2026 IEEE 46th International Conference on Distributed Computing Systems Workshops (ICDCSW) | 979-8-3195-3647-1/26/$31.00 ©2026 IEEE | DOI: 10.1109/ICDCSW72724.2026.00045

Emulating NetQMPI applications with CUNQA: A Decoupled Architecture for HPC Environments

Jorge Vázquez-Pérez[1*], F. Javier Cardama[2], Tomás F. Pena[2,3] and Andrés Gómez[1]

[1]*Galicia Supercomputing Center (CESGA)*
[2]*Research Centre on Intelligent Technologies (CiTIUS) - Universidade de Santiago de Compostela*
[3]*Electronic and Computation Department (DEC) - Universidade de Santiago de Compostela (USC)*

***Abstract*—Distributed Quantum Computing (DQC) promises to scale quantum workloads beyond the limits of monolithic devices by distributing computation across multiple nodes. In this work, we propose combining two recent solutions for programming and simulation of DQC systems: NetQMPI and CUNQA. NetQMPI provides an MPI-inspired programming model for DQC over quantum-network stacks, while CUNQA emulates DQC architectures on HPC by exposing virtual QPUs (vQPUs) with a management and execution interface. This paper proposes a practical integration in which CUNQA serves as the execution backend for NetQMPI, enabling developers to write NetQMPI programs while running them on HPC-based vQPU emulation. We outline the decoupled architecture, the adapter mapping required for backend integration, and the end-to-end application execution flow.**

***Index Terms*—Distributed Quantum Computing, NetQMPI, CUNQA, NetQASM, HPC, virtual QPU, MPI, quantum networking.**

I. Introduction

Scaling quantum computation through a single, monolithic quantum processing unit (QPU) remains difficult in practice because enlarging the device tends to amplify physical and control constraints: as qubit count and wiring density grow, suppressing unintended qubit-qubit interactions becomes increasingly challenging. Phenomena such as crosstalk can introduce correlated errors, thereby reducing the effective computational volume or increasing the demand for extensive error-correction overhead. Distributed quantum computing (DQC) addresses these constraints by interconnecting multiple smaller QPUs and scaling through modular composition: capacity is increased by adding QPUs while keeping each module within an operating regime where noise, crosstalk, and control complexity are more tractable, so that the overall platform can approximate the functionality of a larger processor without requiring a single oversized chip [1].

Given that multi-QPU quantum systems are a plausible near-term direction for scaling, there is a clear need for classical software infrastructure to help developers design and run DQC applications. In [2], Barral et al. explain this landscape and emphasise that progress depends on a full stack of abstraction layers, from hardware and networking capabilities up to programming models and application-level tools. In other words, several missing layers still require dedicated tools to develop distributed quantum applications in a practical and reliable way.

Within this stack, NetQMPI [3] and CUNQA [4] are found. On the one hand, NetQMPI positions itself as a high-level programming framework that translates established concepts from the classical message-passing model into the distributed quantum domain, aiming to simplify the programming of DQC systems by eliminating the need to explicitly specify low-level communications among QPUs. In its current form, NetQMPI is implemented as an interface on top of the NetQASM SDK [5], which means its portability is largely tied to ecosystems that adopt NetQASM-compatible interfaces and backends. In contrast, CUNQA is a backend-level emulator that focuses on the lower part of the stack by emulating DQC architectures in high-performance computing (HPC) environments. Because NetQMPI and CUNQA target different layers (programming model vs. backend), they are conceptually compatible and can be combined in a single workflow, which is exactly the aim of this work[1].

Contributions. The main contributions of this work are the following:

- Decoupled software architecture: We propose a structural redesign of the NetQMPI implementation that strictly separates the user-facing SDK from the execution-facing runtime, ensuring that quantum programs are not intrinsically tied to a specific execution engine.
- Multi-backend integration: We demonstrate the architecture's viability by successfully mapping the Runtime adapters to two distinct environments: the NetQASM stack for quantum network simulation and the CUNQA emulator for HPC-based virtual QPU execution.
- Unified execution workflow: We define a seamless execution lifecycle in which dependency injection enables the same unmodified application script to be deployed across different backends.

