CPSC 416
Capstone Project
A Peer-to-peer Volunteer Computing System for Bitcoin Mining

Poh Leanne Kee
Sebastian Gonzalez Sanchez
Felicia Kwala
Mert Barutcuoglu

Table of Contents

Problem Statement

Volunteer computing (VC) is a form of distributed computing where volunteers can dedicate their spare/idle computing resources to a cause. We would like to define our cause to be in the context of a blockchain miner wanting to verify a cryptocurrency transaction (more details on this in Use Case / Real-World Application).

We would like to design a volunteer computing system utilizing a peer-to-peer network that would fulfill that use case. Users/clients can utilize a service built on our system to distribute their computational workload to peers in the network.

Use Case / Real-World Application

The majority of the public will have access to cheap devices (e.g. phones, laptops) that have limited computing power. We are aiming to democratize Bitcoin mining through our volunteer computing system. Volunteer computing allows us to rely on the gathered computer bandwidth from several small and cheap devices such that it will be comparable to the computer bandwidth of a fast and expensive High-Performance Computing (HPC) gear.

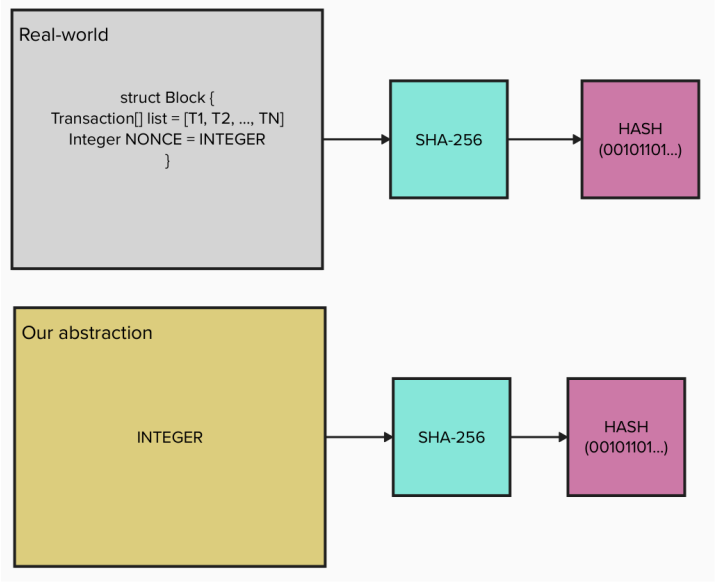
Our Mission: Empower under-resourced miners to compete against the largest actors in the mining space who would otherwise have too much of an unfair advantage for anyone to compete against.

Cryptocurrency Mining

In cryptocurrency, the blockchain represents a ledger that records transaction history. The person who has done the most computational work is the one who gets to register chosen transactions and add a block to the blockchain by finding a nonce that, once added to the miner's tentative block, makes that entire block hash into a particular binary pattern under the SHA-256 algorithm that fulfills a predetermined difficulty level. This is known as the Proof-of-Work (PoW) mechanism. The incentive for verifying the transaction is a small fee that the verifier gets in return.

Crypto mining is the brute-force act of going over a wide range of integers to be used as nonces and trying each and every one of those one by one until one finds a nonce that makes the block's hash match the difficulty level. In practice, this would mean finding a nonce that makes the block hash into a binary pattern with a particular number of trailing zeroes.

Real-World Application



In our design document, we denote the goal of a user as finding some integer that hashes to a binary pattern with a particular number of trailing zeroes. However, in a real-world application, the user would provide a block that consists of a list of transactions and a range of nonce integers to try. For the sake of simplicity, we've abstracted out the block details into a range of integers.

From the perspective of the application, a block structure containing a nonce looks the same as a conventional integer as they are both simply binary patterns of some length that the compiler gives meaning and structure to depending on their data type. Therefore, for the remainder of this design document, the main distributed task is to hash integers instead of hashing Block + Nonce objects to validate the overarching system. We still want to highlight that, if we were to launch this product in the real world, clients, in reality, would have to try to hash blocks alongside ranges of nonces they might want to try as this is the real Bitcoin mining procedure.

Volunteer Incentives

In order to encourage volunteers to contribute their resources, any peer device that has run a portion of the job that returned a successful nonce will get a cut of the crypto reward based on their contribution.

Background

Previous work in the domain includes the Berkeley Open Infrastructure for Network Computing (BOINC) (see References) platform. BOINC utilizes a client-server network model and it differs from our capstone's use case in that it acts as a computational resource for scientific projects. It also requires the client to download input files which it needs to execute the job.

There does not seem to be a large amount of VC services that utilize a peer-to-peer network. However, we did find a paper by Saleh and Shastry — "A new approach for global task scheduling in volunteer computing systems" (see References) that details how a peer-to-peer network model might work for VC. It is similar to how we are distributing our jobs and shares our goal of distributing jobs among nodes as evenly as possible. The key difference is how node discovery among peers is being done and how performance metrics are being measured and how that information is being passed between what they call "super-node" and "sub-node" (which in our case are just parent and child processes).

Relevance

VC is very much a distributed systems problem because it encompasses a pooled resource of devices/servers that are spread all over the world. In a **peer-to-peer network model**, servers can act as clients and clients can act as servers. Although we have been dealing a lot with the client-server network model during the course, there are class concepts that we can apply here, especially with the design of the tracker server. The tracker server serves as a single point of knowledge (and failure!) for the peers, so we have decided to make it a **replicated database** that utilizes a **consensus protocol** to ensure **data consistency** through PAXOS among replicas.

The project also uses a **MapReduce-like** framework to distribute the jobs among peers. This framework is used for the parallel processing of huge data to achieve high performance. Once the data is distributed from the initial server to the peers, each peer will have a map function that produces a key-value pair (Key: [integer_start, integer_end]; Value: Valid Nonce/NULL). If a peer processing an integer range found a valid nonce within that integer it will return to its parent its respective integer range as key and the winning nonce as the value. Otherwise, this value will be NULL to indicate the winning nonce is not found within that range. The key-value pairs will then be returned to the initial server to be reduced into a single final output.

The biggest challenge is **low availability** — peer devices in the network may join or leave the network periodically as well as the high variability in hardware and software configurations of volunteer machines. Other challenges in designing a VC system include task scheduling and load balancing.

There are security concerns such as the injection of malicious code in a network packet between peers (Byzantine fault), but this is not an issue that we will address as it relates more to computer security. Due to the nature of our use case for this VC system, concerns like data privacy and data protection do not apply since there is no data to be transferred and the peer devices are executing pure computational tasks.

Constraints / Assumptions

- All computing devices are homogenous / The heterogeneity of devices is handled for us
- Our system will not tolerate all peer servers being disconnected from the network (ie. our design does not include any core running peer devices)
- We assume there are no malicious actors in our network (no Byzantine fault)

Guarantees to clients

The CAP theorem discussed here applies to the replicas within the Tracker Server system and the peer-to-peer network.

Consistency
Tracker Server
By employing Paxos for our replicated databases in the tracker servers, we establish a model of **strong consistency** within the tracker server system.

Peer-to-peer Network
However, the consistency between peers is **eventual consistency**, ensuring that, eventually, all copies of the distributed data converge to the same value of active IPs. We believe that the system can afford to have the peers have access to the stale list of IP addresses and try the IPs to eventually get an available IP.

Partition Tolerance
Tracker Server
The tracker server system will only tolerate f partitions/failures in a system of 2f+1 nodes before it fails completely. We hope that designing a partition-based tracker server database of active IPs will lead to risk mitigation of network partitions.

