TU Delft

Delft University of Technology

An operating system for executing applications on quantum network nodes

Delle Donne, C.; Iuliano, M.; van der Vecht, B.; Jirovská, H.; van der Steenhoven, T. J.W.; Dahlberg, A.; Skrzypczyk, M.; Montblanch, A. R.P.; Fischer, J.; van Ommen, H. B.

DOI
10.1038/s41586-025-08704-w

Publication date
2025

Document Version
Final published version

Published in
Nature

Citation (APA)
Delle Donne, C., Iuliano, M., van der Vecht, B., Jirovská, H., van der Steenhoven, T. J. W., Dahlberg, A., Skrzypczyk, M., Montblanch, A. R. P., Fischer, J., van Ommen, H. B., Demetriou, N., te Raa, I., Kozlowski, W., Taminiau, T. H., Pawełczak, P., Hanson, R., Wehner, S., & More Authors (2025). An operating system for executing applications on quantum network nodes. *Nature*, *639*, 321-328. https://doi.org/10.1038/s41586-025-08704-w

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

An operating system for executing applications on quantum network nodes

https://doi.org/10.1038/s41586-025-08704-w
Received: 17 July 2024
Accepted: 27 January 2025
Published online: 12 March 2025
Open access

Check for updates

C. Delle Donne[1,2,3,8], M. Iuliano[1,2,8], B. van der Vecht[1,2,3,8], G. M. Ferreira[1,2], H. Jirovská[1,2], T. J. W. van der Steenhoven[1,2], A. Dahlberg[1,2,3], M. Skrzypczyk[1,2,3], D. Fioretto[4], M. Teller[4], P. Filippov[4], A. R.-P. Montblanch[1,2], J. Fischer[1,2], H. B. van Ommen[1,2], N. Demetriou[1,2], D. Leichtle[5], L. Music[5], H. Ollivier[6], I. te Raa[1,2], W. Kozlowski[1,2], T. H. Taminiau[1,2], P. Pawełczak[7], T. E. Northup[4], R. Hanson[1,2] & S. Wehner[1,2,3] ✉

The goal of future quantum networks is to enable new internet applications that are impossible to achieve using only classical communication[1–3]. Up to now, demonstrations of quantum network applications[4–6] and functionalities[7–12] on quantum processors have been performed in ad hoc software that was specific to the experimental setup, programmed to perform one single task (the application experiment) directly into low-level control devices using expertise in experimental physics. Here we report on the design and implementation of an architecture capable of executing quantum network applications on quantum processors in platform-independent high-level software. We demonstrate the capability of the architecture to execute applications in high-level software by implementing it as a quantum network operating system—QNodeOS—and executing test programs, including a delegated computation from a client to a server[13] on two quantum network nodes based on nitrogen-vacancy (NV) centres in diamond[14,15]. We show how our architecture allows us to maximize the use of quantum network hardware by multitasking different applications. Our architecture can be used to execute programs on any quantum processor platform corresponding to our system model, which we illustrate by demonstrating an extra driver for QNodeOS for a trapped-ion quantum network node based on a single $^{40}Ca^{+}$ atom[16]. Our architecture lays the groundwork for computer science research in quantum network programming and paves the way for the development of software that can bring quantum network technology to society.

The first quantum networks linking several quantum processors as end nodes have recently been realized as physics experiments in laboratories[17–23] and fibre networks[24–26], opening the possibility of realizing advanced quantum network applications[2] such as data consistency in the cloud[27], privacy-enhancing proofs of deletion[28], exponential savings in communication[29] or secure quantum computing in the cloud[13,30]. Demonstrations either relied on ad hoc software or chose to establish that hardware parameters were, in principle, good enough to support a given quantum network application, although the application itself was not realized[6,31,32]. Although quantum nodes have been linked at the hardware level[17–26,33], including the design[34–37] and realization[38,39] of network stacks to manage entanglement generation, a critical innovation required to make quantum networks useful is lacking: an architecture enabling the execution of quantum applications.

It is a notable challenge to design and implement an architecture that can enable the execution of arbitrary quantum network applications on quantum processor end nodes (Fig. 1) while enabling programming in high-level software that neither depends on the underlying quantum hardware nor requires the programmer to understand the physics of the underlying devices. In the domain of the conventional internet, the possibility of programming arbitrary internet applications in high-level software has led to the realization of radically new communication applications by diverse communities, which had a transformative impact on our society[40]. What's more, the advent of programmable hardware and new application areas sparked new fields of computer science research and guided further hardware development (for example, network programming and protocols, distributed systems, internet of things and more). A similar development is underway in quantum computing, in which the availability of high-level programming tools allows a broad participation in developing applications[41].

In realizing the first such system architecture, we overcome all challenges below, including both fundamental challenges that are inherent to quantum network applications at any scale, as well as technological challenges that arise from the current state of the art of quantum network hardware.

[1]QuTech, Delft University of Technology, Delft, The Netherlands. [2]Kavli Institute of Nanoscience, Delft University of Technology, Delft, The Netherlands. [3]Quantum Computer Science, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Delft, The Netherlands. [4]Institut für Experimentalphysik, Universität Innsbruck, Innsbruck, Austria. [5]LIP6, CNRS, Sorbonne Université, Paris, France. [6]QAT, DIENS, Ecole Normale Supérieure, PSL University, CNRS, INRIA, Paris, France. [7]Embedded Systems, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Delft, The Netherlands. [8]These authors contributed equally: C. Delle Donne, M. Iuliano, B. van der Vecht. ✉e-mail: s.d.c.wehner@tudelft.nl

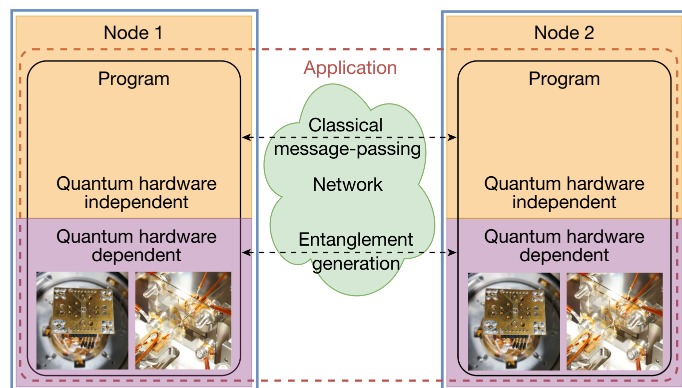

Fig. 1 | Application paradigm. A quantum networking application consists of several programs, each running on one of the end nodes[59]. An end node is a device in a quantum network that executes user applications. A network stack enables entanglement generation between end nodes over a quantum network (Supplementary Fig. 1). The distinct programs at each end node can only interact through: (1) quantum communication (for example, entanglement generation) and (2) classical communication. This allows a programmer to realize security-sensitive applications but prohibits a global orchestration of the quantum execution, such as what we might do in (distributed) quantum computing[70] in which a single quantum program is executed on several nodes. Our architecture allows programs to be written in high-level quantum-hardware-independent software and executed on a quantum-hardware-independent system that controls a hardware-dependent system (QDevice; Fig. 2), such as a NV centre node with a diamond chip (photo taken by authors, left images) or a trapped-ion quantum node[60] (right images). These platforms constitute physically very different QDevice systems but can both be programmed by our architecture.

Design considerations and challenges

Interactive classical-quantum execution

The execution of quantum network applications requires a continuing interaction between the quantum and classical parts of the execution, including interactions between different programs (Fig. 1). For example, during secure quantum computing in the cloud[13,42], the program on the server is waiting for classical messages from a remote client program before continuing the quantum execution at the server. This is in sharp contrast to quantum computing applications, in which a quantum application is a single program that can be executed in one batch, without the need to keep quantum states live while waiting for input from other programs. In quantum computing, only relatively low level and predictable interactions between classical and quantum processing are realized, such as in quantum error correction[43] or mid-circuit measurements[44]. Higher-level classical-quantum interactions in quantum computing[45] do not keep qubits live in memory.

We assume that the programs are divided into classical and quantum blocks of instructions (by a programmer or a compiler). Classical blocks consist of local classical operations executed on a conventional classical processor, as well as networked classical operations (that is, sending messages to remote nodes) executed using network devices. Quantum blocks consist of local quantum operations (gates, measurements, classical control logic), as well as networked quantum operations (entanglement generation) executed on quantum hardware. A single quantum block, in essence, corresponds to a program in quantum computing and may contain simple classical control logic, such as for the purpose of mid-circuit measurements[44].

Different hardware platforms

Interfacing with different hardware platforms presents technological challenges: at present, a clear line between software and hardware has not been defined, and the low-level control of present-day quantum processor hardware has been built to conduct physics experiments. Early microarchitectures[46,47] and operating systems[48,49] for quantum computing do not address the execution of quantum network applications. We thus have to define a hardware abstraction layer capable of interfacing with quantum network processors, including present-day setups.

Timescales

It is a fundamental challenge that different parts of such a system operate at vastly different timescales. For nodes separated by hundreds of kilometres, the duration of network operations is in the millisecond (ms) regime and some applications[2] need substantial local classical processing (ms). By contrast, the time to execute quantum operations on processing nodes is in the regime of microseconds (μs) and the low-level control (including timing synchronization between neighbouring nodes to generate entanglement[50]) requires nanosecond (ns) precision.

Memory lifetimes

Present-day quantum network nodes have short coherence times, posing a technological challenge to ensure that operations are executed within the timeframe allowed by the quantum memory.

