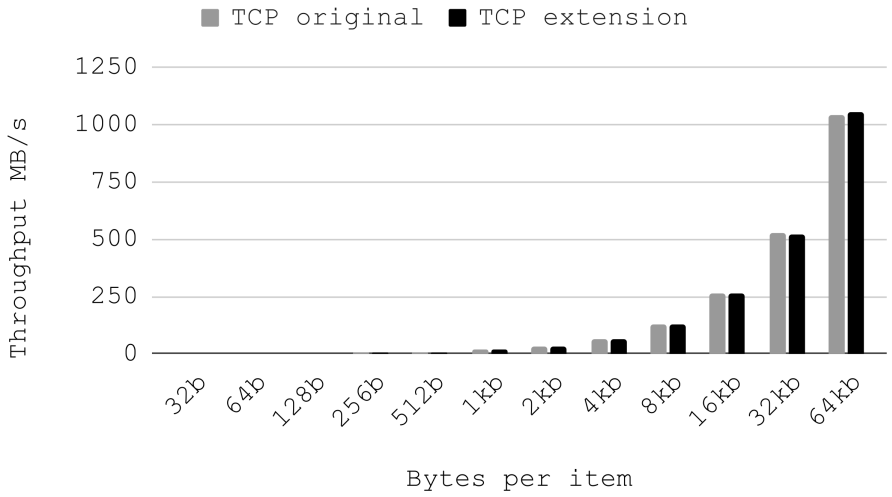
Table 4.1: Comparison of execution times, in seconds, for both TCP and MPI protocols.

Size	TCP original	TCP extension	MPI original	MPI extension
32B	0.521	0.519	0.544	0.549
64B	1.040	1.037	1.062	1.105
128B	2.055	2.041	2.067	2.064
256B	4.091	4.103	4.166	4.207
512B	8.130	8.085	8.202	8.220
1KB	16.141	16.121	16.127	16.232
2KB	32.275	32.235	32.119	32.197
4KB	64.692	64.635	64.294	64.349
8KB	130.413	129.603	128.792	128.840
16KB	261.899	260.332	258.661	258.358
32KB	525.894	522.837	213.422	212.430
64KB	1042.482	1051.243	309.592	306.685

Table 4.2: Throughput results (in MB/s) for the experiments performed with TCP and MPI protocols.



(a) Throughput comparison between original MPI implementation and extension.



(b) Throughput comparison between original TCP implementation and extension.

Figure 4.1.2: Throughput comparison for both MPI and TCP protocols using 100k messages of different sizes, up to 64kb.

To check whether message size influences the performances of the extended communication classes, we tested our applications varying the message sizes, from 32 Bytes up to 64KB, using a total of 100k tasks. In Table 4.2 and Figure 4.1.2 we show the comparison for the throughput achieved by the TCP and MPI tests over increasing message size. As we can see, for all the runs, the achieved performances of our implementation are on par with the original implementation. The achieved throughput is very low for small-sized messages, and it steadily increases together with message size.

However, as can be noted for the MPI transport, throughput decreases at message size of 32KB and increases again with bigger messages. To further investigate this

situation, we analyzed the behaviour with a finer-grained selection of message sizes. For the sake of simplicity, we only report the results for the FastFlow implementation of the MPI transport, since the results of our MPI extension are comparable. In Table 4.3 and Figure 4.1.3 we depict the results for different configurations for the single-protocol MPI implementation of FastFlow. We found out that, with the standard *OpenMPI* configuration, upon reaching the threshold of 20KB messages it records a drop in the amount of messages processed each second. The processed items then stabilize around 5k messages per second, hence the throughput starts increasing again. This can be related to the MPI internal allocation of memory for each of the message sent. MPI offers a lot of parameters to tune the behaviour of the application, thus we experimented with other transports configurations. As we can see, the `ofi+udp` configuration offers slightly better results in terms of messages per second, however the performance drops after a certain message size. The best configuration we have found is the one using the `ob1+tcp` parameter. In this case, as we further show in Figure 4.1.4, the achieved throughput is on par with the "raw" TCP implementation presented in Figure 4.1.2b.

	msg/s		
bytes/msg	**MPI standard**	**MPI ofi+udp**	**MPI ob1+tcp**
16KB	15787	14721	15772
17KB	15745	14738	15718
18KB	15683	13247	15784
19KB	15747	12623	15714
20KB	7320	13034	15770
21KB	7169	12548	15703
25KB	6954	10557	15805
32KB	4724	8782	15803

Table 4.3: Messages per second for the three tested MPI configurations by varying message size.