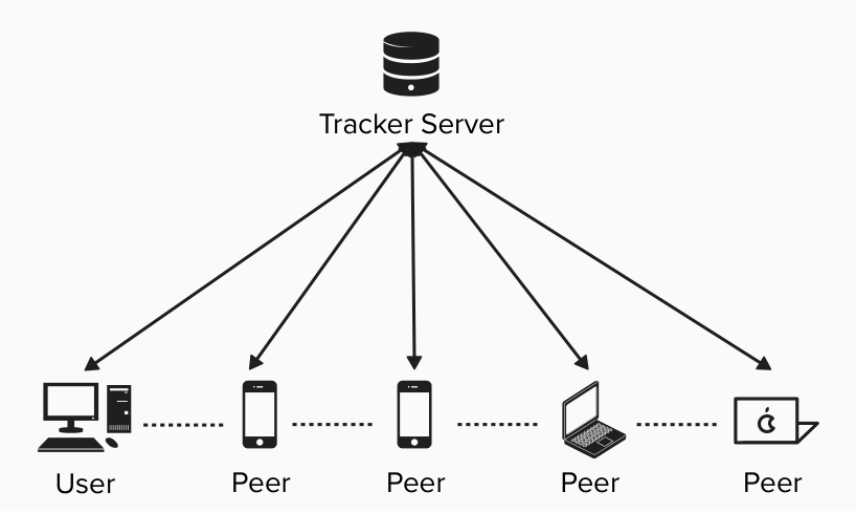
Peer-to-peer Network
If a peer device is completely network partitioned, it will not be able to communicate with the other devices. However, if the peer device is only network partitioned from the Tracker Server and it has a stale list of available IPs, it should still be able to operate with that stale list. If the peer device has just connected to the network and does not have the list yet, it should stay idle until the network partition is resolved.

Availability
Tracker Server

The tracker server system will only tolerate f partitions/failures in a system of 2f+1 nodes before it fails completely. This means it has low availability as it also needs a quorum of nodes to achieve consensus.

Peer-to-peer Network
We choose an eventually consistent model as we want to prioritize the availability of service. We do not need all peers to have the same active IP list at all times. In our workflow, peers who request a list of the active IPs to the tracker server need not have the most up-to-date list of active peers. The client will go over the list of active peers handed by the tracker and will skip any IPs that are non-responsive. Hence, even stale data can be useful as the peer will only need to have enough active peers on that list to submit the required number of subjobs.

High-level Architecture



A volunteer device can be any hardware device that has idle or spare computing resources to contribute. This could be any electronic device with computing power, including a desktop, laptop, or mobile phone. There are 3 components in this diagram:

Tracker Server

The tracker server serves as a single point of knowledge for both clients and peers on the list of all available devices. To ensure fault tolerance, it will be a replicated server that utilizes a consensus mechanism to ensure consistency among replicas. More details here.

Peer/Server

The peer device is responsible for:

- ➢ Sending heartbeat pings to the tracker server and getting a list of available devices
- ➢ Receiving incoming job requests from the client/other peers and distributing the workload to other peer devices if it does not have the capacity to take on the job request fully
- ➢ Executing the job requests that it has assigned to itself
- ➢ Returning the output of the request to the client/peer who requested the job

User

The user sends a request to trigger the client process on any server. The main functionality of the client process is to kickstart the distribution of the jobs to available servers. It is at the top of the job distribution tree (does not have a preceding server) and should not be running the job.

- ➢ Sending heartbeat pings to the tracker server and getting a list of active IP addresses (available devices)
- ➢ Sending job requests to other peer devices
- ➢ Gathering outputs from the peer devices it has sent its job requests to

Anatomy of a Job

in this context consists of having to compute the hash, under some choice of hash algorithm, for every number within a pre-specified range. To parameterize such a job you need to specify three variables: the start of the range, the end of the range, and the hashing algorithm. The choice of this job is made because it is easy to divide this job into disjoint subjobs, which can then be handled by different machines in parallel as the subjobs are not causally connected. This allows us to distribute this job to several computers in the network in a peer-to-peer fashion and make the most out of the available network of idle computational platforms willingly connected to the network.

Under this definition of a job, a subjob X is a subjob of some other subjob Y if X is a subset of Y and they share the same hashing algorithm (e.g. The subjob 10-20 (SHA-256) is a subjob of 10-100 (SHA-256))

The general flow of information for this application starts from a client process which wants to compute the hash values for a range of integers. To achieve this, it partitions this bigger range into smaller ranges, which are then submitted to different active peers in the network. Once that first line of peers has taken all the generated partitions, they will work on that assigned sub-range and potentially partition it even further to continue distributing it.

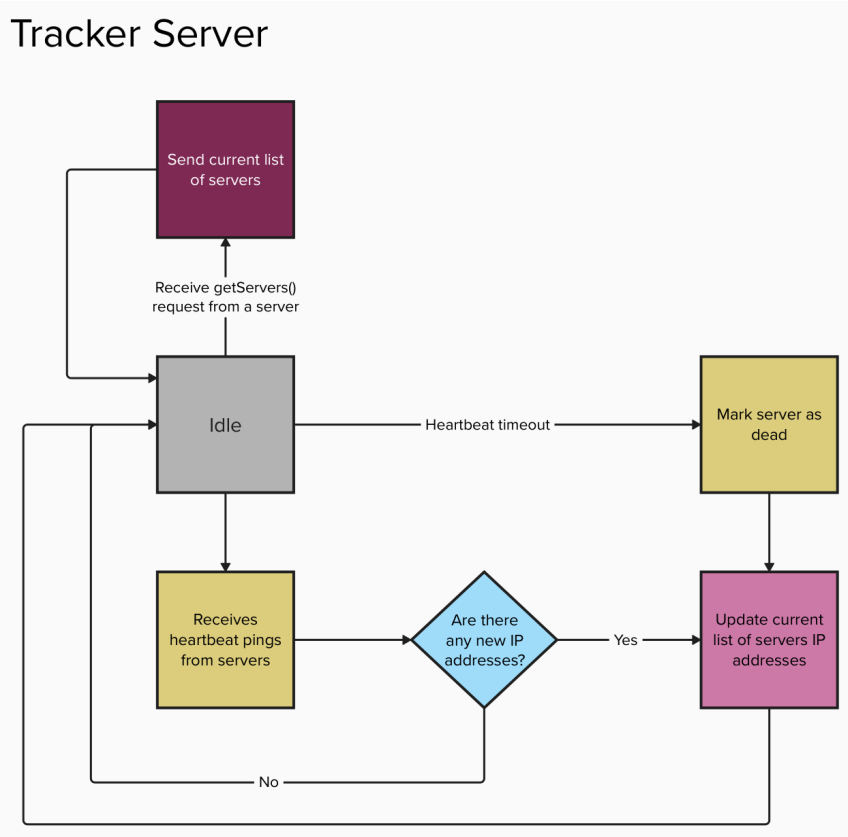
Component Details

Tracker Server

The Tracker Server is the centralized registry in our peer-to-peer computing architecture, responsible for managing the list of active peer devices. It is the only permanent server in our architecture. It receives periodic heartbeat signals from peers to maintain an up-to-date roster of available computational resources, which is essential for job distribution.

Understanding the risks of a single point of failure, our design enhances reliability by employing geo-replication and the Paxos consensus algorithm, enabling fault tolerance that can handle up to f failures in a 2f+1 server configuration. Clients locate the Tracker Server via DNS, ensuring consistent access and network integration. Through DNS, network participants can resolve the Tracker Server's address, ensuring seamless interaction despite its geographical distribution. This server plays a pivotal role in connecting clients with computational resources and upholds the system's availability and consistency across geographically distributed nodes.

Logical Flowchart



Mural Link:
https://app.mural.co/t/cpsc416designprojects6271/m/cpsc416designprojects6271/1697586383217/83c4221983660f301a1b494ac88317ab88919fd3?sender=u285836a032b27745ab421534

Structure of Peer Devices

The application running on the peer devices will be running multiple processes. A single peer device is able to run the following 3 processes simultaneously:

- ➢ **Coordinator**: A process that oversees the subjob process allocation local to its respective peer device.
- ➢ **Client**: Process within a peer that wants to distribute a full job among peers.
- ➢ **Subjob**: Process within a peer running a sub-portion of a larger job submitted by a remote peer.

In the following sections, we will expand on each of these three processes and their purpose.