Scheduling local and network operations

Entanglement is a key resource for quantum network applications[2]. In contrast to classical networking, entanglement is a form of stateful connection already at the physical layer in which both nodes hold one qubit. Our architecture should allow for the execution of applications at end nodes, and these may be separated by a large quantum network that facilitates entanglement generation between them. This can be achieved by the implementation of a network stack[34,35]. A technological challenge arises in the integration of such a network stack with the application stack of QNodeOS when using heralded entanglement generation at the physical layer[34] (as done in all current demonstrations linking quantum processors[25,38]). Heralded entanglement generation requires agreement between neighbouring network nodes to trigger entanglement generation in precise time bins[34], organized into a network schedule[51] that dictates when nodes make entanglement. Such a schedule could be set by a centralized controller[51] or by a distributed protocol (see, for example, ref. 34).

It is a technological challenge to manage the interdependencies between the schedule of local operations, and the networked operations, because, in all current processing node implementations[22,52], entanglement generation cannot be performed simultaneously with local operations[22,53]. Although interdependencies may be mitigated in the future[54], this implies that we cannot schedule (that is, decide when to execute) the execution of local quantum operations independently of the network schedule.

Multitasking

When executing (quantum) network applications, one node is typically idle while waiting for the other node before it can continue execution. For example, a client program may need to wait for a server to finish processing and send a message. It is hence a fundamental challenge how we can increase the utility of the system by performing multitasking[55,56], that is, allowing the concurrent execution of several programs at once to make use of idle times. Consequently, there is a need for managing state and resources for several independent programs, including processes, quantum memory management and entanglement requests.

Architecture

We divide the architecture logically into three main components (Fig. 2; Methods): the classical network processing unit (CNPU) is the logical element responsible for starting the execution of programs and the execution of classical code blocks; the quantum network processing unit (QNPU, realized on classical hardware) is the logical element responsible for governing the execution of the quantum code blocks;

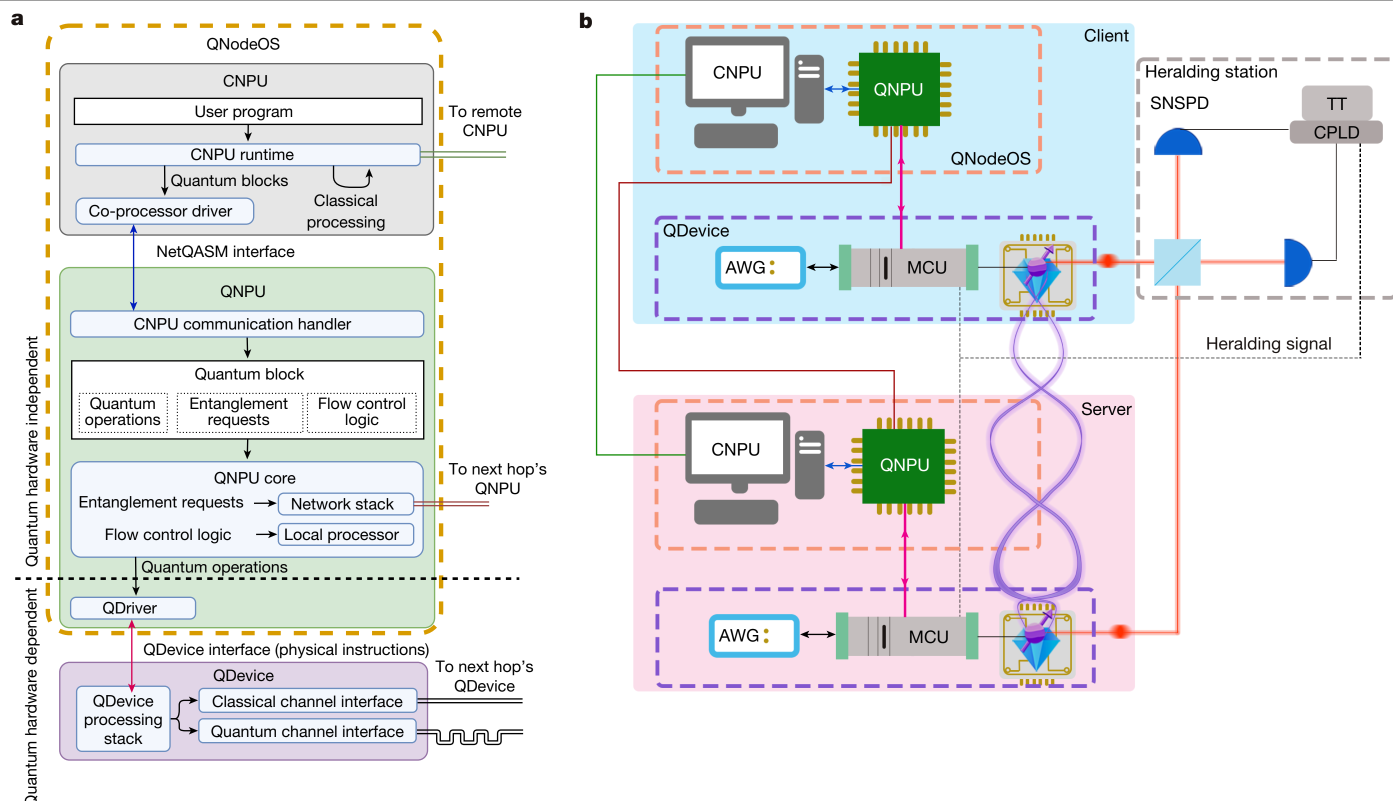

Fig. 2 | QNodeOS architecture. a, QNodeOS consists of a classical network processing unit (CNPU) and a quantum network processing unit (QNPU) (classical system). QNodeOS controls a QDevice (quantum hardware and low-level classical control). **b**, Schematic of our implementation of QNodeOS on a two-node setup in which both QDevices control a single qubit in a diamond NV centre. The CNPU is implemented on a general-purpose PC and the QNPU on an embedded system, connected by means of Gigabit Ethernet (blue). The QNPU connects to its QDevice by means of a serial peripheral interface (SPI) (pink). The two QNPUs (brown) and the two CNPUs (green) connect to each other by means of Gigabit Ethernet. The setup is based on ref. 38 with two QDevices (including AWGs and microcontroller units (MCUs); QDevices communicating over a classical digital input/output (DIO) interface) and a heralding station composed of a balanced 50:50 beam splitter (whose output ports are connected to superconducting nanowire single-photon detectors (SNSPDs) through optical fibres (red)), a time tagger (TT) and a complex programmable logic device (CPLD) that heralds the entanglement generation between QDevices and sends a classical message to the MCU.

the CNPU and the QNPU together form QNodeOS and control the QDevice, which is responsible for executing any quantum operations (gates, measurements, entanglement generation at the physical layer[34]) on the quantum hardware. On starting the execution, the CNPU creates a process (a well-known concept in classical operating systems[57,58]) on the CNPU (a CNPU process), registers the program on the QNPU (through the QNPU's end node application programming interface (API), Supplementary Information section 2.2), which, in turn, creates its own associated QNPU process (including context such as process owner, ID, process state and priority). QNodeOS also defines kernel processes on the QNPU, which are similar to user processes but are created when the system starts (on boot). The CNPU sends quantum blocks to the QNPU in the form of NetQASM subroutines[59]. Classical control logic in quantum blocks is executed by the QNPU processor. Quantum gates and measurements (from any QNPU process) and entanglement instructions (from the network process) are delegated to the QDevice by submitting physical instructions (Methods), after which the QDevice responds back to the QNPU with the result of the instruction (Supplementary Information).

To enable different hardware platforms, we introduce a QDriver realizing the hardware abstraction layer for any hardware corresponding to our minimal QDevice system model (Methods). The QDriver is the only hardware-dependent element of the architecture and is responsible for translating quantum operations, expressed in NetQASM[59], into platform-dependent (streams of) physical instructions to the underlying QDevice. We realize a QDriver for the trapped-ion system of refs. 60,61 and one for NV centres in diamond based on the system of refs. 7,22,38. We validate the trapped-ion QDriver (Fig. 3) by implementing and verifying a set of single-qubit gate operations (Methods) and the QDriver on the NV system as part of the full-stack system evaluation (see below).

To allow for different timescales, we logically divide the architecture into CNPU, QNPU and QDevice, which can thus be realized at different timescale granularities. In our proof-of-concept implementation, we realize the CNPU and QNPU on different devices, reflecting the millisecond timescales of communication between distant nodes (Methods).

Ensuring the necessary interactivity requires architectural as well as implementation choices: as programs may depend on messages from remote nodes, the architecture needs to be able to dynamically handle both classical and quantum blocks, even if not known at runtime. Consequently, it is not possible to preload all quantum blocks of the program into the low-level controller of the QDevice ahead of time, as done in previous physics experiments. Instead, in our system model, the QDevice is capable of executing individual physical instructions similar to a classical CPU. Consequently, the QNPU is continuously ready to receive new NetQASM subroutines from the CNPU and the QDevice can continuously receive and respond to physical instructions from the QNPU (Methods).