In particular, the two additional configurations, namely the `ofi+udp` and `ob1+tcp`, refers to the tuning of the OpenMPI's `mca` parameter. We give, for each configuration, the associated configuration parameters to be paired with the `mpirun` command:

- ofi+udp: `--mca pml cm --mca mtl ofi --mca mtl_ofi_provider_include "udp"`
- ob1+tcp: `--mca pml ob1 --mca btl "self,tcp"`

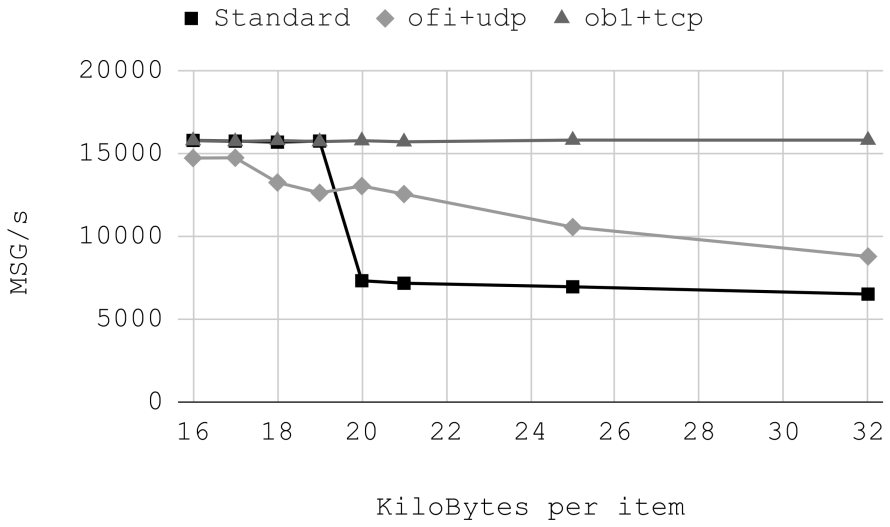


Figure 4.1.3: Messages per second by varying message size for the single-protocol implementation of MPI with three different configurations.

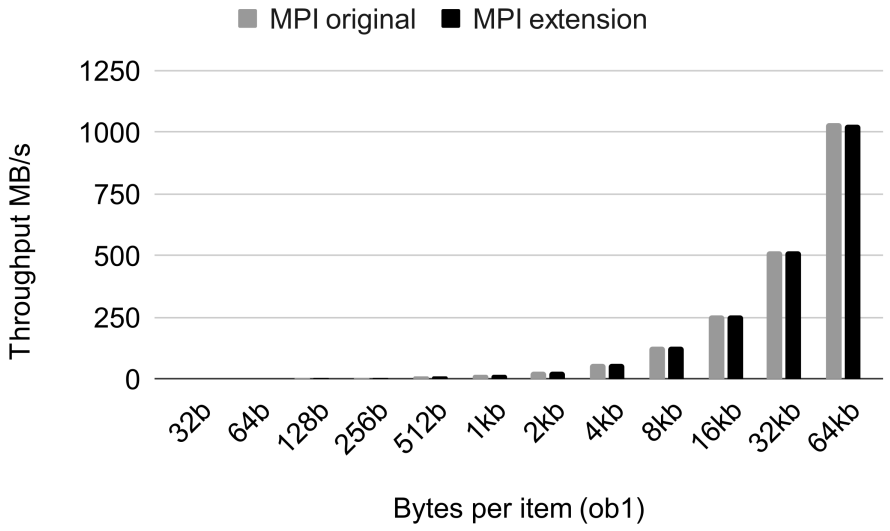


Figure 4.1.4: Best MPI configuration in terms of throughput for both the original MPI implementation and our extension.

This allows us to conclude that the standard configuration does not provide the best achievable performance for our current system architecture, hence extensive care must be given to the configuration phase in order to achieve top performances for each system.

