Cairo University
Faculty of Engineering
Department of Computer Engineering

RunSurge

A Graduation Project Report Submitted

to

Faculty of Engineering, Cairo University

in Partial Fulfillment of the requirements of the degree

of

Bachelor of Science in Computer Engineering.

Presented by
Abdelrahman Mohamed Mahmoud Abdelfatah

Supervised by
Dr/Lydia Wahid

July 1st 2025

Abstract

Modern computational workloads often suffer from a mismatch between demand and available processing power. While many users possess idle computational resources, others are constrained by limited local capabilities. RunSurge addresses this challenge by introducing the Surge Network, a distributed system that bridges this gap—enabling efficient and decentralized code execution by connecting users in need of processing power with contributors offering surplus computational capacity. The objective of this project is to create a scalable, user-friendly, and intelligent system that facilitates the remote execution of user-submitted code across a network of volunteer machines. To achieve this, RunSurge is designed around three core modules:

1. Master Module: Serves as the central coordinator responsible for task scheduling, resource allocation, and result aggregation.
2. Worker Module: Deployed by resource-contributing users, it executes assigned tasks using the local machine's spare capacity.
3. Parallelization Module: Analyzes and transforms submitted code into a form suitable for distributed execution, managing synchronization and data dependencies when necessary.

The system supports two execution modes:

- Automatic Mode: Abstracts complexity from the user, automatically detecting parallelism opportunities and estimating resource requirements.
- Manual Mode: Offers advanced users the flexibility to specify task structure, memory usage, and other execution constraints.

In this report we will be diving deep into the architecture the Communication Module which is the Master Module gRPC Server and the Node Worker.

الملخص

مشاكل الحوسبة الحديثة تعاني في كثير من الأحيان من عدم التوازن بين الطلب وقدرة المعالجة المتاحة حيث يمتلك العديد من المستخدمين موارد حسابية غير مستغلة بينما يواجه آخرون قيودًا بسبب قدراتهم المحلية المحدودة مشروع RunSurge يعالج هذه المشكلة من خلال تقديم شبكة Surge وهي نظام موزع يربط بين المستخدمين الذين يحتاجون إلى قوة معالجة والمستخدمين الذين لديهم موارد فائضة مما يتيح تنفيذًا فعالًا ولا مركزيًا للشفرات البرمجية يهدف هذا المشروع إلى إنشاء نظام ذكي وقابل للتوسع وسهل الاستخدام يتيح تنفيذ الشفرات البرمجية المرسلة عن بُعد عبر شبكة من الأجهزة التطوعية لتحقيق هذا الهدف تم تصميم RunSurge اعتمادًا على ثلاثة مكونات رئيسية:

1. الوحدة الرئيسية تعمل كمنسق مركزي مسؤول عن جدولة المهام وتخصيص الموارد وتجميع النتائج.
2. وحدة العامل يتم نشرها من قبل المستخدمين المساهمين بالموارد وتقوم بتنفيذ المهام الموكلة باستخدام القدرة الفائضة من أجهزتهم.
3. وحدة التوازي تقوم بتحليل الشفرات وتحويلها إلى صيغة قابلة للتنفيذ الموزع مع إدارة التزامن واعتماد البيانات عند الحاجة.

يدعم النظام وضعين للتنفيذ:

- الوضع التلقائي يخفي التعقيد عن المستخدم ويقوم تلقائيًا بالكشف عن فرص التوازي وتقدير متطلبات الموارد
- الوضع اليدوي يمنح المستخدمين المتقدمين القدرة على تحديد بنية المهام واستخدام الذاكرة والمعايير الأخرى للتنفيذ

في هذا التقرير، سنتعمق في معمارية وحدة الاتصال ، والتي تتكون من خادم gRPC للوحدة الرئيسية ووحدة العامل.

ACKNOWLEDGMENT

I extend my deepest gratitude to my beloved family, whose unwavering support and endless encouragement have been the bedrock of my academic journey. Special appreciation goes to my incredible Mom, whose sacrifices, belief in my potential, and countless hours spent nurturing my ambitions provided the constant fuel I needed to persevere. From guiding my very first steps to patiently observing my growth over the years, her love has been a constant source of strength. Thank you for everything; this achievement is as much yours as it is mine.

I also extend my deepest gratitude to Cursor, Google AI Studio, and ChatGPT for their invaluable contributions to our project. The project wouldn’t have been possible without their help.

Table of Contents

Table of Figures

List of Abbreviation

API Application Programming Interface
AST Abstract Syntax Tree
CA Certificate Authority
CPU Central Processing Unit
DDG Data Dependency Graph
DI Dependency Injection
gRPC gRPC Remote Procedure Call
HTTP Hypertext Transfer Protocol
I/O Input/Output
IP Internet Protocol
JWT JSON Web Token
KVM Kernel-based Virtual Machine
mTLS mutual Transport Layer Security
OS Operating System
P2P Peer-to-Peer
PID Process ID
Protobuf Protocol Buffers
QEMU Quick EMUlator
RAM Random-Access Memory
REST Representational State Transfer
RPC Remote Procedure Call
SMB Server Message Block
SSH Secure Shell
TCP Transmission Control Protocol
TLS Transport Layer Security
VM Virtual Machine

Contacts

Team Members

Name	Email	Phone Number
Abdelrahman Mohamed Mahmoud	abdulrahman.fattah02@eng-st.cu.edu.eg	+2 01145385389

Supervisor

Name	Email	Number
Lydia Wahid	LydiaWahid@outlook.com	-

This page is left intentionally empty

1 Introduction

This part of the report focuses on the **Communication and Execution Core**, the foundational components of the RunSurge distributed system. Specifically, it introduces and explains the two primary modules that enable the network's functionality: the **Master Controller**, which serves as the central orchestrator, and the **Worker Node**, which provides the secure execution environment. This section details how these modules communicate over a high-performance **gRPC** framework, ensuring efficient and reliable data transfer. Furthermore, it explains the architecture of the Worker Node, which leverages sandboxed **Virtual Machines (VMs)** to securely execute user-submitted code, guaranteeing isolation and protecting the host system's integrity. These components collectively enable the transformation of scheduled tasks into securely executed results across a distributed network of peer-assisted nodes.

