Future Generation Computer Systems 174 (2026) 107989

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

NetQIR: An extension of QIR for distributed quantum computing

F. Javier Cardama [a,*], Jorge Vázquez-Pérez [a,c], César Piñeiro [a,b], Tomás F. Pena [a,b], Juan C. Pichel [a,b], Andrés Gómez [c]

[a] Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela, Santiago de Compostela, 15782, Spain
[b] Departamento de Electrónica e Computación, Universidade de Santiago de Compostela, Santiago de Compostela, 15782, Spain
[c] Galicia Supercomputing Center (CESGA), Avda. de Vigo S/N, Santiago de Compostela, 15705, Spain

ARTICLE INFO

Keywords:
Distributed quantum computing
Quantum intermediate representation
Quantum internet
Compilers
Teledata
Telegate
Distributed quantum applications

ABSTRACT

The rapid advancement of quantum computing has highlighted the need for scalable and efficient software infrastructures to fully exploit its potential. Current quantum processors face significant scalability constraints due to the limited number of qubits per chip. In response, distributed quantum computing (DQC) — achieved by networking multiple quantum processor units (QPUs)— is emerging as a promising solution. To support this paradigm, robust intermediate representations (IRs) are needed to translate high-level quantum algorithms into executable instructions suitable for distributed systems. This paper presents NetQIR, an extension of Microsoft's Quantum Intermediate Representation (QIR), specifically designed to facilitate DQC by incorporating new instruction specifications. NetQIR was developed in response to the lack of abstraction at the network and hardware layers identified in the existing literature as a significant obstacle to effectively implementing distributed quantum algorithms. Based on this analysis, NetQIR introduces new essential abstraction features to support compilers in DQC contexts. It defines network communication instructions independent of specific hardware, abstracting the complexities of inter-QPU communication. Although the proposed work allows abstraction of the underlying network, it is important to note that it is intended for the development of high-performance code on future modular quantum architectures. Leveraging the QIR framework, NetQIR aims to bridge the gap between high-level quantum algorithm design and low-level hardware execution, thus promoting modular and scalable approaches to quantum software infrastructures for distributed applications. Furthermore, its design may serve as a foundational component for future implementations of distributed quantum standards such as the Quantum Message Passing Interface (QMPI).

1. Introduction

The evolution of computing has progressed from simple mechanical calculators to modern-day classical computers, that have significantly transformed numerous fields, including science, engineering, and everyday life. Despite these advances, classical computers face limitations in solving certain complex problems efficiently, such as factoring large numbers, simulating quantum systems, or optimizing large-scale systems [1,2]. This has led to the emergence of quantum computing, which leverages the principles of quantum mechanics to process information in fundamentally new ways, offering the potential to solve these intractable problems more efficiently than classical computers can achieve [3,4].

Over the last few years, the development of a comprehensive software stack for quantum computing has gained importance in allowing the programming of quantum devices in a scalable and easy way. This software stack includes quantum high-level languages, compilers, and runtime environments designed to enable the programming and execution of quantum algorithms on quantum devices [5,6]. High-level quantum programming languages such as Q# [7], Quipper [8], or Qiskit [9] facilitate the development of quantum algorithms by abstracting the complexities of quantum hardware [10].

For the efficient execution of these algorithms, quantum code compilers play a crucial role. A compiler is a software program that translates high-level languages into low-level instructions that quantum processors can execute [11]. In classical computing, the concept of IR was introduced as an abstract-machine code to facilitate the development of new compilers [12]. This concept was extended in the world of quantum computing to allow a common IR as an intermediate step between high-level and back-end languages. The main objective of using an IR is to facilitate the optimization of quantum codes and, simultaneously, to ensure their compatibility with different hardware backends [13,14].

* Corresponding author.
E-mail address: javier.cardama@usc.es (F.J. Cardama).

https://doi.org/10.1016/j.future.2025.107989
Received 18 April 2025; Received in revised form 18 June 2025; Accepted 20 June 2025
Available online 3 July 2025

One of the critical challenges in quantum computing remains the scalability and noise of the qubits. Current quantum hardware is limited by the number of qubits that can be reliably maintained and manipulated on a single chip, thus complicating the development of more complex algorithms [15]. These limitations have led to the development of new computing approaches, one of which is the design of modular architectures based on DQC. In these architectures, multiple quantum processing units (QPUs) are networked together to work on a problem collaboratively [16–18]. DQC uses both quantum and classical communications to distribute and synchronize computations across QPUs, thereby potentially overcoming the scalability constraints of individual quantum chips [19–21].

DQC introduces a level of complexity over monolithic quantum computing systems (systems consisting of a single QPU) due to the management of classical and quantum networks and its complexities [22]. Because of this, an abstraction model of the complete process of a DQC algorithm, from its high-level specification to the specific back-end where it is supposed to run, must be defined to facilitate the development of tools according to their specific use. Additionally, this need highlights the importance of defining an IR that not only enables the efficient programming of quantum algorithms within this abstraction framework but also addresses the specific challenges inherent to DQC, such as optimized communication protocols and precise synchronization mechanisms across QPUs [23].

In the context of DQC, it is essential to distinguish it from the concept of the Quantum Internet [24,25]. While DQC uses a quantum interconnection network, its focus is on a form of future distributed quantum computing, thus allowing the user to abstract themselves from both classical and quantum networks [26]. Therefore, it is crucial to abstract away the complexities of quantum networking and, instead, define the appropriate high-level instructions within an IR to ensure its efficient utilization. Currently, a wide range of tools are available for simulating quantum communication networks (network-level simulators) [27], such as SquidASM [28] and Simulaqron [29], as well as discrete event simulation for quantum communication at the physical level, including NetSquid [30] and SeQuence [31]. There are also simulators specifically designed for distributed quantum computing, such as QuNetSim, which follows a more point-to-point model, with network simulation handled by the in-process EQNS simulator [32].

In response to the problems encountered in the literature, this paper proposes two main contributions to the state of the art:

- The definition of a **layered abstraction model** for the correct implementation of IRs for DQC by collecting information from the literature, both from standards followed in classical computing and from attempts at quantum computing. This objective will allow other developers to implement new IRs following a model whose efficiency will be demonstrated later.
- The proposal of an IR that implements the abstraction layer model proposed in the previous objective. Our proposal, NetQIR, extends Microsoft's QIR [33], augmenting it with advanced communication and distributed computation directives to support interoperability and scalability, thereby facilitating robust quantum algorithm development in distributed quantum environments and hybrid programming. As IR, the objective is to be a common language that unites different and future optimization, compilation, or scheduling tools.

It is important to note that both the layered abstraction model and the proposed IR aims to contribute to the set of tools for future distributed quantum computing, abstracting from any type of underlying network. The proposed work is not a compiler or an optimizer; it is a specification of an IR designed to facilitate the integration and use of other tools. Therefore, automatic circuit or task partitioning is not within the scope of NetQIR or the proposed work, in the same way that MPI does not automate data or task parallelism.

Fig. 1 illustrates the advantages of using NetQIR as an IR for DQC, highlighting its extension of QIR and, consequently, LLVM. This figure illustrates DQC tools compiled to various quantum network simulators. In the initial approach, without the use of IRs, the number of required compilers is $n \times m$. However, by leveraging an IR such as NetQIR, this complexity is significantly reduced to $n + m$, streamlining the compilation process.[1]

The paper is structured as follows: initially, Section 2 reviews the related work, focusing on existing IRs and programming languages for DQC. Then, Section 3 introduces a layered abstraction model, essential for DQC. It presents the development and network layers, detailing the specific components required to achieve a modular and interoperable architecture for DQC. In addition, Section 4 presents the topics related to the proposed IR, including its specification and the tools developed to facilitate the design of new software for future developers. Subsequently, Section 5 presents a discussion related to the DQC languages analyzed in the related work, along with an evaluation of the different characteristics of the network layer to justify its abstraction in the proposed abstraction layer model. Finally, Section 6 concludes the work and specifies future work.

2. Related work and background

2.1. Background on distributed systems

At the core of any modern computing system lies the operating system (OS), which provides a set of fundamental abstractions to manage hardware complexity and support application execution. The most relevant abstractions include [34]:

- **Processes:** isolated execution contexts with their own memory space and scheduling policies.
- **Memory management:** virtual memory abstraction, page swapping, and memory protection.
- **Input/Output (I/O):** abstract representations of devices, filesystems, and network interfaces.

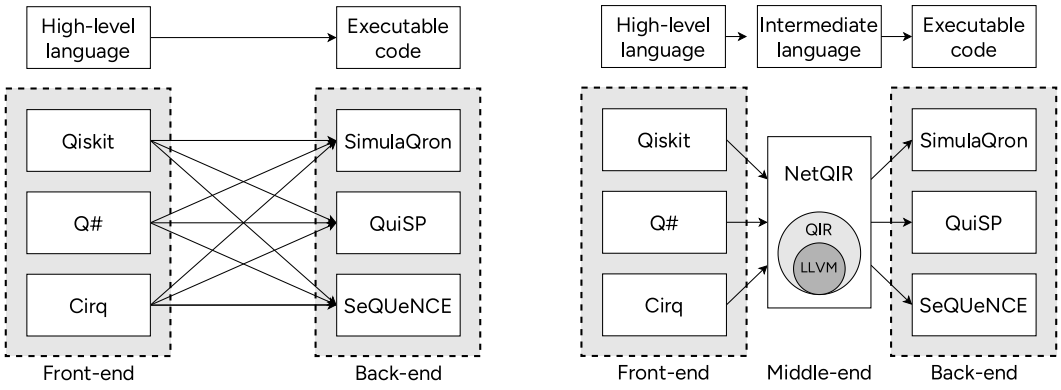
In *distributed systems*, these abstractions are replicated and localized across multiple physical nodes, each typically running its own instance of a full-fledged operating system. This design choice allows each node to manage its own resources — CPU, memory, and I/O devices — independently, leveraging existing OS-level capabilities and simplifying low-level hardware interactions. Fig. 2 shows the levels of a distributed system.

However, distributed applications often require coordination, synchronization, and data exchange across these independent nodes. To support such tasks, a middleware layer is introduced above the OS layer. This middleware provides a programming interface that abstracts away many of the complexities of distributed execution, enabling developers to orchestrate high-level parallel applications without managing low-level networking or hardware-specific details.

One of the most successful and widely adopted middleware standards in classical high-performance computing (HPC) is the *Message Passing Interface* (MPI). In classical HPC, MPI is the *de facto* standard for scalable parallelism across distributed systems. An MPI process is an independent instance of a program with its own local memory, typically mapped to a logical compute unit or hardware core. Each process is assigned a unique *rank* and operates within a *communicator*—a named group of processes that can coordinate using collective or point-to-point communication primitives such as `send` and `recv` or `scatter` and `gather` [36–38].

This abstraction of rank and communicator implies a subsequent mapping by the compiler from logical resource (e.g. rank) to physical

[1] With n as the number of *front-ends* and m of *backends*.



(a) Without IR usage.

(b) Applying NetQIR as IR.

Fig. 1. Comparison between the integration of NetQIR as DQC IR in a compilation scheme between DQC programming frameworks and quantum simulator backends.

