A multi-protocol communication layer for FastFlow's distributed runtime

Federico Finocchio - 516818
MSCS: Data and Knowledge. f.finocchio@studenti.unipi.it

Contents

Introduction

In the past years we are experiencing an explosion in the quantities of produced data, both by sensors and people. These huge amount of data challenge both the storage capabilities of single machines and the network being used to transfer data between multiple machines. Moreover, the amount of computation needed to handle the necessary data for modern applications, like (near) real-time applications that generate near-continuous data, is much bigger than the one offered by current systems. Thus, as we witnessed the shift from sequential to parallel programming, we are now observing a new shift from parallel to distributed computing. Moreover, as the difference between Big Data and High Performance Computing becomes more and more blurred, and due to this new evolution in the way application are developed and deployed, new frameworks, programming models and tools have surfaced to help the programmer to deal with the complexity of designing parallel applications above the underlying distributed infrastructures [1]. In the transition to distributed computing, new extensive parallel processing and new analytics algorithms are needed in order to provide timely and actionable information [2]. However, dealing with scalable and parallel applications that can be deployed across the network is not an easy task, and abstractions are needed in order to allow the application developers to focus on the optimization of the uppermost layer, leaving, among all, communication and synchronization details to experts on the respective fields.

The shift towards 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 often prefer to rely over high-abstraction frameworks to abstract from the distributed challenges and focus only on the application development.

Aggiustare le ultime frasi

The thesis aims at extending the existing distributed-memory runtime system for FastFlow, a C++ structured parallel programming library originally targeting shared-memory platforms. The thesis introduces a novel communication layer, which offers a standardized API in order to abstract from the underlying transport used for communications between remote nodes. The provided abstraction allows the *application programmer* to design new applications, as well as extending existing ones, by leveraging the classes introduced in this work in order to connect different nodes on the network without having to deal with communication specific implementation. The implemented layer provides an automatized way of plugging different protocols inside existing application without breaking the code and with little to zero 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, in order to follow and maintain the LEGO style *philosophy* of the FastFlow library, also the implemented communication layer is composed of a set of building blocks which can be composed in order to address the needs of the specific application. Allowing composability of components is very important in heterogeneous environments such as the cloud, in order to allow easy porting of applications and adaptation to environment changes. To fulfill this need, the communication layer presented in this thesis allows each application subset to communicate using the protocol that is most suited for the task at hand or for the architecture specific limitations. Vendor specific transports can represent a limitation in horizontally scaling applications, in particular if the provided transports are completely *obscure* to the application programmer, which should not care of the underlying communication aspects when programming a parallel application. The implemented building blocks allow for this reason to be run above a multitude of different transports.

The communication layer relies on a multi-protocol RPC framework in order to implement communication between nodes, and the only effort required for the application programmer is related to the declaration of the desired protocol to be used for communication. The transparent API which we aim at providing is completely independent from the underlying communication framework, thus allowing the FastFlow developer to transparently plug his own protocols and communication drivers to tweak the application communication channels. A natural preliminary phase of this work is strictly tied to the analysis of communication frameworks which are available to the application programmer in order to leverage efficiently multiple protocols during communication between distributed groups. The preliminary work is related to the study and analysis on existing libraries that could help the development of FastFlow's communication layer. Given the desire of being as much independent as possible from external libraries, in this document we also analyse which are the dependencies and the limitations tied to the external libraries used to develop the communication layer. In the first part of this work we analysed the Margo [3] library, it's underlying RPC framework called Mercury [3, 4] and the threading library which is used by Margo as a runtime framework, called Argobots [5]. The three analyzed frameworks are part of a much broad set of frameworks which are referred to as *Mochi core*, as specified in [3]. The Mochi core provides a set of frameworks to enable communication, concurrency management and storage with a composition model. As appeared by the testing we performed, but as also described by its developers, the Mercury RPC framework is purposefully built to leverage HPC fabrics vendor protocols, sometimes resulting in a lack of support and faulty implementation for more classical transports like MPI and TCP. This led us to adapt the existing MPI and TCP drivers in the distributed FastFlow RTS in the novel API, in order not to lose two of the most common transports used in HPC and cloud. We show, for each of the analyzed libraries and frameworks, which are their advantages and drawbacks, with special focus on lowering the amount of dependencies and minimizing the knowledge of the communication layer about the data that are exchanged between nodes, in order to create communicating nodes completely agnostic about the types used during computation, allowing also for parallelization of the serialization process and reducing the computation needed at the "edge" of each group.

Besides the limitations we have analyzed, which are mostly related to technical aspects and not on the usability of the presented frameworks, the simplicity of creating communication nodes and connecting them via a high-level communication library such as Margo, without requiring too many modification in the original application code, can be a good driver to experiment and develop an initial set of API calls which will allow to extend the communication layer functionalities and plug in a broader set of protocols which are still not supported by the used frameworks. 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 extensibility with little to no changes in existing code. Besides the limitations we have found during the experimentation with the analyzed libraries, our analysis on those frameworks was anyway important to understand the power and utility of having a high level abstraction on top of very specific APIs which requires a deep understanding of how the underlying infrastructure works. Hence, it is conceptually interesting and it could be very helpful in the implementation of the final communication layer to retain some of the concepts introduced by the *Mochi core* building blocks, which we present in the following sections.

The thesis uses a bottom-up approach to illustrate the overall picture of the used frameworks and to describe internally each of those frameworks. The thesis is organized as follows:

- **Chapter 3**: presents the core framework which provides the building blocks needed to implement RPC calls leveraging multiple transports. We describe the general structure of the framework, which will serve to describe the higher-level framework used to implement the communication layer in FastFlow's distributed runtime;
- **Chapter 4**: describes the runtime system which is used by the higher-level RPC framework in order to manage concurrent execution of different RPCs, to allow asynchronous execution of RPC-related callbacks and to simplify Mercury's progress loop;
- **Chapter 5**: describes the high level framework used for the implementation of the multi-protocol interface for the communication layer in FastFlow's distributed runtime. We show how this framework leverages Mercury and Argobots functionalities to provide a parallel and efficient multi-protocol RPC framework;
- **Chapter 6**: gives insights on the integration of Margo's functionalities into the FastFlow's distributed communication layer. We show pseudo-code of the implemented classes and common use-cases for both the application and runtime programmer;
- **Chapter 7**: focuses on the presentation of sample applications and testing we performed to evaluate the effectiveness of the implemented communication layer.