In our NV QDevice implementation, we address the challenge of interactivity by interleaving specific user-requested pulse sequences (realizing physical instructions sent from QNodeOS) and dynamical decoupling (DD) sequences (protecting quantum memory from

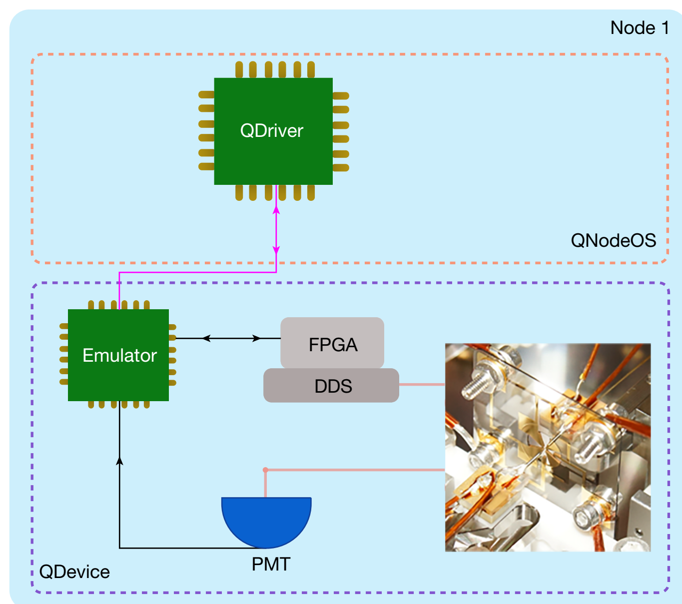

Fig. 3 | Trapped-ion QDevice implementation. Schematic of our implementation of QNodeOS on a single-node setup in which the QDevice contains a single trapped-ion qubit. The QNPU QDriver is implemented on a field-programmable gate array (FPGA) that connects to its QDevice through a SPI (Methods). The setup consists of an emulator that translates between SPI messages and TTL signals, experimental control hardware that includes a FPGA and direct digital synthesis (DDS) modules, a trapped-ion qubit[60] under ultrahigh vacuum (Fig. 1) and a photomultiplier tube (PMT) that registers atomic fluorescence.

decoherence) in an arbitrary waveform generator (AWG)[62]. The DD sequences extend qubit coherence times up to T_{coh} = 13(2) ms, while arbitrary physical instructions can be handled by triggering the corresponding pulse sequence, without knowing them in advance (Methods).

To integrate local operations with the network schedule, our architecture first introduces a QNPU scheduler that can choose which of the ready processes is assigned to the local processor (Fig. 2; Methods) and QDevice. This allows interleaving the execution of different processes directly on the QNPU without incurring delays on the timescale of the CNPU (ms), addressing the challenge of short coherence times. In our implementation, we choose to schedule QNPU processes using a priority-based non-pre-emptive scheduler[63], owing to limited quantum memory lifetimes, which make it undesirable to pre-empt and temporarily store quantum states while halting the execution. Second, we realize a network process as a kernel process, which manages entanglement generation using the network stack[34,35] (implemented in ref. 38 without the ability to execute network applications). Although our architecture can work with any way of setting a network schedule, in our implementation we choose to determine the schedule using a time-division multiple access controller[51], allowing the schedule to be centrally optimized to mitigate present-day memory decoherence. The network process handles entanglement requests submitted by user processes, coordinates entanglement generation with the rest of the network through the time-division multiple access controller, interacts with the QDevice and eventually returns entangled qubits to user processes. User processes enter the waiting state when they need entanglement and become ready again once entanglement was delivered. The network process has the highest scheduling priority and, consequently, is given precedence over the execution of any local quantum operations. We remark that local operations may still be executed during time bins already occupied by the network schedule, if a running non-pre-emptable user process prevents the network process from running, as we indeed observe in our evaluation.

To increase utility, QNodeOS allows several programs to be run concurrently, using the QNPU scheduler from above to enable multitasking[55,56] user processes on the QNPU itself. The QNPU hence needs to keep context for each process, including a virtual quantum memory space (as in classical operating systems[64]). Similar to classical memory management systems[65], a quantum memory management unit (QMMU) on the QNPU manages qubit allocations from processes and translates virtual qubit addresses in NetQASM subroutines to physical addresses in the QDevice. This allows flexibility in translating a virtual qubit address to: (1) a different physical qubit address over time, allowing qubits to be rearranged transparently in the physical memory in the future or (2) a logical qubit address when QNodeOS is executed on top of a processor using quantum error correction[43] (Methods). Entanglement generation between different pairs of processes at remote nodes are distinguished by entanglement request sockets, inspired by classical sockets[66,67], which are established once a user process requests entanglement from the network process. In our implementation, processes of the same priority are scheduled first-come-first-served[65], where the total schedule of the program in our implementation is dependent on both the schedule on the CNPU as well as that on the QNPU (Methods).

Demonstrations

Delegated computation

We first validate our architecture and implementation by the first successful execution of an arbitrary—that is, not preloaded—execution of a quantum network application in high-level software on quantum processors. We implement QNodeOS on a two-node setup of NV centres using one qubit per node (Fig. 1; Methods). We choose to execute an elementary form of delegated quantum computation (DQC)[13] from a client to a server, because the client and server programs jointly realize repetitions of a circuit (Fig. 4a) that triggers all parts of our system (Fig. 4c).

We first verify that the quantum result (fidelity) was found to be above the classical bound[68] >2/3, which verifies that QNodeOS can successfully handle interactive applications consisting of entanglement generation, millisecond-scale memory lifetimes and classical message-passing. The non-perfect fidelity (Fig. 4b) comes mainly from two sources: a noisy entangled state with fidelity 0.72(2) (quantum hardware limitation) and decoherence in the server qubit (depending on T_{coh}) as a result of waiting for several milliseconds (classical software latencies; Fig. 4d).

We proceed to characterize latencies. As expected, we find that the duration the server qubit must remain alive is dominated (>50%) by processing in the CNPU, which could be improved by caching the preparation of S2 and implementing the CNPU and the QNPU on one board (see 'Outlook' section). We observe that CNPU processing time varies greatly (standard deviation 30%; Supplementary Information section 4.6) owing to limited scheduling control over CNPU processes (Methods). Using an a priori estimate of what delays lead to too low a quality of execution (that is, delays that are too long for the server qubit to be stored with sufficiently high quality), we discard application iterations in which the CNPU latencies spiked by more than 8.95 ms. This leads to the discarding of 2% of iterations in post-processing (Methods).

Demonstration of multitasking

We also validate the multitasking capability of QNodeOS by the first concurrent execution of two quantum applications on a quantum network: the DQC application and a single-node local gate tomography (LGT) application on the client (Fig. 5a). The two programs for the client are started in the CNPU at the same time (two CNPU processes, subject to the CNPU scheduler), which means that the QNPU continuously receives subroutines for both programs from the CNPU (two QNPU processes and corresponding subroutines, subject to the QNPU scheduler). This leads to a multitasking challenge directly on the QNPU to schedule the different subroutines received (Fig. 5b). Because the client