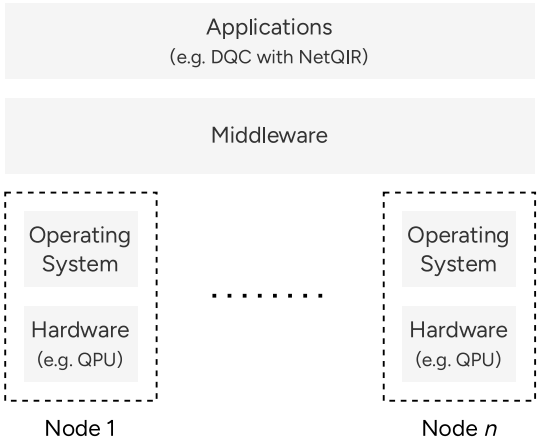


Fig. 2. Levels of a distributed system defined in [35].

resource (e.g. processing unit) typically performed by the compiler or operating system. Work is already underway in the literature on operating systems that allow quantum applications to be run on quantum network nodes such as [39].

Therefore, these abstractions allow the user to exchange messages between the different processes of the distributed system with the aim of collaborating to divide data or tasks. This division is not automatically generated, in any case, by MPI, with the user being responsible for adapting their sequential program to the distributed version.

The goal of the proposed work with NetQIR is the same; NetQIR allows representing circuit partitioning, but it is the user who must modify their sequential circuit to adapt it to a distributed version using the additional features introduced by NetQIR.

NetQIR adapts these ideas to distributed quantum computing, abstracting quantum communication into a clean interface that can coexist with classical workloads and allow for transparent compiler-level optimizations in hybrid quantum–classical applications.

A similar approach is adopted by Quantum Message Passing Interface (QMPI), as proposed by Haner et al. [6]. As its name implies, it is an adaptation of the classical Message Passing Interface (MPI) [36] for quantum communications, achieved by defining analogous point-to-point and collective operations for the quantum pipeline. One of the key differences between NetQIR and QMPI lies in their approach to communication semantics. While NetQIR adopts an MPI-inspired programming style to organize distributed interactions, QMPI replicates classic MPI primitives directly. This direct copying stems from direct problems with the characteristics of quantum computing, in this case, with the no-cloning theorem, as it is not possible to perform copying operations (e.g., broadcast operation). Instead, NetQIR introduces communication abstractions specifically designed to address the unique challenges of distributed quantum computing, as will be detailed in the following sections.

2.2. Related work about intermediate representations in quantum computing

In DQC, the software stack lacks sufficient tools. From high-level languages to lower-level representations — and even development libraries — the literature offers few possibilities, as demonstrated by the state-of-the-art review conducted by Barral et al. [21]. This becomes even more evident when compared to the monolithic case, in which numerous programs, libraries and other software are available for developing quantum applications.

Focusing on the IRs in the monolithic quantum computing case, MLIR [40,41] or SQIR [42] are found, along with QIR [33], backed by the QIR alliance[2] — from now on, it will be referred to as QIR —. The latter is based on the LLVM IR [43] in an attempt to integrate quantum computation into the LLVM infrastructure.[3] In fact, QIR aims to integrate quantum directives with the classical compilation stack, leveraging the advanced LLVM tools to facilitate the generation of highly efficient quantum instructions. In this work QIR will be extended and, therefore, the LLVM IR will be further extended by introducing the necessary directives to perform quantum communications. Throughout this manuscript, this extension will be explained and exemplified.

For DQC, the two most popular specifications in the literature are InQuIR [44] and NetQASM [28]. The first one, InQuIR, is developed from the starting point solely as an IR for DQC. Their primary motivation stemmed from the absence of a dedicated IR for distributed quantum systems. InQuIR stands out for formally defining the grammar of the IR. Using this formalism, InQuIR defines the operational semantics of the IR, which allows it to define and predict how the InQuIR programs will behave under several circumstances. The authors propose some important examples, such as deadlocks and qubit exhaustion, and a roadmap for solving these inconveniences. But InQuIR provides a too low-level approach with explicit generation of the Einstein-Podolsky-Rosen (EPR) pairs and instructions that acknowledge the architecture of the machine, having less control over the form of quantum links.

[2] https://www.qir-alliance.org/.

[3] LLVM is a versatile framework for building compilers and code transformation tools. It lets developers write high-level language code that can be efficiently compiled into machine code for various architectures, with extensive code optimization and analysis support.

As an alternative, as mentioned, NetQASM [28] presents an abstract architecture model composed of an application layer, which is responsible for the classical communications between nodes, and a so-called quantum network processing unit (QNPU), which handles quantum computations and communications. This highlights the scope of NetQASM: the Quantum Internet. It is specifically designed for quantum networks, setting aside *inter-core* communication, which does not require the additional layers that NetQASM introduces. Moreover, NetQASM presents a basic language, called *vanilla*, and a set of variations specially designed for the different quantum architectures, called *flavors*. The authors state that the vanilla version acts as an IR, and the different flavors act as assemblies. The main disadvantage of NetQASM, like IR for DQC, is that its architecture is network-oriented rather than computation-oriented being a proposal closer to the sockets API than to computing communication functions. It also does not consider conditional gates, which are constantly employed in quantum communication protocols, as part of the IR. What is actually done is to perform a measurement, send the result to the application layer and wait until the application layer returns a subroutine with the gate —if the measurement was 1— or without the gate —in the opposite case—.

After discussing the related work about the IRs, it is important to note that NetQIR will not extend QIR arbitrarily. Rather than proposing an ad hoc language, NetQIR builds upon the LLVM-based QIR specification to leverage existing classical optimization and compilation pipelines. This design choice enables hybrid applications to benefit from decades of compiler research, optimization techniques, and mature toolchains originally developed for classical high-performance computing (HPC). By embedding quantum–classical interfaces within an LLVM compatible IR, NetQIR facilitates seamless integration into HPC workflows and heterogeneous computing environments.

In summary, both InQuIR and NetQASM exhibit certain aspects that may represent drawbacks for an IR. Additionally, QMPI defines a high-level standard that is strongly coupled with the classical MPI, introducing various complications and intricacies. NetQIR seeks to address these issues and this manuscript will detail the approach taken to achieve that goal and justify the decisions made in order to do so.

3. Layered abstraction model for distributed quantum computing

Software architectures nowadays strongly rely on abstraction mechanisms as a core principle. These simplify the development of new algorithms, platforms, compilers and tools. DQC architectures follow the same principle. As in the monolithic–and even classical–case, any new IR targeting this paradigm should be hardware-independent and compatible with diverse quantum computing platforms. To achieve this, it is essential to first define a layered abstraction model before designing the DQC IR. To achieve this, it is essential to first define a layered abstraction model before designing the DQC IR.

Fig. 3 depicts the abstraction layers relevant for executing algorithms in a DQC environment. This paper focuses on the computational part of distributed quantum, Therefore, two key layers are defined in our proposed model: the development layer and the network layer. The development layer provides users the necessary tools to design and implement DQC algorithms and software. Meanwhile, the network layer acts as an interface between the development layer and the quantum interconnection network, ensuring seamless interaction while managing its underlying characteristics. Additionally, the network layer interacts with lower layers as needed, further abstracting the physical complexities of the quantum network.

3.1. Network layer

As discussed above, the network layer aims to abstract the particularities of the quantum interconnection network to the development layer. For this purpose, it is necessary to identify this layer's main components, which are the *quantum interconnection network*, the *quantum communication channel* and the *communication protocols*.

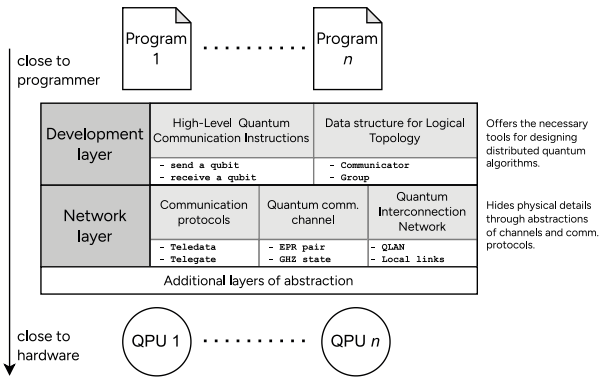


Fig. 3. Abstraction layers relevant to the development of an abstract IR for DQC.

The *quantum interconnection network* abstracts the communication between different quantum computing nodes. This network can be composed of different types of connections, such as quantum network devices, QLANs, or the Quantum Internet. Fig. 4 illustrates a complex example that interconnects quantum computing nodes of different QLANs via the Quantum Internet. This network architecture comprises several QLANs interconnected through the Quantum Internet, allowing quantum computing nodes to communicate with each other. While the quantum interconnection network can abstract a wide spectrum of quantum communication infrastructures — from local optical links to long-range quantum internet protocols — the layered model proposed is specifically designed with a DQC perspective. This implies that, analogously to how the MPI is capable of operating over the Internet but is fundamentally optimized for tightly-coupled HPC clusters, the abstraction model focuses on practical, low-latency interconnections between QPUs within modular or co-located quantum systems.

The main objective of introducing this abstraction layer is not to support long-range quantum communications, but rather to enable efficient and scalable interconnection of modular QPU architectures [45]. In these systems, multiple quantum processors — each with limited qubit capacity — are physically co-located and interconnected to jointly execute a quantum algorithm. Supporting such modular systems requires a flexible software model that can abstract the communication between QPUs without exposing hardware-specific constraints to the developer.

A concrete example of this architectural direction is Xanadu's recent modular quantum computing system, which connects 35 photonic chips using 13 km of optical fiber to construct a distributed quantum processor [46]. Although physically integrated within a local infrastructure, such systems rely on quantum communication channels between QPUs to function as a cohesive unit.

The *quantum communication channel*, as its name indicates, represents the abstraction of the quantum channel responsible of the connection between two quantum nodes. It enables the exchange of quantum information between quantum computing nodes and exploits the principles of quantum mechanics, particularly qubit entanglement. Fig. 4 shows the quantum channel next to a classical channel, which allows operations such as state teleport — an operation that requires both a quantum and a classical channel — to be implemented.

And the last component of the network layer is also its central element: *quantum communication protocols*. They define the fundamental building blocks for exchanging quantum information between quantum computing nodes. Two of the most important communication protocols are *teledata* [47] and *telegate* [48]. These protocols exploit qubit entanglement to enable the exchange of quantum information. Both techniques utilize an entangled EPR pair, where one qubit of the pair resides on a QPU and the other is located on a physically separated QPU. These EPR pairs create a link between the two QPUs, allowing

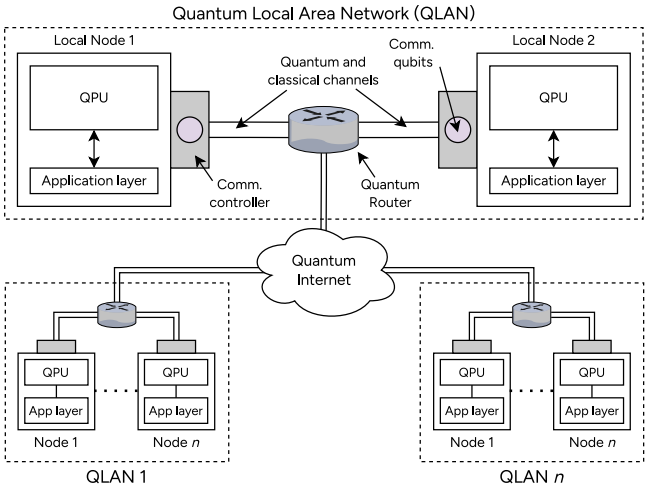


Fig. 4. Complex quantum network architecture interconnecting quantum computing nodes of different QLANs via the Quantum Internet.

quantum data to travel from one QPU to the other by exchanging classical information resulting from measurements of specific qubits. While this work focuses on `teledata` and `telegate`, other communication protocols are also available in the literature [49,50]. Both were selected for this study because they represent two fundamental and widely studied paradigms for distributed quantum communication: quantum state transfer and remote gate application. Their contrasting characteristics make them ideal benchmarks for evaluating abstraction models and compiler decision-making. However, the proposed abstraction model is extensible and can incorporate additional protocols in future work.