1 Background

2 FastFlow

FastFlow is a C++ parallel programming framework originally targeting shared-memory architectures [6] and successively extended for the distributed-memory setting [6]

3 Mercury

Mercury [3, 4] is an asynchronous RPC framework purposefully built to efficiently provide communication services to HPC systems with high-performance fabrics. It 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. Moreover, it provides an asynchronous RPC interface, based on a callback system, that allows transfer of parameters and procedure calls in the context of both local and remote execution in order to completely remove the differentiation of communications between on and off-node.

The library allows the execution of arbitrary functions via a system of function tagging and callbacks, coupled to a queue system that saves 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.

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 abstraction layer functionalities are implemented on top of different plugins. The plugin system can be considered as a *compatibility* layer, which implements the communication functionalities offered by different protocols. Examples of plugins are Libfabric [7, 8], MPI [9] and UCX [10], and multiple protocols are offered by each of the plugin, ranging from vendor-specific protocols to more classic ones like TCP and UDP.

3.1 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 [4], Mercury tries to address two main problems:

- inability to take advantage of HPC 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.

Mercury addresses these limitations by offering an asynchronous and flexible RPC interface specifically tailored for HPC systems. To do this, Mercury exposes an abstracted API to perform asynchronous RPC as well as 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. As stated in [4], 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.

3.2 Architecture

Mercury is built on a two-layer architecture, as shown in **Figure 3.1**, where each layer provides an abstraction for specific functionalities. 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.

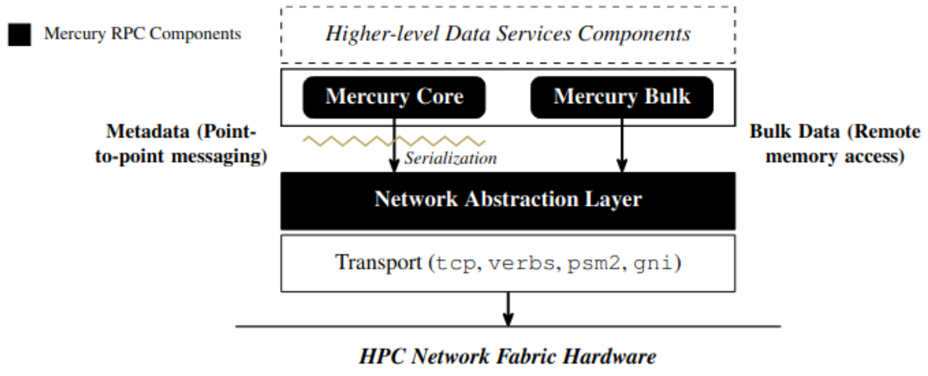
Mercury RPC Components
Higher-level Data Services Components
Mercury Core
Mercury Bulk
Metadata (Point-to-point messaging)
Serialization
Bulk Data (Remote memory access)
Network Abstraction Layer
Transport (tcp, verbs, psm2, gni)
HPC Network Fabric Hardware

Figure 3.1: Mercury RPC components in the software stack.

Mercury extends the functionalities of another project, called *I/O Forwarding Scalability Layer* (IOFSL)[11], 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.

3.2.1 Plugins

The plugin functionalities are referred to as a "*support for various network protocols that can be easily added and selected at runtime*" [12]. 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, Mercury 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 Libfabric, UCX and shared-memory [13] for local nodes communication. We show in §3.5.2 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 pre-defined 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 strings is provided in **Table 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 1: Mercury plugin's initialization format.

Note, however, that plugins may behave differently regarding how the format string is provided, in fact plugins do not manage incomplete or incorrect configuration strings uniformly. Some of these issues are reported in §3.5.2.

3.2.2 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 §3.2.1. Since a node can both provide and ask for services, the only time a server-specific behaviour is defined only refers to initialization, where the user can specify if the current node will be listening for incoming RPCs. Besides this, no more server/client concepts are used in the following.

The functionalities offered by the NAL refers to three main mechanisms:

- expected messages: requires 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, and they 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 operations. In this way, Mercury can easily adapt to protocols which only supports fixed operations, like TCP/IP ones, where one-sided communications are not possible.

3.2.3 RPC Layer

The RPC layer allows nodes to issue remote calls, and it is based on the messaging model described in §3.2.2. RPC requests are based on the common knowledge of origin and target nodes on how to encode and decode function parameters and return values.

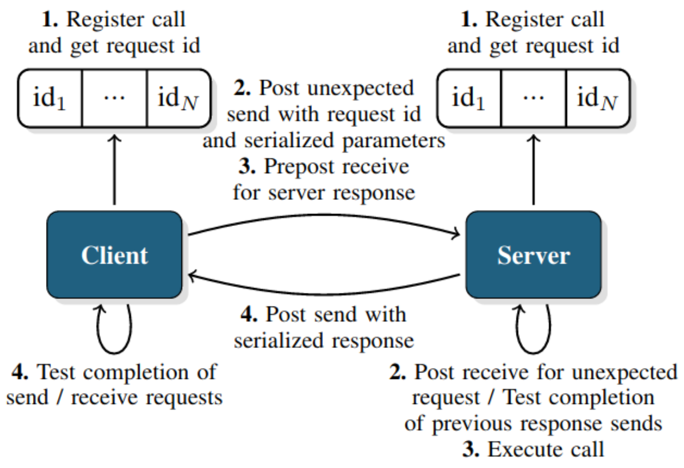
1. Register call and get request id
id_1 ⋯ id_N
1. Register call and get request id
id_1 ⋯ id_N
2. Post unexpected send with request id and serialized parameters
3. Prepost receive for server response
Client
Server
4. Post send with serialized response
4. Test completion of send / receive requests
2. Post receive for unexpected request / Test completion of previous response sends
3. Execute call

Figure 3.2: Execution flow of RPC call.

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 §5.1, since Margo introduces few additional steps, which are the ones that are actually used to implement FastFlow's communication classes. We show in **Figure 3.2** a usual flow of execution needed by both origin and target nodes to execute a 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 a 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.

Two situation may occur during an RPC request, and different mechanisms are used to guarantee full asynchrony:

- 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 specified 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.

3.2.4 Bulk Layer

This layer allows to send large data by avoiding intermediate memory copies. It is performed 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.