4.1.2 Margo performance assessment

In this section we show the results of the tests when using the Margo transport, comparing it with our implementation of the TCP transport. We only test the Margo instances using the `ofi+sockets` configuration string, since, in all the preliminary tests we performed, this seemed the more reliable and consistent plugin. Also in this case, we manually launch each application part in different cluster nodes, communicating via TCP connections using the underlying OFI [17] driver implemented by Margo. We first assess the completion time for an application using an increasing number of fixed-size messages of 1KB each. We present in Table 4.4 the execution times achieved for the TCP and Margo transports and, as we can see, the TCP implementation is much faster at full speed with respect to the Margo transport implementation. The higher completion times can be explained by the fact that Margo does not use a zero-copy approach to send RPC parameters. In fact, it internally copies each of the messages into network buffers and vice versa on both receiver and sender ends.

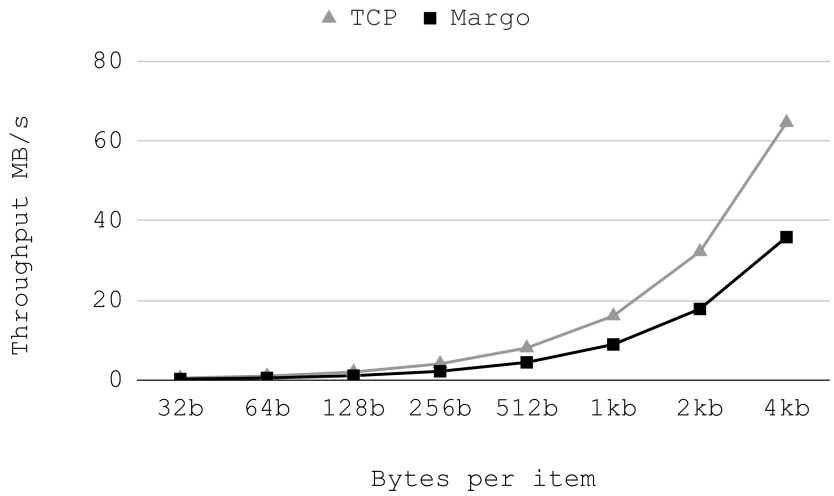


Figure 4.1.5: Throughput comparison for TCP and Margo plugins.

Tasks	TCP	Margo
40000	2.5449	7.6398
50000	3.1739	8.2666
100000	6.3413	11.5098
200000	12.6666	17.6771
400000	25.4031	30.3747

Table 4.4: Comparison of execution times, in seconds, between TCP and Margo plugins.

The exact same reasoning can be made for the throughput achieved with the Margo transport, depicted in Figure 4.1.5. Also in this case, the internal copies performed by

Margo reduces its performance in term of throughput. Moreover, we had to stop at message size of 4KB, since the internal mechanism of Margo allows a maximum of 4KB for the RPC parameters. This value can be increased via the configuration provided by the underlying Mercury framework, but increasing the internal maximum size of accepted messages did not work in our case. Margo developers suggest, in these situations, to rely on bulk functionalities offered by the Mercury framework, but our transport implementation does not provide support to such a functionality.

4.2 A multi-protocol case study

In this section we present a case study to show the usefulness of having a multi-protocol support in a distributed environment. The main aim is to show that, in case one part of the application is executed on machines not providing support for HPC-oriented protocols, we can anyway distribute the application in distinct computing environments and rely both on high-performance protocols and standard ones, like TCP. In our case, the choice of the TCP protocol was mandatory for those nodes which are outside of the *hpc8000* cluster. This is because the nodes of the *hpc8000* machine are not reachable from the outside, hence we had to use SSH tunneling in order to correctly connect the two parts of the application, and this is possible only via a TCP connection. However, we had to restructure the application graph as depicted in Figure 4.2.1, since only one node (*node0*, in the following) is reachable via SSH tunneling. Hence, we used *node0* as a broker node between our personal laptop and VPS, which act, respectively, as a *Source* and *Sink* for the whole application. The nodes inside the cluster are connected by using an MPI connection, instead connections from and to the network are performed using TCP connections.