Fig. 5 shows the basic structure of the *teledata* (see Fig. 5(a)) and the *telegate* (see Fig. 5(b)) protocols. In both techniques, starting from a state $|a\rangle = \alpha|0\rangle + \beta|1\rangle$ in the local QPU_1, it is necessary that the remote QPU_2 can compute using this information via an EPR pair $|\Phi^+\rangle$. Each protocol is elaborated below:

- **Teledata** protocol transmits the state of the qubit $|a\rangle$ in QPU_1 to an empty qubit in QPU_2. This transmission involves teleportation of the quantum state, causing the original qubit to collapse upon measurement and transferring its state to the destination qubit.
- **Telegate** protocol generates a pair in the state $\alpha|00\rangle + \beta|11\rangle$, where the first qubit is in QPU_1 and the second qubit is in QPU_2. The second qubit is used as a control qubit for a controlled operation. Considering that the control qubit is in the state $|a\rangle = \alpha|0\rangle + \beta|1\rangle$, using the second qubit of the pair achieves the same effect as performing a controlled operation in QPU_2 with the state of the qubit in QPU_1.

The main difference between teledata and telegate is that in teledata, the state is transferred, and computation is performed locally at the receiving QPU, whereas in telegate, the state is not transferred; instead, quantum gates are controlled remotely. Table 1 compares both techniques by evaluating four key characteristics. It is important to understand the difference between performing an operation "Locally" and "Remotely". A **local operation** does not require the use of either quantum or classical communications. On the other hand, a **remote operation** involves the use of quantum or classical communications with other QPUs.

1. *Collapsed qubit*: indicates whether the source qubit collapses once the protocol is executed, requiring a qubit reset.
2. *Entanglement result*: refers to the scope affected by the entanglement generated between the remote and local qubits. This entanglement can be local to the computation node or global to the distributed system.

Table 1
Comparative features between teledata and telegate techniques.

Protocol	Collapsed qubit	Entangl.result	Measures	Numbersyncs
Teledata	Yes	Local	Local - Local	1
Telegate	No	Global	Local - Remote	2

3. *Measurements*: describes how the measurements are performed to implement the protocol.
4. *Number of synchronizations*: the number of synchronizations between the QPUs required to execute the communication protocol.

As observed, in the teledata protocol, the qubit collapses when sending the information, necessitating a reset of the qubit afterwards. This occurs because the quantum state is entirely transferred to the target node; thus, operations are performed locally at the destination, and the resulting entanglement is local to the target QPU. Additionally, measurements are performed simultaneously on two qubits local to the QPU_1, requiring only a single synchronization between the two QPUs.

In contrast, in the telegate protocol, the quantum information is shared as a reference without measuring the original qubit, eliminating the need to reset it. Sharing a reference implies that the generated entanglement is global to the distributed system — this means that qubits from different QPUs have been entangled —. Furthermore, an initial measurement is performed at the QPU_1, and a final measurement is conducted on the remote QPU_2, requiring two separate synchronizations between the QPUs, known as *Cat-Entangler* and *Cat-DisEntangler* (see Fig. 5(b)). The compiler determines the timing of the second synchronization (*Cat-DisEnt*), especially when the qubit is no longer in use.

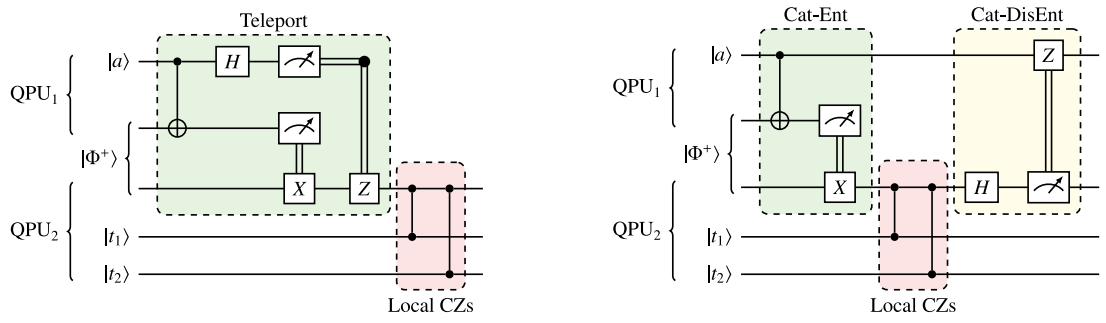
Both protocols have advantages and disadvantages, and there is no clearly superior option. The choice between them depends on the problem to be solved; therefore, the specification of a layered abstraction model will allow the compilation tools to be developed to make an informed decision.

3.2. Development layer

In this subsection, the development layer is introduced, designed to provide users with the necessary instructions to work with DQC algorithms while abstracting away the complexities of the network layer. Specifically, two key components for the development layer, shown in Fig. 3: the *Data Structure for Logical Topology* and the *High-Level Quantum Communication Instructions*. The first component aims to abstract the Quantum Interconnection Network and part of the Quantum Channel by introducing a logical topology data structure that simplifies the development of distributed quantum programs. This logical topology is designed to expose only the number and identity of available quantum processing units (QPUs) to the user, intentionally omitting intermediate network devices such as routers or switches. The second component focuses on abstracting the Quantum Communication Protocols and, to some extent, the Quantum Channel as well, by hiding the implementation details of how communication qubits are generated, through high-level quantum communication instructions.

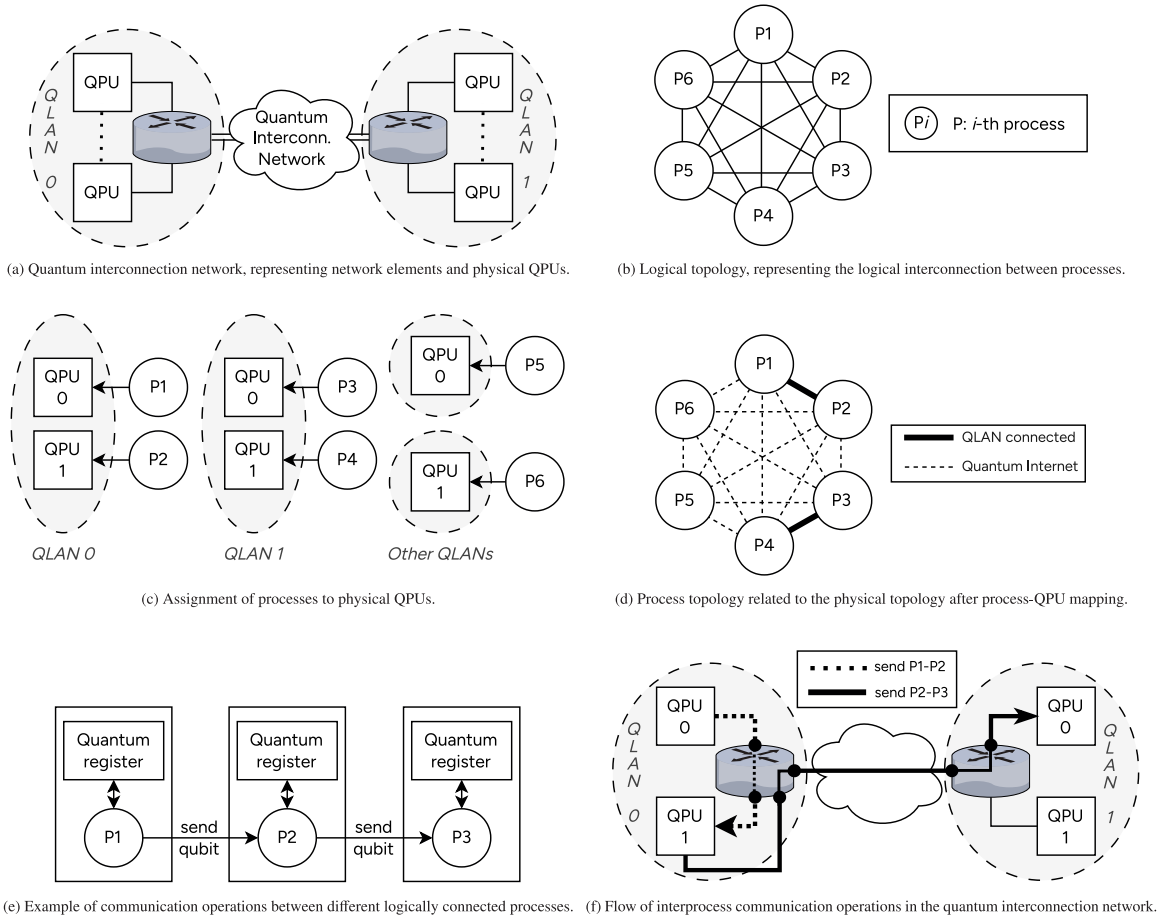
It is important to note that, as in any abstraction model, the compilation tool aims to translate the code from the development layer to physical hardware and the network layer. IRs are developed with the aim of abstracting the underlying hardware and providing a common interface for the development of new applications that can subsequently be used in modern compilers. Therefore, this work seeks to define a model and an IR that performs a complete and correct abstraction of the underlying hardware, providing users with the necessary tools to create their DQC applications. The proposed work is not a compiler.

This allows the different responsibilities involved in running DQC applications to be decoupled, enabling new OSs such as [39] or new application execution environments such as [51] to be developed using a common interface.



(a) Teledata. (b) Telegate.

Fig. 5. Examples of teledata and telegate circuits for the application of CZs.



(a) Quantum interconnection network, representing network elements and physical QPUs.
(b) Logical topology, representing the logical interconnection between processes.
(c) Assignment of processes to physical QPUs.
(d) Process topology related to the physical topology after process-QPU mapping.
(e) Example of communication operations between different logically connected processes.
(f) Flow of interprocess communication operations in the quantum interconnection network.

Fig. 6. Relationship between development layer abstractions and physical network structures of the network layer.

3.2.1. Data structure for logical topology

In this context, a *process* represents a logical execution entity assigned by the OS to a QPU, which participates in the distributed computation. Inspired by the classical HPC model, such as MPI, each process in the abstract model is assigned a rank and can be organized into groups and communicators. These data structures represent logical reorganizations of processes; therefore, a process may belong to multiple communicators or groups. This abstraction enables coordination of distributed tasks independently of the physical network configuration.

Fig. 6 shows the relationship between the network layer and the proposed development layer. Fig. 6(a) shows a simplified quantum interconnection network, which would be used by the development layer to generate a logical process topology, as shown in Fig. 6(b). It is important to note that we are talking about an abstraction of processes, not QPUs or network devices, as these should be abstract to the user.

This abstraction means that users do not need to have a view of the physical topology of the network, which can be highly variable depending on the context and is not always interesting or useful for developing computer programs. In this way, the developer works only with an abstract view of the topology between their running processes. The network layer then manages and optimizes the actual connections within the network, providing an effective interface between the development layer and the physical infrastructure.

Therefore, the IR that implements the development layer abstractions does not need to manage the network layer features, in order to decouple responsibilities. In this case, there are certain operations for translating from the development layer to the network layer that are the subject of the OS.

Fig. 6(c) shows the assignment of processes to processing units, in the case of the DQC, QPUs. This process abstraction is managed

by the OS of each distributed node, as indicated in Section 2.1. Fig. 6(d) shows the logical topology related to the physical topology once processes have been assigned to QPUs, with the aim of showing that some connections between processes are closer than others.