The remainder of this paper is structured as follows. Section II provides background on NetQMPI and CUNQA. Section III details the decoupled architecture and explains the roles of the SDK and Runtime layers. Section IV describes the specific methodology used to add the NetQASM and CUNQA backends. Section V traces the application lifecycle and execution flow. Finally, Section VI concludes the paper.

[1]The code implementing this work can be found in https://github.com/NetQIR/netqmpi

2332-5666/26/$31.00 ©2026 IEEE
DOI 10.1109/ICDCSW72724.2026.00045

II. Background

A. NetQMPI in brief

NetQMPI provides an MPI-inspired programming model for DQC, targeting users who want to express multi-node quantum applications without manually orchestrating low-level networking details. As in classical MPI, the model follows a single-program, multiple-data (SPMD) paradigm, where each node executes the same program while operating on node-local resources, and coordination is expressed through communication primitives. In addition to classical-style messaging, NetQMPI introduces quantum-oriented primitives such as `qsend`/`qrecv` to transfer quantum data between nodes, achieved by consuming shared entanglement and applying the required local operations, as well as quantum-aware collective operations that reflect common coordination patterns in parallel programs (e.g., synchronization, distribution, and aggregation), but adapted to quantum constraints. For a detailed explanation, the reader is referred to [3].

Architecturally, NetQMPI's first implementation, as mentioned before, was implemented as an interface on top of the NetQASM SDK [5]. NetQASM serves as a low-level, universal instruction set architecture (ISA) and programming framework designed specifically for quantum networking [6]. It acts as an assembly-level bridge between high-level applications and the underlying quantum hardware or simulators, providing the fundamental instructions for generating, manipulating, and consuming quantum entanglement across distributed nodes. This design places NetQMPI at a high abstraction level: it exposes a communicator concept to represent groups of participating nodes and to scope communication and collective operations, while delegating device-level execution and network control to the underlying NetQASM stack. As a result, network initialization, connection management, and resource bookkeeping are largely hidden from the application developer, enabling programs to be written in terms of application-level coordination rather than backend-specific control logic. But, as said, NetQMPI's current implementation is tightly coupled to the NetQASM SDK. This has been done as a natural first step to validate the programming model and its semantics; however, such coupling is not fully aligned with NetQMPI's broader objective of acting as a backend-agnostic interface. In this sense, reducing dependency on a single stack is an important direction to better align with the interface-oriented design goal.

B. CUNQA in brief

CUNQA is an emulator of DQC architectures on HPC systems, designed to enable controlled experimentation with multi-QPU configurations and execution workflows [4]. Its core abstraction is the *virtual QPU* (vQPU), defined as a bundle of classical compute resources (e.g., CPU cores, memory, scheduling context) together with a simulation backend that makes it behave, from the user's perspective, like a QPU node in a distributed setting. This abstraction enables the instantiation of multiple QPU-like virtual devices within an HPC environment.

From an operational perspective, CUNQA provides tooling and an API to (i) provision and manage vQPUs, (ii) configure the emulated topology and execution settings, and (iii) submit user-defined quantum tasks to specific vQPUs. By separating the concerns of resource management and task execution, CUNQA supports repeatable experiments in which both the distributed architecture and simulation parameters can be systematically adjusted. In this sense, CUNQA primarily targets the lower part of the DQC software stack, as hinted in the previous section, and can serve as an experimentation layer underneath higher-level programming frameworks.

III. Decoupled Architecture

As noted at the end of Section I, NetQMPI and CUNQA operate at different layers of the distributed quantum software stack. This distinction enables the objective of this work: enabling programs expressed in NetQMPI primitives to be executed within the CUNQA framework. From the user's perspective, this means that distributed quantum protocols can be described simply using the high-level communication and circuit-building directives provided by NetQMPI, whilst, transparently to the user, an underlying platform such as CUNQA handles their execution. In this way, the programmer remains focused on expressing the distributed quantum task without having to deal directly with the mechanisms used to emulate or execute it on a particular backend.

