IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 9 July 2026, accepted 3 August 2026, date of publication 13 August 2026, date of current version 20 August 2026.

Digital Object Identifier 10.1109/ACCESS.2026.3723566

RESEARCH ARTICLE

NetQMPI: An MPI-Inspired Library for Programming Distributed Quantum Applications Over Quantum Networks Using NetQASM SDK

F. JAVIER CARDAMA[1], (Member, IEEE), JORGE VÁZQUEZ-PÉREZ[3], TOMÁS F. PENA[1,2], (Senior Member, IEEE), AND ANDRÉS GÓMEZ[3]

[1]Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela, 15782 Santiago de Compostela, Spain
[2]Departamento de Electrónica e Computación, Universidade de Santiago de Compostela, 15782 Santiago de Compostela, Spain
[3]Galician Supercomputing Center (CESGA), 15705 Santiago de Compostela, Spain

Corresponding author: F. Javier Cardama (javier.cardama@usc.es)

This work has been supported by the "Ministerio de Ciencia, Innovación y Universidades del Gobierno de España" through the European Union NextGenerationEU recovery plan PRTR-C17.I1. This project is funded by the "European Union" through the European High Performance Computing Joint Undertaking (EuroHPC JU) programme under grant agreement 101194491, and by the "Ministerio de Ciencia, Innovación y Universidades del Gobierno de España" (MICIU) and the "Agencia Estatal de Investigación (AEI)" (AEI, 10.13039/501100011033), under the projects PCI2025-163133, PCI2025-163180, and PCI2025-163229. Financial support from the "Ministerio de Economía, Comercio y Empresa del Gobierno de España" under the grants PID2019-104834GB-I00, PID2022-141623NB-I00, and PID2022-137061OB-C22. Financial support provided to CiTIUS (Research Centre on Intelligent Technologies) through the Galician Research Center accreditation 2019-2027 (grant ED431G-2023/04), funded by the "Consellería de Cultura, Educación e Ordenación Universitaria". This research was co-funded by the "European Union" (European Regional Development Fund – ERDF). The authors acknowledge the financial support from the "Consellería de Educación, Cultura y Ordenación Universitaria da Xunta de Galicia" through the predoctoral grant ED481A-2022/241 and IN606A-2025/012.

ABSTRACT Distributed Quantum Computing (DQC) is essential for scaling quantum algorithms beyond monolithic constraints, yet current software ecosystems require tedious manual orchestration of low-level physical resources. This paper presents NetQMPI, a high-level Python framework that adapts the Message Passing Interface (MPI) standard to quantum networks utilizing the Single Program Multiple Data (SPMD) paradigm. Built as a middleware over the NetQASM SDK, NetQMPI completely abstracts physical network topologies and automates resource initialization through a unified Communicator interface. We introduce semantic point-to-point primitives and novel collective operations (expose and unexpose) that safely navigate the constraints of the No-Cloning Theorem by leveraging multipartite entanglement for data distribution. Our comparative analysis demonstrates that NetQMPI effectively decouples algorithmic logic from network scale, reducing code complexity for generating an N-node GHZ state from $\mathcal{O}(N^2)$ to a constant complexity of $\mathcal{O}(1)$. Furthermore, the framework is backend-agnostic, enabling immediate execution on high-fidelity physical simulators such as NetSquid (via SquidASM) and ensuring forward compatibility with future NetQASM-compliant hardware.

INDEX TERMS Distributed quantum computing, high-performance computing, message passing interface (MPI), quantum networks, single program multiple data (SPMD), NetQASM, quantum teleportation.

I. INTRODUCTION

In recent years, quantum computing has emerged as a computational paradigm capable of offering exponential speedups for problems intractable on classical machines [1], [2], such as integer factorization [3] and unstructured search [4]. However, realizing these algorithms at scale requires a substantial number of qubits and robust error correction capabilities, which remain challenging for current hardware in the Noisy Intermediate-Scale Quantum (NISQ) era [5].

To overcome these scaling limitations, Distributed Quantum Computing (DQC) has emerged as a promising path. By interconnecting multiple Quantum Processing Units (QPUs) via a quantum network, it is possible to create a virtual cluster capable of executing large-scale algorithms [6],

The associate editor coordinating the review of this manuscript and approving it for publication was Shadi Alawneh.

[7], [8]. However, orchestrating computations across such a distributed infrastructure introduces engineering complexities [9]. Unlike classical networks, quantum networks require the management of fragile resources such as entanglement, Einstein-Podolsky-Rosen (EPR) pairs, and the tight synchronization of classical and quantum control planes [10], [11], [12].

Nevertheless, the transition from monolithic to distributed quantum algorithms introduces significant software engineering challenges [13]. Most existing software tools require researchers to program at the physical or link layer, where they must explicitly manage elementary operations [6], [14], [15]. At this level, developers are often burdened with defining hardware-specific parameters such as photon emission timing, fiber attenuation, and memory decoherence, while also manually orchestrating network protocols like entanglement generation (EPR pairs), routing, and the tight synchronization of classical and quantum control planes [16].

This low-level approach not only increases development complexity but also couples the algorithmic logic to specific hardware topologies or simulation backends, severely limiting code portability and maintainability. In the realm of classical High-Performance Computing (HPC), this challenge was successfully addressed by the Message Passing Interface (MPI) standard, which unified distributed programming, typically following the Single Program Multiple Data (SPMD) paradigm. While theoretical proposals for a Quantum Message Passing Interface (QMPI) exist [17], the ecosystem still lacks a functional, standardized implementation that bridges the gap between high-level algorithmic design and the execution on rigorous quantum network stacks.

A. CONTRIBUTIONS

In this work, we present **NetQMPI**, a high-level MPI-inspired Python library for programming distributed quantum applications over quantum networks. NetQMPI acts as a middleware layer over the standard NetQASM Software Development Kit (SDK), allowing developers to write distributed quantum algorithms using high-level semantics while ensuring execution on rigorous network simulators.

The main objectives and novel contributions of this work are:

- **Adoption of the SPMD Paradigm:** We propose a shift from role-based scripting to a unified Single Program Multiple Data model. By injecting logical `rank` identifiers, NetQMPI enables a single source file to orchestrate complex protocols across N nodes, thereby decoupling code complexity from network size.
- **Abstraction of Physical Resources:** We introduce the `QMPICommunicator` and semantic primitives (e.g., `qsend`, `qrecv`) that abstract the management of EPR sockets and classical control planes, which allows researchers to focus on algorithmic logic rather than the intricate details of entanglement generation and fidelity management.
- **Novel Collective Operations:** We define and implement quantum-specific collective operations, such as `expose` and `unexpose`. These primitives address the constraints of the No-Cloning Theorem by leveraging multipartite entanglement to share quantum information across the communicator without physical copying.
- **Backend-Agnostic Execution:** By targeting the standardized NetQASM Instruction Set Architecture (ISA), NetQMPI serves as a unified interface that can execute applications on high-fidelity simulators (such as NetSquid via SquidASM) and is forward-compatible with future quantum hardware, bridging the gap between abstract algorithm design and realistic physical simulation.

This paper is organized as follows. Section II introduces the related work including tools for DQC programming and simulation. Section III-C provides an overview of the NetQASM framework, distinguishing between its ISA and the SDK, while highlighting the programming complexity inherent in the current ecosystem. Section IV details the architecture of NetQMPI as well as the creation process on NetQASM SDK and the high-level semantic functions provided. Section V presents a comparative analysis of the various tools in the literature described in the related work section, and, finally, Section VI concludes the paper.

II. RELATED WORK

Recent advances in quantum algorithms have driven considerable efforts in developing software tools for programming and simulating distributed quantum systems. This section presents the most relevant tools in this domain, distinguishing between high-level abstractions and low-level physical simulators.

A. SOFTWARE TOOLS FOR PROGRAMMING DISTRIBUTED QUANTUM SYSTEMS

To manage the complexity of distributed quantum states, several frameworks have been proposed to abstract the underlying physical layer.

QuNetSim [18] provides a high-level Python API for simulating network protocols (e.g., Quantum Key Distribution (QKD), teleportation) without dealing with hardware physics.

- *Strengths:* It offers an extremely gentle learning curve and built-in implementations of standard protocols.
- *Weaknesses:* It is purely a simulator with no path to physical hardware execution; its abstraction level hides the resource management details (like entanglement fidelity) required for realistic system programming.

Interlinq [19] offers a high-level abstraction layer built upon the QuNetSim communication framework, designed to automate circuit partitioning and message scheduling across distributed nodes.

- *Strengths:* It simplifies distributed execution by automating circuit partitioning and teleportation scheduling, allowing monolithic circuits to be run across multiple nodes (“circuit knitting”, i.e., splitting

a quantum program to fit onto smaller devices linked by classical communication [20]) without manual intervention for small-scale systems.
- *Weaknesses:* It suffers from severe scalability issues, where execution time and memory usage grow exponentially with the number of qubits and nodes. Additionally, its automatic partitioner lacks advanced optimization techniques, leading to redundant teleportations and communication overhead, while imposing rigid constraints on node configurations.

QMPI, proposed by Häner et al. [17], introduces the theoretical foundation for applying the Message Passing Interface (MPI) standard to quantum computing.

- *Strengths:* It leverages the mature MPI standard (commands like `Send`, `Recv`, `Bcast`) familiar to the HPC community.
- *Weaknesses:* Although the authors mention a prototype implementation in C++ using classical MPI and multi-threading, this software is closed-source and not publicly available for community verification or deployment. Furthermore, their prototype appears limited to purely functional software simulation. It lacks a defined integration path with an executable, realistic quantum network stack. Consequently, it cannot evaluate vital network parameters such as entanglement generation delays, link-layer fidelity, or memory decoherence. In addition, it does not address how to execute collective operations such as `Bcast` or `Allgather`; these primitives inherently imply data copying in classical MPI, which conflicts with the No-Cloning Theorem, and the proposal offers no physical mechanism to resolve this.
- *Advantage of NetQMPI (This Work)*: In contrast to the closed-source functional prototype of QMPI, NetQMPI provides a fully operational, open-source Python middleware. By natively targeting the standardized NetQASM ISA, NetQMPI bridges the gap between abstract MPI semantics and executable quantum network hardware. This integration allows any user to install the library and seamlessly execute high-level distributed algorithms on top of physically rigorous, discrete-event simulators like NetSquid (via SquidASM) or directly on future NetQASM-compliant quantum clusters.

B. QUANTUM NETWORK SIMULATORS

This category encompasses tools designed to simulate any quantum network. These tools allow the execution of platform-independent code, often relying on lower-level engines for physical accuracy.

SimulaQron [21] focuses on the distributed nature of the network. It runs a simulation where each network node is a separate process on the host machine, communicating via classical sockets to imitate a real distributed architecture.