Finally, if we focus on end-user development, Fig. 6(e) shows an example of point-to-point communications between processes, where it can be seen that the instruction does not take into account the physical topology, but simply seeks to comply with the logical topology between processes. Fig. 6(f) shows the physical path followed for the execution of these instructions.

In the following, these high-level instructions for sending quantum messages are cited in more detail.

3.2.2. High-level quantum communication instructions

High-level directives conform the last piece of abstraction of the development layer. With a clearly defined semantic behavior, they are able to abstract the underlying communication protocols in DQC. This approach offers two key advantages: first, it enables the development of distributed quantum algorithms while abstracting the complexity of the underlying communication mechanisms; second, it provides the compiler with precise semantic information for each function, facilitating both optimization and the selection of the most suitable communication protocol. These instructions should prioritize fundamental computational operations, such as data transmission, reception and collective processing, rather than exposing lower-level physical or network mechanisms like entanglement generation, which should remain transparent to the user. It is important to note that blocking communication functions involve synchronization, as do other instructions such as the Barrier instructions. These operations are the responsibility of the OS of the distributed system.

These semantic instructions also allow to improve the management of the OS and the compiler, for example, to improve the fidelities of the result. When using quantum networks, as in the classical counterpart, not only is there an overhead due to communications, but additional noise is incorporated into the result. In the case of quantum computing this is crucial, since an intense use of quantum connections can cause the fidelity of the result to decrease. Using high-level instructions together with the logical topology between processes allows the operating system to better manage its allocation of processes to QPUs, minimizing communications as much as possible.

As an example of how a lack of abstraction can negatively impact both performance and software quality, consider the `entSwap` instruction defined by InQuIR. This instruction explicitly specifies the entanglement swapping procedure, which enables the connection of two quantum nodes that are not directly linked. However, this is fundamentally a low-level problem, as the development layer should not have to manage the connectivity of quantum nodes. Exposing this detail to the development layer might lead users to invoke `entSwap` unnecessarily, resulting in inefficient calls. Allowing the network — and even lower-level layers — to handle connectivity issues would contribute to more robust software, as these unneeded calls would not be performed.

4. NetQIR: a quantum intermediate representation for distributed quantum computing

This section introduces the IR proposed in this paper: NetQIR, an extension of QIR for DQC. NetQIR is defined according to the layered abstraction model presented in Section 3. NetQIR aims to fulfill the development layer by abstracting from the underlying network, being useful for future distributed quantum computing in any classical-quantum interconnection network. The responsibilities of managing the physical resources would be left to the OS, as discussed above. It is important to emphasize that an IR is fundamentally a formal specification intended for future developers. To that end, a specification has been created and a Python Software Development Kit (SDK) developed to test and work with it. This approach allows users to fully understand the specification by experimenting with actual code, thereby facilitating the production of software that employs NetQIR as an IR. Moreover, a grammar has also been developed in ANTLR to allow programmers to translate NetQIR code to specific backends, such as simulators or real systems.

4.1. NetQIR specification

This subsection details the NetQIR specification. In doing so, NetQIR defines both data structures and functions. The data structures include two components: `%Comm` and `%Group`, which correspond to the Communicator and Group described in Section 3.2.1, respectively. Regarding functions, NetQIR defines a set of state functions, data structure functions, and communication functions — the core focus of this work —. It is important to note that the state functions do not relate to the quantum state of the system but rather to the internal state of the NetQIR execution environment. In addition to this document, the authors provide a more comprehensive specification and detailed documentation on GitHub [52].[4]

4.1.1. State functions

State functions serve as breakpoints where the underlying layers of NetQIR's abstraction can be defined. For example, these functions provide a point where the compiler can determine when to query and establish connections between different quantum or classical devices. NetQIR introduces two state functions inspired by similar solutions in classical distributed computing frameworks, such as MPI. These functions are:

- `__netqir__initialize()`, which initializes the execution environment.
- `__netqir__finalize()`, which terminates the environment.

These functions establish a structured workflow for DQC, ensuring proper initialization and finalization of the execution context.

4.1.2. Operate datatypes functions

In order to abstract from the physical topology, as the development layer explained in Section 3 aims, NetQIR needs to implement a logical topology. For this purpose two already mentioned data structures have been added: `%Comm` and `%Group`. These will allow the organization of the processes in groups and the establishment of logical topologies that the processor will then be able to link with its physical version. In this abstract model, a process refers to a logical unit of execution associated with a QPU, responsible for performing computations and participating in distributed tasks. This concept, inspired by the notion of processes in classical HPC frameworks like MPI, enables the grouping and coordination of distributed quantum operations. Additionally, data type functions are defined to create or modify the described types and to obtain information about their content at runtime.

NetQIR, as it has been spurred along this work, works akin to MPI. Here another example of the similarities arises, because two key variables are associated with the so-called `comm_world`: the process `rank` and the communicator `size`. Consequently, both functions will be included:

- `__netqir__comm_rank`: returns the process `rank` inside the specified communicator.
- `__netqir__comm_size`: operation which, from a `%Comm` object, returns the number of nodes in that communicator.

Moreover, there are also functions established to create, modify or delete `%Comm` and `%Group`, and, in addition, operations to establish new logical network topologies. For further information on these functions and their use, the reader is referred to the specification [52].

[4] https://netqir.github.io/netqir-spec/.

Table 2
NetQIR functions: point-to-point and collective.

Point-to-point communication functions			
Sending functions		Receiving functions	
_netqir_qsend_array	(Array*, i32, i32, Comm*)	_netqir_qrecv_array	(Array**, i32, i32, Comm*)
_netqir_qsend_array_teledata	(Array*, i32, i32, Comm*)	_netqir_qrecv_array_teledata	(Array**, i32, i32, Comm*)
_netqir_qsend_array_telegate	(Array*, i32, i32, Comm*)	_netqir_qrecv_array_telegate	(Array**, i32, i32, Comm*)
_netqir_qsend	(Qubit*, i32, Comm*)	_netqir_qrecv	(Qubit**, i32, Comm*)
_netqir_qsend_teledata	(Qubit*, i32, Comm*)	_netqir_qrecv_teledata	(Qubit**, i32, Comm*)
_netqir_qsend_telegate	(Qubit*, i32, Comm*)	_netqir_qrecv_telegate	(Qubit**, i32, Comm*)
_netqir_measure_send_array	(Array*, i32, i32, Comm*)	_netqir_measure_recv_array	(i1*, i32, i32, Comm*)
_netqir_measure_send	(Qubit*, i32, Comm*)	_netqir_measure_recv	(i1*, i32, i32, Comm*)
Collective communication functions			
_netqir_scatter	(Array*, i32, Array*, i32, i32, Comm*)	_netqir_expose	(Qubit*, i32, Comm*)
_netqir_scatter_teledata	(Array*, i32, Array*, i32, i32, Comm*)	_netqir_expose_array	(Array*, i32, i32, Comm*)
_netqir_scatter_telegate	(Array*, i32, Array*, i32, i32, Comm*)		
_netqir_gather	(Array*, i32, Array*, i32, i32, Comm*)	_netqir_reduce	(Array*, i32, Array*, i32, i32, Comm*)
_netqir_gather_teledata	(Array*, i32, Array*, i32, i32, Comm*)	_netqir_reduce_teledata	(Array*, i32, Array*, i32, i32, Comm*)
_netqir_gather_telegate	(Array*, i32, Array*, i32, i32, Comm*)	_netqir_reduce_telegate	(Array*, i32, Array*, i32, i32, Comm*)

4.1.3. Communication functions

NetQIR proposes a large set of semantic instructions to improve the construction of DQC algorithms without the need to know the underlying communication protocols. Operations are defined to send and receive classical data, and, within quantum communications, two large sets are created: point-to-point instructions and collective communication routines. These functions are defined in Table 2 and explained below.

Collective communication operations are particularly relevant in distributed quantum algorithms, as they enable efficient coordination among multiple QPUs. NetQIR extends classical collective patterns, such as scatter and gather, to the quantum domain, while introducing new abstractions tailored for quantum-specific needs. Among them, the expose operation stands out as a novel contribution. This directive allows multiple QPUs to act upon a shared logical qubit without transferring its state explicitly, leveraging global entanglement to minimize resource consumption. By abstracting the communication protocol and leaving the implementation details to the compiler, expose exploits the layered model's advantages to reduce synchronization overhead and optimize the use of communication qubits in distributed computations.

4.1.3.1. Point-to-point communication. Point-to-point communication in quantum computing parallels that in classical computing, where one node sends or receives information to or from another node. The primary difference is that in the classical case, the information is purely classical, whereas in quantum computing, the information can be classical or quantum. Table 2 lists the directives responsible of communication in quantum computing divided into two subgroups: sending and receiving functions. Each sending function corresponds to a receiving one, both of which block the execution of the quantum program. This design ensures that for each send operation at a node there is a corresponding receive operation that unblocks it at the destination node, and vice versa. Mismatches between these could cause an incorrect behavior or compilation errors.

The most basic sending function is `__netqir__qsend`, representing the part of the circuit on the sending QPU. Additionally, the function `__netqir__measure_send` corresponds to sending a classical bit resulting from a measurement, enabling users to develop custom quantum communication protocols. Each of these functions has an `array` variant for sending or receiving arrays of qubits. It is also important to highlight that the abstraction introduced throughout this work enables the compiler to select the most appropriate communication protocol based on the execution context. If the user wishes to specify a particular protocol, they can use the specific version of the selected function, for example `__netqir__qsend_teledata` in case of wanting to use the *teledata* protocol at sending.[5]

4.1.3.2. Collective communication. While the qubit sending and receiving functions are essential primitives, they may not always be the most efficient choice. Collective communication directives address this by involving multiple QPUs in coordinated operations. They resemble those in classical distributed computing, aiding comprehension for HPC computing users. These functions include `scatter`, `gather`, `reduce` and `expose`.

- `__netqir__scatter` function distributes an array of qubits from one QPU to several others, enabling parallel processing.
- `__netqir__gather` function collects qubits from multiple QPUs into a single QPU.
- `__netqir__reduce` directive allows collecting information from multiple remote qubits and applying an operation to obtain a final result. Using `reduce` simplifies code complexity and enhances computational efficiency compared to sequences of `qsend` and `qrecv`.
- `__netqir__expose` directive, which shares a reference to a qubit with other QPUs, allowing modifications visible to the entire distributed system. This is particularly useful in operations where all nodes need to use a qubit as a target or control, such as in the distributed Quantum Fourier Transformation (QFT) algorithm [53], as illustrated in Fig. 8. In this circuit, a sequence of controlled-phase gates is applied between one target qubit and several control qubits located in different QPUs. Using `expose`, the target qubit can be made available to all control units without physically transferring its state, allowing each QPU to apply its operation as if the qubit were local. Additionally, Fig. 9 shows a possible implementation of the expose operation using a GHZ state to connect the QPUs, achieving the state $\alpha|00\ldots00\rangle + \beta|11\ldots11\rangle$ with the exposed state being $\alpha|0\rangle + \beta|1\rangle$. This implementation is not part of the development layer but represents one of the strategies available to the compiler could choose depending on the underlying physically network.

Similar to point-to-point functions, collective directives have *teledata* and *telegate* variants. Fig. 7 illustrates the use of scatter and gather operations using the teledata protocol.

[5] Notably, if a node uses `__netqir__qsend_teledata` to send, the receiving node must use `__netqir__qrecv_teledata` to receive. Mismatched protocols lead to incorrect behavior. The general functions `__netqir__qsend` and `__netqir__qrecv` offer flexibility by not specifying a protocol, allowing the other node to define it.