1.1 Motivation and Justification

Building a robust and secure hybrid peer-to-peer computing network presents key challenges in node management, task execution, and data communication. To address this, the RunSurge system introduces three core architectural components:

- Solves the Discovery Problem: Pure P2P discovery can be complex and slow. A central Master makes finding other nodes and resources extremely fast and efficient.
- Enables secure python code execution by executing the code inside a **VM.**
- Scales Data Transfer: By enabling direct peer-to-peer data transfer, the Master avoids becoming a bottleneck for large data payloads. The bandwidth and processing for data transfer are distributed among the peers, which is a core P2P advantage.

1.2 Document Organization

This report is organized into five main chapters, followed by several appendices that provide supporting information and documentation.

Chapter 2 details the design of the system: including its architecture, key modules, and design constraints.

Chapter 3 explains the testing process: covering both unit and integration testing along with evaluation results.

Chapter 4 summarizes the outcomes: discusses limitations, and suggests future improvements.

2 System Design and Architecture

This chapter details the design of the Communication and Execution Core, the engine that drives the RunSurge distributed network. We will cover how the central Master Controller, the distributed Worker Nodes, and their sandboxed Virtual Machine (VM) environments all work together. The architecture is built on gRPC, a modern communication framework chosen for its speed and reliability in handling everything from simple status updates to the transfer of large data files between nodes. This section will walk through the entire lifecycle of a task to show how the system coordinates work, ensures code is run securely, and manages the network to enable scalable, peer-assisted computing.