The deployment and execution is, also in this case, manual and required additional steps. The laptop, running node *G0*, communicate via an SSH tunnel to the *hpc8000* machine. The reachable node of *hpc8000*, *node0*, is running two broker FastFlow nodes inside the cluster, namely *G1* and *G4*. Their purpose is to receive and forward messages using the network interface. The internal `a2a` building blocks of the original application are running inside *node1* and *node2* of the *hpc8000* cluster. All the messages are forwarded to our publicly reachable VPS by means of TCP connections.

The rest of the application is launched in the cluster environment using the `mpirun` command. Each of the *dgroups* in the cluster is effectively using two transports in the same *dgroup*, namely MPI and TCP. The *dgroups* handled by *node0* in the cluster are multi-protocol both for sending and receiving tasks, since they listen over TCP connections from the network, use the MPI transport to communicate internally, and they ship results over the network using again a TCP connection.

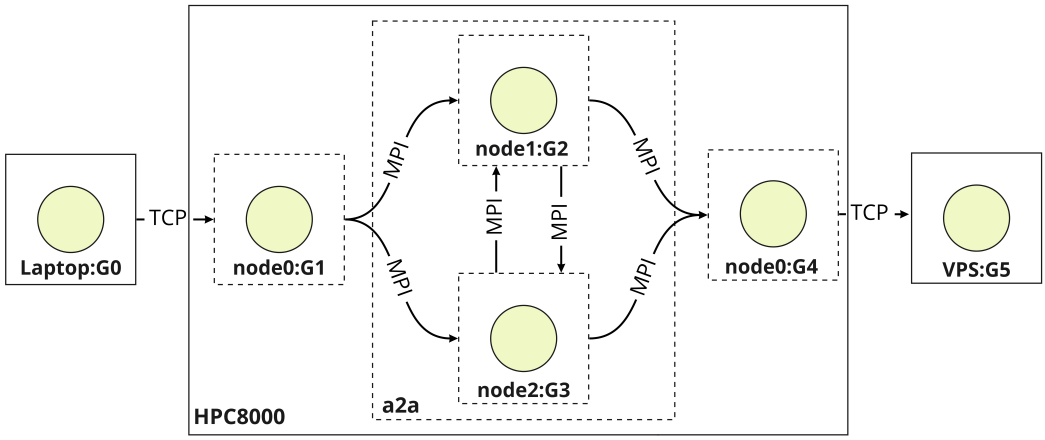


Figure 4.2.1: Case study for a multi-protocol application.

As a final remark, this particular implementation of the application in Figure 4.1.1, has much lower performances when compared to the same application executed only in the cluster environment. TCP connections at the edge of the application graph, together with the additional overhead introduced by the SSH tunnel, are the main cause of the lower performance of this example. Just as a reference, we executed the 100k message test, with message size of 1KB, recording an execution time of 63 seconds, opposed to the 6 seconds required by the TCP and MPI examples running exclusively in the cluster.

Nevertheless, this example shows the effectiveness of using a multi-protocol approach in a cloud environment, where edge nodes may not be compatible with HPC-oriented transports. As we show, multiple computing environments can be connected using the same distributed programming framework and connections can be handled in such a way that multiple protocols are used in the same distributed groups.

In Listing 4.2.1 we show the configuration file for the case-study application. We use the extended syntax for multi-protocol support, presented in Section 3.5.2 to map each *dgroup* with the necessary communication transport. As we can see, each node can specify the preferred transports to listen for incoming messages by means of the `"endpoint"` key. It's the *Sender*'s responsibility to correctly specify the transport to use for its output connections. For example, both *G2* and *G3* groups have to set the specific protocol to use in their communications with *G4*. In contrast, *G4* does not need to specify the required protocol for its communication toward *G5*, since the latter only listens for TCP connections.

```
{
  "groups": [
    {
        "name" : "G0",
        "endpoint" : "localhost",
        "out": ["G1:tcp"]
    }, {
        "name" : "G1",
        "endpoint": "tcp:8001,mpi://node0",
    }, {
        "name" : "G2",
        "endpoint": "mpi://node1",
        "out": ["G4:mpi"]
    }, {
        "name" : "G3",
        "endpoint": "mpi://node2",
        "out": ["G4:mpi"]
    }, {
        "name" : "G4",
        "endpoint": "mpi, tcp:8002://node0"
    }, {
        "name" : "G5",
        "endpoint": "tcp:8003://vps-ip"
    }
  ]
}
```

Listing 4.2.1: JSON configuration file for the case-study example.