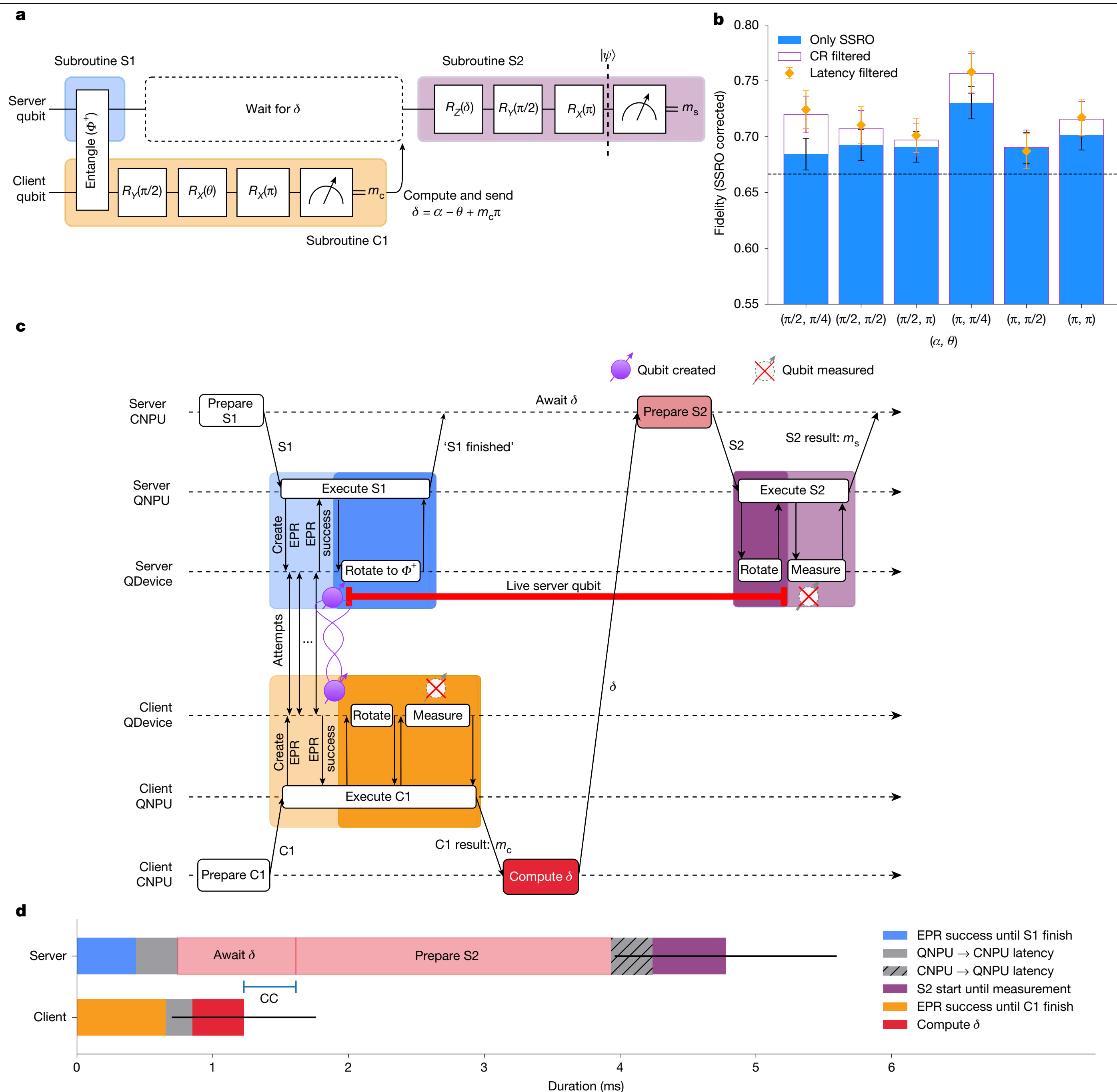

Fig. 4 | Delegated computation between two NV centre nodes using QNodeOS. a, DQC circuit (effective computation: single-qubit rotation $R_Z(\alpha)$; Methods). The DQC application consists of k circuit repetitions (varying measurement bases for tomography on $|\psi\rangle$) realized by two programs: the DQC-client program (client node, repeating the sequence 'quantum block (C1, orange)–classical block (computing δ)' k times) and the DQC-server program (server node, repeating 'quantum block (S1, blue)–classical block (receiving δ)–quantum block (S2, purple)' k times). Client and server produce entanglement $|\Phi^+\rangle = (|00\rangle + |11\rangle)/\sqrt{2}$ (S1 and first part of C1). The client performs gates and a measurement, resulting in outcome bit m_c (rest of C1). Client computes δ from m_c and DQC parameters $\alpha \in [0, 2\pi)$ and $\theta \in [0, 2\pi)$ and sends δ. Meanwhile, the server keeps its qubit coherent. On receiving δ, the server applies gates depending on δ, resulting in single-qubit state $|\psi\rangle$ (S2) depending only on α and θ. **b**, Experimental results executing DQC for six different sets of (α, θ) parameters ($k = 1{,}200$, that is, 7,200 executions of circuit Fig. 4a). Fidelity to $|\psi\rangle$ estimated using single-qubit tomography (1,200 measurement results per data point) and corrected for known tomography errors (SSRO, blue), post-selected for charge-resonance (CR) check validation (purple) and post-selected for latencies (orange) (Methods). **c**, Sequence diagram including the interaction CNPU–QNPU–QDevice for one execution of the DQC circuit (repeated $k = 1{,}200$ times in each experiment) (time flows to the right; not to scale). CNPUs prepare NetQASM subroutines (C1, S1, S2) and send them to their respective QNPUs. CNPUs perform classical computation (message δ). QNPUs execute subroutines, sending physical instructions to their QDevices. Entanglement is generated by QDevices performing a batch of attempts (triggered by a 'create EPR' physical instuction), resulting in the heralding of a two-qubit entangled state rotated to $|\Phi^+\rangle$ by the server. **d**, Processing times and latencies while server qubit is live (time frame red line in **c**, averaged over all 7,200 circuit executions except executions with latency spikes; see Methods), including CNPU–QNPU communication latencies, CNPU processing on both nodes and client–server communication latency (CC) (average total of about 4.8(±0.8) ms; error bars (standard deviation) for the sum of individual segments (per segment: Supplementary Information section 4.6)).

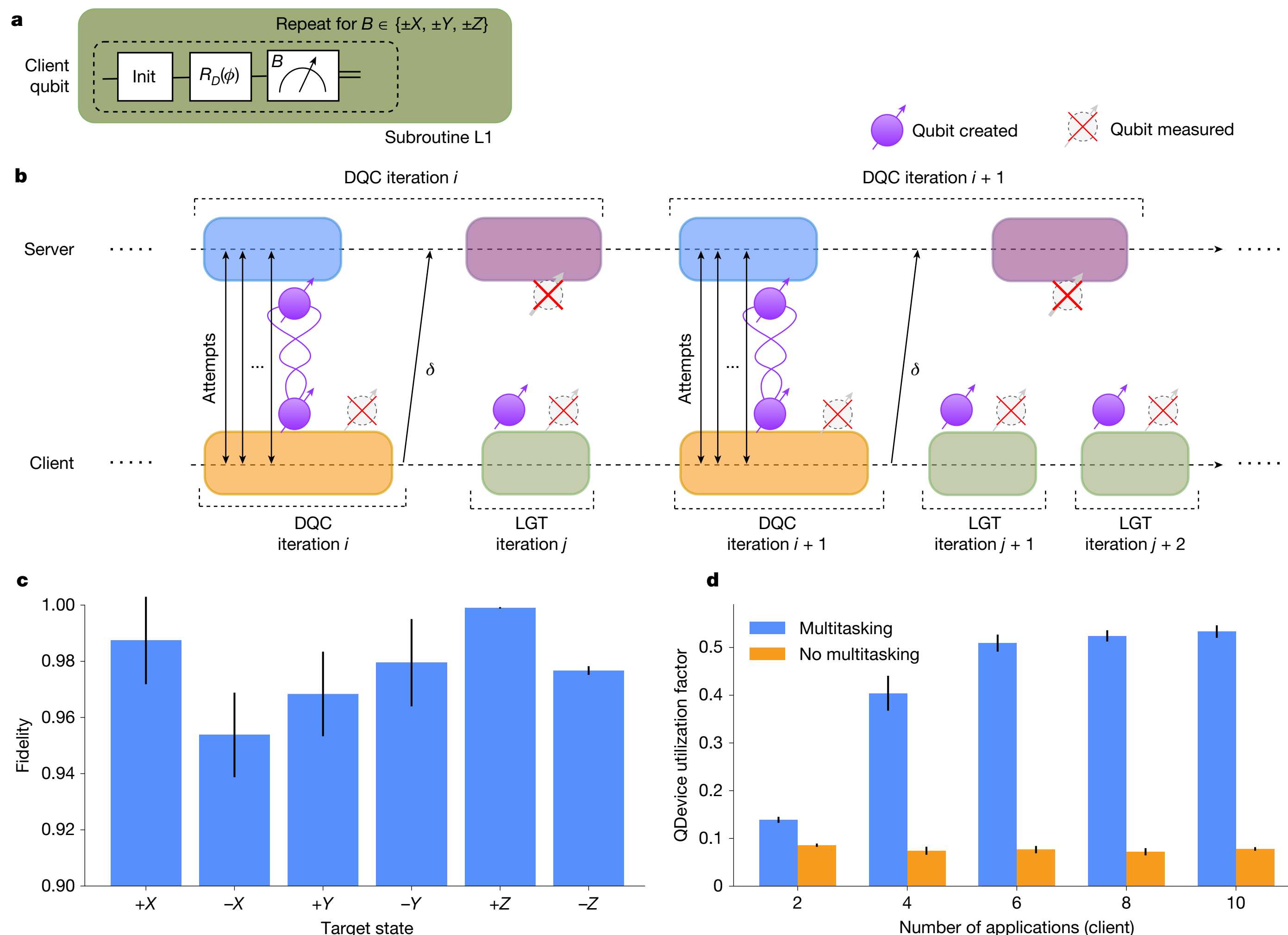

Fig. 5 | Multitasking experiment on two NV centres with QNodeOS. a, LGT circuit. A single NetQASM subroutine (L1) executes six times for bases $B \in \{\pm X, \pm Y, \pm Z\}$: initialize to $|0\rangle$, rotate around axis $D \in \{X, Y\}$ by angle ϕ, measure in B. The LGT application consists of a single LGT program, submitting L1 to the QNPU (fixed D and ϕ) k times successively. **b**, Sequence diagram illustrating concurrent execution (multitasking) of DQC (Fig. 4) and LGT on the client: two DQC circuit repetitions (Fig. 4a; two subroutines on the client (orange), four on the server (blue and purple)) and three LGT circuit repetitions (three subroutines, green). The client QNPU receives subroutines for the DQC program and the LGT program, which the QNPU scheduler can interleave: while the server executes S2 (purple), the client can not yet execute the next S1 (orange), as it involves joint entanglement generation. In idle time, the client can execute LGT subroutines (number can vary). **c**, Results of multitasking LGT (client) and DQC (on both server and client): for each input pair $(D, \phi) \in \{(X, 0), (X, \pi), (Y, \pi/2), (Y, -\pi/2), (X, -\pi/2), (X, \pi/2)\}$ (six cardinal states $\{\pm X, \pm Y, \pm Z\}$): simultaneously (1) a single LGT program was initiated on the client ($k = 1{,}000$); (2) a single DQC-client program was initiated on the client ($k = 200$ successive subroutines); and (3) a single DQC-server program was initiated on the server ($k = 200$, that is, 400 successive subroutines), resulting in a total of 6,000 LGT subroutine executions and 36,000 LGT measurement results, yielding fidelity estimates for the LGT quantum state before measurement. The results are the same as running LGT on its own (no multitasking; Supplementary Information section 5.2). **d**, Scaling number of programs on the client. For $N \in \{1, 2, 3, 4, 5\}$, we initiate simultaneously: (1) N LGT programs (each using $k = 100$) on the client; (2) N DQC-client programs on the client (each $k = 60$); and (3) N DQC-server programs on the server (each $k = 60$). This results in $2N$ programs simultaneously active on the client, each continuously submitting subroutines from the CNPU to the QNPU. Each experiment was repeated but with multitasking disabled. The plot shows the utilization factor of the QDevice (fraction of time spent executing instructions), corrected for variable entanglement generation duration (Methods), with (blue) and without (orange) multitasking, showing that multitasking can increase device utilization. Error bars are standard deviation.

has only one qubit, the multitasking of DQC and LGT never results in both programs having a quantum state alive on the client; therefore, multitasking should not affect the fidelity of LGT. We observe interleaved execution of DQC quantum blocks and LGT quantum blocks on the client node (Fig. 5b). The LGT application produces a quantum result (fidelity; Fig. 5c) equal to that in the scenario in which we run LGT on its own (not interleaved by DQC circuit executions), as expected.