2.1 Block Diagram

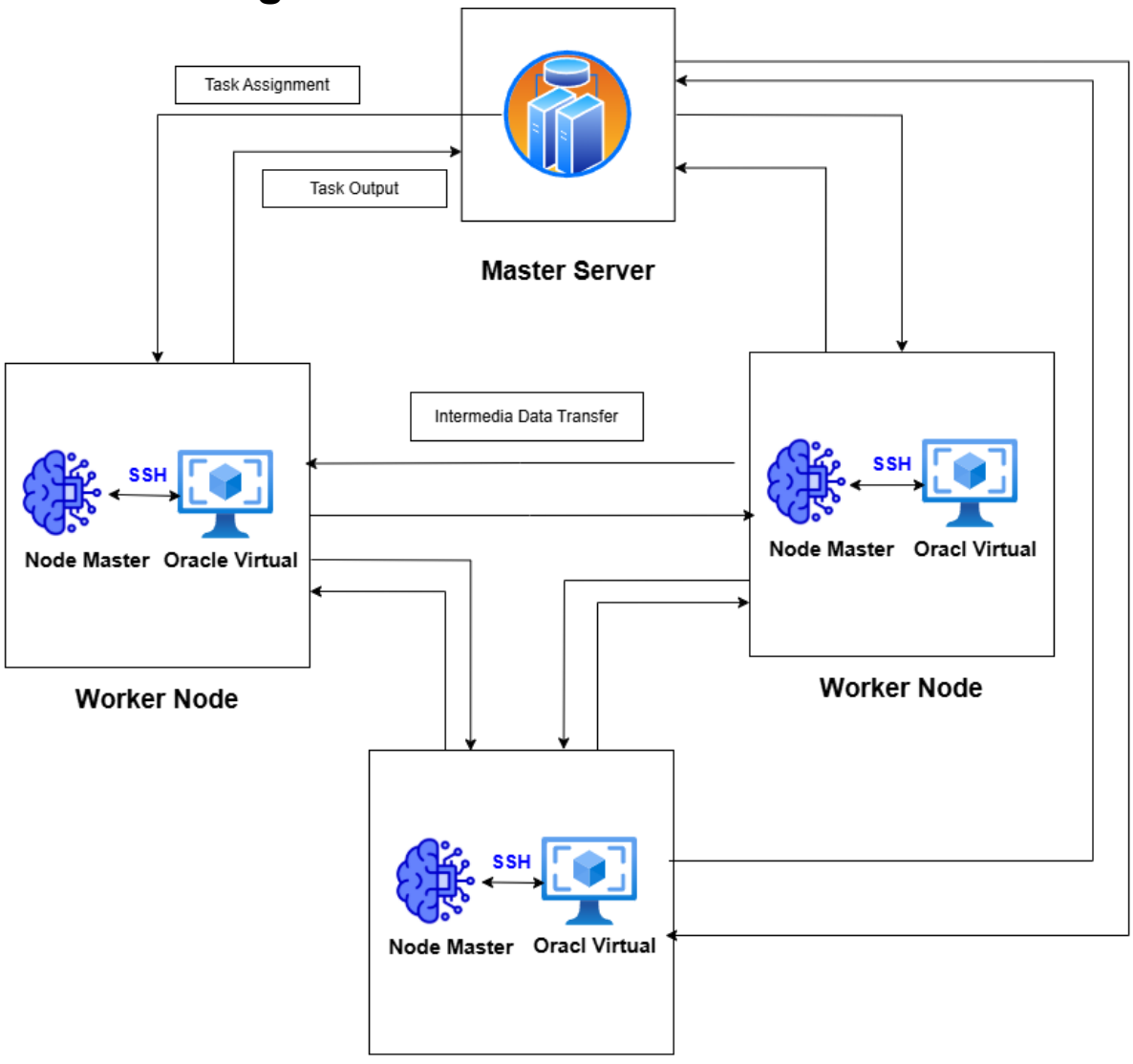

Figure 2.1: Block Diagram

2.2 Modular Decomposition

The RunSurge system is designed as a distributed computing framework that enables users to execute Python code by leveraging the idle computational resources of other users on the network. At its core, the system functions by breaking down a user's job into executable tasks, scheduling these tasks on available Worker Nodes, coordinating the transfer of necessary data, and securely executing the code in isolated environments. The entire process is orchestrated by a central Master Controller that serves as the single source of truth and coordination for the network. The architecture is composed of three primary, interacting modules: the **Master Controller**, the **Worker Node**, and the **Parallelization Module**. We will be talking in depth about Master Controller and Worker Node.

2.2.1 Master Controller: The Central Orchestrator

The Master Controller is the brain of the RunSurge network, responsible for managing the entire lifecycle of a job from submission to completion. Its functionality is exposed through two distinct server interfaces operating concurrently to serve different clients: a FastAPI (HTTP) server for user interaction and a gRPC server for internal network communication.

Functionally, the Master's responsibilities can be broken down as follows:

- User and Worker Management:

  The Master handles the registration, authentication, and lifecycle of both users and **Worker Nodes**. When a user logs in via the web interface, the **Master** validates their credentials. Similarly, when a **Worker Node** comes online, it must register with the Master by providing its credentials and machine fingerprint. The Master, in turn, assigns a unique ID to the worker and maintains its status (e.g., active, offline, busy) in a central database. This registry is kept up-to-date through a gRPC-based **NodeHeartbeat** mechanism, where each active worker periodically sends a signal to confirm its liveness and report its current resource load.

- Job and Task Orchestration:
  When a user submits a job, the Master receives the code and associated data files. It then interacts with the Parallelization Module to break the code down into a dependency graph of executable tasks. The Master is responsible for scheduling these individual tasks onto suitable Worker Nodes based on their reported resource availability and the estimated requirements of each task.

- Data Coordination and Location Services:
  A key function of the Master is to act as a directory service for all data within the network. It tracks which node holds which piece of data—whether it's initial input data uploaded by a user, or intermediate data produced by a completed task. When a worker is assigned a task, the Master proactively notifies it of the network locations of all required input data via the NotifyData gRPC call. This prevents workers from having to poll or search for data, streamlining the execution process.

- Serving and Storing Data:
  The Master also functions as the initial repository for user-submitted code and data. It provides gRPC streaming endpoints (StreamPythonFile, StreamData) from which workers can download the necessary files.

2.2.2 Worker Node: The Execution Engine

The Worker Node is the component run by users who wish to contribute their computational resources to the network. Each Worker is an independent agent that registers with the Master, receives tasks, executes them securely, and exchanges data with its peers. Its functionality is managed by the WorkerManager, which orchestrates several key sub-components.

- Task Execution and VM Management:
  The primary function of a worker is to execute tasks. To ensure security, all user code is run inside an isolated Virtual Machine (VM) managed by the VMTaskExecutor. This executor is responsible for starting, stopping, and interacting with the VM. It uses an SMB (Server Message Block) share to create a bridge between the host machine and the guest VM, allowing for the seamless transfer of script and data files into the sandboxed environment. The TaskProcessor module within the worker handles the complete lifecycle of an assigned task, from fetching dependencies to invoking the script inside the VM and monitoring its execution.

- Data Fetching and Caching:
  Upon receiving a task assignment and data location notifications from the Master, the worker's DataFetcher (or MasterClient in some contexts) initiates the data download process. It establishes direct gRPC connections to peer workers or the Master to stream the required files. These files are stored in the **task** directory to be used by the python script. Downloading the data directly from nodes increases the efficiency and throughput of the whole job.

- Communication and Status Reporting:
  The Worker runs its own gRPC server (WorkerServicer) to listen for incoming requests from the Master or peer nodes. It responds to status checks (GetWorkerStatus), accepts task assignments (AssignTask), and serves its own data to other workers (StreamData). It also proactively communicates with the Master to send heartbeats and report when a task is completed successfully or has failed.