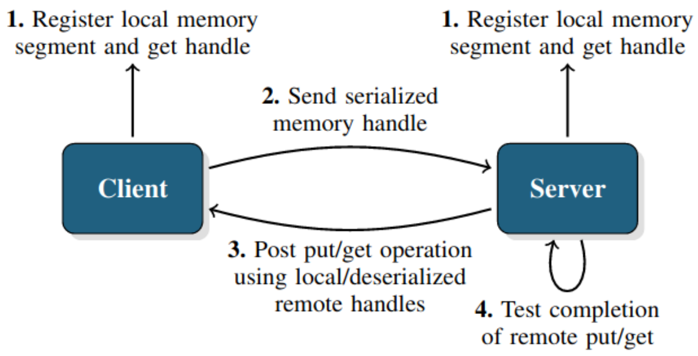
1. Register local memory segment and get handle
1. Register local memory segment and get handle
2. Send serialized memory handle
Client
Server
3. Post put/get operation using local/deserialized remote handles
4. Test completion of remote put/get

Figure 3.3: Execution flow of bulk request.

A typical execution flow for a bulk request is depicted in **Figure 3.3**. 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. 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.

3.3 Resilience and Fault Tolerance

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 aborted.

A basic flow of events related to the cancellation of an operation is shown in **Figure 3.4**.

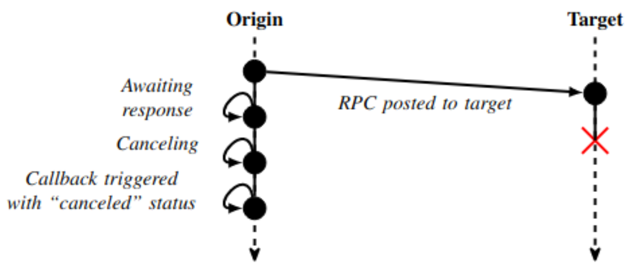

Figure 3.4: Cancellation of an RPC operation.

3.4 Dependencies

As reported by [14], the dependencies are mostly related to the intended plugin to integrate inside a specific application. No further requirements are specified by the Mercury's developers as needed for the installation and use of their framework.

3.5 Mercury analysis

Mercury comes with lots of functionalities and drawbacks. We list in this section all the advantages and the main limitations of the Mercury framework encountered during the testing phase, which are therefore shared by all the higher-level libraries which are based on it.

3.5.1 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 at all 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.

3.5.2 Limitations

Mercury, however, is 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 of the presented framework. As specified in §3.2.1, MPI plugin is deprecated and not maintained [15]. In fact, Mercury's developers suggest the use of libfabric plugin to leverage efficient HPC communication mechanisms, and UCX plugin as a general purpose plugin. As specified in [12], 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.

Further limitations found during the testing of this library are mostly related to the limited support of the Libfabric plugin paired with TCP transport [16], which is intended to use for testing and debugging applications on machines that do not provide high performance fabric protocols [17]. Problems about this specific transport were already reported [17, 18, 19], and during testing the main that we noticed are:

- During reply phase of RPC request using `ofi+tcp`, information about public IP of origin node are not used. This resulting in a lost packet in case the origin node was behind NAT, probably due to the *emulated* source addressing, as stated in [8]. The same does not happen by using `ofi+sockets`, which use is discouraged as per [16];
- Various problems related to termination of nodes which issued a request that couldn't be fulfilled, as also reported in [18];
- As per [20], the sockets provider has been deprecated in favor of the tcp one.

Other plugins, such as UCX [21], showed some problem in connecting nodes situated in different networks, mostly due to the fact that the plugin provided by Mercury struggles to bind the socket address to the specified one. However, it is able to establish a connection between endpoints situated in the same subnetwork.

Given the requirements for the FastFlow distributed version, the internal limitations of the Mercury framework could be too tight to allow its extensive utilization as a final communication library. However, the provided plugins and functionalities could anyway serve as a prototyping library in order to develop FastFlow communication layer APIs, in order to painlessly switch to a different communication framework in the future.

In fact, as we show in §5, the composition of the Mercury framework with higher-level abstractions allows to easily develop communication nodes to handle multiple endpoints at once without struggling with the explicit progress mechanism provided by Mercury. For this reason, in the next sections we introduce the libraries used by the higher-level abstractions which use Mercury framework as a core building block to provide RPC functionalities in an automatized way.

4 Argobots

Argobots [5, 22] is a lightweight low-level threading framework, which offers a portable library interface and allows specialized runtime management to the user. The main aim of Argobots is to provide a mapping between high-level abstraction to low leve implementations [5], 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, which are non-preemptable and offers, respectively, different models of execution. Work units are executed by OS-level threads, which in this context are referred to as Execution Streams (ESs). Each ES can be associated to a set of *pools*, which are containers of work units, and execute tasks in the order provided by *scheduler* entities. Argobots allows to define various scheduler (included custom ones), which determine the order of execution of each work unit inside the pools. Scheduler entities can be “plugged” at runtime to change the strategy of execution, based on requirements adapting to the computation at hand. Moreover, work units can be dinamically moved to a different pool in order to allow computation on a different ES.

A generic Argobots application is depicted in **Figure 4.1**, which is built by following the execution flow shown in **Figure 4.2**. We can see how different pools can be associated to the same ES and how the scheduler provides the work units to the execution flow, which is sequential and guarantees progress. We describe in the next sections the characteristics of each building block of an Argobots application.

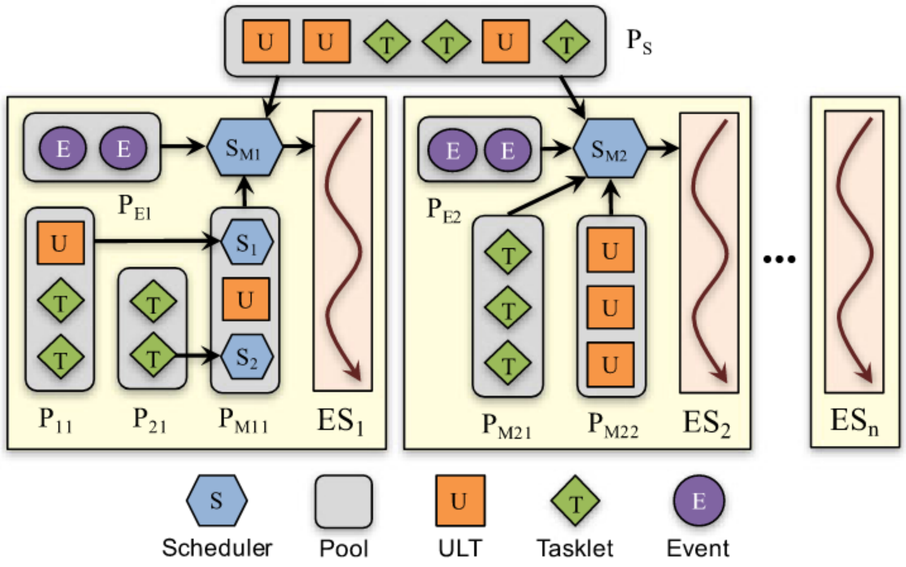

Figure 4.1: Argobots execution model.