We further test multitasking by scaling up the number of programs executed concurrently on the client node, up to five DQC and five LGT programs running at the same time on the client. The interleaved execution of subroutines of different programs increases device utilization (fraction of time spent on executing physical instructions) on the client QDevice compared with the same scenario but with multitasking disabled (Fig. 5d). As expected, we observe that LGT subroutines were scheduled to be executed between DQC subroutines, resulting in lower client QDevice idle time. When multitasking one DQC and one LGT program, we observe one or two subroutines between DQC iterations in most cases (LGT subroutine duration approximately 2.4 ms; Supplementary Information section 5.3). We observe cases in which both server and client QDevice remain idle, which could be improved in part by smarter CNPU-QNPU scheduling algorithms: (1) both the client and the server wait until the start of the next network schedule time bin (time bin length 10 ms); (2) the client QNPU finishes a subroutine for user process P but must wait until the CNPU sends the next

subroutine for P (up to 150 ms for one DQC and one LGT program, but less (up to only 8 ms) when more applications are running, as there are more CNPU processes independently submitting subroutines); (3) the client is ready to perform entanglement generation for DQC, but the next time bin starts only at some future time *t*, preventing activation of the network process. The scheduler activates a user process that runs a LGT circuit, which completes at some time >*t*, delaying the start of the DQC network process, even though the server node was ready at *t*.

Outlook

We designed and implemented the first architecture allowing high-level programming and execution of quantum network applications. Our architecture does not depend on the distance or connectivity between the end nodes, as long as the network stack enables the use of a quantum network to generate entanglement between them. To deploy our system onto nodes separated by several kilometres, one possible improvement to the implementation of our architecture would be to realize the CNPU and the QNPU on two devices on a single system board, ideally with mutual access to a shared memory to avoid millisecond delays in their communication. Such a merger would also allow the definition of a joint classical-quantum executable and processes, opening further doors to reduce latencies by a better scheduling control. Our architecture could also be used to distribute a quantum computing program over several quantum processors by submitting jobs as NetQASM subroutines to the QNPU of each node.

Our work provides a framework for a new domain of computer science research into programming quantum network applications on quantum processors, including new, real-time[69] scheduling algorithms for classical-quantum processes, compile methods for quantum network applications[59] or new programming language concepts including entanglement to make software development even easier, thus advancing the vision to make quantum network technology broadly available.

Online content

Any methods, additional references, Nature Portfolio reporting summaries, source data, extended data, supplementary information, acknowledgements, peer review information; details of author contributions and competing interests; and statements of data and code availability are available at https://doi.org/10.1038/s41586-025-08704-w.

1. Kimble, H. J. The quantum internet. *Nature* **453**, 1023–1030 (2008).
2. Wehner, S., Elkouss, D. & Hanson, R. Quantum internet: a vision for the road ahead. *Science* **362**, eaam9288 (2018).
3. van Meter, R. *Quantum Networking* (Wiley, 2014).
4. Barz, S. et al. Demonstration of blind quantum computing. *Science* **335**, 303–308 (2012).
5. Drmota, P. et al. Verifiable blind quantum computing with trapped ions and single photons. *Phys. Rev. Lett.* **132**, 150604 (2024).
6. Nadlinger, D. *Device-independent Key Distribution Between Trapped-ion Quantum Network Nodes*. Thesis, Univ. Oxford (2022).
7. Hermans, S. L. N. et al. Qubit teleportation between non-neighbouring nodes in a quantum network. *Nature* **605**, 663–668 (2022).
8. Iuliano, M. et al. Qubit teleportation between a memory-compatible photonic time-bin qubit and a solid-state quantum network node. *npj Quantum Inf.* **10**, 107 (2024).
9. Matsukevich, D., Maunz, P., Hayes, D., Duan, L.-M. & Monroe, C. Quantum teleportation between distant matter qubits. *Science* **323**, 486–489 (2009).
10. Langenfeld, S. et al. Quantum teleportation between remote qubit memories with only a single photon as a resource. *Phys. Rev. Lett.* **126**, 130502 (2021).
11. Pfaff, W. et al. Unconditional quantum teleportation between distant solid-state quantum bits. *Science* **345**, 532–535 (2014).
12. Chou, K. S. et al. Deterministic teleportation of a quantum gate between two logical qubits. *Nature* **561**, 368–373 (2018).
13. Broadbent, A., Fitzsimons, J. & Kashefi, E. Universal Blind Quantum Computation. *Proc. 2009 50th Annual IEEE Symposium on Foundations of Computer Science* 517–526 (IEEE, 2009).
14. Doherty, M. W. et al. The nitrogen-vacancy colour centre in diamond. *Phys. Rep.* **528**, 1–45 (2013).
15. Childress, L. & Hanson, R. Diamond NV centers for quantum computing and quantum networks. *MRS Bull.* **38**, 134–138 (2013).
16. Fioretto, D. *Towards a Flexible Source for Indistinguishable Photons Based on Trapped Ions and Cavities*. PhD thesis, Univ. Innsbruck (2020).
17. Moehring, D. L. et al. Entanglement of single-atom quantum bits at a distance. *Nature* **449**, 68–71 (2007).
18. Ritter, S. et al. An elementary quantum network of single atoms in optical cavities. *Nature* **484**, 195–200 (2012).
19. Hofmann, J. et al. Heralded entanglement between widely separated atoms. *Science* **337**, 72–75 (2012).
20. Stockill, R. et al. Phase-tuned entangled state generation between distant spin qubits. *Phys. Rev. Lett.* **119**, 010503 (2017).
21. Stephenson, L. J. et al. High-rate, high-fidelity entanglement of qubits across an elementary quantum network. *Phys. Rev. Lett.* **124**, 110501 (2020).
22. Pompili, M. et al. Realization of a multinode quantum network of remote solid-state qubits. *Science* **372**, 259–264 (2021).
23. Krutyanskiy, V. et al. Entanglement of trapped-ion qubits separated by 230 meters. *Phys. Rev. Lett.* **130**, 050803 (2023).
24. van Leent, T. et al. Entangling single atoms over 33 km telecom fibre. *Nature* **607**, 69–73 (2022).
25. Stolk, A. J. et al. Metropolitan-scale heralded entanglement of solid-state qubits. *Sci. Adv.* **10**, eadp6442 (2024).
26. Knaut, C. M. et al. Entanglement of nanophotonic quantum memory nodes in a telecom network. *Nature* **629**, 573–578 (2024).
27. Ben-Or, M. & Hassidim, A. Fast Quantum Byzantine Agreement. *Proc. Thirty-seventh Annual ACM Symposium on Theory of Computing (STOC '05)* 481–485 (ACM, 2005).
28. Poremba, A. Quantum proofs of deletion for learning with errors. Preprint at https://arxiv.org/abs/2203.01610 (2022).
29. Guérin, P. A., Feix, A., Araújo, M. & Brukner, Č. Exponential communication complexity advantage from quantum superposition of the direction of communication. *Phys. Rev. Lett.* **117**, 100502 (2016).
30. Childs, A. M. Secure assisted quantum computation. *Quantum Inf. Comput.* **5**, 456–466 (2005).
31. Liu, W.-Z. et al. Toward a photonic demonstration of device-independent quantum key distribution. *Phys. Rev. Lett.* **129**, 050502 (2022).
32. Zhang, W. et al. A device-independent quantum key distribution system for distant users. *Nature* **607**, 687–691 (2022).
33. Jing, B. et al. Entanglement of three quantum memories via interference of three single photons. *Nat. Photon.* **13**, 210–213 (2019).
34. Dahlberg, A. et al. A Link Layer Protocol for Quantum Networks. *Proc. ACM Special Interest Group on Data Communication (SIGCOMM '19)* 159–173 (ACM, 2019).
35. Kozlowski, W., Dahlberg, A. & Wehner, S. Designing a quantum network protocol. in *Proc. 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20)* 1–16 (ACM, 2020).
36. Aliro Security, Advanced Secure Networking. https://www.aliroquantum.com (2024).
37. Pirker, A. & Dü, W. A quantum network stack and protocols for reliable entanglement-based networks. *New J. Phys.* **21**, 033003 (2019).
38. Pompili, M. et al. Experimental demonstration of entanglement delivery using a quantum network stack. *npj Quantum Inf.* **8**, 121 (2022).
39. Monga, I., Saglamyurek, E., Kissel, E., Haffner, H. & Wu, W. in *Proc. 1st Workshop on Quantum Networks and Distributed Quantum Computing (QuNet '23)* 31–37 (ACM, 2023).
40. Castells, M. in *Ch@nge: 19 Key Essays on How the Internet Is Changing Our Lives* 127–148 (BBVA, 2013).
41. Quantum Software Development Kits in 2024. AIMultiple: High Tech Use Cases & Tools to Grow Your Business. https://research.aimultiple.com/quantum-sdk/ (2024).
42. Ma, Y., Kashefi, E., Arapinis, M., Chakraborty, K. & Kaplan, M. QEnclave - a practical solution for secure quantum cloud computing. *npj Quantum Inf.* **8**, 128 (2022).
43. Lidar, D. A. & Brun, T. A. *Quantum Error Correction* (Cambridge Univ. Press, 2013).
44. Botelho, L. et al. Error mitigation for variational quantum algorithms through mid-circuit measurements. *Phys. Rev. A* **105**, 022441 (2022).
45. Bharti, K. et al. Noisy intermediate-scale quantum algorithms. *Rev. Mod. Phys.* **94**, 015004 (2022).
46. Bertels, K. et al. Quantum computer architecture toward full-stack quantum accelerators. *IEEE Trans. Quantum Eng.* **1**, 1–17 (2020).
47. Fu, X. et al. eQASM: An Executable Quantum Instruction Set Architecture. Proc. *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)* 224–237 (IEEE, 2019).
48. Giortamis, E., Romão, F., Tornow, N. & Bhatotia, P. QOS: a quantum operating system. Preprint at https://arxiv.org/abs/2406.19120 (2024).
49. Kong, W. et al. Origin Pilot: a quantum operating system for effecient usage of quantum resources. Preprint at https://arxiv.org/abs/2105.10730 (2021).
50. Humphreys, P. C. et al. Deterministic delivery of remote entanglement on a quantum network. *Nature* **558**, 268–273 (2018).
51. Skrzypczyk, M. & Wehner, S. An architecture for meeting quality-of-service requirements in multi-user quantum networks. Preprint at https://arxiv.org/abs/2111.13124 (2021).
52. Drmota, P. et al. Robust quantum memory in a trapped-ion quantum network node. *Phys. Rev. Lett.* **130**, 090803 (2023).
53. Krutyanskiy, V. et al. Light-matter entanglement over 50 km of optical fibre. *npj Quantum Inf.* **5**, 72 (2019).
54. Vardoyan, G., Skrzypczyk, M. & Wehner, S. On the quantum performance evaluation of two distributed quantum architectures. *Perform. Eval.* **153**, 102242 (2022).
55. McCullough, J. D., Speierman, K. H. & Zurcher, F. W. in *Proc. November 30–December 1, 1965, Fall Joint Computer Conference, Part I (AFIPS '65 (Fall, part I))* 611–617 (ACM, 1965).
56. Dennis, J. B. Segmentation and the design of multiprogrammed computer systems. *J. ACM* **12**, 589–602 (1965).
57. Dennis, J. B. & Van Horn, E. C. Programming semantics for multiprogrammed computations. *Commun. ACM* **9**, 143–155 (1966).
58. Tanenbaum, A. S. & Woodhull, A. S. *Operating Systems Design and Implementation* 3rd edn (Prentice-Hall, 2005).
59. Dahlberg, A. et al. NetQASM—a low-level instruction set architecture for hybrid quantum–classical programs in a quantum internet. *Quantum Sci. Technol.* **7**, 035023 (2022).