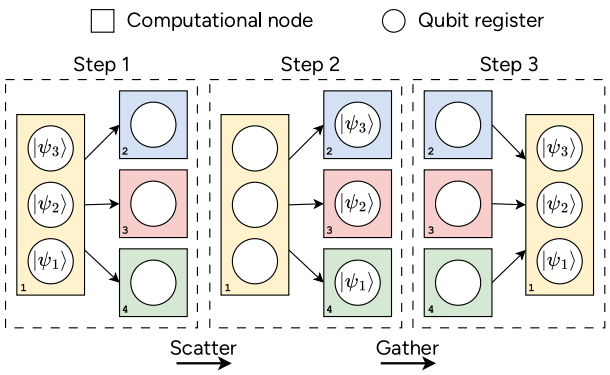


Fig. 7. An example of using a scatter teledata operation on a qubit array (steps 1 to 2) and the inverse gather teledata operation (steps 2 to 3) between 4 QPUs (labeled in each square).

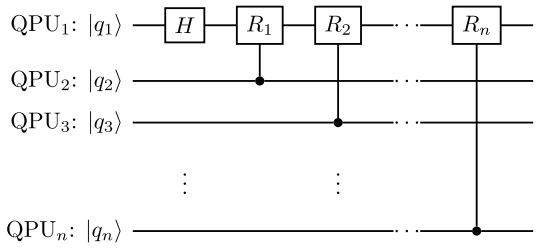


Fig. 8. Use case for the `__netqir__expose` directive on the $|q_1\rangle$ qubit, as it serves as the target for the other remote qubits.

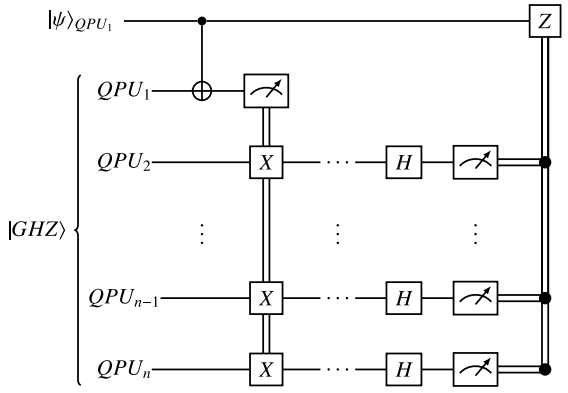


Fig. 9. Possible implementation of the `__netqir__expose` directive on the qubit $|\psi\rangle_{QPU_1}$, which is the target of the rest of the remote qubits. Distributed operations would be performed between `cat-ent` and `cat-dis`.

4.2. NetQIR SDK: PyNetQIR

The NetQIR specification constitutes the central core of an IR, serving as the common starting point for developers in the field of distributed quantum computing. To support developers in building distributed quantum applications, the NetQIR specification is accompanied by an open-source Python SDK designed to facilitate the generation of NetQIR code in Python-based environments. This tool, named PyNetQIR, is available in the project's GitHub repository.[6]

This SDK enables the generation of NetQIR code from high-level Python scripts, following a structured execution model composed of three main components: *Operations*, *Scopes*, and *Executors*. This is provided as a summary, for more information please consult the open source code repository.

- **Operations** represent the basic actions of a NetQIR program. These may include quantum instructions (such as gate applications or qubit transmissions) as well as classical instructions (like conditionals or communicator queries), allowing for hybrid programming. This hybrid approach extends the LLVM framework to integrate quantum semantics while maintaining compatibility with classical logic.
- **Scopes** are hierarchical structures that group operations logically, similar to blocks in traditional programming. The SDK provides builders (e.g., `MainScopeBuilder`) to construct these scopes and manage their contents efficiently.
- **Executors** are responsible for interpreting or compiling the operations within a scope. In the example shown in Fig. 10, a `PrinterExecutor` is used to emit the resulting NetQIR code, although other executors could target simulators or hardware backends in future implementations.

The typical flow of a NetQIR program using this SDK begins by initializing the program and obtaining the global scope and communicator. Fig. 10 shows an example of this process, where a quantum state is transferred from one QPU to another using the `qsend` and `qrecv` directives. Within the main scope, the NetQIR environment is initialized and the rank and size of the communicator are retrieved. By leveraging ranks and communicators, the SDK mirrors the structure of classical distributed frameworks like MPI, allowing developers to adopt familiar patterns when building quantum programs.

A conditional operator based on the process rank is defined: if the rank is 0, the process performs a quantum send (`qsend`); if it is 1, it performs a quantum receive (`qrecv`).[7] The environment is then finalized and the program is executed using the Executor. This example demonstrates how NetQIR supports modular and semantically clear construction of distributed quantum applications using a model inspired by classical distributed computing.

4.3. NetQIR ANTLR grammar

Once the NetQIR specification has been defined and an SDK for translating Python code into NetQIR has been implemented, it becomes essential to provide future developers with a tool for extending NetQIR. This includes developing new high-level languages that compile to NetQIR and translating NetQIR into low-level machine instructions for quantum devices. To facilitate this, a formal grammar definition is introduced, leveraging the ANTLR [54,55] specification to enable structured parsing and transformation of NetQIR code. The primary advantage of defining this grammar is that future developers can choose the target programming language for their NetQIR parser. ANTLR provides automatic code generation from its grammar definitions, supporting a wide range of well-known high-level programming languages. This flexibility facilitates the integration of NetQIR into diverse software ecosystems, enabling seamless adoption across different development environments. The grammar defined for NetQIR is encoded in the GitHub repository `netqir-grammar`.[8] It is important to note that this grammar aims to classify the different categories of NetQIR functions specified earlier. This approach allows developers to generate more specialized listeners for the generated AST tree, providing, for instance, information on whether they are programming a `qsend` function with modifiers such as `teledata` or `array`.

[6] https://github.com/netqir/netqir-sdk.

[7] Several examples, including this one, are available in the repository: https://github.com/NetQIR/netqir-sdk/tree/main/python/examples.

[8] NetQIR grammar: https://github.com/NetQIR/netqir-grammar/.

```python
program = Program()
program.start()  # start the program

gscope = program.get_global_scope()  # get the global scope
comm_world = program.get_comm_world()  # get the communicator world

main = MainScopeBuilder(1, 0) # create a builder for the main scope

main.netqir_initialize() # initialize the NetQIR environment

# Get my rank and the size of the communicator
my_rank = main.get_rank(comm_world)
main.get_comm_size(comm_world, program.get_size_world_register())

# Create a conditional operator
main.conditional(ConditionalType.EQUAL, my_rank, InmediateRank(0),
                 [lambda: main.qsend(comm_world, 0, 1)], # Rank 0
                 [lambda: main.qrecv(comm_world, 0, 0)]) # Rank 1

main.netqir_finalize() # Finalize the NetQIR environment

main.build(gscope) # Build the main scope linked to the global scope

program.run() # Run the program
program.end() # End the program
```

(a) PyNetQIR code.

```
%Qubit = type opaque;
%Result = type opaque;
%Comm = type opaque;
%size_world = alloca i32;

define void @main() {
 %q0 = alloca %Qubit*;

 call i32 @__netqir__initialize();

 %rk0 = alloca i32;
 call i32 @__netqir__comm_rank(%Comm* @netqir_comm_world, i32 %rk0);
 call i32 @__netqir__comm_size(%Comm* @netqir_comm_world, i32 %size_world);

 %t0_result = icmp eq i32 %rk0, 0;
 br i1 %t0_result, label %t1_true, label %t2_false;

%t1_true:                                          Rank 0
 call i32 @__netqir__qsend(%Qubit* %q0, i32 1, %Comm* @netqir_comm_world);
 br label %t3_continue;

%t2_false:                                         Rank 1
 call i32 @__netqir__qrecv(%Qubit* %q0, i32 0, %Comm* @netqir_comm_world);
 br label %t3_continue;

%t3_continue:

 call i32 @__netqir__finalize();
}

declare i32 @__netqir__comm_rank(%Comm*, i32);
declare i32 @__netqir__comm_size(%Comm*, i32);
declare i32 @__netqir__qrecv(%Qubit*, i32, %Comm*);
declare i32 @__netqir__finalize();
declare i32 @__netqir__initialize();
declare i32 @__netqir__qsend(%Qubit*, i32, %Comm*);
```

(b) Generated NetQIR from PyNetQIR code (reduced for document legibility).

Fig. 10. Generation of NetQIR code of teleportation circuit using PyNetQIR.

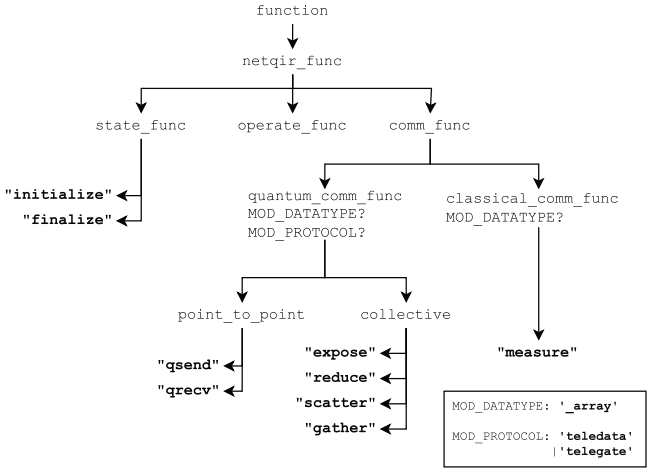


Fig. 11. NetQIR Grammar for the classification of functions in its specification.

Fig. 11 presents a syntax tree illustrating the structure of the defined grammar, where bold elements represent lexical tokens. It can be observed that both quantum and classical communication functions support modifiers, such as "`array`" for sending quantum or classical arrays, and protocol specifications like "`telegate`" or "`teledata`". It is important to note that both modifiers are optional (hence the use of the ? symbol). If no protocol is explicitly specified, the compiler will automatically select the most optimal one based on the execution context.

5. Discussion

This section presents two complementary analyses to support the design rationale behind NetQIR and to evaluate its relevance for distributed quantum computing.

Since it is not feasible to perform a quantitative comparison between IRs — given that they are formal language specifications without associated compilers or optimizers for benchmarking — in this section we conduct a qualitative comparison of various languages designed for distributed quantum computing, followed by a justification of the abstraction layer model through an evaluation of different communication protocols and quantum interconnection networks. In this case, the goal is not to develop a tool that directly improves fidelity, memory management, or execution time. Instead, we propose a model and an IR intended to support the future development of software stack tools for distributed quantum computing.

Therefore, it is proposed a qualitative comparison of different languages designed for distributed quantum and, subsequently, a justification of the abstraction layer model by evaluating different communication protocols and quantum interconnection networks.

First, a **comparative analysis with state-of-the-art languages** is performed to assess how existing DQC intermediate representations and frameworks align with the layered abstraction model proposed in this work. This comparison identifies the extent to which each solution abstracts the complexities of distributed quantum execution at the network and development layers.

Second, a **justification of the layered abstraction model** is provided, with a focus on the benefits introduced by collective communication directives. In particular, this analysis demonstrates how high-level operations, such as `scatter`, `gather`, or the novel `expose`, empower the compiler to select the most appropriate communication strategy depending on the system's topology or the specific requirements of a given algorithm. Rather than prescribing low-level instructions, these abstractions enable optimization and adaptation to the execution context, ultimately leading to more efficient resource usage and scalability. It is important to point out that the objective is to show the reader how the compiler and the OS can use one communication protocol or another depending on the problem to be solved.