However, enabling this workflow requires more than simply placing NetQMPI on top of CUNQA. Any practical implementation of NetQMPI must accommodate different responsibility buckets within the software stack. On the one hand, abstractions are needed to describe the quantum task itself, including circuit construction, communication semantics, and distributed coordination primitives. On the other hand, mechanisms are required for the actual realization of that task, such as process orchestration, resource provisioning, backend interaction, and simulation or execution control. If NetQMPI is to remain faithful to its role as a backend-agnostic interface, the tools used to design and specify distributed quantum programs should not be intrinsically tied to those responsible for executing or emulating them. Rather than replacing the existing execution pathway, the incorporation of CUNQA must be designed so that NetQASM continues to function as part of the execution stack, while CUNQA extends the backend-facing layer with the capabilities needed for orchestration and emulation in HPC environments. In this way, the separation between the user-facing programming layer and the backend-facing execution layer is preserved, while introducing a well-defined bridge between them.

Following this separation, NetQMPI is organized around two clearly differentiated domains. The NetQMPI SDK constitutes the user-facing layer, providing the abstractions required to describe distributed quantum tasks in terms of communication primitives, circuit construction, and coordination semantics. Complementarily, the NetQMPI Runtime serves as the execution-facing layer, providing the mechanisms to realize those abstractions on a concrete platform. As illustrated in

the UML class diagram in Figure 1, this organization allows application developers and platform integrators to operate on distinct parts of the stack while preserving a clear interface between program specification and execution.

A. NetQMPI SDK: High-Level User Abstraction

The SDK package encompasses all the classes and interfaces that users directly employ to develop their distributed applications. Its primary objective is to expose a set of rules and constructs that are entirely independent of the execution environment. Its most important blocks are:

- **`Environment` and `Circuit`:** These serve as the pillars of user-level programming. The `Environment` represents the local context of the node within the SPMD paradigm and acts as a *Factory* to instantiate `Circuit` objects. After creating this `Circuit` object, the user can chain operations onto it regardless of the backend on which these operations will be executed.
- **`OperationContainer`:** The operations appended to the circuit—both local gates such as `H` or `X`, and distributed ones like `expose` or `qsend`—inherit from the abstract base class `Operation`. These are aggregated into an `OperationContainer`, allowing complex sequences of instructions to be treated uniformly before being dispatched to the backend.
- **`QMPICommunicator`:** This class constitutes the cornerstone of the network abstraction. The user interacts with the `QMPICommunicator` to query their `rank` or the network `size`, and to perform the quantum communication directives, such as `qsend`, `qrecv`, `qgather`, etc. These directives are then managed by the selected backend.

B. NetQMPI Runtime: Pluggable Backend Integration

While the SDK abstracts quantum algorithms, the Runtime provides the necessary "behind-the-scenes" infrastructure for developers to plug in new execution engines without altering a single line of SDK code. To achieve this, two important features are implemented at the software design level:

- **`Executor` and Dependency Injection:** The abstract `Executor` class is the core of the Runtime. Its derived classes not only define how to bootstrap processes (`build_app()`) but also act as dependency injectors. During initialization, the `Executor` instantiates the backend-specific network communicator and injects it into the SDK's `Environment`.
- **Decoupling via *Adapters*:** To ensure a fully pluggable integration, each backend must provide its own derived implementations for both the `Executor` and the `Circuit`. For instance, when the SDK's operation container is compiled, the `CunqaCircuitAdapter` class intercepts these abstract operations via its `translate()` method and maps them directly to the native instructions of the CUNQA emulator on the HPC cluster.

This architecture guarantees that NetQMPI is not statically coupled to any specific communication technology or quantum engine, allowing the same script written with the SDK to be seamlessly evaluated across different backends simply by switching the execution adapter.

IV. Adding specific backends: NetQASM and CUNQA

After introducing the separation between the SDK and the Runtime, a natural, practical question arises: how is a new backend actually incorporated into NetQMPI? The answer follows directly from the architectural scheme described in Section III. Rather than modifying the SDK, backend integration is carried out entirely at the Runtime level by providing concrete implementations of the extension points already introduced:

1) An `Executor` implementation must be created to bootstrap the target execution environment and inject the required backend-specific dependencies into the SDK.
2) A concrete `Circuit` adapter must be implemented to translate the abstract operations accumulated in the SDK into the native instruction set of the backend.
3) A concrete `QMPICommunicator` must be provided so that the user-level communication interface can be mapped onto the resources offered by the target platform.