- *Strengths:* It enforces a strict separation of memory between nodes, making it ideal for verifying client-server logic and distributed application flows.
- *Integration:* While it acts as a standalone distributed backend, it is part of the QuTech ecosystem (a suite of standardized quantum networking tools developed at TU Delft) and was a precursor to modern execution environments.

SquidASM [22] is the specialized execution tool for NetQASM applications. It acts as a bridge between the application layer and the physical layer.

- *Strengths:* It allows developers to run NetQASM routines on a simulated network with high fidelity.
- *Integration:* SquidASM is explicitly built on top of NetSquid (see next subsection). It translates the high-level instructions from the SDK into discrete events that NetSquid can process, providing a realistic noise model while maintaining a programmable interface.

C. DISCRETE-EVENT SIMULATORS

At the lowest level of abstraction, these tools model the physical transmission of qubits, time-dependent noise, and hardware components (memories, repeaters, fibers).

NetSquid [23] is the state-of-the-art platform for modeling quantum networks at the physical layer.

- *Strengths:* It handles precise timing of photon emission, fiber loss, and quantum memory decoherence. It serves as the physics engine powering higher-level tools like SquidASM.
- *Weaknesses:* The complexity of defining hardware components makes it cumbersome for general application programming; it is designed for network architects rather than algorithm developers.

SeQUeNCe [24] is a customizable discrete-event simulator with a strong focus on the network layer (routing, resource management).

- *Strengths:* Excellent for studying network topology and protocol stacks, and available as open-source.
- *Weaknesses:* Like NetSquid, it presents a high barrier to entry for software developers simply wanting to write quantum applications using standard programming paradigms.

III. BACKGROUND AND QUANTUM NETWORKING FOUNDATIONS

A. NO-CLONING THEOREM

A fundamental divergence between classical and quantum distributed computing lies in the manipulation of information. In the classical domain, the MPI heavily relies on the ability to perfectly duplicate data across nodes, such as in the `MPI_Bcast` (broadcast) operation. However, in the quantum realm, this straightforward duplication is physically prohibited by the No-Cloning Theorem [25].

The theorem states that it is impossible to create an independent and identical copy of an arbitrary, unknown quantum state. The proof of this theorem is simple. Suppose

there exists a hypothetical unitary cloning operator U that takes an arbitrary state $|\psi\rangle$ and a blank target state $|0\rangle$, copying the information such that:

$$U(|\psi\rangle \otimes |0\rangle) = |\psi\rangle \otimes |\psi\rangle \tag{1}$$

If this operator U is applied to two distinct, arbitrary quantum states, $|\psi\rangle$ and $|\phi\rangle$, the unitarity of the operation (which must preserve inner products) dictates that:

$$(\langle\phi| \otimes \langle 0|)U^\dagger U(|\psi\rangle \otimes |0\rangle) = (\langle\phi| \otimes \langle\phi|)(|\psi\rangle \otimes |\psi\rangle) \tag{2}$$

Because $U^\dagger U = I$ and $\langle 0|0\rangle = 1$, this simplifies to:

$$\langle\phi|\psi\rangle = (\langle\phi|\psi\rangle)^2 \tag{3}$$

This equality is strictly valid only when $\langle\phi|\psi\rangle = 0$ or $\langle\phi|\psi\rangle = 1$. Consequently, a quantum system can only be perfectly cloned if the states belong to a known orthogonal basis. For any unknown superposition, the copying process fails.

The immediate implication for DQC is that quantum information cannot be passively broadcasted or copied to multiple QPUs. Any attempt to measure or duplicate the state will result in its collapse. Therefore, transferring quantum states between nodes necessitates protocols like quantum teleportation, which inherently destroys the original state at the source, or the distribution of multipartite entanglement (e.g., GHZ states) to create shared logical contexts without violating the no-cloning principle.

B. LOW-LEVEL DISTRIBUTED PRIMIVITES: TELEDATA AND TELEGATE

Because the No-Cloning Theorem prevents the direct replication of quantum states, distributed quantum networks must rely on entanglement as a fundamental resource to move information or execute multi-qubit operations across geographically separated nodes. At the physical and link layers, these operations are typically implemented via two foundational protocols: teledata [26] (quantum teleportation) and telegate [27] (distributed quantum gates). The teledata protocol allows the transfer of an unknown quantum state $|\psi\rangle$ from a source node (Alice) to a target node (Bob) without physically copying the data qubit itself. This process, illustrated in Figure 1, strictly requires a pre-shared entangled state, typically an EPR pair in the Bell state $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Alice performs a joint Bell-state measurement on her data qubit and her half of the EPR pair, collapsing the entanglement. She then transmits the two classical bits resulting from this measurement over a standard classical network channel. Finally, Bob uses these classical bits to apply the corresponding local Pauli operations (X and/or Z) to his half of the EPR pair, successfully reconstructing the original state $|\psi\rangle$.

Similarly, executing a multi-qubit gate between nodes, such as a Controlled-NOT (CNOT) where the control qubit resides on Alice's node and the target qubit on Bob's node, cannot be achieved directly. This operation, known as a

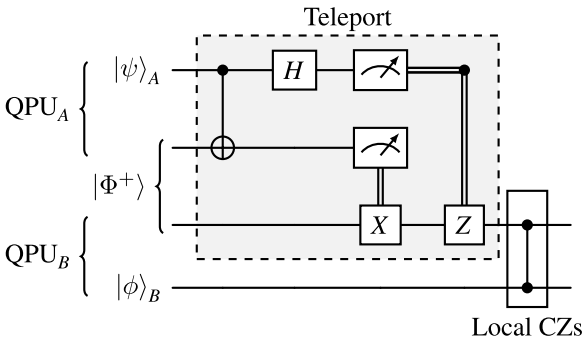


FIGURE 1. Quantum circuit representation of the *teledata* (teleportation) protocol. The source node transfers an unknown quantum state to the target node by consuming a pre-shared EPR pair and transmitting two bits of classical information for local Pauli corrections.

telegate, is depicted in Figure 2, and, as shown, it consumes a pre-shared EPR pair. It requires Alice and Bob to perform local CNOT operations between their respective operational qubits and their halves of the entangled pair, followed by local measurements. The measurement outcomes must be cross-transmitted classically so that both nodes can apply the necessary Pauli corrections to finalize the distributed gate execution.

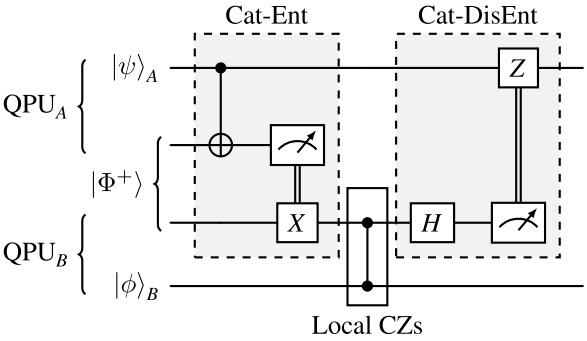


FIGURE 2. Quantum circuit representation of the *telegate* protocol for a distributed CNOT operation. The execution requires the synchronization of local quantum operations, EPR pair consumption, and bilateral classical communication for error correction.

While these primitives are physically robust, exposing them directly to the application layer introduces severe software engineering bottlenecks. As previously demonstrated, implementing a single remote operation requires developers to manually orchestrate EPR socket allocation, explicit hardware synchronization (e.g., waiting for entanglement generation), and tight coordination between the quantum and classical control planes. When scaling these protocols to larger topologies, such as generating a multi-node GHZ state, the requirement to manually code these low-level interactions for every network link leads to verbose, error-prone software that fails to scale efficiently.

C. NetQASM OVERVIEW: ISA AND SDK

The foundation of the work presented in this paper is the NetQASM framework [22], which can be distinguished into

two main components: the Instruction Set Architecture (ISA) and the Software Development Kit (SDK).

NetQASM defines a low-level, platform-independent ISA explicitly designed for quantum networks. It serves as a common language that abstracts the heterogeneity of underlying hardware. Programs compiled into NetQASM assembly can be executed on various backends, ranging from high-fidelity physical simulators like SquidASM (based on NetSquid) or SimulaQron. This architecture separates the quantum logic from the control hardware, allowing for hybrid quantum-classical execution.

To illustrate the granularity of control required by this architecture, Figure 3 presents a snippet of Alice's operations. The execution flow begins with explicit resource management, where the program manually assigns a value to a classical register (`set R0 0`) to identify, allocate (`qalloc`), and initialize (`init`) a local control qubit. Subsequently, the entanglement generation process requires preparing the arguments in specific memory addresses—storing the number of pairs at `@0` and the type at `@1`—before triggering the `create_epr` instruction. Notably, the code must include a `wait_all` instruction to explicitly block execution until the physical hardware confirms the entanglement generation. Finally, the script manually maps the remote entangled qubit to a register (`set R1 1`), performs a local two-qubit gate (`cnot`), measures the result into a classical register (`meas`), and explicitly frees the quantum memory with `qfree`.

```
# NetQASM Assembly (Alice Side)
set R0 0          # Register for Qubit ID
qalloc R0         # Allocate Control Qubit
init R0           # Initialize to |0>

# -- Entanglement Generation --
store 1 @0        # Store arg: number of pairs
store 0 @1        # Store arg: type
create_epr 0 1 0 0 @0
wait_all @0       # Explicit wait for hardware

set R1 1          # Manually assign ID to EPR qubit
cnot R0 R1        # Local CNOT
meas R1 R2        # Measure EPR into Register R2
qfree R1          # Free EPR qubit resource
```

FIGURE 3. Snippet of NetQASM ISA Assembly corresponding to Alice's operations. It highlights the low-level management of registers (e.g., `R0`, `R1`) and memory addresses (e.g., `@0`) required by the architecture.

To facilitate programming without writing raw assembly, the NetQASM SDK provides a Python interface for writing applications for specific network roles (e.g., "Alice" and "Bob"). A central abstraction in the SDK is the *EPR Socket*. By opening an EPR socket between two nodes, the programmer can request the generation of entanglement (e.g., creating Bell pairs) utilizing the underlying network link layer.

Figure 4 demonstrates the implementation of a teleport protocol using the NetQASM SDK, highlighting the steps required to transfer a qubit state from Alice to Bob.

D. PROGRAMMING COMPLEXITY

While the SDK abstracts the physical generation of entanglement, it does not provide high-level primitives for qubit transmission. Crucially, to transfer a quantum state from one node to another, the generation of an EPR pair is merely the first step. The programmer must manually implement the full quantum protocol communication *ad hoc*.