2.3 Overall System Interaction Flow

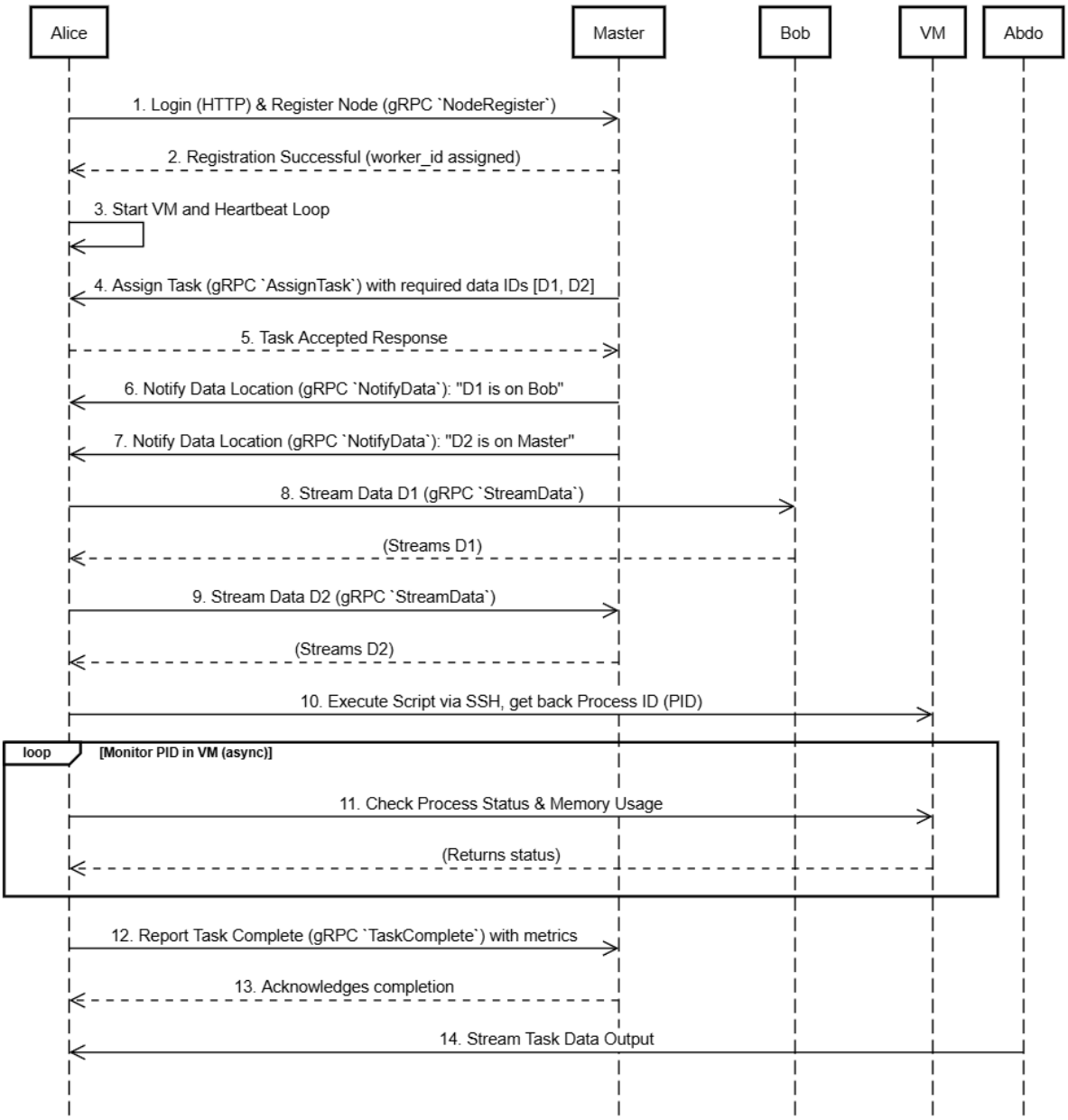

Figure 2.2: Sequence Diagram

The execution of a distributed job in the RunSurge network follows a well-defined sequence of interactions between the Master Controller and the Worker Nodes. The diagram above illustrates the complete lifecycle of a single task, which is detailed in the following steps:

2.3.1 Worker Login and Registration (Alice -> Master)

The process begins when a user starts the Worker Node application (Alice) to join the network. The first action is to establish a secure identity and register its presence. The worker initiates a two-part registration: first, it communicates with the Master's web-facing HTTP API to perform a standard user login with a username and password. Upon successful authentication, it receives a short-lived access token (e.g., a JWT). Immediately following, it uses this token to authenticate with the Master's gRPC server, invoking the NodeRegister RPC. This call transmits the worker's machine fingerprint and resource capabilities (memory, CPU), formally announcing its availability to the distributed network.

2.3.2 Registration Acknowledgment (Master --> Alice)

After the Master receives the NodeRegister request, it validates the provided access token and machine fingerprint. It checks if the worker is already registered or if there are any conflicts. If the registration is valid, the Master creates a new entry in its database for the worker, assigns it a unique, system-wide worker_id, and records its initial resource profile. This worker_id is then sent back to the worker in the gRPC response. This ID will be used in all subsequent communications to identify the worker.

2.3.3 Local Initialization (Alice -> Alice)

Upon receiving its unique worker_id, the worker finalizes its local setup. The most critical step is starting its sandboxed Virtual Machine (VM) environment, which will be used for all task executions. Concurrently, it kicks off a background asyncio task to handle the heartbeat loop. This loop will be responsible for periodically sending NodeHeartbeat messages to the Master, ensuring the Master knows the worker is still alive and responsive.

A key part of this initialization involves establishing a data bridge between the host worker machine and the guest VM. To achieve this, the worker programmatically configures a local directory on the host as a **Server Message Block (SMB)** network share. This process is automated via scripts to ensure consistency. Once the VM has fully booted, the worker establishes an SSH connection to it and issues a command to mount this SMB share within the VM's filesystem. This makes the host's shared folder appear as a native directory inside the sandboxed guest environment, providing a secure and transparent channel for transferring Python scripts and data files required for task execution.