In this way, the application code remains unchanged, while the Runtime determines how that code is executed. Moreover, this integration procedure is the same for all supported backends. The difference between them lies only in how each of these three Runtime components is realized. NetQASM, which already existed in the original version of NetQMPI, is now reorganized according to this explicit scheme, while CUNQA is incorporated using the same process.

A. Step 1: defining the executor adapter

The first modification required to add a backend is implementing a concrete `Executor`. At this stage, the developer determines how processes are initialized, how resources are discovered, and how the backend-specific communicator is attached to the `Environment`. For NetQASM, this role is fulfilled by `NetQASMExecutorAdapter`, which drives execution atop the NetQASM stack and prepares the runtime flow for executing a NetQMPI application over the underlying quantum network infrastructure. For CUNQA, the corresponding class is `CunqaExecutorAdapter`, which connects NetQMPI to the CUNQA environment, including discovering available vQPUs and submitting distributed quantum tasks.

B. Step 2: implementing the circuit adapter

Once the backend can be initialized, the next step is to define how SDK-level operations are translated into native backend instructions. This is done through a concrete `Circuit` adapter. In the NetQASM backend, `NetQASMCircuitAdapter` performs this translation by mapping NetQMPI operations into NetQASM

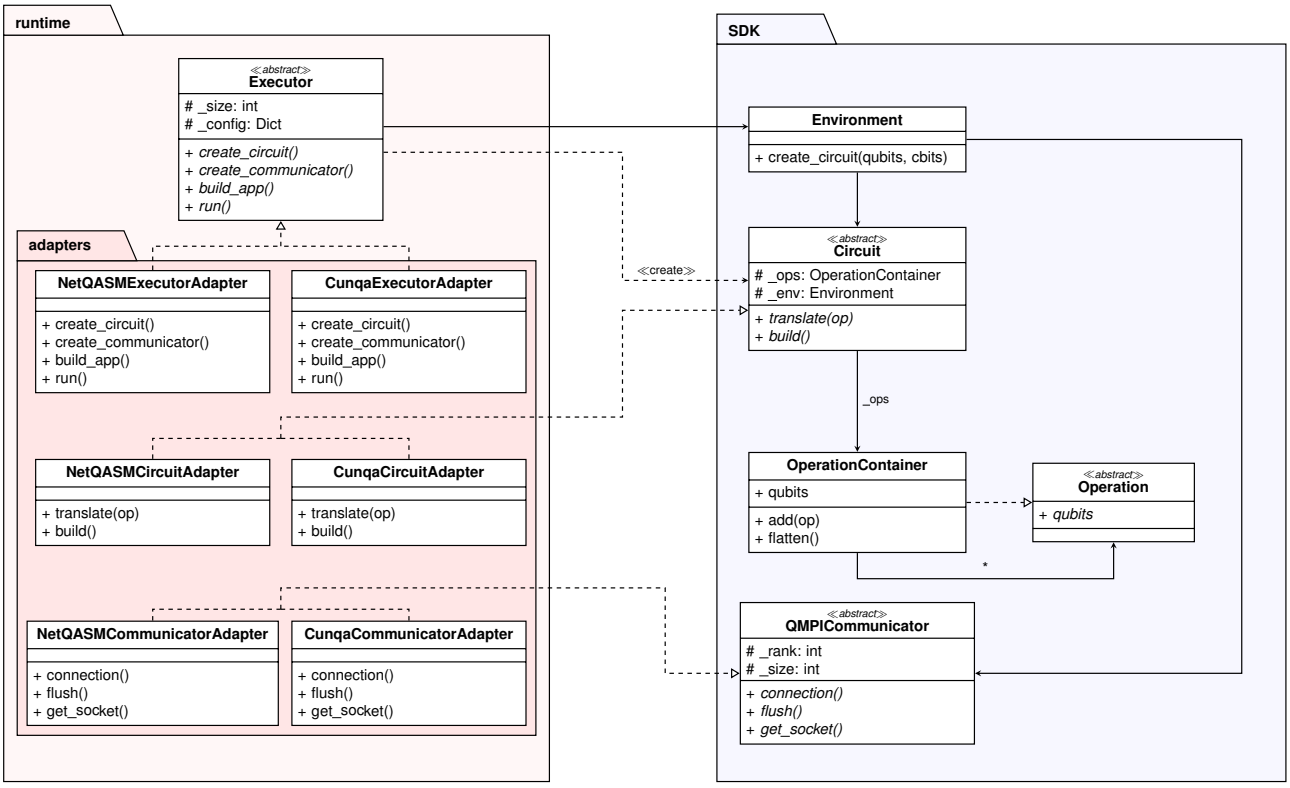

Fig. 1. **UML class diagram of the decoupled NetQMPI architecture.** The design strictly separates the user-facing SDK, which provides backend-agnostic programming abstractions (e.g., Environment, Circuit, QMPICommunicator), from the execution-facing Runtime. Through dependency injection and the Adapter pattern, the Runtime seamlessly integrates specific execution engines, such as NetQASM or CUNQA, without altering the application code.

instructions and routines. In the CUNQA backend, `CunqaCircuitAdapter` carries out the equivalent transformation, generating the circuit and communication structures required by the CUNQA execution model. In both cases, this adapter is the component that keeps backend-specific syntax and semantics isolated from the user-facing programming interface.