Finally, with a future extension to the `dff_run` software module and to the high-level interface of FastFlow distributed runtime, the user will be able to easily create, deploy and run distributed applications. The *dgroups* created with the FastFlow distributed interface will be configured following the JSON configuration file and the transport objects can be created by following the mapping in the `"endpoint"` field. This would provide a flexible and less error-prone way of building heterogeneous distributed applications.

Conclusions

This thesis aimed to introduce the possibility of enabling heterogeneity in the computing nodes participating in a distributed application. A preliminary study on the existing communication frameworks was necessary in order to assess the availability of suitable solutions to introduce multi-protocol support in the FastFlow distributed runtime. The literature seems lacking under this point of view and existing frameworks rarely provide true multi-protocol support to applications. They usually allow to select a specific transport for the whole application, and in case multiple partitions of the same application need to use heterogeneous protocols, further work is required for the programmer in order to orchestrate the various protocols. Moreover, the communication frameworks we analyzed rarely provide support to general-purpose protocols, since the main focus of existing multi-protocol frameworks is the HPC environment. After an initial phase of experimentation, we decided to directly address the challenge of providing a suitable multi-protocol extension to the FastFlow distributed runtime by directly implementing software components to orchestrate multiple protocols in the same application. Our work followed a modular approach in order to provide well-separated responsibilities between the various components and allow extensibility of the provided extensions. The main goal was anyway the one of providing an abstraction that would allow the application developer to easily create, deploy, and run a multi-protocol application, selecting the most suitable protocols based on the characteristics of the used computing environment.

The knowledge gathered in the first exploratory part of our thesis work allowed us to provide an implementation of our multi-protocol components, retaining some of the concepts of existing frameworks but providing support to common transport protocols.

We provide a set of tests to evaluate the implementation of multi-protocol functionalities and assess that no notable overheads to the normal execution of existing distributed applications is introduced. Moreover, we show a case study for the usefulness of multi-protocol functionalities in a cloud environment, where edge nodes are not equipped with the same networking infrastructure as cluster nodes. We effectively connect, using both TCP and MPI connections, a cluster of machines with nodes outside the cluster environment producing and gathering data.

The provided functionalities are still in a preliminary phase and for this reason no interface to the application developer is provided. Our work presents a fully-functional implementation for multi-protocol functionalities in FastFlow distributed runtime, specifically targeting the intermediate representation of distributed applications. All the implementation code, along with the tests we performed, can be found

on our *GitHub* repository[1].

Future works

Our implementation of multi-protocol support is still at a preliminary level, hence new extensions can be introduced to add or improve existing functionalities. The possible improvements at the current stage of development could be:

- High-level API: since we worked at the RTS level of FastFlow, no high-level building block is available to enable support of multi-protocol functionalities. Extending the high-level interface of FastFlow distributed runtime would allow the user to create a multi-protocol distributed application with the same base mechanisms of the existing version.

- Mapping to multi-protocol functionalities: in the single protocol version of FastFlow the user can define the host-specific configurations via a configuration file. With the current extension, however, no interface for multi-protocol functionalities is offered to the user. The next step could be the one of providing a proper mapping between the implemented extensions and the configuration provided by the user configuration file.

- Making multi-protocol functionalities generic: the implemented multi-protocol software components are not generic and they retain the logic behind the communication protocol of the FastFlow distributed runtime. It could be useful to rethink the transport-specific components as generic network components providing a set of network primitives which are used in the upper layers to implement the communication protocol, further abstracting from the used transport mechanisms.

[1] https://github.com/Ulferin/multi-protocol-DFF

Bibliography

[1] David Astorga, Manuel F. Dolz, Javier Fernández, and José García. "A generic parallel pattern interface for stream and data processing". In: *Concurrency and Computation: Practice and Experience* 29 (May 2017). DOI: 10.1002/cpe.4175.

[2] Daniel A. Reed and Jack Dongarra. "Exascale Computing and Big Data". In: *Commun. ACM* 58.7 (June 2015), pp. 56–68. ISSN: 0001-0782. DOI: 10.1145/2699414.