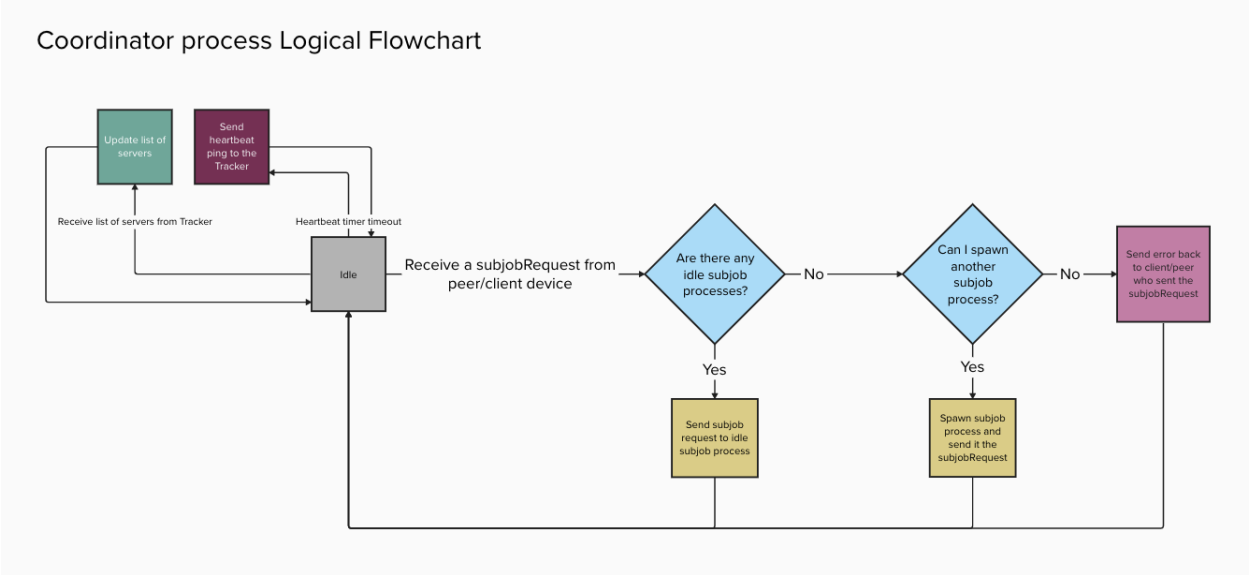
1. Coordinator

When a peer device first installs our volunteer network application in their local device and launches it, it will automatically launch a key process called a Coordinator process. This is the process that is always running on each peer, which is in charge of the following tasks:

1. Ping central tracker server during every heartbeat interval to report itself as an active peer.
2. Regularly request an active IP address list from the central tracker server to maintain an up-to-date list of peers in the network capable of running jobs.
3. When a remote subjob request enters the local peer, the coordinator is in charge of determining whether any idle subjob processes are available or, if possible given current local computational availability, whether it should spawn a new subjob process to handle the incoming load. (See Subjob for more details on the subjob process).

4. When the application is first launched and the coordinator process is instantiated, it is responsible for running an empirical test on the local machine to experimentally determine how many integers a single core can hash under X ms (user-defined variable) running on Y threads (user-defined variable). This number of integers it can hash will be used as the maximum number of integers a given subjob can locally support before further redistributing any remaining integers to other remote peers.

Logical Flowchart



Mural Link:
https://app.mural.co/t/cpsc416designprojects6271/m/cpsc416designprojects6271/1697586383217/83c4221983660f301a1b494ac88317ab88919fd3?sender=u285836a032b27745ab421534

Pseudocode

```
// METHOD WITHIN COORD PROCESS TO HANDLE
// INCOMING SUBJOB REQUEST
// Coordinator process will be listening on some predetermined TCP port

// List of running subjob processes in local peers
Subjob_processes = [SJ1, SJ2, SJ3]

// Subjob example: Compute hash for all integers within [100, 200]
// start_int=100, end_int=200

CoordGetsJob(start_int, end_int)
{
    for SJ in Subjob_processes:
        // Is this subjob process busy?
        if SJ.is_idle():
            SJ.assign_subjob(start_int, end_int)
            return;

    // All existing subjob processes were busy so create a new one
    try:
        new_SJ = create_new_SJ()
        Subjob_processes.append(new_SJ)
        new_SJ.SubJobGetsJob(start_int, end_int)

    Except OutOfMemoryException:
        // Subjob creation will fail of local peer does not have
        // computational capacity for another process
        send(parent, "Sorry, out of capacity. Try again later...")
}
```

2. Client

Terminology Disclaimer: Child = Receiver

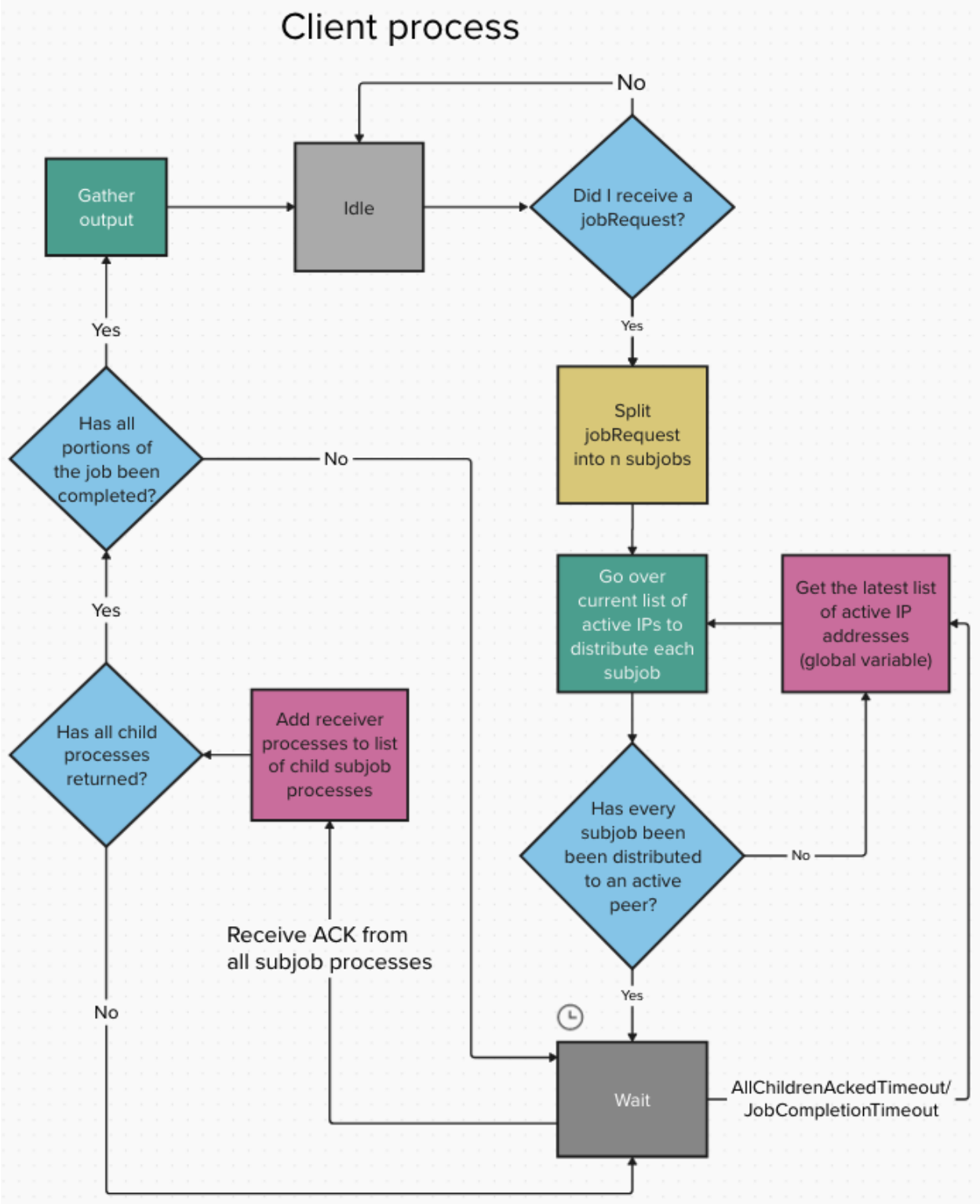
The client process is idle by default and will only be triggered when a user initiates a process (hashing integers) from any peer within the system. The intended client process is outlined below:

1. An end user triggers a request to hashing the integers provided as input from the front end of server X *within the system
2. Server X will split the job into n subjobs, according to the capacity of subjob processes.
3. Server X will also select n number of available servers from its local list of available IP addresses that has been updated by the Coordinator Process.
4. Server X will distribute the subjobs to selected child servers (assuming that Server X is the parent server), according to the Distribution of Jobs section.
5. After the distribution of jobs, Server X will wait for the ACK messages from the selected child servers.
6. After Server X has received the ACK messages from all selected child servers, Server X will then wait for the respective outputs from the selected child servers.
    a. If Server X fails to get an ACK from a child server, Server X will repeat from Step 5 but will only select based on the number of servers that failed to respond instead of n number of servers
7. After Server X has received all the outputs from the selected child servers, Server X will gather and process the outputs to form a single output Y.
    a. If Server X receives a JobCompletionTimeout() from a child server, Server X will repeat from Step 5 but will only select based on the number of servers that failed to respond instead of n number of servers
8. Server X returns output Y as a response to the request initiated by the end-user.