60. Teller, M. et al. Integrating a fiber cavity into a wheel trap for strong ion–cavity coupling. *AVS Quantum Sci.* **5**, 012001 (2023).
61. Teller, M. et al. Heating of a trapped ion induced by dielectric materials. *Phys. Rev. Lett.* **126**, 230505 (2021).
62. Zurich Instruments. HDAWG, 750 MHz Arbitrary Waveform Generator. *Zurich Instruments* https://www.zhinst.com/europe/en/products/hdawg-arbitrary-waveform-generator (2019).
63. Liu, C. L. & Layland, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* **20**, 46–61 (1973).
64. Daley, R. C. & Dennis, J. B. Virtual memory, processes, and sharing in Multics. *Commun. ACM* **11**, 306–312 (1968).
65. Peterson, J. L. & Silberschatz, A. *Operating System Concepts* 2nd edn (Addison-Wesley, 1985).
66. Chesson, G. L. The network UNIX system. *ACM SIGOPS Oper. Syst. Rev.* **9**, 60–66 (1975).
67. Leach, P. et al. The architecture of an integrated local network. *IEEE J. Sel. Areas Commun.* **1**, 842–857 (1983).
68. Massar, S. & Popescu, S. Optimal extraction of information from finite quantum ensembles. *Phys. Rev. Lett.* **74**, 1259 (1995).
69. Ramamritham, K. & Stankovic, J. A. Scheduling algorithms and operating systems support for real-time systems. *Proc. IEEE* **82**, 55–67 (1994).
70. Caleffi, M. et al. Distributed quantum computing: a survey. *Comput. Netw.* **254**, 110672 (2024).

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Open Access This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by-nc-nd/4.0/.

© The Author(s) 2025

Methods

QDevice model

The QDevice includes a physical quantum device that can initialize and store quantum bits (qubits)—which are individually identified by a physical address—apply quantum gates, measure qubits and create entanglement with QDevices on other nodes (either entangle-and-measure or entangle-and-keep[34]), either another end node or an intermediary node in the network. We remark that the ability for two end node QDevices that are not immediate neighbours in the quantum network (but that are separated by other network nodes) to generate entanglement between them relies on the architecture implementing a network layer protocol as part of a network stack[34]. Qubits thereby refers to any possible realization of qubits, including logical qubits realized by error correction. The QDevice exposes the following interface to QNodeOS (Supplementary Information section 2.6): number of qubits available and the supported physical instructions that QNodeOS may send. Physical instructions include qubit initialization, single-qubit and two-qubit gates, measurement, entanglement creation and a 'none' for do nothing. Each instruction has a corresponding response (including entanglement success or failure or a measurement outcome) that the QDevice sends back to QNodeOS.

QNodeOS and the QDevice interact by passing messages back and forth on clock ticks at a fixed rate (100 kHz in our NV implementation, 50 kHz in the trapped-ion implementation). During each tick, at the same time: (1) QNodeOS sends a physical instruction to the QDevice and (2) the QDevice can send a response (for a previous instruction). On receiving an instruction, the QDevice performs the appropriate (sequence of) operations (for example, a particular pulse sequence in the AWG). An instruction may take several ticks to complete, for which the QDevice returns the response (success, fail, outcome) during the first clock tick following completion. The QDevice handles an entanglement instruction by performing (a batch of) entanglement generation attempts[38] (synchronized by the QDevice with the QDevice of the neighbouring node).

QNodeOS architecture

QNodeOS consists of two layers: the CNPU and the QNPU (Fig. 2a; main text and Supplementary Information). Processes on the QNPU are managed by the process manager and executed by the local processor. Executing a user process means executing NetQASM[59] subroutines (quantum blocks) of that process, which involves running classical instructions (including flow control logic) on the local processor of the QNPU, sending entanglement requests to the network stack and handling local quantum operations by sending physical instructions to the QDriver (Fig. 2a). Executing the network process means asking the network stack which request (if any) to handle and sending the appropriate (entanglement generation) instructions to the QDevice.

A QNPU process can be in the following states (see Supplementary Fig. 5 for state diagram): idle, ready, running and waiting. A QNPU process is running when the QNPU processor is assigned to it. The network process becomes ready when a network schedule time bin starts; it becomes waiting when it finishes executing and waits for the next time bin; it is never idle. A user process is ready when there is at least one NetQASM subroutine pending to be executed; it is idle otherwise; it goes into the waiting state when it requests entanglement from the network stack (using NetQASM entanglement instructions[59]) and is made ready again when the requested entangled qubit(s) is(are) delivered.

The QNPU scheduler oversees all processes (user and network) on the QNPU and chooses which ready process is assigned to the QNPU processor. CNPU processes can run concurrently and their execution (order) is handled by the CNPU scheduler. The QNPU scheduler operates independently and only acts on QNPU processes. CNPU processes can only communicate with their corresponding QNPU processes. Because several programs can run concurrently on QNodeOS, the QNPU may have numerous user processes that have subroutines waiting to be executed at the same time. Hence this requires scheduling on the QNPU.

Processes allocate qubits through the QMMU, which manages virtual qubit address spaces for each process and translates virtual addresses to physical addresses in the QDevice. The QMMU can also transfer ownership of qubits between processes, for example, from the network process (having just created an entangled qubit) to a user process that requested this entanglement.

The network stack uses entanglement request sockets (opened by user programs through the QNPU API once execution starts) to represent quantum connections with programs on other nodes. The entanglement management unit maintains all entanglement request sockets and makes sure that entangled qubits are moved to the correct process.

NV QDevice implementation

The two-node network used in this work includes the nodes 'Bob' (server) and 'Charlie' (client) (separated by 3 m) described in refs. 7,22,38. For the QDevice, we replicated the setup used in ref. 38, which mainly consists of: an ADwin-Pro II (ref. 71) acting as the main orchestrator of the setup; a series of subordinate devices responsible for qubit control, including laser pulse generators, optical readout circuits and an AWG (Zurich Instruments HDAWG[62]). The quantum physical device, based on NV centres, counts one qubit for each node. The two QDevices share a common 1-MHz clock for high-level communication and their AWGs are synchronized at the sub-nanosecond level for entanglement attempts.