Together, these two analyses highlight the practical and conceptual value of adopting a layered abstraction model and the role of NetQIR in bridging high-level algorithm design with efficient distributed quantum execution.

Table 3
Qualitative comparison table between the different languages selected for DQC programming.

Language	Network layer			Development layer		Other characteristics	
	Quantum channel	Q. Interconnection network	Communication protocols	High-level quantum comm. instructions	Data structure for logical topology	Real quantum computing inspired	Hybrid programming
NetQASM	∼	∼	✗	✗	✗	✓	✗
InQuIR	∼	✗	✗	✗	✗	✓	✗
QMPI	✓	✓	✓	✓	✓	✗	∼
NetQIR	✓	✓	✓	✓	✓	✓	✓

5.1. Comparison with state-of-the-art languages

This section presents a comparative analysis of existing DQC languages and IRs, focusing on their alignment with the layered abstraction model proposed in Section 3. This model defines essential elements across two layers — the network layer and the development layer — designed to facilitate the efficient development of DQC tools. The evaluation criteria include key aspects such as the ones outlined below:

Network layer: the network layer contains the necessary characteristics to define a correct quantum–classical connection between different nodes, abstracting from the underlying physical complexities, like:

- **Quantum Channel Abstraction**: evaluates whether the language abstracts the quantum channels used for inter-node communication, essential for managing quantum information transfer.
- **Quantum Interconnection Network**: assesses the language's ability to abstract the structure of quantum interconnection networks connecting multiple QPUs, a foundational aspect for scalable QPU architectures.
- **Communication Protocols**: identifies whether the abstraction of the communication protocol is allowed or has to be defined by the user when programming. Abstraction is essential to allow the compiler to optimize techniques according to the context.

Development layer: programming languages for DQC must incorporate features that provides users to perform distributed quantum computing while abstracting away the complexities of the underlying quantum network.

- **High-Level Quantum Communication Instructions**: considers whether the language provides high-level commands to simplify distributed quantum operations, facilitating programming efficiency and code readability. These instructions allow the compiler to provide functions with semantic context, allowing it to perform optimizations between the different possible physical implementations.
- **Data Structures for Logical Topology**: determines the language's support for data structures that abstract the physical topology of the distributed system over a logical topology, allowing easier specification of quantum node relationships and delegating to the compiler the responsibility for matching the code to the target topology.

Other characteristics: key features for an IR to enable the correct and efficient development of new tools.

- **Real Quantum Computing Inspired:** specifies whether the language is intended to perform real quantum computation or only simulated quantum computation. This feature aims to eliminate all languages that include instructions that are not allowed in the quantum model, such as perfect copying of generic states.
- **Hybrid programming:** this feature is crucial for enabling the orchestration between classical and quantum computing devices, facilitating tasks such as optimization problems.

Table 3 shows the comparison between the different languages discussed in Section 2 (related work) together with the IR proposed in this work: NetQIR.

In particular, NetQASM and InQuIR do not fully abstract the quantum channel. InQuIR requires the programmer to explicitly call operations such as `genEnt` to create entangled pairs and `entSwp` to perform entanglement swapping between QPUs. This exposes the physical routing of qubits and limits the flexibility of the compiler to adapt to different network configurations. Similarly, NetQASM uses `create_epr` to establish entanglement between nodes, providing only a minimal abstraction and restricting the channel model to EPR-based links, without considering other communication resources such as GHZ states or multi-party entanglement.

Regarding the quantum interconnection network, both approaches offer at best a partial abstraction. Although entanglement-based communication is used, the fact that the programmer must manage the flow of entangled pairs (e.g., through manual swapping or addressing specific qubits) means that the logical network structure is not decoupled from the physical implementation. For instance, in InQuIR, if a QPU wants to communicate with a non-adjacent node, the user must explicitly define a chain of entSwp instructions. This introduces rigid dependencies on the physical topology and prevents the compiler from transparently handling the routing of quantum information.

Concerning the communication protocols, there is no abstraction in this feature because the programmer has to decide how to interact with the communication qubits. Furthermore, the development layer is not implemented as neither quantum communication instructions nor structures to define a logical topology are defined. Hybrid programming is not allowed in languages such as NetQASM or InQuIR which are specific to quantum device programming.

With respect to QMPI, it allows abstraction in the fields indicated, except that it is not intended for real quantum computation because it has collective operations that are not meaningful due to the non-cloning theorem, such as the `Allscatter` or `Allgather` operation. It is also important to specify that QMPI is a message passing interface, so it is not an IR, just as MPI is not an IR. On the other hand, QMPI, being a message passing interface and not an IR, could allow this type of programming depending on its future implementations.

Finally, NetQIR, the intermediate representation for DQC proposed in this paper, would meet the above requirements by abstracting each part of the layered model. NetQIR, by extending QIR, which in turn extends LLVM, ensures hybrid programming by integrating quantum and classical programming in the same IR.

5.2. Justification of the layered abstraction model

This section aims to analyze the use of different communication protocols such as teledata or telegate against collective operations such as "`expose`". Additionally, it also looks at the comparison between different network topologies. This will allow us to analyze how the computational resources consumed can vary depending on the communication protocol used on different network topologies.

The analysis focuses on how resource consumption varies depending on the selected communication protocol, the physical topology of the quantum network, and the number of QPUs involved. By abstracting these aspects through semantic directives and delegating the decision-making to the compiler, the model allows for adaptation to the underlying infrastructure and algorithmic needs, improving overall efficiency.

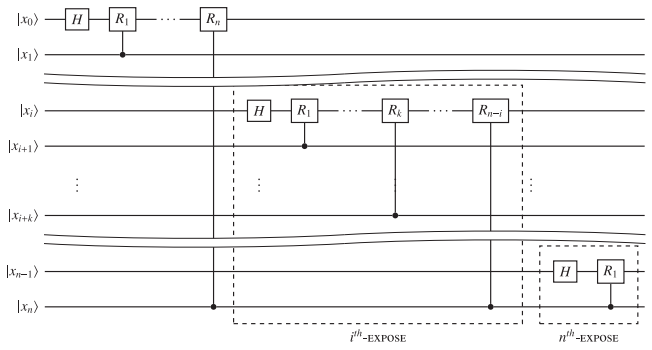


Fig. 12. Circuit for the calculation of the Quantum Fourier Transformation (QFT), where n calculations are performed on n qubits.

Two main metrics are considered: the total number of **communication qubits consumed** during circuit execution, and the number of **communication qubits each QPU must reserve** to support the communication process. These results serve to highlight the benefits of decoupling communication details from the algorithmic description, as promoted by the layered abstraction model.

To do this, the circuit in Fig. 12 was used, which represents the computation of a QFT using $n+1$ qubits. This process can be separated into n epochs, where in epoch i controlled gates are applied to qubit $|x_i\rangle$, equivalent to an `expose` function. In this use case, the QPU i is assigned the qubit $|x_i\rangle$. Therefore, the growth of QPUs will imply a growth in the number of qubits in the circuit.

It is important to note that this partitioning is done on an "ad hoc" basis for the evaluation. No circuit partitioning strategy is used. The objective is to assign a qubit to a QPU in order to maximize the number of communications in the circuit. In this case, QFT is selected as the circuit to be tested as it is resource intensive in terms of gating all the qubits. Therefore, by having each qubit in a distributed QPU, it forces the consumption of communication qubits by having to perform the controlled gate remotely.

On the other hand, the topologies to be tested are shown in Fig. 13, which are the direct connection option and the interconnection option via a communicator.

- **Direct interconnection**: this type of connection is peer-to-peer so that each QPU is connected to each of the other QPUs. This allows a direct connection between each pair of QPUs without unnecessary hops but has the disadvantage of having a tightly coupled network, making it problematic to add new nodes and requiring a large number of communication qubits.
- **Topology via one communicator**: In this case, access to the distributed system is managed through a quantum communicator (or quantum router). Each QPU is connected only to this central node, which is responsible for relaying quantum information between them. One of the main advantages of this abstraction is that it allows the compiler to choose the most suitable strategy depending on the context. In some situations, the communicator may assist the routing process by performing entanglement swapping between QPUs. In other cases, the communicator might directly establish entanglement links with the target QPU and transfer the qubit's state or apply remote operations, acting as an active participant in the communication. This flexibility highlights the benefit of leaving implementation decisions to the lower layers of the stack.

Continuing with the communication protocols evaluated, the analysis considers the use of `teledata`, `telegate`, and `expose` in both topologies. For the `expose` function, the implementation shown in Fig. 9 is used, where a GHZ state is generated to connect all involved QPUs.

In the case of `teledata`, the source QPU must send the qubit to the destination node, allow the operation to be performed, and then retrieve the updated state — requiring two EPR pairs per operation. The `telegate` protocol, which executes the operation remotely without moving the qubit, requires only a single EPR pair. Lastly, the `expose` directive leverages a shared GHZ state, allowing multiple QPUs to access the same logical qubit. This significantly reduces the number of required communication qubits, especially as the number of QPUs increases, since the cost of GHZ generation is shared across the participants.

Fig. 14 shows the results obtained, always from the perspective of resource consumption on a single QPU in the system. As can be seen in Figs. 14(a) and 14(b), the number of qubits consumed for a QPU increases with the QPUs connected, as it implies a higher number of communications between the circuit. With regard to protocols, for both topologies, the protocol with the highest qubit consumption is `teledata`, followed by `telegate`, and finally `expose`, which shows the lowest consumption. This highlights the importance of selecting an appropriate communication protocol for each specific task. By using high-level semantic functions, the compiler gains the flexibility to optimize communication by selecting the most efficient strategy based on the topology and algorithmic context. This case is particularly comprehensive as QFT has been selected as the test circuit, as it has controlled gates between all the qubits.

On the other hand, Fig. 14(c) shows the number of communication qubits each QPU must reserve to communicate with the distributed system. As expected, this number grows linearly in the case of a directly connected topology, requiring one or two additional communication qubits for each new QPU added to the system. In contrast, in the via one communicator topology, this requirement remains constant, as only one or two communication qubits are needed per QPU, regardless of the system size, since the quantum communicator handles the interconnections. As shown in Figs. 14(a) and 14(b) above, this also implies that the consumption of communication qubits is higher, as the information has to flow through the communicator.

It is important to contextualize the resource usage of NetQIR or the IRs in general. While custom low-level code can, in theory, achieve superior performance, this comes at the cost of portability, maintainability, and development complexity. The primary advantage of adopting an IR lies in its ability to act as a unifying abstraction across multiple frontends and backends. This enables the reuse of mature, optimized compiler infrastructures and promotes rapid prototyping and code generation from high-level languages.

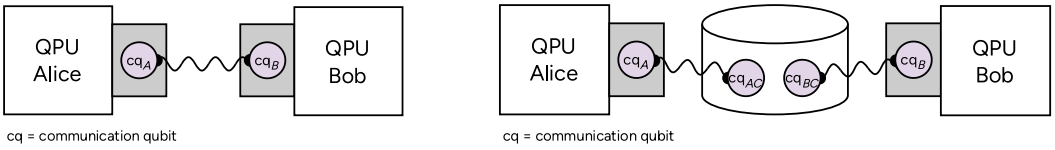
NetQIR inherits these advantages by offering a structured IR for distributed quantum computation, allowing the integration of quantum-specific optimizations without sacrificing compatibility with classical toolchains (due to the extension of QIR and LLVM). In practice, the abstraction introduced by NetQIR empowers developers to generate code more efficiently and consistently across architectures, while still enabling backend-specific optimizations through lower layers. Therefore, while the raw performance of hand-optimized code may remain unmatched, the productivity and correctness benefits of IR-based development — especially in complex distributed environments — make NetQIR a scalable alternative.