* We have limited the servers that are allowed to make a client process request to be within the system so as to keep the integrity and availability of our volunteer computing system.

The performance could be potentially optimized by modifying Step 8 such that Server X executes the output immediately once the output is received from any selected server, instead of waiting for all the outputs to return. This will significantly save execution time if the correct hash is found in the first few outputs, such that Server X will not have to wait for the remaining outputs to return.

Logical Flowchart



Mural Link:
https://app.mural.co/t/cpsc416designprojects6271/m/cpsc416designprojects6271/1697586383217/83c4221983660f301a1b494ac88317ab88919fd3?sender=u1875938edec1118650585686

Failure Scenarios

What happens when the client process times out?
When the client process times out, it could be due to

1) Timeout from waiting for the child servers' ACKs
2) Timeout from waiting for the outputs to be returned from the child servers.

In both cases, it is impossible for the current server to tell if the messages are dropped or if the child servers are suddenly unavailable.

Hence, our proposed solution is for the current server to go through the most updated list of IP addresses (most likely available as a global variable within the program, shared among processes) and attempt to find an updated available peer to resend the sub-job. This guards against both cases as we are able to avoid the previously used child server in case it is unavailable, and we are resending the sub-job so that the output can be resent. This process repeats until the client process is completed.

What happens when not all portions of the client's job have been assigned?
This only happens when there are not enough available peers in the list of active IPs, resulting in some portion of the client's job not being assigned. Our proposed solution is for the current server to go through the most updated list of IP addresses (most likely available as a global variable within the program, shared among processes) and attempt to find an updated available peer. This process repeats until all portions of the client's job have been assigned.

What happens when most peers are triggering the client processes (ie. wanting to execute their jobs) simultaneously?
This will result in every server executing jobs from another server instead of its own client process. This is the worst-case scenario as this scenario is no different from having each server execute its own client process. Hence, this eliminates the benefit of our volunteer computing system which is to leverage the combined bandwidth of other servers. However, while this scenario is not ideal, the processes will still run to completion, but only at the default performance rate of servers.

3. Subjob

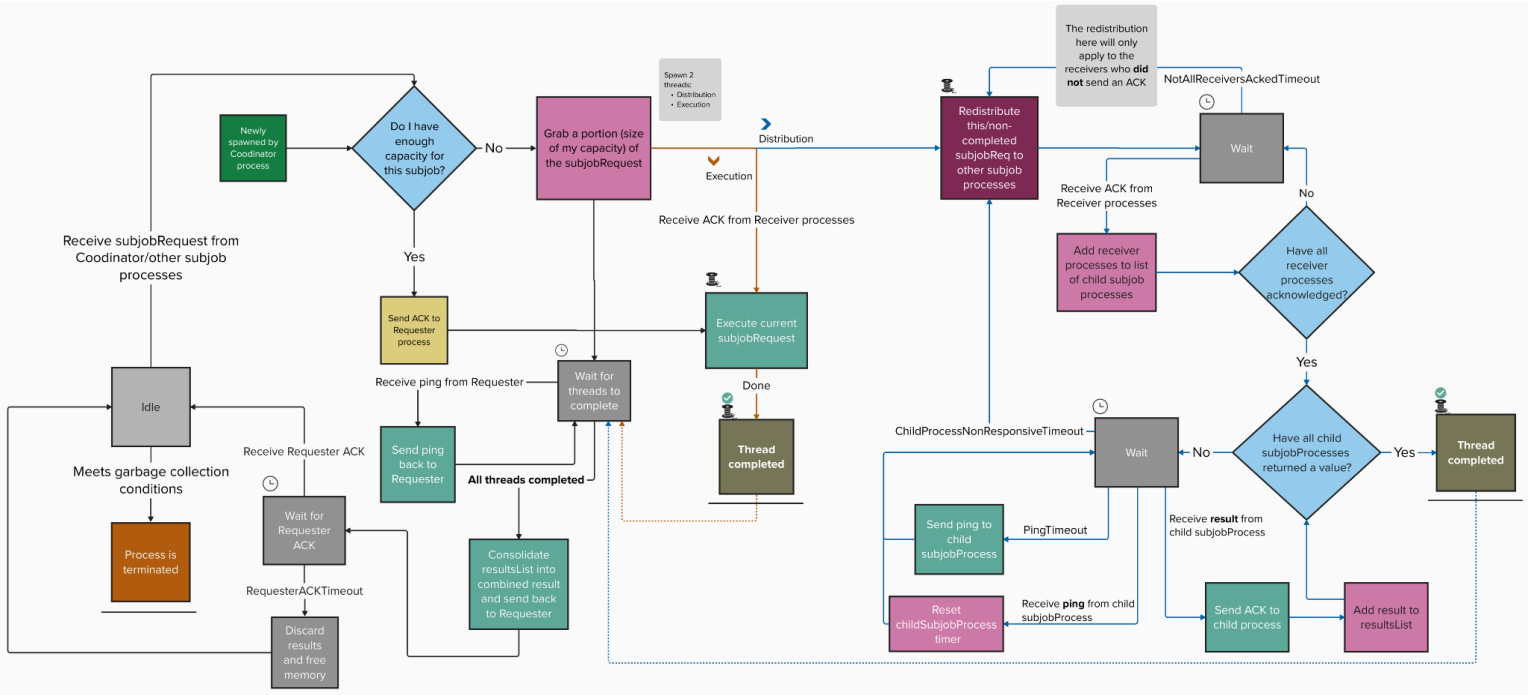
Terminology Disclaimer: Child = Receiver, Parent = Requester

A subjob process is created when the Coordinator process receives a subjob request and the device itself has sufficient resources. A subjob process is responsible for:

1. Distributing/redistributing the subjob request to other subjob processes (on other peer devices)
2. Sending ACK to the Requester (also known as the parent)
3. Executing its own portion of the subjobRequest
4. Receiving and consolidating results from child subjob processes
5. Sending results back to the Requester

The capacity of a subjob process is detailed in [Capacity of Subjob Processes].