For example, to send a single qubit via teleportation [26] using the SDK, the developer must explicitly:

1) Create or request entanglement via the EPR Socket and wait for completion.
   - Alice: line 12.
   - Bob: line 9.
2) Perform local gates (CNOT, Hadamard) between the payload qubit and the entangled qubit.
   - Alice: lines 15–16.
3) Measure the qubits and obtain the classical outcomes.
   - Alice: Lines 17–18.
4) Send the classical bits to the receiver using a separate classical channel.
   - Alice: Line 23.
5) Receiver side: read these bits and apply the corresponding Pauli corrections (X or Z) to recover the state.
   - (Bob: Line 15–21).

This requirement to manually orchestrate classical and quantum synchronization for every transfer leads to verbose and error-prone code. This limitation is the primary motivation for NetQMPI, which encapsulates these complex workflows into simple, atomic communication primitives.

IV. NetQMPI

This section introduces NetQMPI, a high-level Python library designed to simplify the development of distributed quantum applications. Inspired by the standardized Message Passing Interface (MPI) used in classical HPC, NetQMPI adopts the Single Program Multiple Data (SPMD) paradigm, which enables developers to write a single, platform-independent source code that orchestrates quantum operations across multiple network nodes without manually managing the underlying physical entanglement resources.

A. SOFTWARE STACK

Figure 5 illustrates the architectural position of NetQMPI within the quantum network software ecosystem. NetQMPI acts as a middleware layer that sits directly on top of the NetQASM SDK.

A primary objective of this architecture is to bridge the gap between high-level algorithmic design and platform-specific execution. By abstracting the NetQASM SDK, NetQMPI hides the complexity of connection management and explicit instruction compilation. The user application interacts solely with NetQMPI primitives (e.g., `qsend`, `qrecv`), which are then translated at runtime into the corresponding NetQASM subroutines—handling `EPRSocket` lifecycle, `NetQASMConnection` management, and classical control flow transparently.

Alice (Control Node)

```python
# Create a socket to send classical information
socket = Socket("Bob", "Alice", log_config=log_config)

# Create an EPR socket for entanglement generation
epr_socket = EPRSocket("Bob")

with NetQASMConnection(epr_sockets=[epr_socket]) as conn:
    # Create a qubit to teleport
    q = Qubit(conn)

    # Create EPR pairs
    epr = epr_socket.create_keep()[0]

    # Teleport
    q.cnot(epr)
    q.H()
    m1 = q.measure()
    m2 = epr.measure()

# Send the correction information
m1, m2 = int(m1), int(m2)

socket.send_structured(StructuredMessage("Corrections", (
    m1, m2)))

```

Bob (Target Node)

```python
# Create a classical socket
socket = Socket("Alice", "Bob")

# Create an EPR socket
epr_socket = EPRSocket("Alice")

with NetQASMConnection(epr_sockets=[epr_socket]) as conn:
    # 1. Receive EPR qubit
    epr = epr_socket.recv_keep()[0]

    # 2. Flush to sync
    conn.flush()

    # 3. Get and apply the corrections
    m1, m2 = socket.recv_structured().payload

    if m2 == 1:
        epr.X()

    if m1 == 1:
        epr.Z()

    # 4. Final flush to sync
    conn.flush()
```

FIGURE 4. Implementation of a teleport protocol using NetQASM SDK. The developer is responsible for manually orchestrating the EPR socket creation, classical message passing, and synchronization (flush).

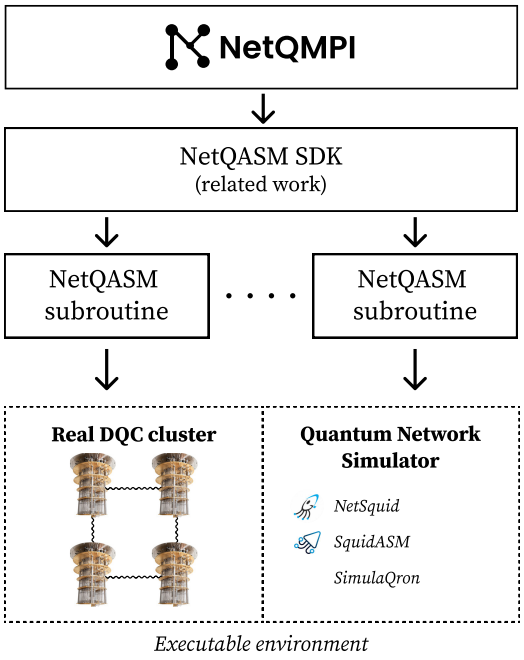


FIGURE 5. The NetQMPI software stack. The library wraps the verbose NetQASM SDK primitives, providing a unified API for the application layer while relying on the standard NetQASM ISA to interface with backend simulators, such as SquidASM or SimulaQron, or real DQC clusters supporting NetQASM.

Because these primitives compile down to the standardized NetQASM ISA, NetQMPI is inherently backend-agnostic. This abstraction allows developers to write code that is completely decoupled from the underlying infrastructure, enabling the execution of the same source code on both *simulated networks* and *physical quantum networks* without modification.

Furthermore, this design leverages the robust capabilities of the ecosystem. Specifically, it allows users to utilize high-fidelity simulators like NetSquid (via the SquidASM interface) without facing its steep learning curve. While NetSquid is the state-of-the-art for modeling physical noise and network delays, programming it directly for application development is notoriously complex. NetQMPI enables researchers to validate algorithms using NetSquid's realistic physics engine while maintaining a high-level, abstract programming model.

B. CONSTRUCTION OF NetQMPI FROM NetQASM SDK

NetQMPI[1] is constructed as a wrapper that fundamentally alters the workflow of the NetQASM SDK. To achieve the transition from the role-based scripts required by the SDK to the Single Program Multiple Data (SPMD) paradigm, NetQMPI relies on a specific internal architecture. Figure 6 illustrates the organization of the source code.

The core orchestration is handled by the `external.py` module, which manages the multiprocess environment mapping. The network state and automated entanglement generation are encapsulated within the `communicator` package. Finally, the specific communication logic is organized under `primitives`, divided into `p2p` and `collective` modules, whose specific implementations will be detailed in Subsections IV-C and IV-D, respectively.

[1]The complete source code is available at: https://github.com/netqir/netqmpi

```
netqmpi/
|-- sdk/
|   |-- external.py
|   |-- communicator/
|   |   |-- communicator.py
|   |-- primitives/
|   |   |-- collective/
|   |   |-- p2p/
```

FIGURE 6. Directory structure of the NetQMPI library. The `external.py` module orchestrates the multiprocess execution, while `communicator.py` manages the automated entanglement generation.

1) SINGLE PROGRAM ARCHITECTURE

In the standard NetQASM SDK, developers must write N separate source files (e.g., `app_alice.py` and `app_bob.py`), one for each node in the network. NetQMPI unifies this into a single file, following the SPMD paradigm.

To enable this, NetQMPI utilizes the abstractions of *rank* and *size*. For readers unfamiliar with HPC terminology, the *rank* is a unique integer identifier assigned to each node in the distributed network, ranging from 0 to $size - 1$, while the *size* represents the total number of participating nodes in the communicator. These values are injected automatically by the library into the user's code at runtime, allowing the developer to define the logic for all nodes in a single function signature.

The user script must accept `rank` and `size` as arguments, which are then used to initialize the `QMPICommunicator`, as shown in Figure 7.

```python
from netqmpi.sdk.communicator import QMPICommunicator

def main(app_config=None, rank=0, size=1):
    # The rank is injected by NetQMPI
    # and used to initialize the local communicator
    COMM_WORLD = QMPICommunicator(rank, size, app_config)
    print(f"Hello, rank={rank} of {size} processes")
```

FIGURE 7. An example of the rank and size variable injection with the global communicator creation.

When the user executes this script using the Command Line Interface (CLI), the `cli.py` module parses the `-n` argument (number of processes) and triggers the `external.py` dispatcher. This dispatcher spawns the simulation processes using the NetQASM SDK, passing the specific rank to each instance. For example, executing the code with three nodes results in:

```
$ netqmpi -n 3 script.py
Hello, rank=0 of 3 processes
Hello, rank=2 of 3 processes
Hello, rank=1 of 3 processes
```

As illustrated in Figure 8, the execution flow is orchestrated by `external.py`. It parses the unified script, defines the network roles based on the requested size, and injects the corresponding rank information into the application's entry point. Finally, it hands over the configuration to the underlying NetQASM SDK, which manages the physical spawning of the runtime processes (e.g., Alice, Bob, Charlie).

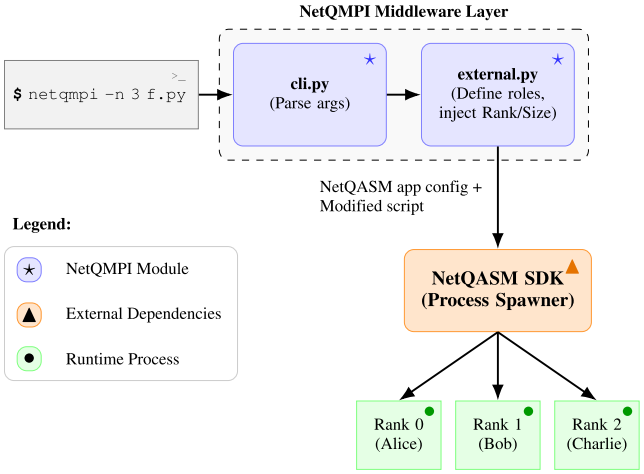


FIGURE 8. Execution flow of a NetQMPI application. The `external.py` module orchestrates the multiprocess environment, injecting rank information into the unified user script and delegating the process management to the NetQASM SDK.

2) AUTOMATED NETWORK INITIALIZATION

The core abstraction that manages the logical connectivity between the distributed nodes is the `QMPICommunicator` class, defined in the `communicator` package.

Conceptually, a communicator acts as a virtualization layer that groups a set of logical processes connected through a logical network. This construct abstracts the underlying complexity of the infrastructure, where logical processes (identified by ranks) are mapped to physical resources (Quantum Processing Units) via a physical interconnection network. This relationship is visually described in Figure 9. Figure 9(a) depicts the physical reality where QPUs are connected via a Quantum Router, while Figure 9(b) illustrates how the software stack (OS/Middleware) maps the high-level Logical Ranks ($1 \ldots n$) to these physical resources transparently.

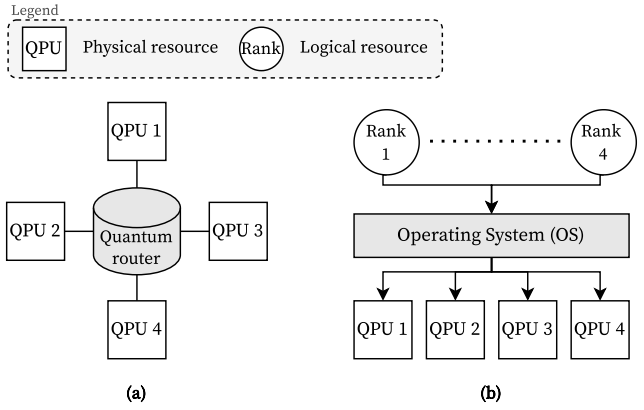


FIGURE 9. Logical abstraction for scheduling physical resources. (a) An example of physical resources interconnected in a physical quantum network. (b) The Communicator provides a logical layer where abstract Ranks are assigned to processes, and the OS/Middleware automatically handles the mapping to the available physical QPUs.