2.3.4 Task Assignment (Master -> Alice)

When the Master's scheduler determines that a task is ready to be executed and that Alice is a suitable candidate (e.g., it is idle and has sufficient resources), the Master initiates the assignment. It sends an AssignTask gRPC command to Alice. This command is designed to be lightweight and efficient; it does not contain the actual code or data. Instead, it includes the unique task_id and a list of identifiers for the required Python script and input data files (e.g., D1, D2).

2.3.5 Task Acceptance (Alice --> Master)

Alice's gRPC server receives the AssignTask request. Its WorkerManager performs a quick, non-blocking check to confirm it can accept the job (e.g., it is in an IDLE state and has available resource slots). If it can, it immediately sends a success response back to the Master. This immediate acknowledgment is crucial, as it allows the Master to quickly move on to scheduling other tasks without waiting for Alice to complete the entire job. The actual processing of the accepted task is offloaded to a background thread or async task on the worker.

2.3.6 6 & 7. Data Location Notification (Master -> Alice)

After assigning the task, the Master takes on the role of a directory service. It looks up the network locations of all data dependencies for the assigned task. For each required data item, the Master sends a proactive NotifyData gRPC message to Alice. For example, it might send one notification stating that data D1 is located on peer worker "Bob" at a specific address, and a second notification indicating that data D2 resides on the Master itself. This "push" model eliminates the need for the worker to poll the Master, streamlining the data acquisition process.

2.3.8 Peer-to-Peer Data Fetching (Alice -> Bob)

Once Alice's TaskProcessor receives a notification for a data item located on a peer (Bob), it initiates a direct, **peer-to-peer** data transfer. It establishes a new gRPC connection to Bob's WorkerService and invokes its **StreamData** RPC. Bob then streams the requested data (D1) in chunks directly to Alice. This decentralized transfer prevents the Master from becoming a data bottleneck and is a key feature of the system's scalability.

2.3.9 Master-Sourced Data Fetching (Alice -> Master)

Similarly, when Alice receives a notification for data located on the Master (D2), it establishes a gRPC connection to the Master's MasterService and calls its StreamData RPC. The Master streams the data in chunks to Alice. At the end of steps 8 and 9, Alice has downloaded all necessary files into a local shared folder that is accessible by its VM.

2.3.10 Sandboxed Execution (Alice -> VM)

With all code and data dependencies in place, the worker instructs its VM to begin execution. It uses an SSH connection to send a command to the VM, typically python3 /path/to/script.py, where the path points to the script in the shared folder. The command is executed in the background within the VM, which returns the Process ID (PID) of the new script process back to the worker host.

2.3.11 Process Monitoring (Alice -> VM Loop)

The worker enters an asynchronous monitoring loop to track the health and progress of the script running inside the VM. Periodically, it sends SSH commands to the VM to check if the process with the given PID is still active and to read its current memory usage from the /proc filesystem. This data is collected to calculate average resource consumption for the task, which is valuable information for the Master's scheduler. The loop terminates when the process is no longer found, indicating it has either completed or crashed.

2.3.12 Reporting Task Completion (Alice -> Master)

Once the script execution is finished, the worker sends a TaskComplete RPC to the Master. This message signals the end of the task and includes a summary of its execution, including important metrics like the total run time and the average memory

used. This data helps the Master refine its resource estimates for future tasks and is used for billing and user earnings calculations.

2.3.13 Master Acknowledgment (Master --> Alice)

The Master receives the TaskComplete message, processes the metrics, and updates the status of the task and the overall job in its database. It then sends a simple acknowledgment back to Alice, confirming that the completion has been recorded. Alice is now free to clean up the task's resources and return to an IDLE state, ready for a new assignment.

2.3.14 Serving Output Data (Abdo -> Alice)

The output data (D3) produced by the completed task now resides on Alice. The Master has updated its data location directory to reflect this. When another worker (Abdo) is assigned a subsequent task that depends on D3, the Master will notify Abdo of Alice's address. Abdo will then initiate a direct peer-to-peer StreamData request to Alice to fetch the data, continuing the distributed data-flow cycle of the network.

2.4 gRPC Communication

The choice of communication protocol is a cornerstone of any distributed system's architecture, directly impacting its performance, reliability, and scalability. For the RunSurge network, which requires frequent, low-latency communication between a central Master and numerous Worker Nodes, gRPC was selected as the exclusive framework for all internal, server-to-server interactions. gRPC, an open-source Remote Procedure Call (RPC) framework developed by Google, provides a robust and high-performance alternative to traditional RESTful APIs over HTTP/1.1.

This section details the key technical advantages of gRPC that made it the ideal choice for our system, including its use of HTTP/2 for advanced network features, its efficient binary serialization format, its support for complex streaming patterns, and its built-in provisions for robust security.

2.4.1 Performance Advantages over Traditional HTTP/REST

While REST APIs using JSON over HTTP/1.1 are ubiquitous and easy to implement, they present significant performance limitations in a high-throughput, distributed

environment like RunSurge. gRPC addresses these limitations directly through its modern technology stack.
HTTP/2 Multiplexing: Unlike HTTP/1.1, which often requires multiple TCP connections to handle concurrent requests or suffers from head-of-line blocking, gRPC is built on HTTP/2. This allows for multiplexing—sending multiple independent RPC calls (requests and responses) concurrently over a single TCP connection. For RunSurge, this is a critical advantage. A worker can be streaming data for one task, sending a heartbeat for another, and receiving a new task assignment, all without the overhead of establishing new connections for each interaction. This dramatically reduces network latency and improves overall system efficiency.