Logical Flowchart

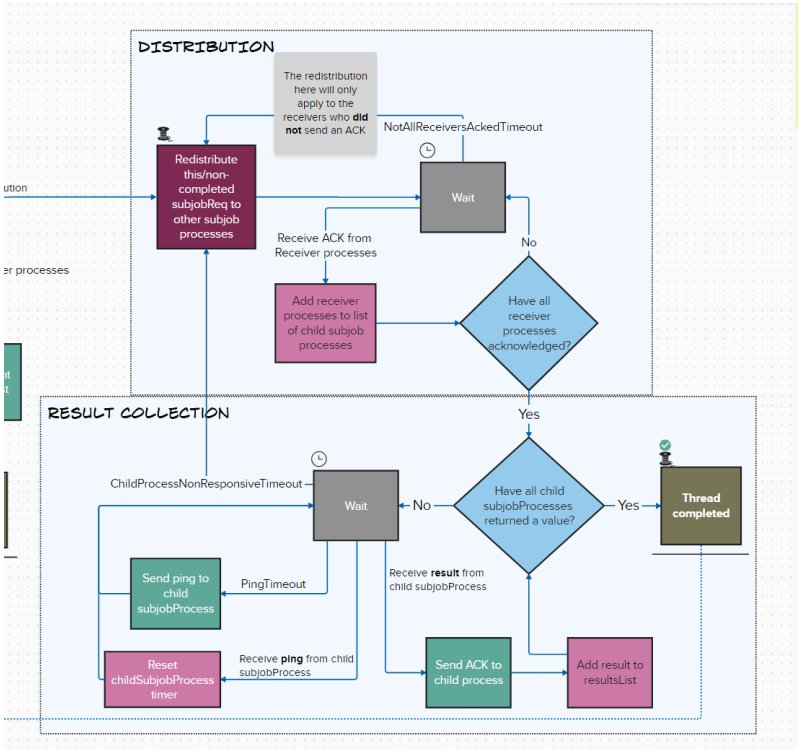


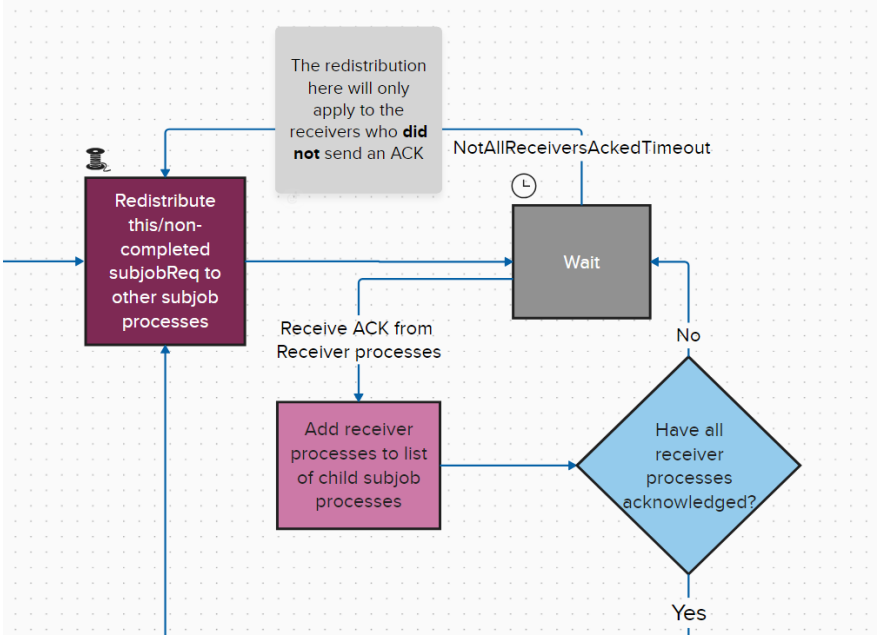
Mural Link:
https://app.mural.co/t/cpsc416designprojects6271/m/cpsc416designprojects6271/1697586383217/83c4221983660f301a1b494ac88317ab88919fd3?sender=u285836a032b27745ab421534

The subjob process should have 2 threads running concurrently: **Distribution** and **Execution**.

Distribution Thread

This thread is responsible for distributing the remaining portions of the subjobRequest and gathering results from child processes.

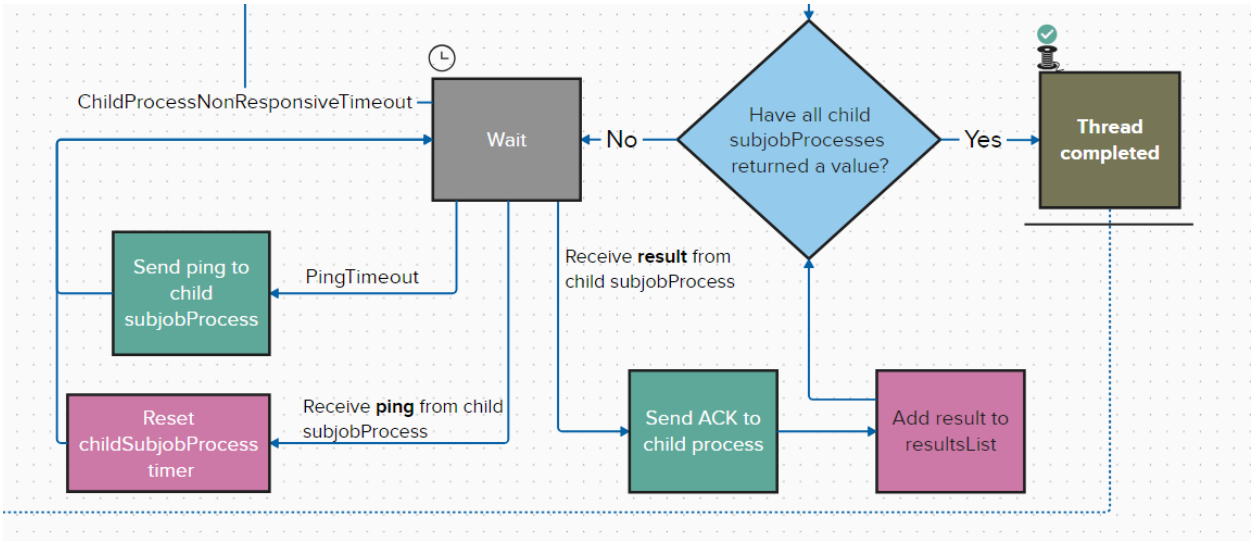




Initial Distribution

The initial distribution consists of distributing portions of the subjobRequest that the current process does not have capacity for. Once it has sent the request, it will wait for the Receiver processes to send an ACK acknowledging that they have received the request. If all receiver processes have acknowledged, then it will move on to the Result Collection phase.

During the Wait phase, this process will resend the Request periodically to receivers who have not ACK-ed. However, if there are receiver processes who did not send an ACK by the time the waiting loop timeout occurs, redistribution will happen.

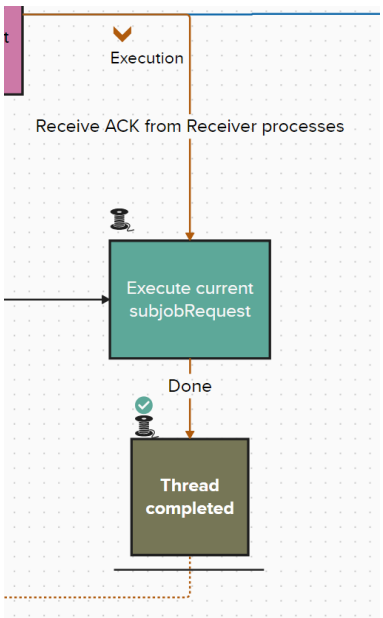


Result Collection / Is My Child Alive?

Once all receiver processes have acknowledged, we move into result collection. In the Wait phase here, it will periodically send a ping to the child process to ensure that it is still alive and working on the subjob. It will also receive results from child processes and add it to a resultsList. If a child process fails to send a ping and the timer runs out, the thread will go back to the distribution phase where it will redistribute the failed/disconnected child's portion of subjobRequest. Once all child processes have returned a value, the thread is complete.