Overall, these results reinforce the value of the proposed layered abstraction model. By abstracting protocol and topology details through collective operations, the compiler is empowered to make optimal decisions, improving scalability and resource efficiency in distributed quantum systems.

6. Conclusions

This work addresses two key objectives aimed at mitigating some of the challenges identified in the literature regarding the development of compilation frameworks and software tools for distributed quantum computing.

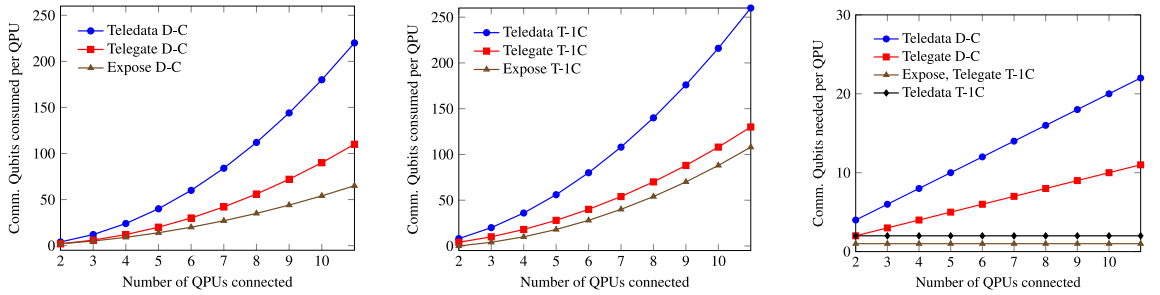
Firstly, the need to establish a common framework for the development of new IRs related to DQC is addressed by proposing the *layered*



(a) Direct interconnection.

(b) Interconnection via a communicator.

Fig. 13. Examples of different quantum network topologies for connecting QPUs within a QLAN.



(a) Consumed qubits in Directly Connected Topology (D-C). Refers to the number of communication qubits that have been measured for each single QPU (per-QPU).

(b) Consumed qubits in Topology via one communicator (T1-C). Refers to the number of communication qubits that have been measured for each single QPU (per-QPU).

(c) Communication qubits needed in the quantum register per-QPU. Refers to how many are needed at the same time for each single QPU.

Fig. 14. Comparison of the number of communication qubits consumed and needed for the QFT circuit, in function of the number of connected QPUs, the communication protocol used and the existing physical topology. It is important to note that, because of how the partitioning of the distributed circuit has been defined, one qubit has been assigned to one QPU, therefore the number of connected QPUs is equal to the number of qubits used exclusively for the QFT.

abstraction model. This model not only provides a scalable architecture but also establishes a foundation for optimization opportunities in DQC. By implementing functions that facilitate quantum data distribution across logical topologies, NetQIR reduces the complexity of DQC programming, allowing compilers to dynamically optimize based on high-level semantic directives. This model effectively addresses challenges observed in other IRs, such as NetQASM and InQuIR, which either lack flexibility in protocol handling or are too closely tied to specific network assumptions.

Secondly, an IR has been proposed in this work that meets the requirements objectively specified in the abstraction layer model, called NetQIR. It is an innovative extension of QIR for DQC, designed to address current limitations in scalability and interoperability in distributed quantum environments, allowing the hybrid programming and the use of LLVM common tools. NetQIR offers a flexible IR designed to handle quantum and classical communications across multiple QPUs by introducing high-level abstractions and communication directives. Unlike previous solutions, NetQIR integrates high-level quantum communication functions — such as point-to-point (`qsend`, `qrecv`) and collective operations (`scatter`, `gather`, `reduce`, `expose`) — enabling developers to program complex distributed algorithms with ease. This design abstracts the underlying network layer, allowing NetQIR to efficiently map communication protocols such as teledata and telegate based on the topology and specific requirements of the quantum network.

In this work, an abstraction model and IR are proposed, aiming to improve the software stack for DQC. Therefore, a compiler or optimizer that improves results such as fidelities or computational resource usage is not being proposed. This implies that, instead of working with empirical results, the focus has been on a discussion of the different existing languages for communication or quantum computing and, on the other hand, the evaluation of the impact on computational resource consumption of different communication protocols in a real problem.

The comparison between languages allows us to observe that not all of them define a direct abstraction like the one proposed in the abstraction layer model in this work. This is important, as the goal is to propose a system similar to the one already used in classical HPC environments, but incorporating the characteristics and constraints of quantum computing (especially since alternatives like QMPI propose operations that do not comply with the no-cloning theorem).

Moreover, being able to evaluate different communication protocols in various interconnection networks has also made it possible to highlight how important it is to abstract away from these features — belonging to the network layer — so that the operating system or compiler can manage which protocol or routing strategy to use.

Although hand-optimized low-level implementations may achieve maximum performance, the use of an IR such as NetQIR offers a more practical trade-off between abstraction and efficiency. By enabling the reuse of mature compilation toolchains and supporting code generation from high-level languages, NetQIR accelerates development and enhances portability across platforms. This abstraction layer is particularly valuable in DQC, where complexity and heterogeneity make manual low-level programming impractical at scale.

It is important to point out that NetQIR aims to establish an IR to develop the necessary tools for the future DQC, which in the NISQ era is not yet available due to the accumulated errors in quantum communication networks for computing. The main objective is to work on having the right languages and tools for when the FTQC era is reached.

Future work on NetQIR should prioritize developing tools that simplify its integration into new software projects. Additionally, testing its interoperability with various quantum backends and exploring advanced optimization techniques would be valuable. A well-designed toolchain could improve the management of distributed resources in NetQIR, potentially reducing communication costs and enhancing qubit allocation strategies to further boost efficiency and scalability in distributed quantum systems.

CRediT authorship contribution statement

F. Javier Cardama: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Jorge Vázquez-Pérez:** Writing – review & editing, Writing – original draft, Supervision, Software, Methodology, Investigation, Formal analysis, Conceptualization. **César Piñeiro:** Visualization, Validation, Supervision. **Tomás F. Pena:** Writing – review & editing, Validation, Supervision, Resources, Project administration,

Investigation, Funding acquisition, Conceptualization. **Juan C. Pichel:** Writing – review & editing, Validation, Supervision, Resources, Project administration, Funding acquisition. **Andrés Gómez:** Writing – review & editing, Validation, Supervision, Resources, Project administration, Investigation, Funding acquisition, Formal analysis, Conceptualization.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used ChatGPT in order to improve language and readability. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Tomas F. Pena reports financial support and article publishing charges were provided by European Union. Tomas F. Pena reports financial support was provided by Government of Spain MINECO. Tomas F. Pena reports financial support was provided by Government of Galicia. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by MICINN through the European Union NextGenerationEU recovery plan (PRTR-C17.I1), the Galician Regional Government through "Planes Complementarios de I+D+I con las Comunidades Autónomas" in Quantum Communication, MINECO (grants PID2019-104834GB-I00, PID2022-141623NB-I00 and PID2022-137061OB-C22), Consellería de Cultura, Educación e Ordenación Universitaria Galician Research Center accreditation 2024–2027 ED431G-2023/04, and the European Regional Development Fund (ERDF).

Data availability

All code is open source and has been indicated by links to the GitHub repository in the manuscript. Everyone is welcome to contribute to open source and to contact us with any questions.

References

[1] R.P. Feynman, Simulating physics with computers, Internat. J. Theoret. Phys. 21 (1982) 467–488, http://dx.doi.org/10.1007/BF02650179/METRICS, URL https://link.springer.com/article/10.1007/BF02650179.

[2] I.L. Markov, Limits on fundamental limits to computation, Nat. 2014 512: 7513 512 (2014) 147–154, http://dx.doi.org/10.1038/nature13570, URL https://www.nature.com/articles/nature13570.

[3] H. Buhrman, R. Cleve, A. Wigderson, Quantum vs. classical communication and computation, in: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, 1998, pp. 63–68.

[4] H. Riel, Quantum computing technology, Tech. Dig. - Int. Electron Devices Meet. IEDM 2021-December (2021) 1.3.1–1.3.7, http://dx.doi.org/10.1109/IEDM19574.2021.9720538.

[5] B. Koen, A. Sarkar, T. Hubregtsen, M. Serrao, A.A. Mouedenne, A. Yadav, A. Krol, I. Ashraf, C.G. Almudever, Quantum computer architecture toward full-stack quantum accelerators, IEEE Trans. Quantum Eng. 1 (2020) http://dx.doi.org/10.1109/TQE.2020.2981074.

[6] T. Haner, D.S. Steiger, T. Hoefler, M. Troyer, Distributed quantum computing with QMPI, Int. Conf. High Perform. Comput. Netw. Storage Anal. SC (2021) http://dx.doi.org/10.1145/3458817.3476172/SUPPL_FILE/TRENDS, URL https://dl.acm.org/doi/10.1145/3458817.3476172.

[7] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, M. Roetteler, Q#: Enabling scalable quantum computing and development with a high-level DSL, in: Proceedings of the Real World Domain Specific Languages Workshop 2018, in: RWDSL2018, Association for Computing Machinery, New York, NY, USA, 2018, http://dx.doi.org/10.1145/3183895.3183901.

[8] A.S. Green, P.L. Lumsdaine, N.J. Ross, P. Selinger, B. Valiron, Quipper: A scalable quantum programming language, ACM SIGPLAN Not. 48 (2013) 333–342, http://dx.doi.org/10.1145/2499370.2462177/SUPPL_FILE/PLDI154.ZIP, URL https://dl.acm.org/doi/10.1145/2499370.2462177.

[9] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F.J. Cabrera-Hernández, J. Carballo-Franquis, A. Chen, C.-F. Chen, J.M. Chow, A.D. Córcoles-Gonzales, A.J. Cross, A. Cross, J. Cruz-Benito, C. Culver, S.D.L.P. González, E.D.L. Torre, D. Ding, E. Dumitrescu, I. Duran, P. Eendebak, M. Everitt, I.F. Sertage, A. Frisch, A. Fuhrer, J. Gambetta, B.G. Gago, J. Gomez-Mosquera, D. Greenberg, I. Hamamura, V. Havlicek, J. Hellmers, L. ukasz Herok, H. Horii, S. Hu, T. Imamichi, T. Itoko, A. Javadi-Abhari, N. Kanazawa, A. Karazeev, K. Krsulich, P. Liu, Y. Luh, Y. Maeng, M. Marques, F.J. Martín-Fernández, D.T. McClure, D. McKay, S. Meesala, A. Mezzacapo, N. Moll, D.M. Rodríguez, G. Nannicini, P. Nation, P. Ollitrault, L.J. O'Riordan, H. Paik, J. Pérez, A. Phan, M. Pistoia, V. Prutyanov, M. Reuter, J. Rice, A.R. Davila, R.H.P. Rudy, M. Ryu, N. Sathaye, C. Schnabel, E. Schoute, K. Setia, Y. Shi, A. Silva, Y. Siraichi, S. Sivarajah, J.A. Smolin, M. Soeken, H. Takahashi, I. Tavernelli, C. Taylor, P. Taylour, K. Trabing, M. Treinish, W. Turner, D. Vogt-Lee, C. Vuillot, J.A. Wildstrom, J. Wilson, E. Winston, C. Wood, S. Wood, S. Wörner, I.Y. Akhalwaya, C. Zoufal, Qiskit: An open-source framework for quantum computing, 2019, http://dx.doi.org/10.5281/zenodo.2562111.