Binary Serialization with Protocol Buffers: Traditional REST APIs typically use JSON, a text-based format that is human-readable but computationally expensive to parse and verbose in size. gRPC, by contrast, uses Protocol Buffers (Protobuf) as its default serialization format. Protobuf messages are defined in .proto files and are serialized into a compact binary representation. This binary format is significantly smaller on the wire and much faster for machines to parse than JSON, resulting in lower bandwidth consumption and reduced CPU load on both the Master and the Worker nodes. For a system that frequently communicates status and transfers data, this efficiency is paramount.

2.4.2 Advanced Streaming Capabilities

One of the most powerful features of gRPC, and a primary reason for its selection, is its native support for advanced streaming. While traditional HTTP APIs struggle with streaming, gRPC makes it a first-class feature.

- **Bidirectional Streaming**: gRPC supports four streaming modes, including client-streaming, server-streaming, and **full-duplex bidirectional streaming**. This allows for complex, long-lived conversations between services. In RunSurge, we leverage this for efficient data transfer. For example:

    - The Master uses server-streaming (StreamData RPC) to send large data or script files to a worker in manageable chunks, preventing either side from having to load the entire file into memory.

    - The worker uses client-streaming (ReceiveData RPC) to upload a large output file to another node. The client sends a stream of data chunks, and the server responds with a single acknowledgment once the entire file is received.

This capability is essential for handling the large datasets and code files common in computational jobs, making data transfer a non-blocking and memory-efficient operation.

2.4.3 Strongly-Typed Contracts and API Evolution

gRPC enforces a "contract-first" approach to API design. The services and message structures are defined in .proto files, which act as a single source of truth for the entire system's communication API. From these files, gRPC's tooling automatically generates client stubs and server skeletons in Python.

This provides several key benefits:

- Reduced Integration Errors: Since both the Master and Worker repositories generate their communication code from the same .proto files, it virtually eliminates runtime errors caused by mismatched data types, missing fields, or incorrect endpoint names.

- Clear and Self-Documenting API: The .proto files serve as clear, language-agnostic documentation for the API.
- Robust API Evolution: Protocol Buffers are designed for forward and backward compatibility. New fields can be added to messages without breaking older clients or servers, which is crucial for maintaining a distributed system where not all nodes may be updated simultaneously.

2.4.4 Integrated Security Model: mTLS and Token-Based Authentication

Securing a distributed network is of utmost importance. gRPC provides a robust, integrated security model that is simpler to implement than manually securing traditional HTTP APIs.

- Transport Layer Security with mTLS: gRPC is built on top of TLS, and it natively supports mutual TLS (mTLS) for encrypting all traffic and authenticating both the client and the server. In our architecture, this means we can configure the Master and each Worker with their own TLS certificates, potentially issued by a shared, private Certificate Authority (CA). When a worker connects to the master, not only does the worker verify the master's identity, but the master also verifies the worker's identity based on its certificate. This creates a highly

secure, zero-trust communication channel where all participants are authenticated.

- Per-RPC Authentication with Tokens: In addition to transport-level security, gRPC allows for per-call authentication credentials, which we leverage for user-level authorization. When a worker registers or sends updates, it includes a JSON Web Token (JWT) in the metadata of the gRPC request. We implemented a gRPC interceptor on the Master server that intercepts every incoming RPC call. This interceptor inspects the JWT, validates its signature, and extracts the user and worker identity from its claims. It can then check if that user has the necessary permissions to perform the requested action. If the token is invalid, expired, or lacks the required permissions, the interceptor can immediately reject the request with an UNAUTHENTICATED status code before it ever reaches the application logic. This provides a clean and centralized way to enforce access control across the entire gRPC API.

2.5 Worker Node

The Worker Node is the operational workhorse of the RunSurge network, responsible for executing user-submitted tasks in a secure and isolated environment. Each worker is an independent application run by a contributing user, designed to manage local resources, process assigned tasks, and communicate its status back to the Master Controller. Its architecture prioritizes security, performance, and reliability through a combination of virtualized sandboxing, efficient data handling, and robust task lifecycle management. To achieve this, all user code is executed within a sandboxed Virtual Machine (VM), which provides strong isolation from the host operating system. Data and scripts are transferred into this secure environment using the industry-standard Server Message Block (SMB) protocol, ensuring a reliable and high-performance channel for file sharing between the host and the VM.

2.5.1 Task Execution Workflow

The primary function of a worker is to process tasks assigned to it by the Master. This process follows a well-defined, asynchronous workflow orchestrated by the worker's central WorkerManager class to ensure the node remains responsive while handling long-running computational jobs.

2.5.1.1 Task Assignment:

The workflow begins when the worker's gRPC server receives an AssignTask request from the Master. The request is immediately acknowledged, and the WorkerManager offloads the task to a dedicated TaskProcessor.

2.5.1.2 Spawning a Task-Specific Thread:

To prevent blocking the main gRPC server thread, the TaskProcessor spawns a new, dedicated thread for each assigned task. This allows the worker to handle multiple tasks concurrently (up to its configured limit) and remain responsive to other network requests, such as heartbeats or data requests from peer nodes.