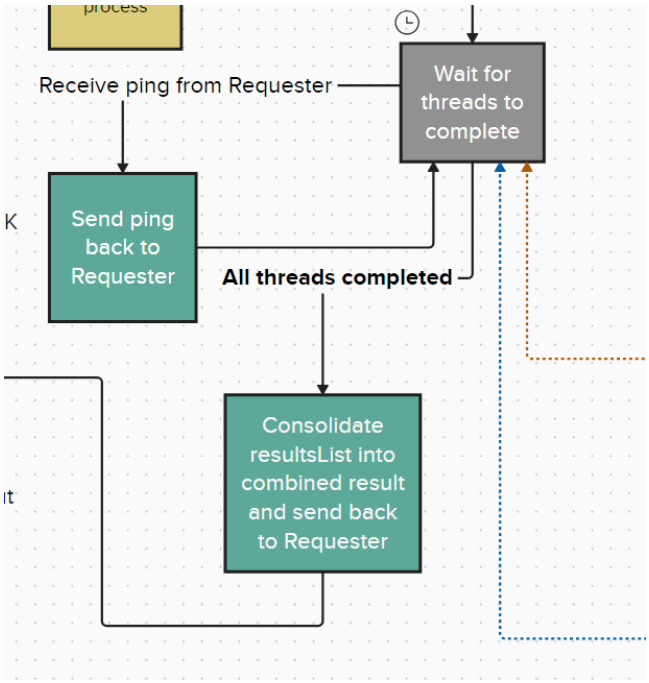
Redistribution

Redistribution will only happen for portions of the subjobRequest that were sent but not acknowledged by the Receiver. In the redistribution phase, the receiver process is presumed dead and this process should direct the subjobRequest to other available peers based on the global list of IP addresses that is updated by the Coordinator process.



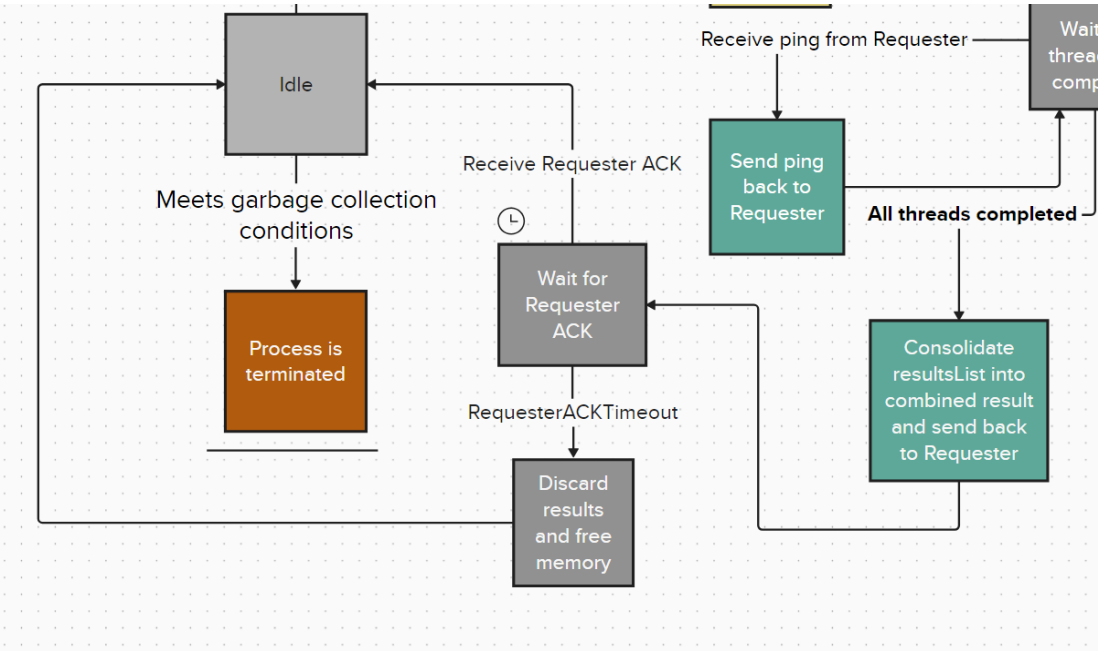
Execution Thread

The execution thread is solely responsible for executing the subjobRequest of the current process.



Main Thread

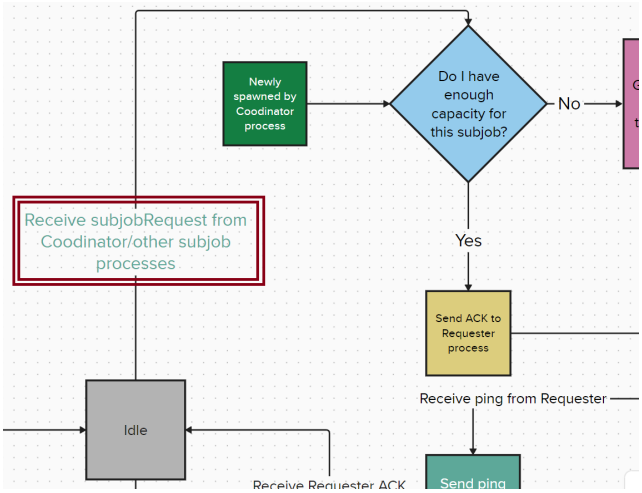
Once the main thread has grabbed its portion of the subjobRequest and spawn 2 new threads — Distribution and Execution, it will enter a Wait phase where it waits for those 2 threads to complete. As it waits, it will also be the one receiving pings from Parent processes and it is responsible to send a ping back to the Parent.



The important thing to note is that the current process will only send the results to the Parent once all child subjobProcesses have finished executing their subjobRequests.

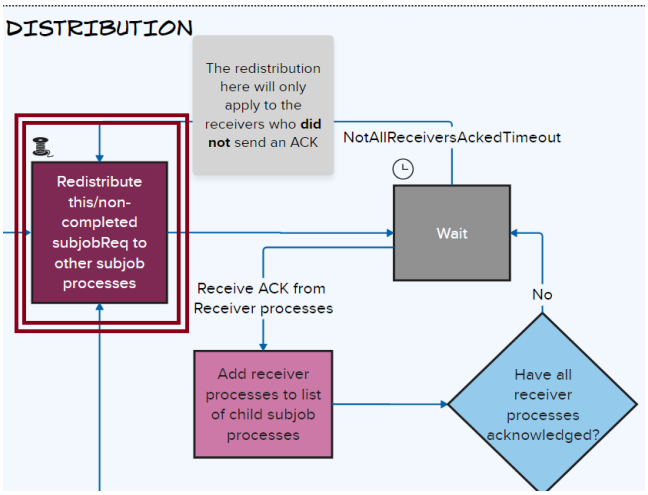
If the Parent/Requester does not send an ACK acknowledging it has received the results, the current process will discard the results and free the memory associated with it.

Pseudocode



The following pseudocode relates to "Receive subjobRequest from Coordinator/other subjob processes".

```
// METHOD WITHIN SUBJOB PROCESS
// This method handles a subjob assigned to the receiving subjob process

// Static maximum number of integers (capacity metric) that
// a Subjob process can take
max_num_ints = 20

SubJobGetsJob(start_int, end_int)
{
    // How many integers within this subjob?
    num_integers = end_int - start_int

    if max_num_ints < num_integers:
        // Start computing local subjob portion
        // on new thread. This thread would also be in charge
        // of monitoring any subjob children it connects to
        // and should wait for all of its redistributed subjobs
        // to finish before recombining all results.
        take_on_subjob(start_int, start_int + max_num_ints)

        // Launch new thread that will redistribute the remaining
        // integers that were not processed by current subjob
        redistribute(start_int + max_num_ints + 1, end_int)

    else:
        // Subjob has enough capacity to compute entire range
        take_on_subjob(start_int, end_int)
}
```



The following pseudocode corresponds to a method in the Distribution thread. More specifically, the "Redistribute this/non-completed subjobReq to other subjob processes".

```
// METHOD WITHIN SUBJOB PROCESS
// This method handles an subjob being redistributed by current subjob to
// other subjobs in remote peers

// Arbitrary number of children chosen to redistribute excess job
NUM_CHILDREN = 10

// Global List of active ip addresses maintained by coordinator process
active_ips = [...]

redistribute(start_int, end_int)
{
    // Split job interval into NUM_CHILDREN partitions
    // and store reference to each child in array
    child_reference = []
    for part_start, part_end in partition_job(start_int, end_int, NUM_CHILDREN)
        has_accepted = False
        peer = null

        while !has_accepted:
            // Try to submit subjob to random active peer
            peer = choose_random(active_ips)
            has_accepted = submit_subjob(peer, part_start, part_end)

            // Store reference to peer that accepted subjob
            child_reference.append(peer)

    // Wait for all peers to return output and recombine result. This method handles
    // child death and potential re-spawning of new children
    wait_and_collect_results(child_reference)
}
```

Failure Scenarios

What happens if the device crashes/is network partitioned?