This approach mirrors the architecture of classical distributed computing, allowing developers to focus on the algorithmic logic rather than the physical topology. For a comprehensive definition of this abstraction model in distributed quantum computing, we refer the reader to the architectural framework presented in [14], specifically Figure 6.

From an implementation perspective, the communicator automates resource management that is currently handled manually in the NetQASM SDK. In NetQASM, establishing a network requires the user to instantiate a `NetQASMConnection` and define specific `EPRSocket` objects for every pair of communicating nodes, leading to an $O(N^2)$ initialization complexity in fully connected topologies. NetQMPI resolves this by performing the following automated steps during initialization:

1) It assigns a unique `rank` to each logical process.
2) It iterates through the network configuration and automatically instantiates the necessary mesh of `EPRSocket` between all participating ranks.
3) It stores these sockets in an internal routing table within the communicator instance, making the management of entanglement resources transparent to the user.

Furthermore, NetQMPI handles the synchronization of the quantum control flow. While the NetQASM SDK requires manual `flush()` invocations, NetQMPI primitives include implicit flushing mechanisms or offer explicit `flush()` operations to ensure state consistency across the network.

C. POINT TO POINT FUNCTIONS

The fundamental requirement of a distributed quantum system is the ability to transfer quantum states between nodes. NetQMPI encapsulates the standard Quantum Teleportation Protocol into two atomic primitives: `qsend` and `qrecv`.

Crucially, these primitives represent a shift towards **high-level semantic communication**. From the programmer's perspective, the operation is defined simply as sending a specific qubit to a destination rank. All references to low-level hardware resources—such as explicit EPR pair generation, entanglement fidelity, specific QPU identifiers, or classical control signaling—are completely abstracted away. The developer focuses solely on the logical flow of quantum information.

1) THE `QSEND` PRIMITIVE

The `qsend(qubit, dest_rank)` function abstracts the complex role of the sender ("Alice"). When this semantic function is invoked, the library transparently handles the following underlying operations:

1) Looks up the pre-established `EPRSocket` associated with the logical `dest_rank` in the routing table.
2) Requests the creation of an EPR pair from the network layer.
3) Performs the local Bell measurement (CNOT and Hadamard gates) between the user's payload qubit and the entangled qubit.
4) Measures the qubits and automatically transmits the classical measurement outcomes (2 bits) to the destination rank using the classical control plane.

2) THE `QRECV` PRIMITIVE

The `qrecv(source_rank)` function abstracts the role of the receiver ("Bob"). It returns a handle to the received quantum state, allowing the program to continue execution immediately. Internally, the library:

1) Waits for the hardware confirmation of the entanglement generation.
2) Listens for the classical correction bits sent by `source_rank`.
3) Applies the necessary Pauli-X and Pauli-Z corrections to recover the original state fidelity.

This encapsulation drastically simplifies the development process. As shown in Figure 10, the complex coordination required by the raw SDK (discussed in Section III-C) is reduced to intuitive, single-line commands.

```python
def main(app_config=None, rank, size):
    # Initialize the communicator
    comm = QMPICommunicator(rank, size, app_config)

    if rank == 0:
        # Sender Logic (Alice)
        q = Qubit(comm.connection)
        q.H() # Prepare state |+>

        # Semantic send: "Send qubit q to Rank 1"
        # No EPR management required
        comm.qsend(q, 1)
        print("Rank 0: Qubit sent")

    elif rank == 1:
        # Receiver Logic (Bob)
        # Semantic receive: "Get qubit from Rank 0"
        q_received = comm.qrecv(0)

        # The qubit is ready for use
        m = q_received.measure()
        print(f"Rank 1: Qubit received, measure: {m}")
```

FIGURE 10. Example of point to point code using the `qsend` and `qrecv` primitives of NetQMPI.

D. COLLECTIVE OPERATIONS

A significant advantage of the MPI paradigm is the ability to perform collective operations involving multiple nodes simultaneously. While point-to-point communication allows for specific data transfer, collective primitives abstract common data movement patterns, reducing code verbosity and potential errors. NetQMPI implements these operations by leveraging the underlying point-to-point primitives.

1) SCATTER AND GATHER

The `qscatter` and `qgather` primitives allow for the distribution and collection of quantum states among a group of processes.

The `qscatter` operation takes a list of qubits residing in a source process (*root*) and distributes them sequentially to all other processes in the communicator. To strictly satisfy

the constraints imposed by the No-Cloning Theorem, it is crucial to clarify that `qscatter` does not copy or replicate any quantum state. Unlike classical scattering operations that can duplicate data buffers, `qscatter` operates exclusively on an ensemble of physically distinct and independent qubits pre-allocated at the root node. The primitive sequentially transfers the absolute ownership of exactly one unique qubit from the list to each participating target rank. Because each destination process receives a completely separate physical entity and no state duplication is attempted, the operation remains entirely compliant with quantum mechanics. From an implementation standpoint, these operations are abstractions of a loop of point-to-point transfers. For instance, a `qscatter` is functionally equivalent to the root node executing a `qsend` inside a `for` loop targeting each rank in the communicator.

Figure 11 illustrates the movement of qubits during these operations. In Figure 11(a), the initial state is shown where the root holds a register of qubits. Figure 11(b) shows the result of a scatter operation (or the start of a gather), where each qubit has been teleported to its corresponding rank.

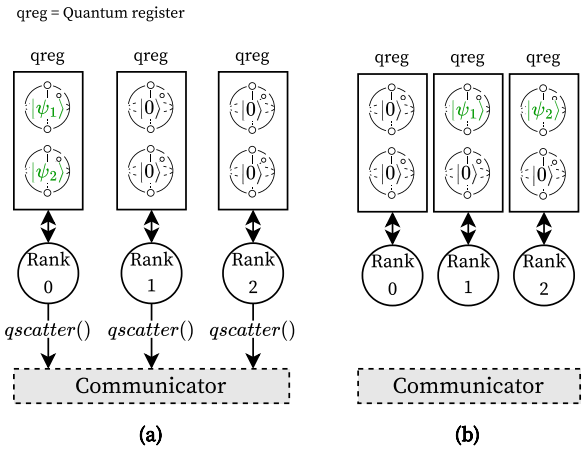


FIGURE 11. Visual representation of the `qscatter` operation. (a) Initial state where all qubits reside in the root process. (b) Distributed state where qubits have been moved to their respective ranks. The operations rely on iterative point-to-point teleportations.

A practical implementation is shown in Figure 12. In this example, the root process initializes a list of qubits in the superposition state $|+\rangle$ (using Hadamard gates) and distributes one qubit to each node in the network using `qscatter`. Finally, each node measures its local qubit independently.

2) EXPOSE AND UNEXPOSE

In classical MPI, the `MPI_Bcast` operation copies data from one process to all others. However, as previously mentioned, in quantum computing, the No-Cloning Theorem prohibits creating independent copies of an arbitrary unknown quantum state. Therefore, a direct quantum broadcast is physically impossible.

```python
def main(app_config=None, rank, size):
    comm = QMPICommunicator(rank, size, app_config)
    conn = comm.connection
    ROOT_RANK = 0
    qubits = []

    # 1. Root initializes the qubits
    if rank == ROOT_RANK:
        qubits = [Qubit(conn) for _ in range(size)]
        for q in qubits:
            q.H() # Apply Hadamard

    # 2. Scatter: Distribute from Root to all
    # Returns the specific qubit assigned to this rank
    local_qs = comm.qscatter(qubits, ROOT_RANK)

    m = local_qs[0].measure())
```

FIGURE 12. Example of the `qscatter` collective primitive in NetQMPI. The root process initializes a list of qubits in the superposition state $|+\rangle^{\otimes n}$, scatters them to all ranks, and each rank measures its local qubit.

To address this, NetQMPI introduces the `expose` and `unexpose` primitives, as proposed in [14]. Instead of copying the state, the `expose` operation creates a shared context where a specific qubit (or set of qubits) becomes conceptually accessible to other processes for collective operations, without violating the no-cloning principle.

Mathematically, the objective of `expose` is to transition the system from an initial state where the root process R_0 holds an arbitrary superposition

$$|\psi\rangle_{initial} = \alpha|0\rangle_{R_0} + \beta|1\rangle_{R_0}, \tag{4}$$

to a global state where this quantum information is distributed across all n participating ranks $(R_0, \ldots, R_n)$.

$$|\psi\rangle_{exposed} = \alpha(|0\rangle_{R_0} \otimes \cdots \otimes |0\rangle_{R_n}) + \beta(|1\rangle_{R_0} \otimes \cdots \otimes |1\rangle_{R_n}) \tag{5}$$

This shared state enables collective controlled operations on the quantum information distributed across all nodes. The implementation relies on entangling the root qubit with a pre-shared GHZ state across all nodes, followed by local measurements and classical corrections to distribute the quantum information, as illustrated in Figure 13.

Crucially, because this operation alters the accessibility scope of the qubits, it is mandatory to release the resource once the collective computation is finished. The `unexpose` primitive serves this purpose, returning the system to a consistent state where the qubits are once again private to their owner or properly measured.

To illustrate the practical utility of the `expose` primitive, consider distributed algorithms that require global phase estimation or multi-controlled operations, such as the distributed Quantum Fourier Transform (QFT) or Shor's algorithm. In these scenarios, a single control qubit must interact with multiple target qubits spread across the network. By generating a shared GHZ context via `expose`, NetQMPI enables the application of distributed controlled-phase gates without forcing the developer to manually orchestrate the underlying entanglement distribution for each target node, drastically reducing the algorithmic implementation complexity.

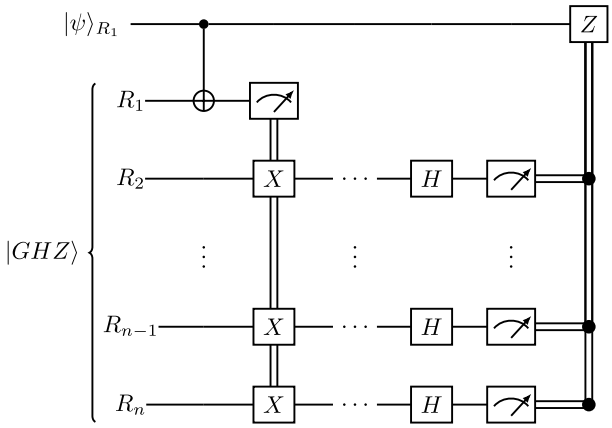


FIGURE 13. Quantum circuit representation of the `expose` and `unexpose` operation. The root qubit is entangled with a pre-shared GHZ state across all nodes, followed by local measurements and classical corrections to distribute the quantum information.