2.5.1.3 Data Acquisition:

Within this new thread, the TaskProcessor first acquires all necessary dependencies. It uses the MasterClient to stream the required Python script from the Master and any input data from its designated location, which could be the Master or a peer worker. All downloaded files are stored in a local shared directory that is accessible by the worker's Virtual Machine.

2.5.1.4 Sandboxed Execution:

Once all dependencies are present, the task thread invokes the VMTaskExecutor to run the Python script inside the secure VM environment. The script is executed in the background within the VM, and its Process ID (PID) is returned to the task thread for monitoring.

2.5.1.5 Task Monitoring and Completion:

The task thread then enters a monitoring loop, continuously checking the status of the process inside the VM. After the process finishes, the worker reports the task's completion, along with performance metrics, to the Master.

2.5.2 VM Management for Secure Sandboxing

Executing untrusted code from various users poses a significant security risk to the host machine. To mitigate this, RunSurge enforces strong isolation by executing every task within a dedicated Virtual Machine (VM). The VMTaskExecutor module is responsible for the complete lifecycle of this sandboxed environment.

- **Initial Approach with QEMU:**
  Our initial implementation utilized **QEMU**, a powerful open-source machine emulator. QEMU was chosen for its flexibility and excellent command-line automation capabilities. However, performance testing revealed significant drawbacks in our environment. When running on Windows hosts without specific hardware acceleration drivers like KVM (which is Linux-specific. This resulted in slow VM startup times and noticeable I/O performance degradation, which would unacceptably increase the overhead for every task executed on the network.

- **Migration to Oracle VirtualBox for Performance:**
  To address these performance issues, we migrated the VMTaskExecutor to use **Oracle VirtualBox**. As a Type 2 hypervisor, VirtualBox directly leverages hardware virtualization extensions (Intel VT-x and AMD-V) available on modern CPUs. This transition yielded a dramatic improvement in performance. VM boot times were reduced significantly, and the execution speed of code within the VM was near-native. VirtualBox also provides a comprehensive command-line interface (VBoxManage).

2.5.3 High-Performance Data Transfer with SMB

A crucial component of the execution pipeline is the mechanism for transferring files (scripts and data) from the host worker machine into the isolated guest VM. While VirtualBox offers a built-in "Shared Folders" feature, we opted for a more robust and performant industry-standard protocol: **Server Message Block (SMB)**.

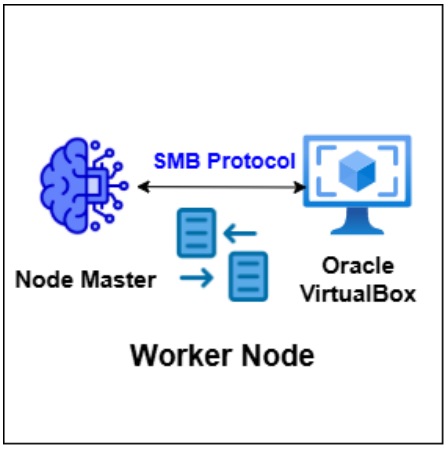

Figure 2.3: SMB protocol

The VMTaskExecutor programmatically configures a local directory on the host as an SMB share. The guest OS within the VM then mounts this network share. This

approach was chosen over the default VirtualBox file sharing for several key advantages:

- **Robustness and Reliability:** SMB is a mature, battle-tested network protocol designed for reliable file sharing. It handles network state and data integrity more effectively than proprietary guest addition drivers, leading to fewer transfer errors and more consistent performance.
- **Higher Throughput:** In many environments, especially with larger files, a native SMB connection provides higher data transfer speeds compared to the VirtualBox Guest Additions shared folder mechanism, which can sometimes be a performance bottleneck.
- **Generality:** SMB is a universal standard. This design makes our VM setup more portable and less dependent on specific hypervisor guest tools, allowing for potential future support of other hypervisors (like Hyper-V or VMware) with minimal changes to the data transfer logic.

2.5.4 Task Monitoring

Once a script is executing inside the VM, it is vital to monitor its behavior to ensure it runs within its expected bounds and to gather performance metrics. The dedicated thread spawned for each task on the Worker Node is responsible for this close-up monitoring. This process provides real-time insights and ensures that the system can react appropriately to both successful and failed executions.

The execution itself is initiated via an SSH command that is carefully constructed to run the user's script in the background and capture its outputs:
Generated bash

Figure 2.4: Asynchronous Task Execution Command

This command has several key parts:

- nohup: Ensures the process continues running even if the SSH session is terminated, making it resilient.

- python3 ...: The actual execution of the user's script from its location in the shared directory.
- 1> log.out 2> error.err: This redirects the standard output (stdout) to log.out and the standard error (stderr) to error.err within the task's directory. This is crucial for post-execution analysis.
- & echo $!: This runs the command in the background (&) and immediately prints (echo) the Process ID ($!) of the newly created background process. The Worker captures this PID to begin monitoring.

With the process running, the monitoring thread performs two main functions:

- **Resource Monitoring:** The thread periodically establishes an SSH connection into the running VM and queries the /proc filesystem to inspect the process using its captured PID. It specifically tracks the VmSize field from /proc/{PID}/status to get a real-time measurement of the task's memory usage. This data is aggregated throughout the task's lifetime to calculate the average memory consumption, a critical metric that is reported back to the Master for billing and for refining future scheduling decisions.

- **Error and Completion Checking**: The monitoring loop continuously checks for the existence of the process PID. If the PID disappears from the process list, it signals that the process has terminated. At this point, the monitoring thread performs a final check by reading the contents of the error.err file via **SSH**. The presence of any content in this file indicates that the user's script raised an unhandled exception or wrote to standard error, signifying a failed task. This failure is then reported to the Master.