We address the challenge of limited memory lifetimes by using DD. While waiting for further physical instructions to be issued, DD sequences are used to preserve the coherence of the electron spin qubit[72]. DD sequences for NV centres can prolong the coherence time (T_{coh}) up to hundreds of milliseconds (ref. 7) or even seconds[73]. In our specific case, we measured $T_{coh} = 13(2)$ ms for the server node, corresponding to about 1,300 DD pulses. The discrepancy to the state of the art for similar setups is because of several factors. To achieve such long T_{coh}, a thorough investigation of the nuclear spin environment is necessary to avoid unwanted interactions during long DD sequences, resulting in an even more accurate choice of interpulse delay. Other noise sources include unwanted laser fields, the quality of microwave pulses and electrical noise along the microwave line.

A specific challenge arises at the intersection of extending memory lifetimes using DD and the need for interactivity: to realize individual physical instructions, many waveforms are uploaded to the AWG, in which the QDevice decodes instructions sent by QNodeOS into specific preloaded pulse sequences. This results in a waveform table containing 170 entries. The efficiency of the waveforms is limited by the waveform granularity of the AWG that corresponds to steps that are multiples of 6.66 ns, having a direct impact on the T_{coh}. We are able to partially overcome this limitation by using the methods described in ref. 74. Namely, each preloaded waveform, corresponding to one single instruction, has to be uploaded 16 times to be executed with sample precision. To not fill up the waveform memory of the device, we apply the methods in ref. 74 only to the DD pulses that are played while the QDevice waits for an instruction from the QNPU, whereas the instructed waveforms (gate/operation + first block of the XY8 DD sequence) are padded according to the granularity, if necessary.

The list of physical instructions supported by our NV QDevice is given in Supplementary Information section 3.1.

NV QNPU implementation

The QNPUs for both nodes are implemented in C++ on top of FreeRTOS[75], a real-time operating system for microcontrollers. The stack runs on a dedicated MicroZed[76], an off-the-shelf platform based on the Zynq 7000 SoC, which hosts two ARM Cortex-A9 processing cores, of which only one is used, clocked at 667 MHz. The QNPU was implemented on top of FreeRTOS to avoid reimplementing standard

operating system primitives such as threads and network communication. FreeRTOS provides basic operating system abstractions such as tasks, intertask message-passing and the TCP/IP stack. The FreeRTOS kernel—like any other standard operating system—cannot however directly manage the quantum resources (qubits, entanglement requests and entangled pairs) and hence its task scheduler cannot take decisions based on such resources. The QNPU scheduler adds these capabilities (Supplementary Information section 2.5).

The QNPU connects to peer QNPU devices by means of TCP/IP over a Gigabit Ethernet interface (IEEE 802.3 over full-duplex Cat 5e). The communication goes through two network switches (Netgear JGS524PE, one per node). The two QNPUs are time-synchronized through their respective QDevices (granularity 10 μs), as these already are synchronized at the microsecond level (common 1-Mhz clock).

The QNPU device interfaces with the QDevice's ADwin-Pro II through a 12.5-MHz serial peripheral interface (SPI) interface, used to exchange 4-byte control messages at a rate of 100 kHz.

NV CNPU implementation

The CNPUs for both nodes are a Python runtime executing on a general-purpose desktop machine (four Intel 3.20-GHz cores, 32 GB RAM, Ubuntu 18.04). The choice of using a high-level system was made as the communication between distant nodes would ultimately be in the millisecond timescale and this allows for ease of programming the application. The CNPU machine connects to the QNPU device through TCP over a Gigabit Ethernet interface (IEEE 802.3 over full-duplex Cat 8, average ping round-trip time of 0.1 ms), through the same single network switch as mentioned above (one per node), and sends application registration requests and NetQASM subroutines over this interface (10 to 1,000 bytes, depending on the length of the subroutine). CNPUs communicate with each other through the same two network switches.

Scheduler implementations

We use a single Linux process (Python) for executing programs on the CNPU. CNPU 'processes' are realized as threads created within this single Python process. Python was chosen because the NetQASM SDK is implemented in Python. When running several programs concurrently, a pool of such threads is used. Scheduling of the Python process and its threads is handled by the Linux operating system. Each thread establishes a TCP connection with the QNPU to use the QNPU API (including sending subroutines and receiving their results) and executes the classical blocks for its corresponding program.

Both the CNPU and the QNPU maintain processes for running programs. The CNPU scheduler (standard Linux scheduler, see above) schedules CNPU processes, which indirectly controls in which order subroutines from different programs arrive at the QNPU. The QNPU scheduler handles subroutines of the same process priority on a first-come-first-served basis, leading however to executions of QNPU processes not in the order submitted by the CNPU (Supplementary Information section 5.3).

Using only the CNPU scheduler is not sufficient because: (1) we want to avoid millisecond delays needed to communicate scheduling instructions across the CNPU and the QNPU; (2) user processes need to be scheduled in conjunction with the network process (meeting the challenge of scheduling both local and network operations), which is only running on the QNPU; and (3) QNPU user processes need to be scheduled with respect to each other (for example, a user process is waiting after having requested entanglement, allowing another user process to be run, as observed in the multitasking demonstration).

Sockets and the network schedule

In an entanglement request socket, one node is a 'creator' and the other is a 'receiver'. As long as an entanglement request socket is open between the nodes, an entanglement request from only the creator suffices for the network stack to handle it in the next corresponding time bin, that is, the 'receiver' can comply with entanglement generation even if no request has (yet) been made to its network stack.

Trapped-ion implementation

The experimental system used for the trapped-ion implementation is discussed in refs. 60,61 and is described in detail in ref. 77. The implementation itself is described in ref. 16. We confine a single $^{40}Ca^{+}$ ion in a linear Paul trap; the trap is based on a 300-μm-thick diamond wafer on which gold electrodes have been sputtered. The ion trap is integrated with an optical microcavity composed of two fibre-based mirrors, but the microcavity is not used here. The physical-layer control infrastructure consists of C++ software, Python scripts, a pulse sequencer that translates Python commands to a hardware description language for a field-programmable gate array (FPGA) and hardware that includes the FPGA, input triggers, direct digital synthesis (DDS) modules and output logic.

QNodeOS provides physical instructions through a development FPGA board (Texas Instruments, LAUNCHXL2-RM57L (ref. 78)) that uses a SPI. We programmed another board (Cypress, CY8CKIT-143 (ref. 79)) that translates SPI messages into TTL signals compatible with the input triggers of our experimental hardware.

The implementation consisted of sequences composed of seven physical instructions: initialization, $R_X(\pi)$, $R_Y(\pi)$, $R_X(\pi/2)$, $R_Y(\pi/2)$, $R_Y(-\pi/2)$ and measurement. First, we confirmed that message exchange occurred at the rate of 50 kHz as designed. Next, we confirmed that we could trigger the physical-layer hardware. Finally, we implemented seven different sequences. Each sequence was repeated 10^4 times, which allowed us to acquire sufficient statistics to confirm that our QDriver results are consistent with operation in the absence of the higher layers of QNodeOS.

Metrics

Both classical and quantum metrics are relevant in the performance evaluation: the quantum performance of our test programs is measured by the fidelity $F(\rho, |\tau\rangle)$ of an experimentally obtained quantum state ρ to a target state $|\tau\rangle$, in which $F(\rho, |\tau\rangle) = \langle\tau|\rho|\tau\rangle$, estimated by quantum tomography[80]. Classical performance metrics include device utilization $T_{util} = 1 - T_{idle}/T_{total}$, in which T_{idle} is the total time that the QDevice is not executing any physical instruction and T_{total} is the duration of the whole experiment, excluding time spent on entanglement attempts (see below).

Experiment procedure NV demonstration

Applications are written in Python using the NetQASM SDK[59] (code in Supplementary Information), with a compiler targeting the NV flavour[59], as it includes quantum instructions that can be easily mapped to the physical instructions supported by the NV QDevice. The client and server nodes independently start execution of their programs by invoking a Python script on their own CNPU, which then spawns the threads for each program. During application execution, the CNPUs have background processes running, including QDevice monitoring software.

A fixed network schedule is installed in the two QNPUs with consecutive time bins (all assigned to the client–server node pair) with a length of 10 ms (chosen to be equal to 1,000 communication cycles between QNodeOS and QDevice as in ref. 38) to assess the performance without introducing a dependence on a changing network schedule. During execution, the CNPUs and QNPUs record events including their time stamps. After execution, corrections are applied to the results (see below) and event traces are used to compute latencies.

DQC

Our demonstration of DQC (Fig. 4) implements the effective single-qubit computation $|\psi\rangle = H \circ R_Z(\alpha) \circ |+\rangle$ on the server, as a simple form of blind quantum computing that hides the rotation angle α from the server when executed with randomly chosen θ and not performing

tomography. The remote entanglement protocol used is the single-photon protocol[81–83] (Supplementary Information section 3.1).

Filtering

Results, with no post-selection, are presented including known errors that occur during the tomography single-shot readout (SSRO) process (Fig. 4b, blue) (details on the correction in the Supplementary Information of ref. 22). We also report the post-selected results in which data are filtered on the basis of the outcome of the charge-resonance check[84] after one application iteration (Fig. 4b, purple). This filter enables the elimination of false events, specifically when the emitter of one of the two nodes is not in the right charge state (ionization) or the optical resonances are not correctly addressed by the laser fields after the execution of one iteration of DQC.