[10] M.A. Serrano, J.A. Cruz-Lemus, R. Perez-Castillo, M. Piattini, Quantum software components and platforms: Overview and quality assessment, ACM Comput. Surv. 55 (2022) 164, http://dx.doi.org/10.1145/3548679, URL https://dl.acm.org/doi/10.1145/3548679.

[11] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[12] J. Stanier, D. Watson, Intermediate representations in imperative compilers, ACM Comput. Surv. 45 (2013) http://dx.doi.org/10.1145/2480741.2480743, URL https://dl.acm.org/doi/10.1145/2480741.2480743.

[13] T.S. Metodi, S.D. Gasster, Design and implementation of a quantum compiler, in: E.J. Donkor, A.R. Pirich, H.E. Brandt (Eds.), Quantum Information and Computation VIII, vol. 7702, International Society for Optics and Photonics, SPIE, 2010, p. 77020S, http://dx.doi.org/10.1117/12.852548.

[14] K. Hietala, R. Rand, S.-H. Hung, X. Wu, M. Hicks, Verified optimization in a quantum intermediate representation, 2019, URL https://arxiv.org/abs/1904.06319v4.

[15] C.G. Almudever, L. Lao, X. Fu, N. Khammassi, I. Ashraf, D. Iorga, S. Varsamopoulos, C. Eichler, A. Wallraff, L. Geck, A. Kruth, J. Knoch, H. Bluhm, K. Bertels, The engineering challenges in quantum computing, Proc. the 2017 Des. Autom. Test Eur. DATE 2017 (2017) 836–845, http://dx.doi.org/10.23919/DATE.2017.7927104.

[16] R. Beals, S. Brierley, O. Gray, A.W. Harrow, S. Kutin, N. Linden, D. Shepherd, M. Stather, Efficient distributed quantum computing, Proc. R. Soc. A: Math. Phys. Eng. Sci. 469 (2013) http://dx.doi.org/10.1098/RSPA.2012.0686, URL http://dx.doi.org/10.1098/rspa.2012.0686orvia. http://rspa.royalsocietypublishing.org.

[17] S.W. Loke, From distributed quantum computing to quantum internet computing: An overview, 2022, URL https://arxiv.org/abs/2208.10127v2.

[18] R. Wakizaka, Towards reliable distributed quantum computing on quantum interconnects, ACM Int. Conf. Proceeding Ser. (2023) 114–116, http://dx.doi.org/10.1145/3594671.3594691, URL https://dl.acm.org/doi/10.1145/3594671.3594691.

[19] R.V. Meter, T.D. Ladd, A.G. Fowler, Y. Yamamoto, Distributed quantum computation architecture using semiconductor nanophotonics, Int. J. Quantum Inf. 8 (2009) 295–323, http://dx.doi.org/10.1142/S0219749910006435, URL http://arxiv.org/abs/0906.2686.

[20] S. Rodrigo, S. Abadal, E. Alarcon, C.G. Almudever, Will quantum computers scale without inter-chip comms? A structured design exploration to the monolithic vs distributed architectures quest, 2020 35th Conf. Des. Circuits Integr. Syst. DCIS 2020 (2020) http://dx.doi.org/10.1109/DCIS51330.2020.9268630.

[21] D. Barral, F.J. Cardama, G. Díaz, D. Faílde, I.F. Llovo, M.M. Juane, J. Vázquez-Pérez, J. Villasuso, C. Piñeiro, N. Costas, J.C. Pichel, T.F. Pena, A. Gómez, Review of distributed quantum computing. From single QPU to high performance quantum computing, 2024, URL https://arxiv.org/abs/2404.01265v1.

[22] S.-H. Wei, B. Jing, X.-Y. Zhang, J.-Y. Liao, C.-Z. Yuan, B.-Y. Fan, C. Lyu, D.-L. Zhou, Y. Wang, G.-W. Deng, et al., Towards real-world quantum networks: A review, Laser & Photonics Rev. 16 (3) (2022) 2100219.

[23] F.T. Chong, D. Franklin, M. Martonosi, Programming languages and compiler design for realistic quantum hardware, Nature 549 (7671) (2017) 180–187.

[24] S. Wehner, D. Elkouss, R. Hanson, Quantum internet: A vision for the road ahead, Science 362 (6412) (2018) eaam9288, http://dx.doi.org/10.1126/science.aam9288, URL https://www.science.org/doi/abs/10.1126/science.aam9288.

[25] J. Illiano, M. Caleffi, A. Manzalini, A.S. Cacciapuoti, Quantum internet protocol stack: A comprehensive survey, Comput. Netw. 213 (2022) 109092, http://dx.doi.org/10.1016/j.comnet.2022.109092.

[26] K. Azuma, S.E. Economou, D. Elkouss, P. Hilaire, L. Jiang, H.-K. Lo, I. Tzitrin, Quantum repeaters: From quantum networks to the quantum internet, Rev. Modern Phys. 95 (4) (2023) 045006.

[27] O. Bel, M. Kiran, Simulators for quantum network modelling: A comprehensive review, 2024, URL https://arxiv.org/abs/2408.11993. arXiv:2408.11993.
[28] A. Dahlberg, B.V.D. Vecht, C.D. Donne, M. Skrzypczyk, I.T. Raa, W. Kozlowski, S. Wehner, NetQASM—a low-level instruction set architecture for hybrid quantum–classical programs in a quantum internet, Quantum Sci. Technol. 7 (2022) 035023, http://dx.doi.org/10.1088/2058-9565/AC753F, URL https://iopscience.iop.org/article/10.1088/2058-9565/ac753f.
[29] A. Dahlberg, S. Wehner, SimulaQron—a simulator for developing quantum internet software, Quantum Sci. Technol. 4 (2018) 015001, http://dx.doi.org/10.1088/2058-9565/AAD56E, URL https://iopscience.iop.org/article/10.1088/2058-9565/aad56e.
[30] T. Coopmans, R. Knegjens, A. Dahlberg, D. Maier, L. Nijsten, J.d. Filho, M. Papendrecht, J. Rabbie, F. Rozpędek, M. Skrzypczyk, L. Wubben, W. de Jong, D. Podareanu, A. Torres-Knoop, D. Elkouss, S. Wehner, NetSquid, a NETwork simulator for quantum information using discrete events, Commun. Phys. 2021 4: 1 4 (2021) 1–15, http://dx.doi.org/10.1038/s42005-021-00647-8, URL https://www.nature.com/articles/s42005-021-00647-8.
[31] X. Wu, A. Kolar, J. Chung, D. Jin, T. Zhong, R. Kettimuthu, M. Suchara, SeQUeNCe: A customizable discrete-event simulator of quantum networks, Quantum Sci. Technol. 6 (2020) http://dx.doi.org/10.1088/2058-9565/ac22f6, URL https://arxiv.org/abs/2009.12000v1.
[32] S. Diadamo, J. Nötzel, B. Zanger, M.M. Beşe, QuNetSim: A software framework for quantum networks, IEEE Trans. Quantum Eng. 2 (2021) http://dx.doi.org/10.1109/TQE.2021.3092395.
[33] QIR Alliance: https://qir-alliance.org. URL https://github.com/qir-alliance/qir-spec.
[34] A.S. Tanenbaum, H. Bos, Modern Operating Systems, Pearson Education, Inc., 2015.
[35] G.F. Coulouris, J. Dollimore, T. Kindberg, Distributed Systems: Concepts and Design, pearson education, 2005.
[36] M.P. Forum, MPI: A message-passing interface standard, 1994.
[37] J. Bruck, D. Dolev, C.-T. Ho, M.-C. Roşu, R. Strong, Efficient message passing interface (MPI) for parallel computing on clusters of workstations, in: Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures, 1995, pp. 64–73.
[38] A. Skjellum, N. Doss, K. Viswanathan, A. Chowdappa, P. Bangalore, Extending the message passing interface (MPI), in: Proceedings Scalable Parallel Libraries Conference, 1994, pp. 106–118, http://dx.doi.org/10.1109/SPLC.1994.376998.
[39] C. Delle Donne, M. Iuliano, B. Van Der Vecht, G. Ferreira, H. Jirovská, T. Van Der Steenhoven, A. Dahlberg, M. Skrzypczyk, D. Fioretto, M. Teller, et al., An operating system for executing applications on quantum network nodes, Nature 639 (8054) (2025) 321–328.
[40] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, O. Zinenko, MLIR: Scaling compiler infrastructure for domain specific computation, in: 2021 IEEE/ACM International Symposium on Code Generation and Optimization, CGO, 2021, pp. 2–14, http://dx.doi.org/10.1109/CGO51591.2021.9370308.
[41] A. McCaskey, T. Nguyen, A MLIR dialect for quantum assembly languages, in: 2021 IEEE International Conference on Quantum Computing and Engineering, QCE, IEEE, 2021, pp. 255–264, http://dx.doi.org/10.1109/QCE52317.2021.00043.
[42] K. Hietala, R. Rand, S.-H. Hung, X. Wu, M. Hicks, A verified optimizer for quantum circuits, Proc. ACM Program. Lang. 5 (POPL) (2021) http://dx.doi.org/10.1145/3434318.
[43] L. Foundation, LLVM Assembly Language Reference Manual, URL https://releases.llvm.org/2.6/docs/LangRef.html.
[44] S. Nishio, R. Wakizaka, InQuIR: Intermediate representation for interconnected quantum computers, 2023, URL https://arxiv.org/abs/2302.00267v1.
[45] I. Quantum, Technology for the quantum future: Development roadmap, 2025, URL https://www.ibm.com/quantum/technology#roadmap. Online, (Accessed 16 June 2025).
[46] S.K. Moore, The future of quantum computing is modular, IEEE Spectr. (2025) URL https://spectrum.ieee.org/quantum-computers. (Accessed 16 June 2025).
[47] C.H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, W.K. Wootters, Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels, Phys. Rev. Lett. 70 (1993) 1895–1899, http://dx.doi.org/10.1103/PhysRevLett.70.1895, URL https://link.aps.org/doi/10.1103/PhysRevLett.70.1895.
[48] D. Gottesman, I.L. Chuang, Demonstrating the viability of universal quantum computation using teleportation and single-qubit operations, Nature 402 (6760) (1999) 390–393, http://dx.doi.org/10.1038/46503.
[49] D. Ferrari, A.S. Cacciapuoti, M. Amoretti, M. Caleffi, Compiler design for distributed quantum computing, IEEE Trans. Quantum Eng. 2 (2021) http://dx.doi.org/10.1109/TQE.2021.3053921.
[50] D. Ferrari, S. Carretta, M. Amoretti, A modular quantum compilation framework for distributed quantum computing, IEEE Trans. Quantum Eng. 4 (2023) http://dx.doi.org/10.1109/TQE.2023.3303935.
[51] B. van der Vecht, A.T. Yücel, H. Jirovská, S. Wehner, Qoala: An application execution environment for quantum internet nodes, 2025, URL https://arxiv.org/abs/2502.17296. arXiv:2502.17296.
[52] J. Vázquez-Pérez, F.J. Cardama, NetQIR/netqir-spec: v0.0.1, 2024, http://dx.doi.org/10.5281/zenodo.13142521.
[53] N.M. Neumann, R. van Houte, T. Attema, Imperfect distributed quantum phase estimation, in: Computational Science–ICCS 2020: 20th International Conference, Amsterdam, the Netherlands, June 3–5, 2020, Proceedings, Part VI 20, Springer, 2020, pp. 605–615.
[54] T.J. Parr, R.W. Quong, ANTLR: A predicated-LL (k) parser generator, Softw.: Pr. Exp. 25 (7) (1995) 789–810.
[55] T. Parr, The Definitive ANTLR 4 Reference, The Pragmatic Bookshelf, 2013.