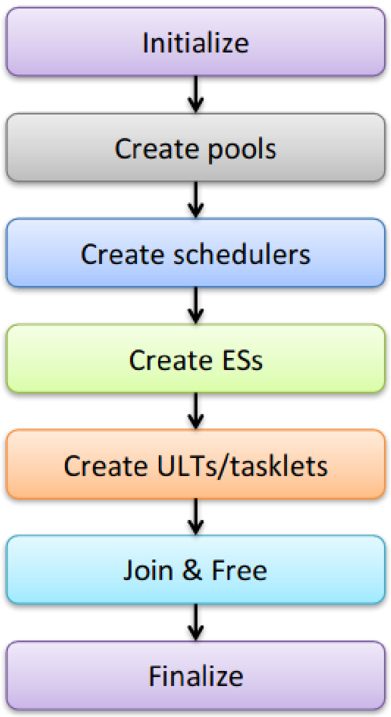

Figure 4.2: Argobots application flow of execution.

4.1 Building blocks

4.1.1 Execution Streams

Executions streams represents a sequential and independent instruction stream, which can consist of one or multiple non-preemptive work units (namely, ULTs and tasklets), it is mapped to a Pthread and can be bound to a hardware processing element such as CPU core or hardware thread. The work units to be executed are contained in one (or more) pools, and they are retrieved from one (or more) schedulers. Scheduling policies can be tweaked during runtime, and they determine the order in which work units flow to the execution stream and are thereby executed sequentially.

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. However, synchronization is needed between work units executed in different ES running in parallel.

4.1.2 Schedulers

Schedulers are responsible to deliver work units to the ES they are associated with, and they can implement different scheduling policies, which determines how work units are retrieved from pools and handled to the ES. Schedulers can be composed and scheduled just like work units, in order to compose different scheduling approaches dynamically, related to the computation at hand. Since a scheduler is considered as a work unit, it can be inserted into a pool and executed, which translates in a change of scheduling policy for the ES in charge, which will shift back to its original scheduler when the computation associated with the current scheduler is over.

4.1.3 ULTs and Tasklets

User-level threads and tasklets, also called *work units*, represent two different types of workflow. User-level threads are independent execution units, which are executed in user space and can yield control to the scheduler, they can be dynamically migrated on a different execution stream during execution, since they have a private stack and context-saving capabilities. Tasklets, on the other hand, are considered to be “lighter” than ULTs, since they 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.

4.1.4 Pools

Pools are containers of work units, and their sole purpose is to act as a uniform way of associating work to ES, indirectly, through schedulers. The only property associated with pools is the one regarding access, which can be set as private or shared. Shared pools may be used to implement a “work-stealing” strategy between two or more ESs. Pools are also used internally by the ES to receive asynchronous events.

A pool associated with a running or stacked scheduler defines a set of work units ready to execute, which have to be controlled by the application programmer, since Argobots does not implicitly defines dependencies between work units. Hence, synchronization mechanisms offered by the library must be used to control the control flow of different work units. Various operations can be performed over pools, like migration of work units to or from a specified pool, creation, destruction, and of course pop and push operations.

4.2 Operations

Argobots defines a set of operations which are common to all work units, made exception for some of them which are not available to Tasklets, given their execution model limitations. Implemented operations refer to:

- Creation: allows the creation of work units, which are then inserted in a specific pool in a ready state. The type of pool and the associated scheduler(s) will determine the time and context of execution;
- Join: work units can be joined by other ULTs, which wait for their termination;
- Yield: a work unit can cooperatively yield control to the scheduler which is executing in the current ES at the time of yielding. The scheduler will then gather the next work unit to be executed following the implemented policy;

- Yield_to: a ULT can decide to yield control to a specific ULT in the same ES. This operation avoids the overhead of one context switch by bypassing the scheduler;
- Migration: allows work units to be migrated between pools;
- Synchronization: ULT can rely over usual synchronization mechanisms implemented by Argobots, such as mutexes, condition variables etc.

4.3 Memory Management

Argobots is intended to be used in fine-grained dynamic environments, where creation and destruction of work units, as well as context switch between them, take place at high frequencies.

Since, as shown in [5], memory allocation and deallocation contribute to most of the time required to create and destroy a work unit, Argobots developed its own memory allocator. The memory allocator creates, for each ES, a memory pool which is subsequently used to manage all creation and destruction calls of work units. The memory pool is held private per ES, in order to avoid heap access synchronization upon creation of work units by means of the same ES. The size of the pool is tied to the number of work units which are spawned and allows to return memory to the system upon their destruction, if the pool reached a given threshold.

Context switching, on the other hand, has a lower cost in terms of memory, compared to creation and destruction of work units, and it is only related to ULTs, since tasklets are atomic operations and can't yield to other work units. Context switching costs are mostly associated with:

- ULT suspension: context of the currently running ULT must be saved and context of the next ULT must be resumed. However, the first step can be avoided if the yielding ULT is terminating and will not be resumed later;
- ULT join: the join operation suspends the caller and yield control to the scheduler, until the joined ULT terminates its execution. However, if two ULTs are in the same ES, the joining ULT can directly yield control to the joined ULT, bypassing the context switch to the scheduler entity.

5 Margo

Margo [3] is a 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, a physical 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 wrappers based on Argobots runtime in order to present asynchronous communication mechanisms as a user-level thread communication model. Communication facilities are handled by ULTs, which can be suspended and resumed as communication proceeds;
- progress loop abstraction: polling and communication events are internally managed by a ULT. 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;
- renewed polling strategy: Margo allows both busy and idle mode, which allows to tweak Margo for both performance and resource consumption needs.