E. END-TO-END EXECUTION FLOW

To clarify in detail how NetQMPI reconciles high-level SPMD semantics with physically faithful quantum network simulation, we present a systematic, stepwise analysis of the execution lifecycle of a distributed quantum application. As a running example, we consider the execution of a simple point-to-point communication operation (`qsend` / `qrecv`) invoked via the command-line interface (`netqmpi -n 2 app.py`). In this architecture, NetQMPI relies on NetSquid via SquidASM to handle the underlying physical simulation. The corresponding execution proceeds through the following hierarchical stages:

1) **Application Launch and Context Injection:** The execution begins in the `cli.py` module, which parses the user's request for an N-node topology. The `external.py` dispatcher takes control, instantiating the multiprocessing environment. It spawns N independent asynchronous processes, transparently injecting the specific `rank` and `size` variables into the namespace of each process before executing the unified user script.
2) **Backend Instantiation and Mesh Synchronization:** Upon script execution, the `QMPICommunicator` is initialized. At the implementation level, NetQMPI communicates with the NetQASM SDK to allocate the corresponding logical QPUs. The communicator iterates through the routing table and establishes a fully connected mesh of `EPRSocket` and classical `Socket` objects. To prevent race conditions during physical initialization, NetQMPI imposes a hidden classical barrier, ensuring all nodes are actively listening before quantum instructions are evaluated.
3) **Primitive Invocation and ISA Compilation:** When the user invokes `comm.qsend(q, dest=1)`, NetQMPI translates this semantic instruction into a deterministic sequence of NetQASM SDK calls. It requests an EPR pair via the destination's socket, applies the local Bell-state measurement (CNOT and Hadamard), and extracts the classical outcomes. Crucially, the NetQASM SDK compiles these operations into NetQASM ISA assembly (e.g., `create_epr`, `wait_all`, `meas`) and pushes the instruction queue to the simulated Quantum Node Operating System (QNodeOS).
4) **Discrete-Event Physical Simulation:** At this layer, the backend agnostic nature of NetQMPI delegates execution to the physics engine (e.g., NetSquid via SquidASM). The simulator processes the `create_epr` instruction by modeling the actual stochastic physical phenomena: it simulates photon emission from the quantum network interface, applies optical fiber attenuation, models the time-of-flight delay across the predefined link distance, and applies the corresponding T_1/T_2 memory decoherence models to the stored qubits while they await the `wait_all` synchronization.
5) **State Recovery and Application Yield:** Once the physical simulator confirms the entanglement generation and measurement, the classical bits are routed through the simulated classical channels. The receiving rank (executing `comm.qrecv(0)`) catches the classical payload, triggers the NetQASM operations for Pauli-X and Pauli-Z corrections on its local memory, and successfully reconstructs the quantum state. NetQMPI finally yields the resulting `Qubit` object back to the Python application layer, allowing the user's algorithm to proceed seamlessly.

This rigid, multi-layered execution pipeline guarantees that while the programmer experiences the simplicity of classical MPI, the underlying execution strictly adheres to the physical, temporal, and noise constraints modeled by the discrete-event quantum simulator.

F. ADVANCED PRIMITIVES: REDUCTIONS AND BARRIERS

While point-to-point and collective data movement primitives cover the immediate requirements of most distributed algorithms, classical MPI also heavily relies on synchronization barriers (`MPI_Barrier`) and reduction operations (`MPI_Reduce`). In the quantum domain, implementing a global synchronization barrier is straightforward via classical control plane messaging, ensuring all nodes reach a specific execution point before proceeding with their quantum instruction queues. However, quantum reductions require physical operations to combine states. A distributed quantum reduction could be mapped to collective parity checks or multi-qubit gates, such as a distributed Toffoli (CCNOT) gate, where the state of multiple remote control qubits is "reduced" into a single target qubit [28]. Developing efficient, hardware-native protocols to support these quantum reductions as high-level NetQMPI primitives remains a promising avenue for future framework extensions.

V. COMPARATIVE AND USE CASES

To evaluate the effectiveness of the proposed abstractions, we perform a comparative analysis against the current state-of-the-art tools for distributed quantum programming. We selected the generation of a distributed Greenberger–Horne–Zeilinger (GHZ) state across N nodes as the benchmark scenario. This task is chosen because it requires multi-partite entanglement and coordinated control flow, serving as a standard "Hello World" for the Quantum Internet.

A. BENCHMARK DEFINITION

The goal is to generate the maximally entangled state $|\text{GHZ}\rangle = \frac{1}{\sqrt{2}}(|00\ldots0\rangle + |11\ldots1\rangle)$ shared among N distinct network nodes, where each rank holds exactly one qubit.

As shown in Figure 14, the protocol involves the following steps:

1) One root node (e.g., Rank 0) prepares one qubit in superposition ($|+\rangle$).
2) The root node performs a sequence of distributed CNOT operations targeting the qubits of the remaining $N-1$ nodes.
3) In a distributed setting without shared memory, every "Remote CNOT" implies a complex teleportation protocol or entanglement swapping routine involving classical communication and corrections.

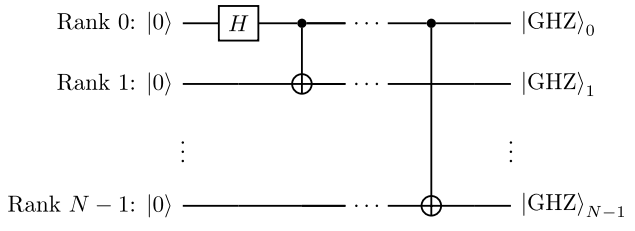


FIGURE 14. Distributed GHZ State Generation across *N* nodes. The root node (Rank 0) prepares a qubit in superposition and performs distributed CNOTs to entangle the qubits at the target nodes (Ranks 1 . . . *N* − 1). Each Remote CNOT involves entanglement distribution and classical communication.

B. QUALITATIVE ANALYSIS: PROGRAMMING COMPLEXITY

We compare the implementation workflow in NetQMPI against three categories of tools: High-level Simulators (QuNetSim), Raw SDKs (NetQASM/SquidASM), and Discrete-Event Engines (NetSquid/SeQUeNCe).

1. NetQMPI (This Work): Following the SPMD paradigm, the solution is written in a *single source file*. The logic scales automatically with the number of processes. The developer uses the `rank` variable to distinguish between the root (Control) and the others (Targets). The distributed CNOT is abstracted and the `expose` primitive is used to create the shared state in a single call. *Complexity: O*(1) in terms of file management. The code remains constant regardless of N.

```python
def main(app_config=None, rank=0, size=3):
    comm = QMPICommunicator(rank, size, app_config)

    # 1. Root creates the base qubit
    qubits = []
    q = Qubit(comm.connection)
    if rank == 0:
        q.H() # Superposition
        qubits.append(q)

    # 2. Collective Operation
    # The root exposes the qubit to all ranks (control of
      remote CNOT)
    comm.expose(qubits, root_rank=0)

    # 3. Perform local operations (if my rank is not root
     )
    if rank != 0:
        qubits[0].cnot(my_qubit)

    comm.unexpose(root_rank=0)
```

Listing 1. NetQMPI: GHZ generation (Single File).

2. NetQASM SDK (SquidASM / SimulaQron): As these tools operate on a role-based paradigm, the developer must explicitly define the behavior for each node. For a 3-node GHZ state, this requires writing three separate application flows (e.g., `app_alice.py`, `app_bob.py`, `app_charlie.py`). Furthermore, a Remote CNOT is not a native primitive. The developer must manually implement the "telegate" protocol, which involves generating EPR pairs, performing local measurements, sending classical bits via a separate socket, and applying corrections. *Complexity: O*(N) in terms of source files and code maintenance. Extending the system from 3 to 4 nodes requires writing a completely new script for the new node and updating the root script to handle the new connection.

3. NetSquid / SeQUeNCe: These are discrete-event simulators, not programming frameworks. Before implementing the GHZ logic, the user must manually construct the network topology in Python: defining quantum memories, fiber lengths, photon sources, and noise models. The algorithm itself is then defined as a protocol class attached to these physical entities. *Complexity:* Extremely High. The code is dominated by hardware definitions rather than algorithmic logic.

C. CODE IMPLEMENTATION EXAMPLES

To substantiate the complexity analysis, we present code snippets required to implement the GHZ state generation across the evaluated platforms.

1) NETQMPI (SPMD PARADIGM)

Listing 1 shows the full implementation in NetQMPI. The logic is defined in a single block. The `expose` primitive automatically handles the underlying distributed CNOTs and entanglement distribution to create the shared state defined in Eq. 5. Changing the size from 3 to 50 nodes only requires changing the command line argument `-n 50`.

```python
# --- app_alice.py (Root) ---
# 1. Explicit Socket Creation
# Alice needs separate sockets for each target
sock_bob = EPRSocket("Bob")
sock_charlie = EPRSocket("Charlie")
csock_bob = Socket("Alice", "Bob")       # Classical
csock_charlie = Socket("Alice", "Charlie") # Classical

with NetQASMConnection("Alice", epr_sockets=[sock_bob,
     sock_charlie]) as c:
    q = Qubit(c)
    q.H()

    # 2. Manual Remote CNOT to Bob
    epr_b = sock_bob.create_epr()[0]
    q.cnot(epr_b)
    m_b = epr_b.measure()
    c.flush()
    csock_bob.send(str(m_b)) # Send correction bits

    # 3. Manual Remote CNOT to Charlie
    # Logic must be duplicated for the new node
    epr_c = sock_charlie.create_epr()[0]
    q.cnot(epr_c)
    m_c = epr_c.measure()
    c.flush()
    csock_charlie.send(str(m_c))

# --- app_bob.py (Target 1) ---
sock_alice = EPRSocket("Alice")
csock_alice = Socket("Bob", "Alice")

with NetQASMConnection("Bob", epr_sockets=[sock_alice])
     as c:
    q = Qubit(c)
    epr = sock_alice.recv_epr()[0]
    epr.cnot(q)
    c.flush()

# --- app_charlie.py (Target 2) ---
# New file required for the third node
sock_alice = EPRSocket("Alice")
csock_alice = Socket("Charlie", "Alice")

with NetQASMConnection("Charlie", epr_sockets=[sock_alice
     ]) as c:
    q = Qubit(c)
    epr = sock_alice.recv_epr()[0]
    epr.cnot(q)
    c.flush()
```

Listing 2. NetQASM SDK: GHZ implementation (Alice, Bob, Charlie).