If a device fails, all processes within that device will not be able to communicate with other processes and vice versa. Here I will only detail this failure scenario from the perspective of a subjob process.
A couple of scenarios play out here:-

- If it is an existing child process of a parent process, the parent process will detect that the child process has failed because it has not received a response from it (see ChildProcessNonResponsiveTimeout in diagram), and it will redistribute the subjob portion of that failed child process to some new child subjob process on a working peer machine.
- If it is a parent process, the child will only find out about its failure when it is waiting for it to acknowledge the result. Once the timer reaches 0 for the parent to send an ACK to the child, the child will discard the result and free the memory.

What happens when the subjob process crashes before it sends the resultsList?

If it crashes before it sends the results, the results will forever be lost and another peer device will have to pick up the same job and essentially re-execute the request.

What happens if the ACKs between child and parent processes get lost?
ACK messages are sent in the two following scenarios:

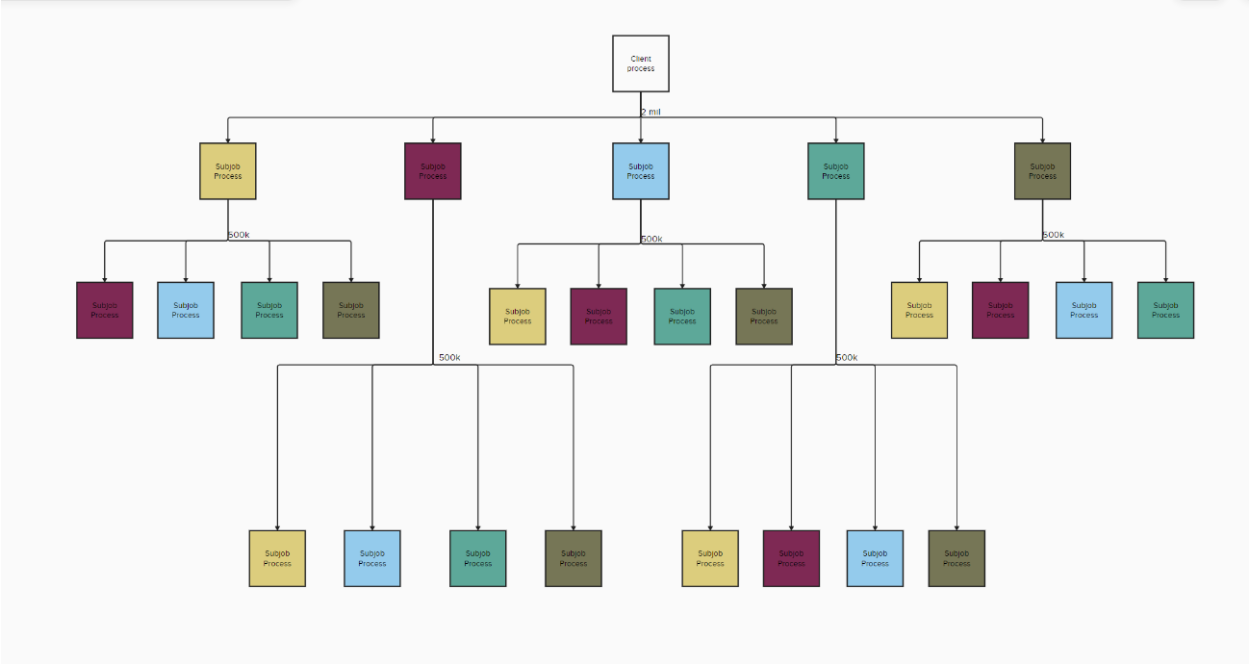
1. When a parent process receives a resultList from the child process, it sends an ACK to the child process indicating it has received it.
2. When a child process takes on a subjob, it sends an ACK to the parent indicating it is taking on the portion of the subjobRequest.

In 1, if the ACK gets lost, the resultList will be discarded.

In 2, if the ACK gets lost, the parent will assume the child is not taking on the Request and is going to assign it to a different subjobProcess. Practically speaking, when heartbeat pings are exchanged between parent and child processes, the jobRequest id should be included in the ping so that the parent can identify if the child should not be executing a Request and send it an error message.

Distribution of Jobs

The efficiency of our Peer-to-peer Volunteer Computing System hinges on how effectively it distributes computational tasks across a dynamic network of volunteer peers.



[Mural Link]

Here's a detailed explanation of the job distribution workflow:

Client Peer Initialization:

- ➢ A server, upon deciding to initiate a client process, will refer to the last requested list of available IP addresses sent from the Tracker Server.
- ➢ This list is crucial as it offers the latest data on peers that are active and can contribute computational power to the mining task.

Distributed Job Allocation:

- ➢ The server holding the client process then iterates over the list of active peers using their IP addresses.
- ➢ It splits the job into smaller sub-jobs, assigning each portion to a different peer until every part of the job is distributed, i.e., the process is repeated or every pear and it is further split. The size of the subjobs is dynamically determined based on the system's benchmark for optimal performance to avoid redundancy and maximize computational efficiency.

Acknowledgment and Monitoring:

- ➢ For each subjob sent out from the client process, the client process waits for an acknowledgment (ACK) from the receiving peer's subjob process, ensuring that the work has been accepted.
- ➢ A timeout mechanism is in place if an ACK is not received, prompting the client to reassign it to another peer.
- ➢ Received ACKs trigger the client to add the responding peers to a list of child subjob processes, which is monitored for status updates and eventual result collection.

Result Compilation:

- ➢ As peers work on their assigned subjobs, they periodically report progress or completion, at which point, results are sent back to the client.
- ➢ The client compiles these partial results into a comprehensive outcome which represents the completed job

On the **peer** side, here is how a job is handled:

Resource Assessment:

- ➢ Incoming jobs are evaluated by the Coordinator process, measuring against the current load and resource availability of the peer device. This depends on the free capacity of the peer process.
- ➢ This process ensures that the job is feasible for the peer to perform without overloading its resources.

Subjob Execution or Redistribution:

- ➢ If the peer can handle the job, it takes on the subjob. Otherwise, it performs a redistribution of workload.
- ➢ The decision to redistribute is based on the size of the task and the peer's predefined benchmarks. If the task is below a certain size, the system deems it inefficient to redistribute further; the peer then fully processes the task to avoid latency that could stem from additional network communication.

Execution and Reporting:

- ➢ Each subjob process is responsible for a portion of the workload and on completion, sends results back to the parent who requested the job..
- ➢ During execution, if a peer encounters issues or cannot fulfill the task in a timely manner, the Coordinator of the parent process may elect to initiate a redistribution of the remaining work to safeguard the system's overall performance.

To enhance robustness, the system features fault tolerance and recovery mechanisms that ensure continuity in case of errors or peer dropouts. The deliberate design of job distribution and the built-in redundancies allow our system to persist and adapt, maintaining effective operations and progression toward successful job runs.

Capacity of Subjob Processes

All subjob processes are created equal and will have a static capacity. The capacity of the process can be determined by answering the question “How many hashes can we compute under λ milliseconds?”. More practically speaking, the longest time we will wait for any given subjobRequest will be determined based on the average time it takes for a Request to be executed with network latency factored in. This would most likely change the timeout from milliseconds to seconds. Since this would require additional metrics and benchmark tests, we will provide our reasoning for determining the capacity here.

Benchmark

We would have to run a hash benchmark on the device to determine how many hashes per second it can generate. The benchmark would consist of running as many hashes in 100 mili-seconds in a test process and calculating the rate of hashes per second from that.

Let’s assume that the benchmark gives us a value X h/s (hashes per second). If we assume that each subjob process will utilize a single CPU/GPU core, then the hashes/second for a single process would be:

$$\alpha = \frac{X}{Number\ of\ CPU/GPU\ cores} \text{ h/s}$$

where X = benchmarked value of hashes per second

Then, we would run a network benchmark to determine how long on average it takes to send and receive packets from the device. As an example, we could take the average of 5 round trips from this device to 5 other peer devices. Let’s denote this value to be θ.