C. Step 3: providing the communicator implementation

The final step is to implement the communication layer required by the backend. Although the user interacts only with the generic `QMPICommunicator` interface, each backend must provide its own concrete implementation to map rank management and inter-node communication to the underlying platform resources. For NetQASM, this functionality is provided by `NetQASMCommunicator`, which encapsulates the backend communication resources, including the NetQASM connection itself, the management of EPR sockets between ranks, and the creation of classical sockets when needed. For CUNQA, the corresponding implementation is `CunqaCommunicator`, which adapts CUNQA's communication facilities to the common `QMPICommunicator` interface while exposing the rank and communicator size expected by the SDK.

Summarizing, backend integration in NetQMPI does not require changes to the SDK abstractions already used by the programmer. Instead, support for a new execution platform is obtained by implementing these three Runtime components in accordance with the backend's execution and communication model. NetQASM and CUNQA demonstrate that the same integration workflow can support substantially different environments while preserving a uniform programming interface at the SDK level.

V. Application Lifecycle and Execution Flow

To fully comprehend how the decoupled architecture operates in practice, it is essential to trace the lifecycle of a distributed quantum application within the framework. As NetQMPI adheres to the SPMD execution model, the developer writes a single, high-level Python script (e.g., `app.py`) using the NetQMPI SDK and this script is then executed in parallel across N virtual nodes. Figure 2 illustrates the end-to-end execution flow, detailing the interactions between the user space, the Runtime layer, and the SDK layer. The process is orchestrated through five sequential steps:

1) **Initialization and Backend Selection:** The lifecycle begins in the user space via the Command Line Interface (CLI). By executing a command such as `netqmpi -n N app.py`, the user defines the number of distributed