5.1 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 5.1**, 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 5.2** 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 don't 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 5.3** 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 **5.4** and **5.5**. 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   hash_val;
    hg_uint64_t  size;
    hg_bulk_t    bulk_handle;
} ff_rpc_in_t;
```

Listing 5.1: RPC type definition.

```c
hg_return_t
hg_proc_ff_rpc_in_bulk_t(hg_proc_t proc,
                         void* data) {

    ...
    // Retrieve input data structure
    ff_rpc_in_t* in = (ff_rpc_in_t*)data;

    // Write/read data to/from Margo's
    // send/receive buffer carried by 'proc'
    hg_proc_hg_int64_t(proc,
            &in->hash_val);

    // We call a 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 5.2: Packing routine definition. Allows the Margo framework to manage data from/to the network buffer used internally to ship data during RPC calls. Each of the type-specific routines allows data copies which are aware of the size of data to be packed/unpacked.

```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;
    const struct hg_info* hgi;
    margo_instance_id     mid;

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

    // Retrieve objects to identify current RPC
    // and get back registered data
    hgi = margo_get_info(handle);
    mid = margo_hg_info_get_instance(hgi);

    // Here data may be retrieved and used
    // internally in the RPC call

    margo_free_input(handle, &in);
    return;
}
DEFINE_MARGO_RPC_HANDLER(ff_rpc)
```
Listing 5.3: RPC declaration and definition

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

    // Initialize the Margo instance used to perform
    // Margo-related calls
    margo_instance_id mid;
    mid = margo_init(listening_addr, MARGO_CLIENT_MODE, 1,
    1);
    ...
    my_rpc_id = MARGO_REGISTER(mid, "ff_rpc",
                    ff_rpc_in_t, ff_rpc_out_t, NULL);

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

    // Creates the handle for the RPC call
    hg_handle_t handle;
    margo_create(mid, svr_addr,
                    my_rpc_shutdown_id, &handle);
    margo_forward(handle, NULL);
    ...
}
```

Listing 5.4: 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 the Margo instance used to perform
    // Margo-related calls
    margo_instance_id mid;
    mid = margo_init(protocol, MARGO_SERVER_MODE, 1, 1);
    my_rpc_id = MARGO_REGISTER(mid, "ff_rpc",
                    ff_rpc_in_t, ff_rpc_out_t, ff_rpc);

    margo_wait_for_finalize(mid);
}
```

Listing 5.5: 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.

5.2 Margo analysis

As we pointed out in §3.5, most of the limitations are tied to the support of the various plugins and their functionalities. Margo, besides sharing all these limitations, introduce an incredibly easy interface to implement multi-homed RPC services, which greatly simplifies the development of RPC-based FastFlow nodes.

The main limitation of the Margo library 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 used to ship RPC data through the network. However, further investigation is needed at this point in time to analyze how those limitations can be avoided, without relying on the bulk functionalities, which however remain a suitable fallback method if data copies are unavoidable. However, a further 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 in the underlying layers and how it handles configuration strings. The introduced dependencies are the following:

- Mercury: as described in §3, in particular requires for this framework to be built with `-DMERCURY_USE_SYSTEM_BOOST:BOOL=OFF -DMERCURY_USE_BOOST_PP:BOOL=ON`, which enable pre-processors macro in Mercury in case BOOST library is not present in the system. Since we do not want to be dependend on BOOST library, this is necessary since Margo uses these macros internally;
- Argobots: as described in §4;
- json-c: A JSON implementation for C language[23], needed internally by Margo in order to generate configurations based on json-formatted strings provided at initialization of Margo instances.

6 Implementation

6.1 Communication classes

We developed FastFlow's communication classes by extending the `ff_node_t` class, in such a way the developed classes can be used in any context where a `ff_node_t` can be used. However, we show that they naturally sit at the extremes of a pipeline building block, since they offer functionalities to receive and forward data from and through the network.

The classes we implemented are strictly tied to the concepts of receiver and sender node, and they make use of Margo's capabilities of developing multi-endpoint services without requiring to manually handling progress loops. At the current stage of development, there is no personalized behaviour based on the endpoint which received a call to the registered RPCs. This represents the next main step in the evolution of the implemented classes. (De)Multiplexing, however, should be fairly easy to handle thanks to the internal mechanisms that Margo offers in order to identify the origin node which issued a request. Having personalized behaviour, depending on the specific endpoint, is very important to deal with the grouping concept introduced since the distributed version of FastFlow has been developed. Particularly pathological cases, where a personalized behaviour must be implemented given the specific endpoint on which the request is received, benefit from the automatized mechanisms offered by Margo. Moreover, the simplicity in handling multiple endpoints via the implemented classes allow to test a multitude of situations with little to no code changes.

6.1.1 Receiver node

Receiver node is a FastFlow `ff_node_t` that relies on a Margo instance initialized with `MARGO_SERVER_MODE`. This allows the receiver node to wait for incoming RPC requests at the specified addresses. A list of addresses can be provided at initialization, translating in a receiver node listening for the same set of RPC functionalities on all of the provided endpoints. **Listing 6.1** shows pseudo-code of the implemented receiver node.

```
struct receiverStage: ff_node_t<Task> {
    std::vector<margo_instance_id> mids // Margo instances to be used during servicing

    // Parametrized constructor which registers all specified endpoints with the
    // provided configuration strings.
    receiverStage(std::array<char*> endpoints, std::array<char*> configs) {
        ...
        for(i < num_endpoints) {
            // init_endpoint initializes a Margo server instance on the provided address
            // and config string. The endpoint allocates a specific ES and pool to handle
            // all the requests coming through the specified address.
            // Returns an handle to the created Margo instance
            mids[i] = init_endpoint(endpoints[i], configs[i]);

            // Code to get the end address (to be forwarded and used by the sender node)
            // and various debugging can be put here, by using the mids generate by the init
            // phase
            ...

            // Use the initialized mids to register the set of RPCs that will be offered by
            // this receiver node.
            // Responses for the registered RPCs are disabled.
            register_service(mids[i])
        }

    }

    // The receiver node has nothing to do in the service method, it can simply wait
    // for termination of all the listening endpoints and forward the EOS at the end.
    // 'task' parameter is ignored in this stage, since it acts as a sort of 'endo-stream'
    // generator for the other stages in the pipe.
    Task* svc(Task* task) {
        wait_for_finalize(mids);
        return EOS;
    }
}
```

Listing 6.1: Receiver node pseudo-code.