We can then define the capacity of a subjob process to be as follows:

$$Capacity = \alpha \bullet \theta$$

where α (h/s) = Hashes per second a subjob process can run **on a single process**,
θ (ms) = Average packet RTT (round-trip time) from this device to other devices

To allow even further tunability, we can allow the user to specify the maximum computational time relative to the average packet RTT for a process. It would be a tunable field on the user interface of the application program that provides float selections where minimum is 1.0 and maximum is 4.0. For example, if the user wants the max time to be twice as long as the average packet RTT, he would select 2.0.

Side Note about Low Number of Peer Devices

As a consequence of our design, specifically once a Parent process is disconnected, all its Child processes will discard its results, our system is not very robust if there are a low number of peer devices connected.

The height of our distribution tree comes out to be

$$Height,\ h\ =\ log_{\#\ of\ peer\ devices}\frac{Size\ of\ problem\ space}{Subjob\ process\ capacity}$$

For every device that fails, we would be losing (as a factor of h)

$$\sum_{i=1}^{h}\frac{(k-1)^{i-1}}{k^{i}}$$

where k = # of peer devices, h = height of distribution tree

Assume that we have 5 peer devices.
The problem space for finding a valid none is 2^{32} and if we assume the capacity of a subjob process to be approximately 125,000 (calculated by multiplying 2.5 million hashes /s with a 0.05 second network latency), our height of the tree would be as follows

$$h\ = log_5\frac{2^{32}}{125{,}000}\ \approx\ 6\ (rounded\ down)$$

And our loss would be

$$\sum_{i=1}^{6}\frac{4^{i-1}}{5^{i}}\approx\ 0.74$$

This means that for every device that gets disconnected, we would lose a whopping amount of **~74%** of hashes computed. If we scale this up to use 100 devices as opposed to 5 devices, we would lose **~0.06%** of hashes computed. Conclusively, it is evident that our system is more suited for high volume of peer devices and it would not perform as well with a low number of peers connected.

Design Alternatives

Other alternatives that we considered:-

- Peer-to-peer network implementation: Using a distributed hash table in place of the tracker server
- Peer-to-peer network implementation: Using the gossip protocol to get availabilities of all peers
- Using a client-server network model with a global job scheduler (similar to BOINC)
- Storing the result on the child subjobProcess if the parent is disconnected from the network, however the drawback of this is that it will consume more memory and the parent may never come back alive.

Scalability

The peer-to-peer network model of our design makes it more scalable since there are no overloaded components/central server. However, in terms of geographic locations of peer devices, there would be a network latency overhead for devices that are located really far from each other. To mitigate this issue, peer devices should only be connected to our network if they satisfy a network latency limit. This filtering could be done in the network benchmark portion of Benchmark.

Since the tracker server is a replicated database, the scalability of our program could be increased by increasing the number of databases and placing them in specific geographic locations that maximizes reach.

Methodology and Implementation Strategy

To effectively demonstrate the capabilities of our peer-to-peer model and validate the correctness of our job distribution algorithm, we will strategically abstract certain complexities inherent to distributed systems. Our methodology pivots on a simplified but focused parallel implementation within a single-machine, fate-shared, environment. This approach serves as a simulation sandbox, approximating a "perfect distributed system" where the complexities of networking, consensus mechanisms such as Paxos, and multi-device coordination are temporarily set aside.

Implementation Overview

Single-Machine Parallelism: We replicate the behavior of a distributed network of servers within one machine by creating isolated processes that represent servers. Each server process

houses multiple threads, each thread emulating an independent process, i.e. a subjob, in the server.

Data Structures and Programming Language: Our software model will include structs that capture the essences of our core components—Client, Tracker, Coordinator, and Peer—and their interactions. We will utilize GoLang (Go) for our development efforts due to its inherent support for concurrent programming and powerful primitives for managing goroutines and channels, which align perfectly with our model's needs.

Test-Driven Development (TDD): The single-machine design allows us to employ Test-Driven Development, where unit tests are written in advance of implementation to validate our algorithm and system behavior under various conditions.

Environment Isolation: By operating in a 'fate-shared' environment, where processes are co-located on a single host, we eliminate network-related unreliability and focus solely on system logic and inter-process coordination.

Benefits of the Chosen Approach

Complexity Management: Abstracting away the lower-level distributed system challenges streamlines our focus on the peer-to-peer algorithm and the interaction between logical components.

Testing and Verification: It becomes practical to write comprehensive unit tests, which offer immediate feedback to developers and ensure individual components meet their specified behaviors. This is a huge benefit as formal verification for distributed system correctness wouldn’t be achievable under the project’s scope and timing limits.

Benchmarking: By manipulating job distribution granularity and measuring system performance under different configurations, we can determine the optimal threshold for subtask division. This benchmarking is a critical step towards optimizing our system for real-world distributed environments.

The current methodology sets the stage for a controlled and precise evaluation of our system's capabilities. Our learnings here will directly inform how we scale out and address the nuanced realities of networking, device heterogeneity, and fault tolerance in a genuine distributed setting. The endpoint of this methodology is not only to affirm our system's theoretical foundations but also to produce practical, tested components ready for extension into a fully distributed infrastructure.

Evaluation and Metrics

To accurately measure the success of our Peer-to-peer Volunteer Computing System, we will employ a comprehensive evaluation strategy using Go's benchmarking tools. This approach will allow us to collect detailed performance data and quantify the efficiency of our system under simulated conditions.

Benchmarking Metrics

Elapsed Time: Our primary metric will be the time taken to complete jobs as the system scales in complexity and load. We will measure the total time from job initiation to the completion of all subjobs, capturing the impact of job distribution and concurrent processing.

Thread Utilization: We will monitor the number of threads in use as a function of the active job processes. This metric will help us understand the system's concurrency demands and how efficiently it utilizes computational resources.Using Go's built-in benchmarking facility, we can perform systematic performance testing, where we plot the number of threads (representing separate job processes in a distributed environment) against the time elapsed. This will yield a performance graph that helps us optimize the job distribution granularity. Our goal is to ensure the time gained from parallel processing of subjobs offsets the overhead introduced by managing these subjobs.

Performance Evaluation

Unit Testing: Comprehensive unit tests will form the foundation of our evaluation process, ensuring that individual components behave as expected. We will specifically test for correctness in job assignment, task execution, and result aggregation within our multithreaded model.

Varying Input Sizes: To test the scalability and robustness of our job separation algorithm, our system will be subjected to a wide range of input sizes, challenging the job distribution logic and its adaptability to changing workloads. During these evaluation phases, we will be searching for an optimal threshold for task subdivision. This is the point at which the overhead of creating additional subjobs is balanced by the time saved due to parallel execution, resulting in the most efficient operation of our system.

Real-World Consideration: While our initial evaluation focuses on a single-machine model, the insights gained will be essential for scaling to a true distributed environment. We will look to extend the benchmarking methodology to distributed hardware in subsequent phases, tracking similar metrics but with a heightened emphasis on network-induced latencies and real-world communication overhead.

The systematic evaluation of our peer-to-peer computing system, combined with targeted metric analysis, will provide us with the actionable insights needed to refine and optimize the

distribution of computational tasks. This data-driven approach ensures that our system not only theoretically satisfies our project goals, but also demonstrates practical efficacy and real-world performance resilience.

References

Background Reading
Berkeley Open Infrastructure for Network Computing (BOINC)
https://boinc.berkeley.edu/trac/wiki/SoftwareDevelopment
https://github.com/BOINC/boinc/wiki/BoincPapers

Global task scheduling in volunteer computing system
https://link.springer.com/article/10.1007/s41870-022-01090-w

Capacity of Subjob Processes
https://stackoverflow.com/questions/46386273/whats-a-good-typical-network-timeout
https://stackoverflow.com/questions/5578702/using-timeout-in-p2p-architecture
https://www.gemini.com/cryptopedia/crypto-mining-rig-bitcoin-mining-calculator-asic-miner#section-gpu-mining-takes-over
https://bitcointalk.org/index.php?topic=1628.0