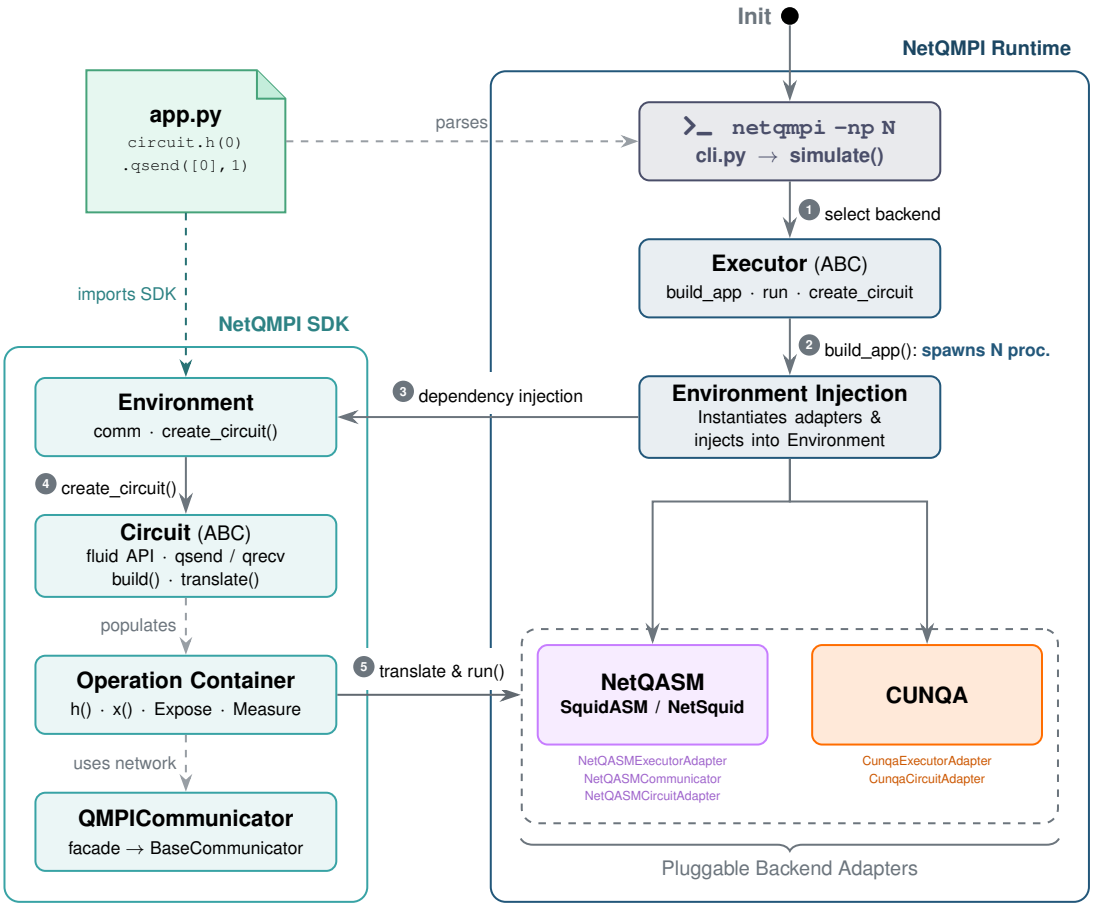

Fig. 2. **End-to-end execution flow of a NetQMPI application.** The diagram illustrates the five-step lifecycle orchestrated by the decoupled architecture: (1) CLI initialization and backend selection, (2) dynamic dependency injection by the Executor, (3) parallel process spawning following the SPMD model, (4) backend-agnostic circuit construction via the SDK, and (5) translation and dispatch of the abstract operations into the native instructions of the selected execution environment (NetQASM or CUNQA).

processes (N) and the target script. The user can also add the flags `--netqasm` or `--cunqa` to select the backend. The CLI parses this input, initializes the runtime environment via the `simulate()` function, and instantiates the backend selected.

2) **Dependency Injection:** For each process, the Executor links the abstract SDK with the concrete backend. This is done in the `build_app()` method, where the necessary backend-specific adapters are instantiated and dynamically injected into the SDK's `Environment`. This crucial step ensures that the SDK remains entirely unaware of the underlying emulator's specific implementation.
3) **Process Spawning:** Again, the Executor takes control and orchestrates the SPMD model by spawning N independent parallel processes. Each process corresponds to a distinct node in the simulated quantum network. In the case of CUNQA, this is done in the `run()`, while in the NetQASM backend resides in the `build_app()` method.
4) **Circuit Initialization:** Once the `Environment` is fully constructed and injected, the user's script invokes the `create_circuit()` method. This returns an abstract `Circuit` object, providing a fluid API for the programmer. As the script executes, it populates the `OperationContainer` with both local quantum gates (e.g., `h()`, `x()`) and network-aware primitives (e.g., `expose`, `qsend()`) through the `Communicator`.
5) **Translation and Execution:** After the circuit is fully populated, the execution phase concludes with the `translate & run()` routine. The injected backend adapters traverse the `OperationContainer`, translating the framework-agnostic instructions into the native syntax of the selected backend. Finally, the translated operations are dispatched to the backend for the execution of the circuit.

By strictly adhering to this execution flow, NetQMPI effectively shields users from the complexities of both parallel process management and low-level emulator APIs, offering

a seamless, unified programming experience for distributed quantum algorithms.

