Parallel Framework

Parallel is a command-and-control distributed botnet. There are 2 characters that participate in this system.

- Chief – facilitates connection to the client and distribution of resources in the botnet.
- Worker – reports to a Chief and completes tasks for the group based on given inputs.

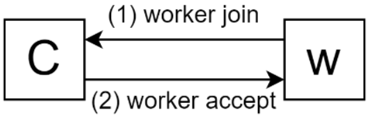
The Design Philosophy

I want the capability to process data using *modules* that run in isolated environments on multiple computers and behave as an **asynchronous** distributed system. Modules contain instructions for data splitting, merging results, and processing. Users will define their applications as a module package which is uploaded to the Chief. This network is called a *Company,* and it participates in a higher-level peer-2-peer network.

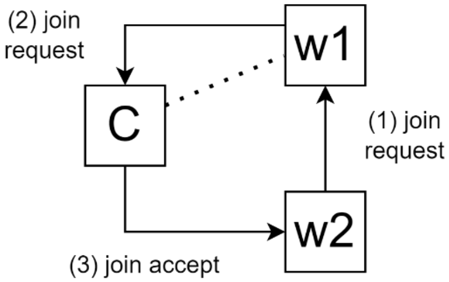
I want this framework to be accessible for people who want to use the benefits of distributed parallel processing and do not have the software infrastructure to do it.

Creating a Network

The network is built with TCP sockets and http web sockets. A network is **started by instantiating a chief**. To add workers, instantiate a worker process that connects to the network through the chief or another worker. If the new worker connects to another worker, its request will be relayed to the chief.



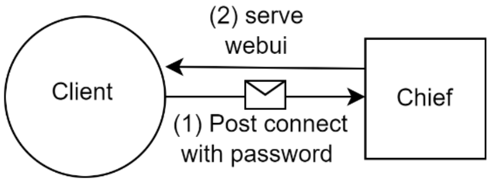
- (1), worker w sends a request to join the network directly to the chief C. (2), C accepts the request, adds the new worker to the network and assigns the worker's supervisor as itself.



- (1), worker w2 attempts to join the network via a join request to another worker w1. (2) w1 is not the chief so it relays the request to the chief C. (3), C accepts the request, adds the new worker, and assigns that worker's supervisor to itself.

Using The Network as a Client

You will access this system through a web-ui on the chief. The client is only able to use the network through the chief and is not allowed to contact its workers. From this point the client can upload resources to the network. The client can also start jobs through the user interface. Clients may only have direct contact with one chief, but chiefs can have multiple clients.



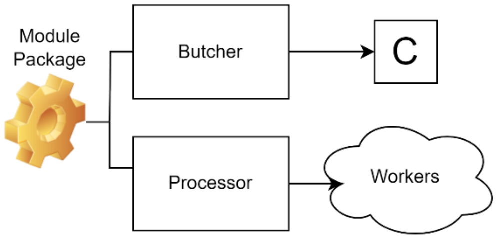
- (1), The client attempts to access the user-interface through a web-ui form. (2) the chief serves the web-ui or refuses the client

File Distribution with Web Sockets

Every process on this network operates its own http client/server. We use the initial TCP connections as the contacts list. There should be some kind of standard directory in the protocol where the files are stored for each character. **The chief generates and allocates fragments on demand** as it processes requests from workers and delivers them over its http server with the *splitter module*. After a fragment is sent to a worker that worker is *locked* until the chief receives a response from the worker or it times out at which we can do something else. This enables us to have concurrent access across companies to larger groups of workers. This also allows us to potentially run more than 1 job at a time. The client uploads files with post requests to the chief and downloads data with get requests like a normal rest API.

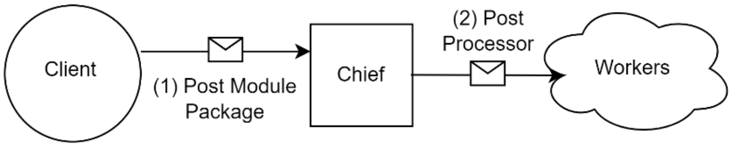
Modules

Modules are python programs that are developed by the user for deployment on the botnet. They run in an isolated environment for security measures. We can maybe log the std I/O of that process to collect data.



Every task requires a module package. A module package consists of 2 components, a **Butcher,** and a **Processor**. The Butcher is a pair of modules named *Split.py* and *Merge.py* used only by the chief. These files are defined by the user and must adhere to the constraints of this protocol to function properly. The Butcher generates fragments on demand by loading parts of the dataset into memory one at a time. The fragments are cached and destroyed when the subtask is completed. It is also responsible for merging results.