6.1.2 Sender node

Sender node, on the other hand, relies over Margo instance initialized with `MARGO_CLIENT_MODE`, since we do not expect this node to be listening on any RPC request for now. The current version of the sender node allows to contact only a single endpoint at a given address. The address to contact for this specific node must be provided at initialization, and at the moment it is not possible to change the remote service address. In the next versions the possibility to contact multiple addresses, in order to implement the pathological grouping case described before, will be implemented following the same approach used for the receiver stage. Pseudo-code for this class is provided in **Listing 6.2**.

```
struct senderStage: ff_node_t<Task> {
    margo_instance_id       mid;        // Margo object to use for communications
    hg_addr_t               svr_addr;   // Server address listening for incoming RPCs

    senderStage(char* addr) {
        // Initialize the Margo instance as a client node and creates the necessary
        // objects to register RPC calls and addresses to issue the requests to the
        // specified address.
        mid = init_endpoint(addr);

        // This is the exact same function that is called by the receiver, but in this
        // case we don't need to provide implementation of the actual RPC function.
        register_service(mid);
    }

    Task* svc(Task* task) {
        // The task received by the previous stage is packed into the RPC registered
        // type and forwarded to the sender via an RPC call. The current node can
        // proceed since no response is expected.
        ff_rpc_in in = pack_task(task);
        forward_task(mid, in);
        return GO_ON;
    }

    void svc_end() {
        // Upon termination forwards a shutdown RPC to the connected receiver node.
        // The sender has nothing to do more than cleaning Margo resources.
        forward_shutdown(mid, shutdown_id);
        finalize(mid);
    }
};
```

Listing 6.2: Sender node pseudo-code.

6.2 RPC based service

The developed classes allow distributed groups to communicate through a fixed set of RPC calls, which are shared among receiver and sender nodes and used for all the communications between the various groups. Two RPCs are offered by the communication service, and they refer to:

- `ff_rpc`: this RPC is used during the whole lifetime of the FastFlow application, and it is used by the sender node, upon reception of a stream element, to forward the current stream element to the receiver node it is connected to. The sender node packs the data in the RPC input type, as described in **§5.1**, and ships them issuing a forward call to the connected endpoint. Since no response is expected from the RPC call, the sender can proceed immediately.

- `ff_rpc_shutdown`: this RPC is only used upon reception of an EOS object from the stream. In this case no data needs to be sent through the network, but only a signal that the stream has ended, in order to allow the receiving node to gracefully terminate its execution and forward EOS accordingly to the local group's nodes. Signaling an EOS propagates through the network, since the receiver node will generate an actual EOS object when the `ff_rpc_shutdown` callback is executed that will allow local nodes to terminate as per FastFlow's execution flow. At that point, the next sender node will issue an end-of-stream RPC and the whole FastFlow application will terminate. Note, however, that a receiver node will not forward an EOS object unless all the listening endpoints have received a shutdown RPC.

It is important to point out that this set of RPC callbacks need to be registered only once per Margo instance. This means that, once for each listening endpoint, the receiver node must register the same set of RPCs, but it is completely agnostic on the number of nodes that will issue RPC requests on the same endpoint.

6.3 Splitting taxonomy

In this section we analyze the characteristics of different connections that take place between remotely connected groups, we show that very specific types can be determined whether the splitting happens in a *horizontal* or *vertical* fashion. We describe the specific categories depending on different splitting strategies. To better explain the identified categories, we introduce the concepts of *level*, *horizontal* and *vertical* splits. Considering a generic FastFlow application as a pipeline of stages, we naturally associate a *level* to each stage incrementally. In addition, we define as *horizontal* a split which creates two groups sitting at the same level of the original pipeline, instead we define as *vertical* a split which creates groups operating at two different levels of the pipeline. Note, however, that a *vertical* split can internally contain further *vertical* or *horizontal* splits. Given these definitions, we can identify two categories that describe each of the connections in remotely-connected groups, specifically:

- *internal*: elements to be sent/received are related to a connection that takes place between groups that are split horizontally. Connections of this type can be linked to splittings that divides an original FastFlow building block in two groups belonging to the same pipeline level, for example an *all-to-all* building block divided by distributing left and right workers in two groups;

- *external*: elements to be sent/received are related to connections that take place between two groups at different levels of the original pipeline. These connections are typically related to vertical splits of the original FastFlow application.

To better illustrate this taxonomy, we present a simple example which contains every concept we introduced up to now. In **Figure 6.1** there is a sample FastFlow application composed of an *all-to-all* building block with two left workers and two right workers. By performing an *horizontal* "cut", we want to split the *all-to-all* in two different groups, each of them with one left worker and one right worker. If we consider this *all-to-all* block to be in the middle of a pipeline, the groups will finally be as depicted in **Figure 6.2**. We show the structure of the main software entities that handle the splitting. As we can see, the amount of nodes participating in the application increases to facilitate communication between remote groups, both via *internal* connections (LS1 to RS2) and via *external* ones (ff_receiver to LS2).

We will proceed now by describing the bigger picture by delving into particulars and explaining each of the involved parts and messy communication lines. The reasoning is very simple once the way communications are handled is clear, but it can seem pretty complicated at the start. We depicted in different colours the data flow which involve a stream element as the same "origin" object. When the colour of a communication line changes, also the provenance attached to the considered object will change. We describe in the following the different concepts depicted in the picture below:

- left/right box: they are abstracted representations of *internal* nodes in remotely connected groups. Left and right boxes of horizontally split groups, belonging to the same original building block, are logically connected by mean of their own local sender and receiver nodes. Since they respectively emulate a left and right worker, a stream element received by a left box (and hence originally shipped by a right box in a remote group) will be forwarded to a right worker in the current local group.

- red channel: carries stream elements which are part of the normal flow of execution, for example streaming elements flowing through the stage of a pipeline one after the other.

- purple/green channel: represents *internal* connection channels. They allow communication between boxes of different groups in order to maintain the *all-to-all* semantic in a distributed environment.