A. Code example: distributed superposition

To demonstrate the practical implications of this decoupled architecture, Listing 1 presents a complete NetQMPI application in which one node prepares a qubit in superposition and transmits it to a neighboring node in the network topology. It is worth noting that the implementation relies exclusively on the `Environment` and `QMPICommunicator` abstractions. From the programmer's perspective, routing the quantum state only requires invoking `comm.qsend()` and `comm.qrecv()`. Since backend-specific dependencies are injected dynamically during the initialization stage, the very same script can be executed either on a simulated quantum network through NetQASM or on an HPC cluster through CUNQA by simply adding the `--netqasm` or `--cunqa` flag to the command line, without modifying any part of the application logic.

```python
from netqmpi.sdk.environment import Environment

def main(env: Environment = None):
    comm = env.comm
    rank = comm.rank

    next_rank = comm.get_next_rank(rank)
    prev_rank = comm.get_prev_rank(rank)

    with comm:  # Execute only within this block
        circuit = env.create_circuit(num_qubits
            =1, num_clbits=1)

        if rank == 0:
            circuit.h(0)
            comm.qsend(circuit, [0], next_rank)
        else:
            comm.qrecv(circuit, [0], prev_rank)
            circuit.measure(0, 0)

    results = comm.results  # Execution results

    if rank != 0:
        print_info(f"measure: {results}")
    else:
        print_info("teleport. complete", rank)
```

Listing 1. Distributed superposition with NetQMPI

To execute the program, the user simply invokes an MPI-like command such as:

```
netqmpi -n NUM_PROC file.py --cunqa --shots=1024
```

VI. Conclusions

This work presents a structural evolution of the NetQMPI framework, introducing a decoupled architecture that strictly separates the user-facing programming model from the underlying execution backends. By establishing a clear boundary between the NetQMPI SDK and the Runtime, we have enabled the seamless integration of diverse execution environments without requiring any modifications to the application code.

We demonstrated this capability by successfully integrating two completely distinct backends into NetQMPI: CUNQA, an HPC-based emulator of DQC architectures, and the NetQASM stack, a low-level instruction set architecture and programming framework for quantum networking. This integration provides developers with a highly versatile toolchain: programmers can leverage NetQMPI's high-level message-passing primitives to express multi-node quantum coordination, while relying on CUNQA to orchestrate and emulate those operations efficiently across compute nodes in an HPC cluster. Ultimately, this decoupled approach shields developers from low-level API complexities, fosters software portability, and brings the community one step closer to a robust, hardware-agnostic ecosystem for DQC.

Acknowledgements

This work was funded by the European Union EuroHPC program with grant agreement 101194491 and by MICIU/AEI/10.13039/501100011033 with project number PCI2025-163229, the Agencia Estatal de Investigación (Spain) (PID2022-141623NB-I00), the Consellería de Cultura, Educación e Ordenación Universitaria Galician Research Center accreditation 2024–2027 ED431G-2023/04, and the European Regional Development Fund (ERDF).

References

[1] S. W. Loke, "From distributed quantum computing to quantum internet computing: an overview," *arXiv*, 8 2022.

[2] D. Barral, F. J. Cardama, G. Díaz-Camacho, D. Faílde, I. F. Llovo, M. Mussa-Juane, J. Vázquez-Pérez, J. Villasuso, C. Piñeiro, N. Costas, J. C. Pichel, T. F. Pena, and A. Gómez, "Review of distributed quantum computing: From single QPU to high performance quantum computing," *Computer Science Review*, vol. 57, p. 100747, 8 2025.

[3] F. J. Cardama and T. F. Pena, "NetQMPI: An MPI-inspired library for programming distributed quantum applications over quantum networks using netqasm sdk," *ArXiV*, 12 2025.

[4] J. Vázquez-Pérez, D. Expósito-Patiño, M. Losada, Álvaro Carballido, A. Gómez, and T. F. Pena, "CUNQA: a distributed quantum computing emulator for HPC," *arXiv:2511.05209*, vol. 1, 11 2025.

[5] A. Dahlberg, B. V. D. Vecht, C. D. Donne, M. Skrzypczyk, I. T. Raa, W. Kozlowski, and S. Wehner, "Netqasm—a low-level instruction set architecture for hybrid quantum–classical programs in a quantum internet," *Quantum Science and Technology*, vol. 7, p. 035023, 6 2022.

[6] J. Illiano, M. Caleffi, A. Manzalini, and A. S. Cacciapuoti, "Quantum internet protocol stack: A comprehensive survey," *Computer Networks*, vol. 213, p. 109092, 8 2022.