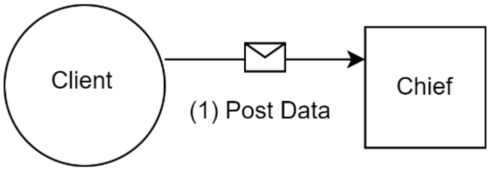
The Processor module consists of 1 or more python files used by the workers. To upload a module package containing required resources for a job we have the following procedure.

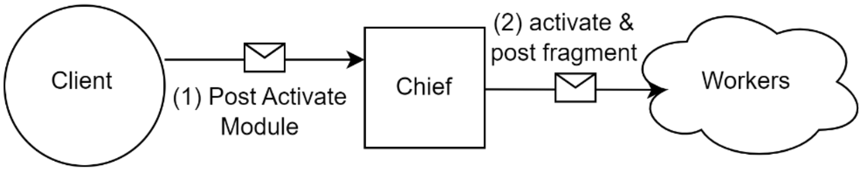


(1) The client posts the package to the chief. (2) The chief sends the processor to everyone in its directory. Modules are cached in workers so they can be used repeatedly without requesting the webserver between different jobs.

Data Distribution & Network Activation

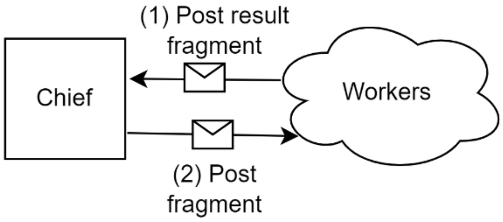
The data that is delivered across this network must be compatible with the modules defined by the user. To upload data to the network we post the whole dataset to the chief as the client. Triggering the chief to send a fragment causes it to be created on demand with the splitter.

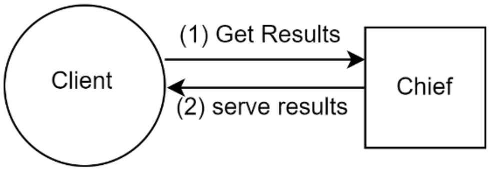




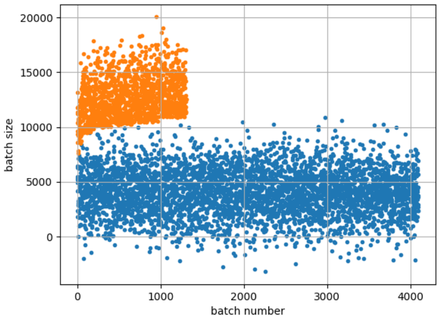
(1) triggers the activation, (2) provides the worker with the endpoint directory to retrieve fragments from, it also provides the worker with a fragment to start with.

As each worker completes their task it will notify the chief by posting their results and the chief will unlock the worker. The chief will then send another fragment and lock, this is **the main processing loop.** When the chief knows that all tasks have been completed it will use the *Merge* module to recombine the resulting data. The chief will then add the merged results to its server. The client can download and view the results.





Choosing the best fragment size for each job



The runtime of our system is dependent on the number of post requests sent per job, which is inversely proportional to the size of each fragment. Fragment sizes are a function of the user's python generator, and we can make no assumptions about its output so we must consider **sparse** input spaces, (fragments are different sizes). We can implement **batching** to handle this. The goal is to limit the total amount of POST requests while normalizing the size of the batch. We $\sqrt{n}$ want to look at the entire input space, only some of it. The size of the window could be the last ￼ fragment sizes in an input space with n fragments. So, if we have the 9th fragment, we look at the last 3 fragments. If the size of the 9th fragment is smaller than the max size of the last 3 fragments, then we save it. Otherwise, we just send it. We end up with a total number of batches proportional to the size of the input space equal to 1/e. This model follows the Poisson distribution for the probability of an event occurring in a fixed interval.

$$P(k,\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

Unlike many examples online our case is time independent, the fixed interval is our input space, so a "time" unit is whenever a batch is made. Using this algorithm, we can model the probability of the next batch being formed as we iterate over any input space with the expression,

$$\alpha = |inputs|,\ k = \frac{1}{\alpha},\ \lim_{k \to \infty} \left(\frac{1^k e^{-1}}{k!} \right) = \left(\frac{k!}{e} \right) = e^{-1}$$

we end up with a function that converges at e^{-1}. Our choice for the portion of the input space to evaluate does not affect the outcome. So, if we have 10 inputs, we get $10e^{-1}$ batches, that's it. And the **average** batch size is converging towards *e* for any input space.