3 System Testing and Verification

This chapter outlines the process undertaken to validate the correctness and functionality of the developed RunSurge system. It describes the testing setup and the multi-layered strategy used to verify each module individually before performing full integration testing to ensure reliable, end-to-end behavior.

3.1 Testing Setup

To simulate a realistic distributed environment, the testing setup involved deploying the Master and multiple Worker nodes on separate machines within a local area network. This required configuring static IP addresses and firewall rules to ensure all nodes could discover and communicate with the Master's gRPC and HTTP servers. A

suite of Python scripts with varying complexities was prepared to serve as realistic test jobs for the system.

3.2 Module Testing

Before full system integration, key components were tested in isolation to validate their core functionality. The Worker Node's VMTaskExecutor was a primary focus. We initiated the VM startup and manually established SSH connections to verify its state. Key tests included:

- **File Sharing and Execution:** We confirmed that the SMB share was correctly mounted inside the VM and that files placed in the host's shared folder were immediately accessible for execution by the guest OS.
- **Process Monitoring:** We ran test scripts inside the VM and used ps and cat /proc/{PID}/status commands via SSH to verify that our monitoring logic for process liveness and memory usage was accurate.

This hands-on, modular testing ensured that the foundational components of the worker, such as secure execution and monitoring, were robust before connecting them to the full network.

3.3 Integration Testing

Once individual modules were validated, we performed end-to-end integration testing. The Master server was started, followed by one or more Worker nodes on the network. We tested the entire job lifecycle:

1. Verified that workers could successfully register with the Master and maintain their active status via heartbeats.
2. Submitted a complete Task (processed by the Parallelization Module) to the Master.
3. Tracked the job as the Master assigned tasks, workers received notifications, and data was correctly streamed between peer nodes and the Master.
4. Confirmed that tasks were executed correctly inside the worker VMs and that completion status and metrics were reported back to the Master.
5.

The final outputs produced by the distributed execution were compared against the results from running the original script sequentially to verify correctness. This end-to-end testing confirmed that all modules functioned cohesively as a single, distributed system.

We tested 4+ nodes on different systems and there were no issues in testing.

asdasdasdasd

4 Conclusions and Future Work

This chapter summarizes the project's outcomes, highlighting the key features of the implemented distributed system, the challenges faced, the experience gained, and promising directions for future enhancements.

4.1 Faced Challenges

The development of the Master-Worker architecture presented several key challenges:

- **Concurrent Server Management:** Implementing and managing two concurrent servers (FastAPI for HTTP and a gRPC server) within a single Master application required careful handling of Python's asyncio to ensure both could run without blocking and share core resources.
- **Virtual Machine Performance:** Our initial choice of QEMU for virtualization proved to be a performance bottleneck. Migrating to Oracle VirtualBox provided the necessary execution speed but required re-engineering the automation for VM control and file sharing.
- **Dependency Injection in a Hybrid System:** Integrating services that relied on FastAPI's Depends system with the gRPC servicer, which lacks a native DI framework, required designing a parallel, manual dependency provisioning pattern using context managers to ensure logic could be reused.

4.2 Gained Experience

This project provided invaluable hands-on experience in building a complex, distributed system from the ground up:

- **Hybrid System Architecture:** Learned to design and implement an application serving multiple protocols (HTTP and gRPC) that share a common core logic and database backend.
- **High-Performance Networking with gRPC:** Acquired expertise in using gRPC for building robust, server-to-server APIs, including defining services with Protocol Buffers, implementing streaming RPCs for large data transfer, and using interceptors for authentication.
- **System-Level Automation and Virtualization:** Developed practical skills in programmatically controlling hypervisors like VirtualBox and automating system-level configurations, such as setting up SMB network shares for secure, sandboxed execution environments.
- **Asynchronous Programming:** Mastered the use of Python's asyncio to build a highly concurrent Master server capable of managing numerous simultaneous connections and I/O-bound operations efficiently.

4.3 Future Work

This project establishes a solid foundation for a peer-assisted computing network. Future work can expand upon its capabilities in the following directions:

- **Enhanced Fault Tolerance:** Implement a more sophisticated mechanism for handling worker failures, where the Master can automatically re-queue and reschedule an interrupted task on another available node to ensure job completion.
- **Support for Dockerized Execution:** As an alternative to full VMs, add support for executing tasks within lightweight Docker containers. This would significantly reduce startup overhead and resource consumption on worker nodes, allowing for faster task turnaround.
- **Full mTLS Security:** Implement a private Certificate Authority (CA) to issue TLS certificates to the Master and all workers, enabling mutual TLS (mTLS) for a zero-trust security posture across all internal gRPC communication.

5 References

[1] Google LLC, "gRPC Documentation," gRPC.io. [Online] Available: https://grpc.io/docs/
[2] Microsoft Corporation, "[MS-SMB]: Server Message Block (SMB) Protocol," Microsoft Open Specifications. [Online]. Available: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-smb/f210069c-7086-4dc2-885e-861d837df688
[3] The QEMU Project, "QEMU User Documentation," qemu.org. [Online]. Available: https://www.qemu.org/docs/master/
[4] Oracle Corporation, "Oracle VM VirtualBox User Manual," virtualbox.org. [Online]. Available: https://www.virtualbox.org/wiki/Documentation