Further filtering (Fig. 4b, latency filtered) is performed on those iterations that showed latency not compatible with the combination of T_{coh} of the server and the average entangled state fidelity. For this filter, a simulation (using a depolarizing model, based on the measured value T_{coh}; Supplementary Information section 4.4) was used to estimate the single-qubit fidelity (given the entanglement fidelity measured above) as a function of the duration the server qubit stays live in memory in a single execution of the DQC circuit (Fig. 4a). This gives a conservative upper bound of the duration as 8.95 ms, to obtain a fidelity of at least 0.667. All measurement results corresponding to circuit executions exceeding 8.95 ms duration were discarded (146 out of 7,200 data points).

Other main sources of infidelity not considered in this analysis of the outcome include, for instance, the non-zero probability of double excitation for the NV centre[83]. During entanglement generation, the NV centre can be re-excited, leading to the emission of two photons that lower the heralded entanglement fidelity. The error can be corrected by discarding those events that registered, in the entanglement time window, a photon at the heralding station (resonant zero-phonon line photon) and another one locally at the node (off-resonant phonon sideband photon).

Finally, the dataset presented in Fig. 4b (not shown chronologically) was taken in 'one shot' to prove the robustness of the physical layer, therefore no calibration of relevant experimental parameters was performed in between, leading to possible degradation of the overall performance of the NV-based setup.

The single-qubit fidelity is calculated with the same methods as in ref. 8, measuring in the state $|i\rangle$ and in its orthogonal state $|-i\rangle$, provided that we expect the outcome $|i\rangle$, whereas the two-qubit state fidelity is computed taking into account only the same positive-basis correlators (*XX*, *YY*, *ZZ*).

Multitasking: delegated computation and LGT

In the first multitasking evaluation, we concurrently execute two programs on the client: a DQC-client program (interacting with a DQC-server program on the server) and a LGT program (on the client only) (Fig. 5). The client CNPU runtime executes the threads, executing the two different programs concurrently. The client QNPU has two active user processes, each continuously receiving new subroutines from the CNPU, which are scheduled with respect to each other and the network process.

Estimates of the fidelity (Fig. 5b) include the same corrections as in the Supplementary Information of ref. 22. To assess the quantum performance of the LGT application, we used a mocked entanglement generation process on the QDevices (executing entanglement actions without entanglement) to simplify the test: weak-coherent pulses on resonance with the NV transitions, which follow the usual optical path, are used to trigger the complex programmable logic device in the entanglement heralding time window. This results in comparable application behaviour for DQC (comparable rates and latencies; Supplementary Information section 5.1) with respect to multitasking on QNodeOS.

Multitasking: QDevice utilization when scaling number of programs

We scale the number of programs being multitasked (Fig. 5d). We observe how the client QNPU scheduler chooses the execution order of the subroutines submitted by the CNPU. DQC subroutines each have an entanglement instruction, causing the corresponding user process to go into the waiting state when executed (waiting for entanglement from the network process). The QNPU scheduler schedules another process ((56%, 81%, 99%) for ($N = 1, N = 2, N > 2$)) of the times that a DQC process is put into the waiting state (demonstrating that the QNPU schedules independently from the order in which the CNPU submits subroutines). The number of consecutive LGT subroutines (of any LGT process; LGT block execution time approximately 2.4 ms) that is executed between DQC subroutines is 0.83 for $N = 1$, increasing for each higher N until 1.65 for $N = 5$, showing that indeed idle times during DQC are partially filled by LGT blocks (Supplementary Information section 5.3).

Device utilization (see 'Metrics' section) quantifies only the utilization factor between entanglement generation time windows to fairly compare the multitasking and non-multitasking scenarios. In both scenarios, the same entanglement generation processes are performed, which—hence—have the same probabilistic durations in both cases. To avoid inaccurate results owing to this probabilistic nature, we exclude the entanglement generation time windows in both cases.

Data availability

The datasets that support this manuscript and the software to analyse them are available at https://doi.org/10.4121/6aa42f05-6823-4848-b235-3ea19e39f4ae.

Code availability

The application software development kit used for writing program code is open-sourced on GitHub[85]. The QNodeOS source code is not open source at present, but site visits are possible for academic researchers.

71. ADwin-Pro II – flexible and modular. https://www.adwin.de/us/produkte/proII.html (2024).
72. De Lange, G., Wang, Z., Riste, D., Dobrovitski, V. & Hanson, R. Universal dynamical decoupling of a single solid-state spin from a spin bath. *Science* **330**, 60–63 (2010).
73. Abobeih, M. H. et al. One-second coherence for a single electron spin coupled to a multi-qubit nuclear-spin environment. *Nat. Commun.* **9**, 2552 (2018).
74. Corna, A. Efficient generation of dynamic pulses. *Zurich Instruments* https://www.zhinst.com/europe/en/blogs/efficient-generation-dynamic-pulses (2021).
75. FreeRTOS. Real-time operating system for microcontrollers and small microprocessors. *FreeRTOS* https://www.freertos.org/ (2024).
76. MicroZed. Development board based on the Zynq-7000 SoC. *Avnet* https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/microzed/ (2024).
77. Teller, M. *Measuring and Modeling Electric-field Noise in an Ion-cavity System*. PhD thesis, Univ. Innsbruck (2021).
78. LAUNCHXL2-RM57L. Hercules RM57Lx LaunchPad Development Kit. *Texas Instruments* https://www.ti.com/tool/LAUNCHXL2-RM57L (2024).
79. CY8CKIT-143A. *Infineon Technologies* https://www.infineon.com/cms/en/product/evaluation-boards/cy8ckit-143a/ (2024).
80. Paris, M. & Rehacek, J. *Quantum State Estimation* Vol. 649 (Springer, 2004).
81. Cabrillo, C., Cirac, J. I., García-Fernández, P. & Zoller, P. Creation of entangled states of distant atoms by interference. *Phys. Rev. A* **59**, 1025–1033 (1999).
82. Bose, S., Knight, P. L., Plenio, M. B. & Vedral, V. Proposal for teleportation of an atomic state via cavity decay. *Phys. Rev. Lett.* **83**, 5158–5161 (1999).
83. Hermans, S. L. N. et al. Entangling remote qubits using the single-photon protocol: an in-depth theoretical and experimental study. *New J. Phys.* **25**, 013011 (2023).
84. Robledo, L., Bernien, H., van Weperen, I. & Hanson, R. Control and coherence of the optical transition of single nitrogen vacancy centers in diamond. *Phys. Rev. Lett.* **105**, 177403 (2010).
85. NetQASM. *GitHub* https://github.com/QuTech-Delft/netqasm (2024).

Acknowledgements This research was supported by the Quantum Internet Alliance through the European Union's Horizon 2020 programme under grant agreement no. 820445 and from the Horizon Europe programme grant agreement no. 101080128. S.W. acknowledges support from a Dutch Research Council (NWO) Vici grant. M.I., A.R.-P.M. and R.H. also acknowledge support from the NWO through the Spinoza Prize 2019 (no. SPI 63-264). This project has also received financing from the European Research Council (ERC) under the European Union's

Horizon 2020 research and innovation programme (grant agreement no. 852410). H.B.v.O. acknowledges support from the joint research program "Modular quantum computers" by Fujitsu Limited and Delft University of Technology, co-funded by the Netherlands Enterprise Agency under project number PPS2007.

Author contributions S.W. conceived the project; A.D., M.S. and S.W. developed the initial ideas; C.D.D., B.v.d.V., W.K., M.S., I.t.R., P.P. and S.W. designed the QNodeOS architecture; C.D.D., B.v.d.V., W.K., I.t.R., G.M.F., T.J.W.v.d.S., H.J., M.S. and P.P. implemented the operating system; M.I. prepared the nitrogen-vacancy setup, with help from A.R.-P.M., J.F., H.B.v.O. and N.D.; C.D.D., M.I., B.v.d.V., W.K., I.t.R., T.E.N., R.H. and S.W. devised the experiments; M.I. and B.v.d.V. performed the nitrogen-vacancy experiments, analysed the data and discussed the results with all of the authors.; D.F., M.T. and P.F. prepared the trapped-ion setup; C.D.D., D.F. and M.T. performed the trapped-ion integration tests; B.v.d.V. analysed the delegated computation protocol, with the help of D.L., L.M. and H.O.; M.I., B.v.d.V., T.E.N. and S.W. wrote the main text, with input from all authors; C.D.D., M.I., B.v.d.V., W.K., P.P., T.E.N. and S.W. wrote the Supplementary Material; T.H.T., P.P., T.E.N., R.H. and S.W. supervised the research; S.W. supervised the collaboration.

Competing interests C.D.D., B.v.d.V., A.D., M.S., I.t.R., W.K. and S.W. have filed a patent on the QNodeOS architecture.

Additional information
Supplementary information The online version contains supplementary material available at https://doi.org/10.1038/s41586-025-08704-w.
Correspondence and requests for materials should be addressed to S. Wehner.
Peer review information *Nature* thanks Claudio Cicconetti and the other, anonymous, reviewer(s) for their contribution to the peer review of this work. Peer reviewer reports are available.
Reprints and permissions information is available at http://www.nature.com/reprints.