2) NetQASM SDK (ROLE-BASED)

In the raw SDK, the logic is strictly separated by roles. Listing 2 demonstrates the code required to generate a GHZ state among three nodes (Alice, Bob, and Charlie).

Crucially, note that the developer must explicitly instantiate the connection objects (`EPRSocket` for quantum links and `Socket` for classical communication) *before* the main logic, which reveals the scalability bottleneck: adding a fourth node (Dave) forced us to:

1) Create a completely new file `app_charlie.py`.
2) Modify `app_alice.py` to manually instantiate the new sockets for Charlie (`sock_charlie`, `csock_charlie`) and duplicate the teleportation logic.

3) NetSquid (DISCRETE-EVENT / HARDWARE)

NetSquid operates at a lower level of abstraction. Before implementing the GHZ protocol, the user must define the physical hardware. Listing 3 illustrates the verbosity of setting up just the nodes and channels. The actual algorithm

```python
# 1. Define Hardware Components
qproc_alice = Node("Alice", num_positions=2,
    mem_noise_models=...)
qproc_bob = Node("Bob", num_positions=2, ...)

# 2. Define Physical Connections
channel_a_b = QuantumChannel("A->B", length=20,
    models={"delay_model": FibreDelayModel()})

# 3. Network Configuration
network = Network("GHZ_Net")
network.add_node(qproc_alice)
network.add_connection(qproc_alice, qproc_bob,
    channel_to=channel_a_b)

# 4. Protocol Implementation (Not shown: ~100 lines)
# Requires defining state machines for photon emission
```

Listing 3. NetSquid: Hardware setup.

requires defining a class inheriting from `NodeProtocol` for each entity.

Due to the excessive length and technical complexity of the full implementation, the remaining algorithmic code is omitted. It is crucial to emphasize that this comparison does not aim to undermine the utility of discrete-event simulators like NetSquid, but rather to highlight the conceptual gap between physical layer modeling and high-level application development. These tools serve fundamentally different purposes: while discrete-event engines are indispensable for validating hardware fidelity and noise models, they are not designed for algorithmic prototyping. Thus, the level of detail presented here demonstrates that these are not entities directly comparable to NetQMPI, but instead underlying layers that our proposed approach aims to abstract away.

4) QuNetSim (HIGH-LEVEL SIMULATOR)

QuNetSim provides a centralized view similar to a network orchestrator. While the algorithmic part is concise, it requires significant configuration code to set up the simulation environment. The developer must manually instantiate hosts, configure the network topology, establish connections, and manage the start/stop lifecycle of the simulation engine (see Listing 4). Furthermore, this code runs purely as a simulation and cannot be compiled to real hardware instructions.

D. QUANTITATIVE ANALYSIS: LINES OF CODE (LOC)

To evaluate the development effort and scalability of the proposed solution, we analyzed the implementation of the GHZ generation protocol across the selected platforms. We utilize lines of code (LOC) as a metric for software complexity, counting only logical lines (excluding comments and imports).

Table 1 summarizes the asymptotic complexity required for both setting up the network topology and implementing the computational logic.

Trend Analysis and Scalability: The projection of code growth as the network size N increases is illustrated in Figure 15. The analysis reveals three distinct trends:

- **Linear/Quadratic Growth (State of the Art):** In frameworks like the NetQASM SDK or NetSquid, the

```python
# 1. Manual Network Setup
network = Network.get_instance()
alice = Host('Alice')
bob = Host('Bob')
charlie = Host('Charlie')

# 2. Manual Connection Management
alice.add_connection('Bob')
alice.add_connection('Charlie')
bob.add_connection('Charlie') # ... and others ...
alice.start(); bob.start(); charlie.start()
network.start()

# 3. Protocol: Manual Circuit Construction
def ghz_protocol_alice(host):
    # a. Create Root Qubit
    q_root = Qubit(host)
    q_root.H()

    # b. Entangle and Send to Bob
    q_bob = Qubit(host)
    q_root.cnot(q_bob) # Local CNOT

    host.send_qubit('Bob', q_bob)

    # c. Entangle and Send to Charlie
    q_charlie = Qubit(host)
    q_root.cnot(q_charlie)
    host.send_qubit('Charlie', q_charlie)

    # ... Wait for acknowledgments ...

def ghz_protocol_target(host):
    # Targets must explicitly receive the qubit
    q = host.get_data_qubit('Alice')
    print(f"{host.host_id} received part of GHZ")

# 4. Execution
t1 = alice.run_protocol(ghz_protocol_alice)
t2 = bob.run_protocol(ghz_protocol_target)
t3 = charlie.run_protocol(ghz_protocol_target)
t1.join(); t2.join(); t3.join()
network.stop(stop_hosts=True)
```

Listing 4. QuNetSim: Network setup & Orchestration.

TABLE 1. Comparison of Lines of Code (LOC) complexities required for setting up the network topology and implementing the GHZ generation as the number of nodes N increases.

Framework	Files	LOC (Network)	LOC (Comp.)	Paradigm
NetQMPI	**1**	$\mathcal{O}(1)$	$\mathcal{O}(1)$	**SPMD (Abstract)**
QuNetSim	1	$\mathcal{O}(N^2)$	$\mathcal{O}(1)$	Centralized Sim
NetQASM SDK	N	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$	Role-based
NetSquid	N	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$	Hardware Descr.

complexity grows significantly with N. Each new node requires the creation of separate source files, manual instantiation of connection sockets ($\mathcal{O}(N^2)$ for fully connected topologies), and duplication of control logic. This verbose approach forces the developer to write boilerplate code that is essentially identical but structurally necessary, dramatically increasing the probability of introducing copy-paste errors or synchronization bugs.

- **Constant Trend (NetQMPI):** In contrast, NetQMPI maintains a flat complexity curve ($\mathcal{O}(1)$). Thanks to the SPMD paradigm and collective primitives, the code written for $N = 3$ is identical to the code for $N = 100$. The logic is encapsulated in loops (e.g., `expose`) that iterate over the communicator size dynamically.

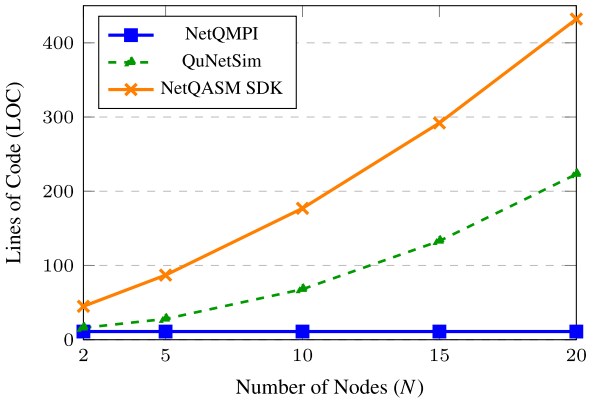


FIGURE 15. Comparison of Lines of Code (LOC) required to implement the GHZ state generation protocol as the network size (N) increases. NetQMPI maintains constant complexity due to its SPMD collective operations.

Implications for Quantum Software Engineering: This reduction in code complexity is critical for the development of distributed quantum applications. By decoupling the program logic from the network size, NetQMPI allows for rapid prototyping and ensures a bug-free implementation of complex protocols. In a domain where debugging distributed quantum states is notoriously tricky, minimizing the codebase surface reduces the potential for implementation errors, allowing researchers to focus on algorithmic correctness rather than infrastructure management.

E. PERFORMANCE ESTIMATION MODEL

To formally evaluate the runtime overhead of NetQMPI's primitives, we adopt the SENDQ performance model introduced by Häner et al. [17] for distributed quantum computing. Following their methodology, we assume that classical communication latency is negligible compared to the execution time of logical quantum operations and entanglement generation.

The model is defined by the following primary parameters:

- S: The number of qubits used to store EPR pairs (per node).
- E: The upper bound on the time it takes for a node to establish a logical EPR pair with any other node.
- N: The number of quantum nodes in the distributed network.
- D: The delay incurred due to local quantum computation (e.g., local gates and measurements).
- Q: The number of logical qubits available for computation (per node).

Using these parameters, we can estimate the theoretical execution time (T) for the core NetQMPI operations:

1) INITIALIZATION OVERHEAD (EPR SOCKETS MESH)

While NetQMPI successfully reduces the application-layer code complexity for network setup to a constant $\mathcal{O}(1)$ for the programmer, the physical underlying infrastructure must still allocate the communication channels. For a fully connected logical topology of N nodes, the `QMPICommunicator`

automatically instantiates a total of $N(N-1)/2$ EPR sockets across the entire network.

However, evaluating this initialization overhead as a quadratic cost ($\mathcal{O}(N^2)$) is misleading in a distributed architecture. Each individual quantum node operates independently and is only responsible for establishing connections with its $N-1$ peers. Consequently, the local initialization overhead per node scales linearly, $\mathcal{O}(N)$. Because all nodes in the communicator execute the initialization routines concurrently, the total initialization time of the system, T_{init}, is bounded by this local linear complexity rather than the global sum.

Under the SENDQ model, assuming a node processes its $N-1$ socket allocation requests sequentially due to local hardware limits (e.g., a single quantum network interface), the initialization time is given by:

$$T_{\text{init}} = \mathcal{O}((N-1)E) \tag{6}$$

This demonstrates that NetQMPI efficiently leverages the distributed nature of the hardware, bounding the setup latency to a linear scale despite managing a fully connected mesh.

2) POINT-TO-POINT (`QSEND` / `QRECV`)

Transmitting a single qubit via standard teleportation requires establishing exactly one EPR pair and performing local Bell measurements. Thus, the execution time is bound by:

$$T_{\text{p2p}} = E + \mathcal{O}(D) \tag{7}$$

This operation temporarily consumes one qubit from the available S buffer pool at both the source and destination nodes during the entanglement generation phase.

3) SCATTER (`QSCATTER`)

The `qscatter` collective involves a root node sequentially distributing $N-1$ distinct qubits to the remaining nodes. Assuming the root node is constrained by a minimal EPR storage ($S=1$), it must initiate EPR pair generation strictly sequentially. The total time scales linearly with the number of target nodes:

$$T_{\text{scatter}} = (N-1)E + \mathcal{O}(N \cdot D) \tag{8}$$

4) EXPOSE / UNEXPOSE (GHZ CONTEXT GENERATION)

To create the shared state required by the `expose` primitive, the root node prepares a base qubit from its computational pool (Q) and performs a sequence of distributed CNOT operations targeting the $N-1$ nodes. Since each remote CNOT requires the consumption of an EPR pair, a sequential initialization under $S=1$ yields:

$$T_{\text{expose}} = (N-1)E + \mathcal{O}(N \cdot D) \tag{9}$$

Optimization Note: While the application layer complexity for the developer remains strictly $\mathcal{O}(1)$ in NetQMPI, the physical execution time depends heavily on the hardware limits S and Q. If a node possesses sufficient EPR storage ($S \geq N-1$) and the network layer permits concurrent entanglement generation, the physical overhead could theoretically be optimized. However, governed by the standard SENDQ bounds, the $(N-1)E$ scaling accurately reflects the theoretical latency for establishing entangled contexts in near-term distributed topologies.