The connections, internally, are determined by each remote group by means of the configuration file provided upon initialization and by the information that are exchanged during the startup phase. In this way, each receiver/sender node can associate correctly the ID of each channel to the various entities that are created after the splitting is performed.

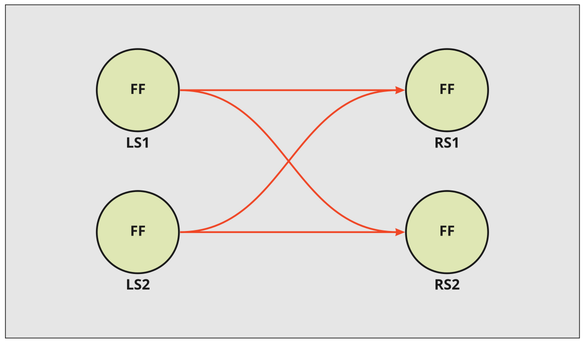

Figure 6.1: Sample A2A building block.

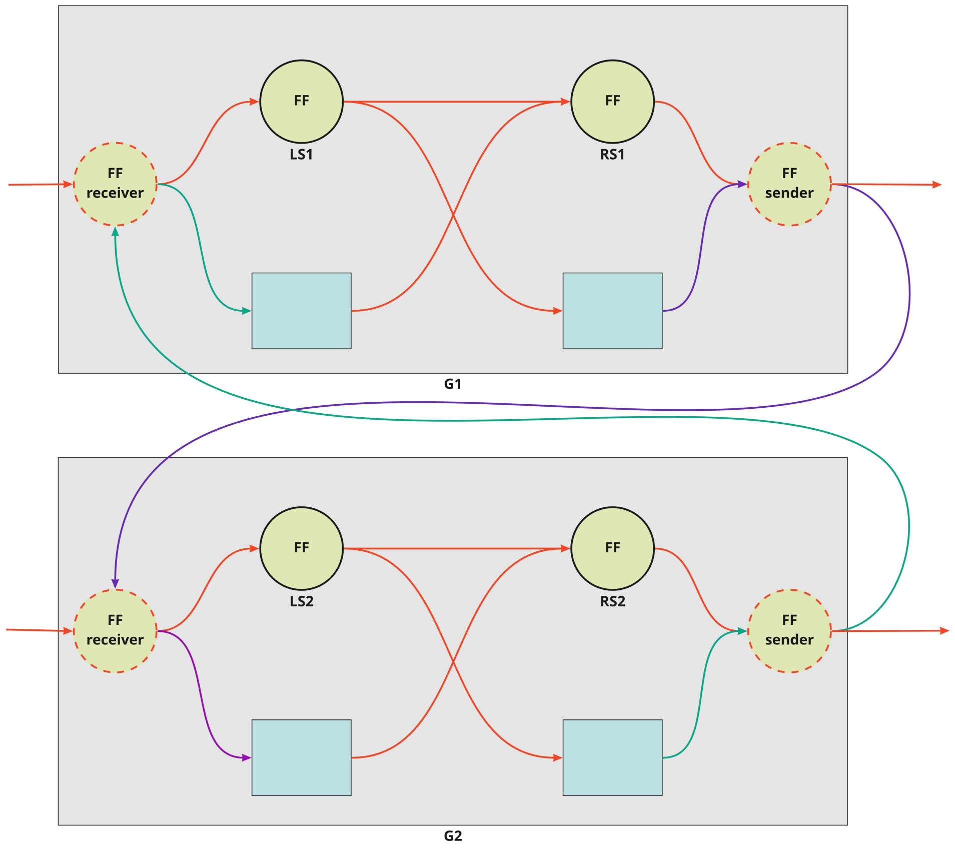

Figure 6.2: Pathological case representing the splitting of an A2A node into two groups which needs to communicate internally in order to retain the original semantic of the A2A building block.

7 Testing

In this section we show the testing application we have developed to assess the functionalities of the implemented classes described in §6.

7.1 A sample application

In **Figure 7.2** we show a sample application developed to test the implemented classes for communication. As it can be seen, the structure of the application is very simple, we have three groups communicating in a pipeline fashion, by means of implemented `sender` and `receiver` nodes. Communication between groups can happen with the different plugins provided by the Mercury library, and switching from one plugin (transport) to another is only a matter of passing as configuration string the desired plugin (transport) to use for communication.

Figure 7.1 shows the legend for the nodes and communication channels used in the sample application. We have two types of nodes, sequential ones which are standard `ff_node_t`, and Margo-injected nodes which are used for communication between distributed groups. Simple plain arrows indicates usual FastFlow shared memory channel, instead bullet-tailed arrows indicate a distributed connection between two nodes. Note, however, that "distributed" here only means that two groups are connected through Margo calls, but the nodes can reside in the same machine and communicate with the shared memory plugin provided by Margo, as it happened in part of the testing phase.

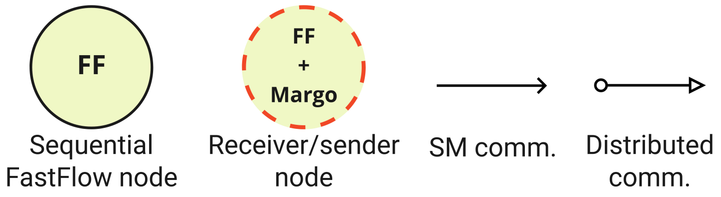

Figure 7.1: Node legends

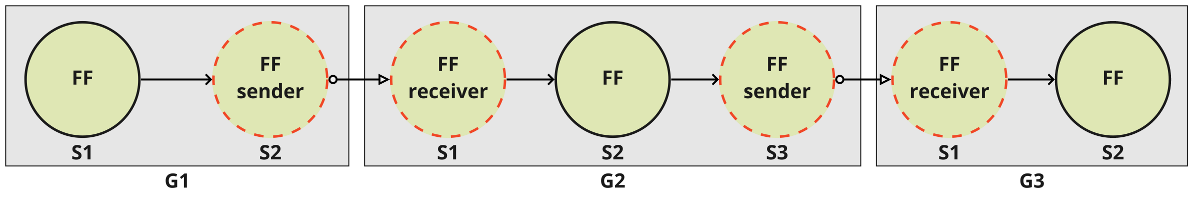

Figure 7.2: A sample testing application composed of three main groups which are connected by using the implemented classes using Margo library.