[3] Cristian Ramon-Cortes, Pol Alvarez, Francesc Lordan, Javier Alvarez, Jorge Ejarque, and Rosa M. Badia. "A survey on the Distributed Computing stack". In: *Computer Science Review* 42 (2021), p. 100422. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2021.100422.

[4] *Apache Hadoop*. URL: https://hadoop.apache.org/ (visited on 08/01/2022).

[5] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664.

[6] Eman Shaikh, Iman Mohiuddin, Yasmeen Alufaisan, and Irum Nahvi. "Apache Spark: A Big Data Processing Engine". In: Nov. 2019, pp. 1–6. DOI: 10.1109/MENACOMM46666.2019.8988541.

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink™: Stream and Batch Processing in a Single Engine". English. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015), pp. 28–38. URL: https://research.tudelft.nl/en/publications/apache-flink-stream-and-batch-processing-in-a-single-engine.

[8] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0262530864.

[9] Murray Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming". In: *Parallel Computing* 30.3 (2004), pp. 389–406. ISSN: 0167-8191. DOI: 10.1016/j.parco.2003.12.002.

[10] Javier Fernández Muñoz, Manuel F. Dolz, David del Rio Astorga, Javier Prieto Cepeda, and J. Daniel García. "Supporting MPI-Distributed Stream Parallel Patterns in GrPPI". In: *Proceedings of the 25th European MPI Users' Group Meeting*. EuroMPI'18. Barcelona, Spain: Association for Computing Machinery, 2018. ISBN: 9781450364928. DOI: 10.1145/3236367.3236380.

[11] Johan Enmyren and Christoph W. Kessler. "SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems". In: *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*. HLPP '10. Baltimore, Maryland, USA: Association for Computing Machinery, 2010, pp. 5–14. ISBN: 9781450302548. DOI: 10.1145/1863482.1863487.

[12] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. "SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters". en. In: *International Journal of Parallel Programming* 49.6 (Dec. 2021), pp. 846–866. ISSN: 0885-7458, 1573-7640. DOI: 10.1007/s10766-021-00704-3.

[13] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. "Fastflow: High-Level and Efficient Streaming on Multicore". In: Mar. 2014. ISBN: 9780470936900. DOI: 10.1002/9781119332015.ch13.

[14] Nicolò Tonci. "A Distributed-Memory run-time for FastFlow's building-blocks". MA thesis. University of Pisa, 2022.

[15] Nicolò Tonci, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. "Distributed-memory FastFlow Building Blocks". In: *15th International Symposium on High-level Parallel Programming and Applications* (July 2022), Porto, Portugal, To appear in International Journal of Parallel Programming, Springer. ISSN: 0885-7458.

[16] Tom Shanley. *Infiniband*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321117654.

[17] Paul Grun, Sean Hefty, Sayantan Sur, David Goodell, Robert D. Russell, Howard Pritchard, and Jeffrey M. Squyres. "A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency". In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 2015, pp. 34–39. DOI: 10.1109/HOTI.2015.19.

[18] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. "UCX: An Open Source Framework for HPC Network APIs and Beyond". In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 2015, pp. 40–43. DOI: 10.1109/HOTI.2015.13.

[19] Robert B. Ross, George Amvrosiadis, Philip Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K. Gutierrez, Robert Latham, Bob Robey, Dana Robinson, Bradley Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, and Qing Zheng. "Mochi: Composing Data Services for High-Performance Computing Environments". In: *J. Comput. Sci. Technol.* 35.1 (Jan. 2020), pp. 121–144. ISSN: 1000-9000. DOI: 10.1007/s11390-020-9802-0.

[20] Steffen Ernsting and Herbert Kuchen. "Algorithmic skeletons for multi-core, multi-GPU systems and clusters". In: *International Journal of High Performance Computing and Networking* 7 (Apr. 2012), pp. 129–138. DOI: 10.1504/IJHPCN.2012.046370.

[21] Blair Archibald, Patrick Maier, Robert Stewart, and Phil Trinder. "YewPar: Skeletons for Exact Combinatorial Search". In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '20. San Diego, California: Association for Computing Machinery, 2020, pp. 292–307. ISBN: 9781450368186. DOI: 10.1145/3332466.3374537.