F. QUANTITATIVE EVALUATION ON SquidASM PHYSICAL BACKEND

To complement the theoretical SENDQ analysis with concrete simulation data, we executed the distributed GHZ benchmark directly on the SquidASM discrete-event backend [22]. For this experiment, we exclusively utilized the point-to-point primitives (`qsend` and `qrecv`), mimicking the underlying physical network load of collective operations without invoking them directly.

We note that this evaluation exercises the point-to-point primitives directly; the collective operations (`expose`, `unexpose`, ...) were not executed on the physical backend in this study, as their communication pattern for $N=3$ is equivalent to two sequential `qsend` or `qrecv` calls. A dedicated execution of the collective primitives, particularly to characterize scheduling overhead when the underlying EPR sockets are shared across multiple concurrent CNOTs, is left for future empirical work.

The protocol follows the logical flow where Rank 0 allocates N qubits locally, builds a local N-qubit GHZ state via a Hadamard gate and $N-1$ CNOT gates, and subsequently loops over $i = 1, \ldots, N-1$ executing `comm.qsend(q_i, i)`. Concurrently, each target rank i executes `comm.qrecv(0)` to recover its share of the distributed entangled state.

1) HARDWARE CONFIGURATION

The simulation employs a generic quantum device parameterized to reflect near-term nitrogen-vacancy (NV) center specifications, consistent with the default values reported in the SquidASM `NVQDeviceConfig` [22]. The physical parameters include: longitudinal relaxation time $T_1 = 1\,\text{s}$, transverse coherence time $T_2 = 300\,\text{ms}$, electron-spin initialization $t_{\text{init}} = 2\,\mu\text{s}$, single-qubit microwave gate time $t_{1q} = 5\,\text{ns}$, two-qubit electron-carbon (EC) gate time $t_{2q} = 500\,\mu\text{s}$, and optical read-out time $t_{\text{meas}} = 10\,\mu\text{s}$. Gate-level depolarizing probabilities are set to $p_{1q} = 0.001$ and $p_{2q} = 0.008$. Quantum links are modeled as *DepolariseLinkConfig* channels with a cycle time of $t_{\text{cycle}} = 100\,\mu\text{s}$, corresponding to a 10 km optical-fiber round-trip propagation. Two independent experiments were conducted, with results presented in Figures 16 and 17.

2) EXPERIMENT A: GHZ FIDELITY VS. LINK NOISE

The network size is fixed at $N=3$ and local gate noise is disabled to strictly isolate the effect of EPR-pair quality on the distributed state fidelity. The EPR-pair fidelity F_{link} is swept over five values $\{0.55, 0.65, 0.75, 0.85, 0.95\}$ with deterministic generation per cycle ($p_{\text{success}} = 1$). For each operating point, 200 independent shots are executed. The

performance measure is the *GHZ fidelity proxy*[2]:

$$\hat{F}_{\mathrm{GHZ}} = P(\text{ all nodes agree in } Z\text{-basis }) \tag{10}$$

which, for an ideal $|\mathrm{GHZ}_N\rangle$ state, equals 1. Under the symmetric depolarizing noise model, this proxy relates to the true state fidelity by:

$$\hat{F}_{\mathrm{GHZ}} = F_{\mathrm{GHZ}}\left(1 - \frac{1}{2^{N-1}}\right) + \frac{1}{2^{N-1}} \tag{11}$$

which, for $N = 3$, reduces to $\hat{F}_{\mathrm{GHZ}} = \frac{3}{4}F_{\mathrm{GHZ}} + \frac{1}{4}$, yielding the inverse estimator $F_{\mathrm{GHZ}} = (4\hat{F}_{\mathrm{GHZ}} - 1)/3$.

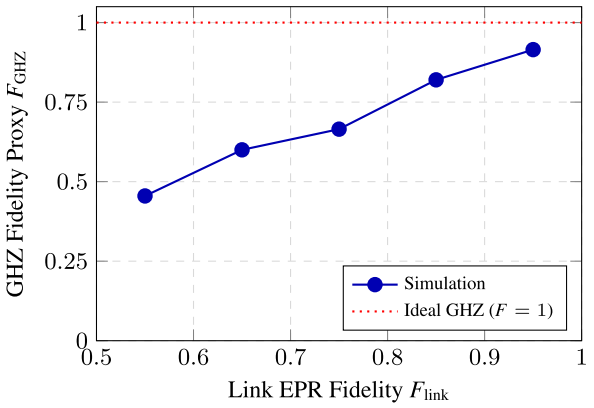


FIGURE 16. Impact of link-layer noise on distributed GHZ state generation. Simulated over the SquidASM backend ($N = 3$, 200 shots). The GHZ fidelity proxy $\hat{F}_{\mathrm{GHZ}}$ is shown as a function of EPR-pair fidelity. The fidelity degrades from 0.915 to 0.455, consistent with the $F_{\mathrm{GHZ}} \approx F_{\mathrm{link}}^2$ scaling expected for sequential teleportation channels. The dotted line marks the ideal target ($F = 1$).

Figure 16 demonstrates that $\hat{F}_{\mathrm{GHZ}}$ degrades monotonically from 0.915 at $F_{\mathrm{link}} = 0.95$ to 0.455 at $F_{\mathrm{link}} = 0.55$. Applying Eq. (11), the inferred GHZ state fidelities are $F_{\mathrm{GHZ}} = 0.887$ and $F_{\mathrm{GHZ}} = 0.273$ at the respective extremes. The sequential teleportation of two qubits introduces two independent depolarizing channels; under this model, the GHZ fidelity scales as $F_{\mathrm{GHZ}} \approx F_{\mathrm{link}}^2$, predicting values of 0.902 and 0.303, falling within the statistical uncertainty of the simulation. These results confirm that link-layer EPR fidelity is the primary bottleneck in state distribution, and that NetQMPI transparently inherits the rigorous physical noise models of the underlying simulated hardware.

3) EXPERIMENT B: SIMULATED EXECUTION TIME VS. N

In this experiment, links are configured with $F_{\mathrm{link}} = 0.85$ and a success probability per cycle of $p_{\mathrm{success}} = 0.10$, representative of near-term NV-center photonic interfaces. The number of nodes is varied from $N = 2$ to $N = 5$, performing 30 independent simulation runs per configuration. The discrete-event simulation clock (t_{sim}) captures the full stochastic history of the quantum network.

[2]Crucially, this proxy is derived solely from Z-basis measurement statistics and is therefore insensitive to phase coherence between the $|0\ldots0\rangle$ and $|1\ldots1\rangle$ branches; it constitutes a lower-bound-style diagnostic rather than a full state fidelity.

The measured mean simulated clock times are 1.94 ± 0.79 ms, 3.60 ± 0.79 ms, 6.15 ± 1.99 ms, and 8.68 ± 1.86 ms for $N = 2, 3, 4, 5$, respectively. A linear regression yields a slope of $\approx$2.28 ms per additional node (Figure 17).

This empirical result directly validates the theoretical SENDQ bounds established in our performance model. Each additional node appends exactly one teleportation step to the sequential protocol ($S = 1$), contributing two independent latency components:

1) **EPR generation latency** (E): With $p_{\mathrm{success}} = 0.10$ and $t_{\mathrm{cycle}} = 100\,\mu$s, the expected number of generation cycles follows a geometric distribution with a mean of 10, resulting in $E = 10 \times 100\,\mu\mathrm{s} = 1$ ms.
2) **Local gate overhead** ($\mathcal{O}(D)$): Each step executes one EC gate for the local GHZ extension and one EC gate within the Bell measurement protocol, yielding a deterministic latency of $2 \times t_{2q} = 2 \times 500\,\mu\mathrm{s} = 1$ ms.

The first-order SENDQ prediction per node is thus $E + \mathcal{O}(D) \approx 2$ ms, which closely aligns with the empirical slope of 2.28 ms. The slight overestimate is attributable to measurement overhead (t_{meas}) and classical correction signaling latency, second-order terms properly subsumed within the $\mathcal{O}(D)$ notation.

The non-negligible standard deviations arise from the geometric distribution of EPR generation attempts. The $(N-1)E + \mathcal{O}(N \cdot D)$ complexity accurately predicts not just the theoretical scaling, but the realistic stochastic latency experienced in actual physical topologies managed by NetQMPI.

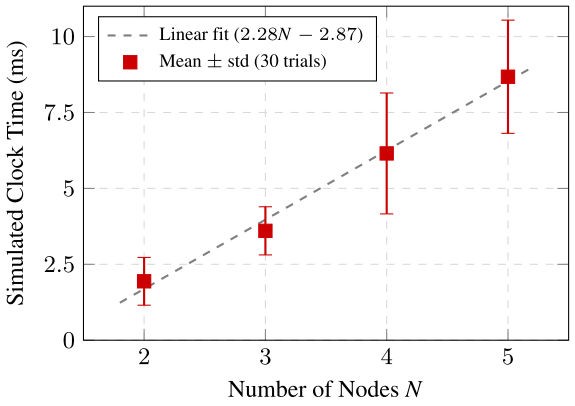


FIGURE 17. Scalability of NetQMPI point-to-point primitives. Evaluated on the SquidASM discrete-event simulator tracking execution time for GHZ state generation (30 trials, NV-center hardware). The plot displays the mean $\pm 1\sigma$ simulated clock time. The linear fit (slope $\approx$2.28 ms/node) confirms the $(N-1)E + \mathcal{O}(N \cdot D)$ scaling of the SENDQ performance bound, with $E = 1$ ms (EPR generation) and $\mathcal{O}(D) \approx 1$ ms (two-qubit gate overhead per step). Error bars reflect the geometric distribution of EPR generation attempts.

VI. CONCLUSION

This work introduced NetQMPI, a high-level Python framework designed to abstract low-level distributed quantum network management and bring the structural scalability of classical High-Performance Computing (HPC) to the

quantum domain. By shifting from labor-intensive, role-based scripting to a unified Single Program Multiple Data (SPMD) paradigm, NetQMPI successfully decouples distributed application development from specific hardware topologies.

The core findings of this study demonstrate that NetQMPI reduces the network initialization and file management complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(1)$ lines of code for fully connected topologies. Furthermore, the introduction of quantum-specific collective operations, such as `expose` and `unexpose`, effectively resolves the conflict between classical data dissemination and the No-Cloning Theorem by leveraging pre-shared multipartite entanglement. Moving beyond purely theoretical specifications in literature, NetQMPI provides a functional, operational software environment that seamlessly maps high-level distributed abstractions to rigorous physical network stacks.