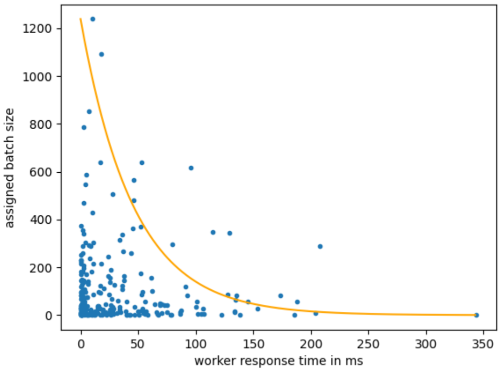
Workload Management

We know the distribution of worker nodes' performance in a network follows a normal distribution. Performance is a metric measured as how fast they can return batches. Now that the inputs are also normalized, we can calculate the standard deviation of the input window and find out how far a batch is away from the mean batch size. We also need to calculate the standard deviation for response times and find the workers' distance from the means.

We can formulate a matching problem using the size of the batch and the performance of a worker node to decide who gets what. The *Gale-Shapely* algorithm is a famous method used to solve stable matching problems. There are *m* batches in the cache and *w* free workers. It will be preferable to calculate our cost function as a dimensionless value. Therefore, *preference* is the *difference in z-scores squared* of the free workers and the batches in the chief's fragment cache.

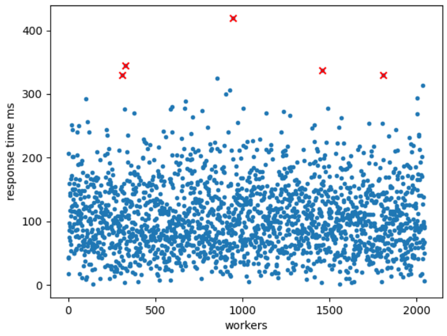
$$\Delta(x) = \frac{x-\mu}{\sigma},\ \ p(w,m) = \big(\Delta(w) - \Delta(m)\big)^2$$

Preference where μ is the mean and σ is the standard deviation. We will end up with an assignment profile that looks like this graph on the right. I think it follows the model of the orange line, but more testing needs to happen. It represents the function that batch assignments are trying to get under (the goal of the matching problem). $M = \max(batch_{sizes})$, $s(t) = \frac{M}{e^{\frac{x}{\sigma}}}$, where σ is the standard deviation of performance. You must understand with this algorithm it is unlikely but not impossible that late nodes get large batches. So how do we assign each incoming batch?



At first batches are sent to all workers and the chief waits for the first 2 workers to return batches to calculate the first round of z-scores. When any worker returns a batch **and** has an empty job queue, we call this a *batch request.* The chief invokes the generator to create *w* new batches and then *Gale-Shapely* for all workers who have returned at least 1 batch. Workers who are busy will add their assigned batches to a queue. Workers that have not yet responded are not part of the stable matching problem yet and extra batches are also ignored by the algorithm. Even though sources say that it's for equal sized groups the algorithm behaves the same if we have way more batches than workers and takes $O(w^2)$ operations. The number of *batch requests* that it will now take to exhaust the generator is approximately $requests = \frac{m}{w}$. Once the generator is exhausted, batch requests no longer yield new batches because every fragment is either assigned to a worker or in the chief's cache waiting to be assigned at a later round. Every subsequent round will work to empty the fragment cache. Any workers who request a batch **during** matching are ignored because we are guaranteed if there are at least w batches before matching that this worker will be assigned a batch during matching that it will be able to use (pigeon-hole).

Failure Detection



We can also use the performance standard deviation model to kick workers when they fall over 4 standard deviations of the mean, so we need to find when the *z-score* of a node is over 4. This may need to be calculated asynchronously but it will only be used by the chief. We only need to know the last response time of each worker. **A worker fails if its performance metric fails**. This means that when the chief's timer on the late node reaches over the threshold, we defined it will stop and we can disconnect from the worker or ignore it. Will workers get kicked every round? Do I actually care if they are slowing my system down anyways? Does this make my system semi-synchronous if the failure of a worker dependent on the performance of its peers?