[22] Massimo Torquati. "Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns." PhD thesis. University of Pisa, 2019.

[23] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. "HPX: A Task Based Programming Model in a Global Address Space". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS '14. Eugene, OR, USA: Association for Computing Machinery, 2014. ISBN: 9781450332477. DOI: 10.1145/2676870.2676883.

[24] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. "Argobots: A Lightweight Low-Level Threading and Tasking Framework". In: *IEEE Transactions on Parallel and Distributed Systems* 29.3 (2018), pp. 512–526. DOI: 10.1109/TPDS.2017.2766062.

[25] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. "A View of the Parallel Computing Landscape". In: *Commun. ACM* 52.10 (Oct. 2009), pp. 56–67. ISSN: 0001-0782. DOI: 10.1145/1562764.1562783.

[26] *HDFS Architecture Guide*. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Introduction (visited on 08/07/2022).

[27] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

[28] Stephen Kaisler, Frank Armour, J. Alberto Espinosa, and William Money. "Big Data: Issues and Challenges Moving Forward". In: *2013 46th Hawaii International Conference on System Sciences*. 2013, pp. 995–1004. DOI: 10.1109/HICSS.2013.645.

[29] John Bloomer. *Power Programming with RPC*. USA: O'Reilly & Associates, Inc., 1992. ISBN: 0937175773.

[30] *XDR: External Data Representation standard*. Request for Comments RFC 1014. Internet Engineering Task Force, June 1987. URL: https://datatracker.ietf.org/doc/rfc1014/.

[31] W. Shane Grant and Randolph Voorhies. *Cereal - A C++11 library for serialization*. 2017. URL: http://uscilab.github.io/cereal/ (visited on 08/07/2022).

[32] *Boost Serialization*. URL: https://www.boost.org/doc/libs/1_73_0/libs/serialization/doc/index.html (visited on 08/07/2022).

[33] *Cap'n Proto: Introduction*. URL: https://capnproto.org/ (visited on 08/07/2022).

[34] *Protocol Buffers*. URL: https://developers.google.com/protocol-buffers (visited on 08/07/2022).

[35] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. "Java RMI performance and object model interoperability: experiments with Java/HPC++". In: *Concurrency: Practice and Experience* 10.11-13 (1998), pp. 941–955. DOI: https://doi.org/10.1002/(SICI)1096-9128(199809/11)10:11/13<941::AID-CPE391>3.0.CO;2-T.

[36] Jerome Soumagne, Dries Kimpe, Judicael Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert Ross. "Mercury: Enabling remote procedure call for high-performance computing". In: *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. 2013, pp. 1–8. DOI: 10.1109/CLUSTER.2013.6702617.

[37] Pieter Hintjens. *ZeroMQ*. O'Reilly Media, Mar. 2013. ISBN: 9781449334062. URL: https://www.oreilly.com/library/view/zeromq/9781449334437/.

[38] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. June 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf (visited on 05/24/2022).

[39] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 97–104. ISBN: 978-3-540-30218-6.

[40] *MPICH Overview — MPICH*. URL: https://www.mpich.org/about/overview/ (visited on 05/24/2022).

[41] Scott Pakin. "Myrinet". en. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1239–1247. ISBN: 9780387097664. DOI: 10.1007/978-0-387-09766-4_486.

[42] Jerome Soumagne, Philip H. Carns, and Robert B. Ross. "Advancing RPC for Data Services at Exascale". In: *IEEE Data Eng. Bull.* 43.1 (2020), pp. 23–34. URL: http://sites.computer.org/debull/A20mar/p23.pdf.

[43] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and P. Sadayappan. "Scalable I/O forwarding framework for high-performance computing systems". In: *2009 IEEE International Conference on Cluster Computing and Workshops*. 2009, pp. 1–10. DOI: 10.1109/CLUSTR.2009.5289188.

[44] *Mercury - Network Abstraction Layer*. URL: https://mercury-hpc.github.io/user/na/ (visited on 05/24/2022).

[45] *JSON-C - A JSON implementation in C*. URL: https://github.com/json-c/json-c/wiki (visited on 07/25/2022).