Through its native integration with the NetQASM ecosystem, the framework acts as a backend-agnostic middleware. This allows application developers to validate complex distributed algorithms over state-of-the-art discrete-event simulators like NetSquid (via SquidASM) without facing prohibitive learning curves, while simultaneously guaranteeing that identical codebases will remain executable on future physical quantum hardware platforms.

While the expose and unexpose primitives were validated architecturally and their cost analytically modeled (Section IV-D), their direct empirical execution on a physical backend remains an item for future work.

In summary, NetQMPI represents a significant step forward in quantum software engineering. It enables developers to write readable, maintainable, and bug-free distributed quantum applications, paving the way for the implementation of complex distributed algorithms.

ACRONYMS

DQC	Distributed Quantum Computing.
EPR	Einstein-Podolsky-Rosen.
HPC	High-Performance Computing.
ISA	Instruction Set Architecture.
LOC	lines of code.
MPI	Message Passing Interface.
NISQ	Noisy Intermediate-Scale Quantum.
QKD	Quantum Key Distribution.
QMPI	Quantum Message Passing Interface.
QPU	Quantum Processing Unit.
SDK	Software Development Kit.
SPMD	Single Program Multiple Data.

REFERENCES

[1] H. Buhrman, R. Cleve, and A. Wigderson, "Quantum vs. classical communication and computation," in *Proc. 13th Annu. ACM Symp. Theory Comput. (STOC)*, 1998, pp. 63–68.

[2] I. L. Markov, "Limits on fundamental limits to computation," *Nature*, vol. 512, no. 7513, pp. 147–154, Aug. 2014. [Online]. Available: https://www.nature.com/articles/nature13570

[3] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Rev.*, vol. 41, no. 2, pp. 303–332, 1999, doi: 10.1137/S0097539795293172.

[4] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. 28th Annu. ACM Symp. Theory Comput. (STOC)*, 1996, pp. 212–219, doi: 10.1145/237814.237866.

[5] J. Preskill, "Quantum computing in the NISQ era and beyond," *Quantum*, vol. 2, p. 79, Aug. 2018. [Online]. Available: https://quantum-journal.org/papers/q-2018-08-06-79/

[6] 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," *Comput. Sci. Rev.*, vol. 57, Aug. 2025, Art. no. 100747. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013725000231

[7] M. Caleffi, M. Amoretti, D. Ferrari, J. Illiano, A. Manzalini, and A. S. Cacciapuoti, "Distributed quantum computing: A survey," *Comput. Netw.*, vol. 254, Dec. 2024, Art. no. 110672. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128624005048?via%3Dihub

[8] R. Van Meter, W. J. Munro, and K. Nemoto, "Architecture of a quantum multicomputer implementing Shor's algorithm," in *Proc. Workshop Quantum Comput., Commun., Cryptogr.*, vol. 5106, 2008, pp. 105–114, doi: 10.1007/978-3-540-89304-2_10.

[9] A. Ovide, S. Rodrigo, M. Bandic, H. Van Someren, S. Feld, S. Abadal, E. Alarcon, and C. G. Almudever, "Mapping quantum algorithms to multi-core quantum computing architectures," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2023, pp. 1–5. [Online]. Available: https://ieeexplore.ieee.org/document/10181589

[10] S. Wehner, D. Elkouss, and R. Hanson, "Quantum internet: A vision for the road ahead," *Science*, vol. 362, no. 6412, Oct. 2018, Art. no. eaam9288. [Online]. Available: https://www.science.org/doi/10.1126/science.aam9288

[11] J. Illiano, M. Caleffi, A. Manzalini, and A. S. Cacciapuoti, "Quantum internet protocol stack: A comprehensive survey," *Comput. Netw.*, vol. 213, Aug. 2022, Art. no. 109092. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128622002250

[12] K. Azuma, S. E. Economou, D. Elkouss, P. Hilaire, L. Jiang, H.-K. Lo, and I. Tzitrin, "Quantum repeaters: From quantum networks to the quantum internet," *Rev. Modern Phys.*, vol. 95, no. 4, Dec. 2023, Art. no. 045006. [Online]. Available: https://journals.aps.org/rmp/abstract/10.1103/RevModPhys.95.045006

[13] S. W. Loke, "From distributed quantum computing to quantum internet computing: An overview," 2022, *arXiv:2208.10127*.

[14] F. J. Cardama, J. Vázquez-Pérez, C. Piñeiro, T. F. Pena, J. C. Pichel, and A. Gómez, "NetQIR: An extension of QIR for distributed quantum computing," *Future Gener. Comput. Syst.*, vol. 174, Jan. 2026, Art. no. 107989. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X25002845

[15] G. Díaz-Camacho, I. F. Llovo, F. J. Cardama, I. Bautista, D. Faílde, M. M. Juane, J. Vázquez-Pérez, N. Costas, T. F. Pena, and A. Gómez, "Benchmarking distributed quantum computing emulators," 2025, *arXiv:2512.01807*.

[16] A. S. Cacciapuoti, M. Caleffi, F. Tafuri, F. S. Cataliotti, S. Gherardini, and G. Bianchi, "Quantum internet: Networking challenges in distributed quantum computing," *IEEE Netw.*, vol. 34, no. 1, pp. 137–143, Jan. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8910635

[17] T. Häner, D. S. Steiger, T. Hoefler, and M. Troyer, "Distributed quantum computing with QMPI," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2021, pp. 1–15. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3458817.3476172

[18] S. Diadamo, J. Nötzel, B. Zanger, and M. M. Bese, "QuNetSim: A software framework for quantum networks," *IEEE Trans. Quantum Eng.*, vol. 2, pp. 1–12, 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9465750

[19] R. Parekh, A. Ricciardi, A. Darwish, and S. DiAdamo, "Quantum algorithms and simulation for parallel and distributed quantum computing," in *Proc. IEEE/ACM 2nd Int. Workshop Quantum Comput. Softw. (QCS)*, Nov. 2021, pp. 9–19. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9651415

[20] C. Piveteau and D. Sutter, "Circuit knitting with classical communication," *IEEE Trans. Inf. Theory*, vol. 70, no. 4, pp. 2734–2745, Apr. 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10236453

[21] A. Dahlberg and S. Wehner, "SimulaQron—A simulator for developing quantum internet software," *Quantum Sci. Technol.*, vol. 4, no. 1, Jan. 2019, Art. no. 015001. [Online]. Available: https://iopscience.iop.org/article/10.1088/2058-9565/aad56e

[22] A. Dahlberg, B. van der Vecht, C. Delle Donne, M. Skrzypczyk, I. te Raa, W. Kozlowski, and S. Wehner, "NetQASM—A low-level instruction set architecture for hybrid quantum–classical programs in a quantum internet," *Quantum Sci. Technol.*, vol. 7, no. 3, Jul. 2022, Art. no. 035023. [Online]. Available: https://iopscience.iop.org/article/10.1088/2058-9565/ac753f

[23] T. Coopmans, R. Knegjens, A. Dahlberg, D. Maier, L. Nijsten, J. de Oliveira Filho, M. Papendrecht, J. Rabbie, F. Rozpędek, M. Skrzypczyk, L. Wubben, W. de Jong, D. Podareanu, A. Torres-Knoop, D. Elkouss, and S. Wehner, "NetSquid, a NETwork simulator for QUantum information using discrete events," *Commun. Phys.*, vol. 4, no. 1, p. 164, Jul. 2021. [Online]. Available: https://www.nature.com/articles/s42005-021-00647-8

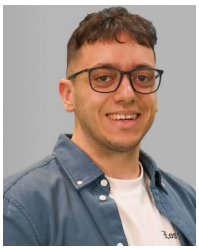
[24] X. Wu, A. Kolar, J. Chung, D. Jin, T. Zhong, R. Kettimuthu, and M. Suchara, "SeQUeNCe: A customizable discrete-event simulator of quantum networks," *Quantum Sci. Technol.*, vol. 6, no. 4, Oct. 2021, Art. no. 045027. [Online]. Available: https://iopscience.iop.org/article/10.1088/2058-9565/ac22f6

[25] W. K. Wootters and W. H. Zurek, "A single quantum cannot be cloned," *Nature*, vol. 299, no. 5886, pp. 802–803, Oct. 1982.

[26] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters, "Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels," *Phys. Rev. Lett.*, vol. 70, no. 13, pp. 1895–1899, Mar. 1993. [Online]. Available: https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.70.1895

[27] D. Gottesman and I. L. Chuang, "Demonstrating the viability of universal quantum computation using teleportation and single-qubit operations," *Nature*, vol. 402, no. 6760, pp. 390–393, Nov. 1999.

[28] I. F. Llovo, G. Díaz-Camacho, N. C. Lago, and A. G. Tato, "Network-assisted collective operations for efficient distributed quantum computing," *IEEE Trans. Quantum Eng.*, vol. 6, pp. 1–14, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/11197054



F. JAVIER CARDAMA (Member, IEEE) received the B.S. degree in computer engineering from the University of Santiago de Compostela (USC), Spain, in 2021, and the M.S. degree in high-performance computing and the Ph.D. degree in information technology research from the Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), USC, in 2022 and 2026, respectively. He is currently a Postdoctoral Researcher with CiTIUS, USC. His research interests include distributed quantum computing and high-performance computing, specifically on the development of software frameworks and architectures for the integration of quantum technologies in HPC centers.

JORGE VÁZQUEZ-PÉREZ is currently a Predoctoral Researcher with the University of Santiago de Compostela (USC), Spain, and a Research Technician at Galicia Supercomputing Center (CESGA). His research interests include the intersection of high-performance computing (HPC) and quantum computing, with a particular interest in distributed quantum architectures, intermediate representations (such as QIR), and the development of compilation tools and software frameworks for heterogeneous quantum-classical systems.

TOMÁS F. PENA (Senior Member, IEEE) received the Ph.D. degree in physics from the University of Santiago de Compostela (USC), Santiago, Spain, in 1994. He is currently a Full Professor with the Department of Electronics and Computer Science, USC, and a Senior Member of the Research Center in Intelligent Technologies (CiTIUS), Santiago de Compostela, Spain. His research interests include distributed quantum computing, high-performance computing, parallel systems architecture, optimization of performance in irregular codes and with sparse matrices, and big data technologies.

ANDRÉS GÓMEZ received the Ph.D. degree in physics from the University of Santiago de Compostela (USC), Spain. He is currently the Head of the Applications and Projects Department, Galicia Supercomputing Center (CESGA). He leads the quantum computing activities at CESGA, overseeing the strategic deployment of the QMIO quantum infrastructure and coordinating various research and development initiatives, including the Quantum Spain Project. His research interests include high-performance computing, cloud technologies, and the practical implementation of distributed quantum computing algorithms and applications in industrial and scientific environments.