Moreover, the standard FastFlow nodes depicted in **Figure 7.2** can be any composition of standard FastFlow building blocks. We showed here, for simplicity, a single sequential ff_node_t. The important thing to notice is that sender and receiver nodes are mandatory, respectively as last and first nodes, in order to compose distributed groups correctly. The depicted groups represents, potentially, the three situations that may occur when splitting a standard FastFlow application in a distributed one, that are:

- (G1): group with only out remote connections. In this case only a senderStage is needed as a last node in the application pipeline. The pipeline is most likely to generate elements from an endo/eso-stream;
- (G2): group with both in/out remote connections. Both receiverStage and senderStage are needed, respectively as first and last nodes in the pipeline. All internal nodes receive and send stream elements using Margo-injected nodes;
- (G3): group with only in remote connections. Only a first receiverStage pipeline node needed.

In **Listings 7.1**, we show a skeleton with required library calls necessary to initialize Margo and Argobots environment, which are necessary steps before composing FastFlow building blocks with the implemented communication nodes.

```
int main(int argc, char** argv)
{
    ...

    // Setting up main Argobots instance
    margo_set_environment(NULL);
    ABT_init(0, NULL);

    {
        // Here we place code to build each of the groups composing
        // the application, namely G1, G2, G3.
    }

    // Finalizing Argobots
    ABT_finalize();
}
```

Listing 7.1: Skeleton of a sample FastFlow application using Margo as communication layer.

After having initialized the environment with the required library calls, we proceed by building the groups that have to be executed as follows:

- group G1: implemented in **Listing 7.2**, creates two nodes, the first one is a simple FastFlow ff_node_t generating a stream of elements to forward to other nodes, the second one is a senderStage initialized with the address of the receiverStage to which elements will be forwarded using the Margo communication mechanisms;
- group G2: implemented in **Listing 7.3**, builds three stages, where the first and the last one are communicator nodes, respectively a receiverStage and a senderStage. The middle node is a simple ff_node_t which simply lets tasks flow between the two nodes communicating with groups G1 and G3;

- group G3: implemented in **Listing 7.4**, builds a `receiverStage` and a standard `ff_node_t` node which simply prints the tasks received.

```cpp
firstStage  first(stream_len);
senderStage sender(receiver_addr);
ff_Pipe<float> pipe(first, sender);
if (pipe.run_and_wait_end()<0) {
    error("running pipe");
    return -1;
}
```

Listing 7.2: G1 node composition

```cpp
// Build addresses vector
...

receiverStage receiver(addresses);
forwardStage first;
senderStage sender(receiver_addr);
ff_Pipe<float> pipe(receiver, first, sender);
if (pipe.run_and_wait_end()<0) {
    error("running pipe");
    return -1;
}
```

Listing 7.3: G2 node composition

```cpp
// Build the addresses vector
...

receiverStage receiver(addresses);
forwardStage first;
ff_Pipe<float> pipe(receiver, first);
if (pipe.run_and_wait_end()<0) {
    error("running pipe");
    return -1;
}
```

Listing 7.4: G3 node composition

With the implemented classes, building various groups communicating between each other is very simple and straightforward. Moreover, the way communications are abstracted by the underlying frameworks, makes it easy and painless to extend functionalities of the communication nodes. Adding new protocols requires zero effort and no code modifications from the user point-of-view, since the only step to follow is to define a Mercury-accepted string with the preferred protocol to use during communication and use it at initialization of the receiver node.

Note that, since the receiver stages can be initialized with a vector of strings which represents all the endpoints to which the receiver will listen on, we can (and we must, to allow correct termination) compose multiple instances of groups (G1) and (G2), based on the amount of endpoints created, respectively by (G2) and (G3). For example, if we create a group (G2) with two listening endpoints, in order to allow the whole application to terminate correctly, we must run two (G1) connecting, in turn, to both the endpoints.

References

[1] Cristian Ramon-Cortes et al. “A survey on the Distributed Computing stack”. In: *Computer Science Review* 42 (2021), p. 100422. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2021.100422. URL: https://www.sciencedirect.com/science/article/pii/S1574013721000629.

[2] Stephen Kaisler et al. “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.

[3] Robert B. Ross et al. “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. URL: https://doi.org/10.1007/s11390-020-9802-0.

[4] Jerome Soumagne et al. “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.

[5] Sangmin Seo et al. “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.

[6] Marco Aldinucci et al. “Fastflow: High-Level and Efficient Streaming on Multicore”. In: Mar. 2014. ISBN: 9780470936900. DOI: 10.1002/9781119332015.ch13.

[7] Paul Grun et al. “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.

[8] *Mercury - Network Abstraction Layer. OFI plugin.* URL: https://mercury-hpc.github.io/user/ofi/.

[9] *Mercury - Network Abstraction Layer. MPI plugin.* URL: https://mercury-hpc.github.io/user/na/#deprecated-plugins.

[10] Pavel Shamis et al. “UCX: an open source framework for HPC network APIs and beyond”. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE. 2015, pp. 40–43.

[11] Nawab Ali et al. “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.

[12] *Mercury - Network Abstraction Layer.* URL: https://mercury-hpc.github.io/user/na/.

[13] *Mercury - Network Abstraction Layer. Shared Memory plugin.* URL: https://mercury-hpc.github.io/user/sm/.

[14] *mercury-hpc.* URL: https://github.com/mercury-hpc/mercury.

[15] *Mercury - Issue #489*. URL: https://github.com/mercury-hpc/mercury/issues/489.

[16] *Mercury - Network Abstraction Layer. Available plugins.* URL: https://mercury-hpc.github.io/user/na/#available-plugins.

[17] *OFI/Libfabric - tcp sockets fabric provider*. URL: https://ofiwg.github.io/libfabric/v1.11.1/man/fi_tcp.7.html.

[18] *Mercury - Issue #418*. URL: https://github.com/mercury-hpc/mercury/issues/418.

[19] *Mercury - Issue #332*. URL: https://github.com/mercury-hpc/mercury/issues/332.

[20] *Open Fabric Interfaces*. URL: https://github.com/ofiwg/libfabric.

[21] *Mercury - Network Abstraction Layer. UCX plugin.* URL: https://mercury-hpc.github.io/user/na/#ucx.

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

[23] *JSON-C - A JSON implementation in C*. URL: https://github.com/json-c/json-c/wiki.