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.

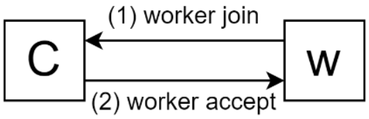
The Design Philosophy

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.

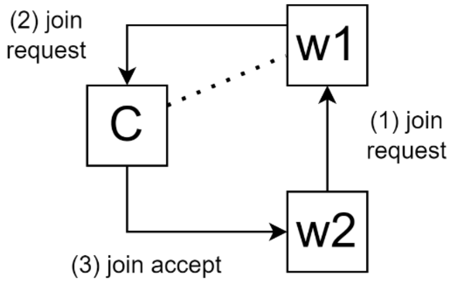
I want this framework to be accessible for people who want to use the benefits of distributed parallel processing and do not have the software infrastructure to do it.

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.



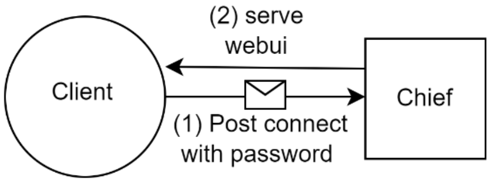
- (1), worker w sends a request to join the network directly to the chief C. (2), C accepts the request, adds the new worker to the network and assigns the worker's supervisor as itself.



- (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.

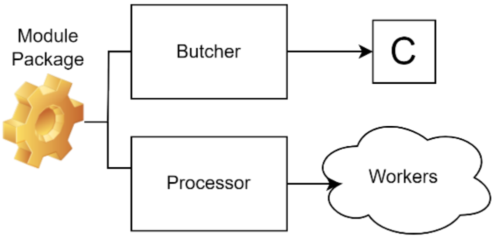


- (1), The client attempts to access the user-interface through a web-ui form. (2) the chief serves the web-ui or refuses the client

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.

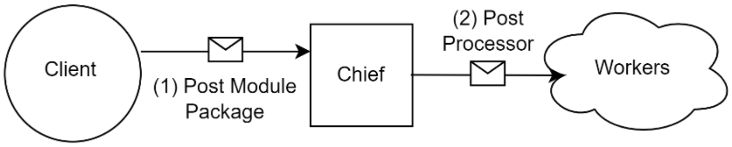
Modules



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.

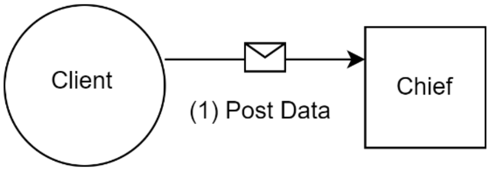
The Processor module consists of 1 or more python files used by the workers. To upload a module package containing required resources for a job we have the following procedure.

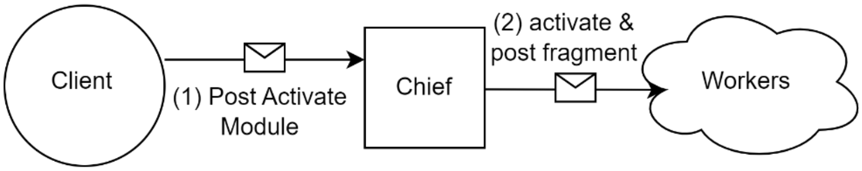


(1) The client posts the package to the chief. (2) The chief sends the processor to everyone in its directory. Modules are cached in workers so they can be used repeatedly without requesting the webserver between different jobs.

Data Distribution & Network Activation

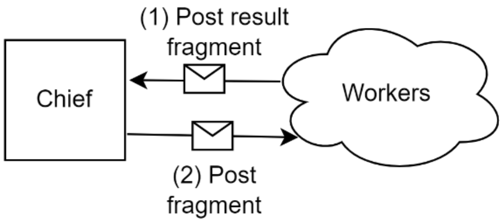
The data that is delivered across this network must be compatible with the modules defined by the user. To upload data to the network we post the whole dataset to the chief as the client. Triggering the chief to send a fragment causes it to be created on demand with the splitter.

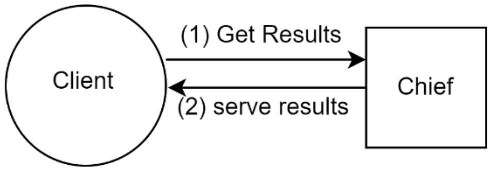




(1) triggers the activation, (2) provides the worker with the endpoint directory to retrieve fragments from, it also provides the worker with a fragment to start with.

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. We can implement **batching** to handle this. The goal is to limit the total amount of POST requests while maximizing the size of the batch. We don't want to look at the entire input space, only some of it. The size of the window will be the last $\sqrt{n}$ 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 (e^{-1}). 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. So, we can express the probability of the next batch forming over the input space modeled by this equation,

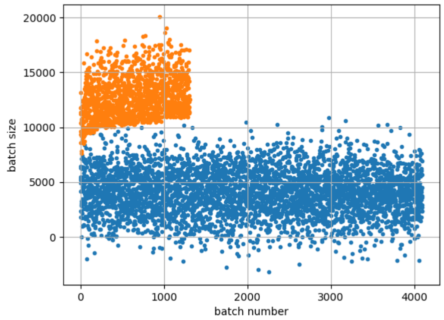
$$\alpha = |inputs|,\ k = \frac{1}{\alpha},\ \lim_{k\to\infty}\left(\frac{1^k e^{-1}}{k!}\right) = \left(\frac{k!}{e}\right) = e^{-1}$$

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 roughly $\frac{10}{e}$ batches, that's it. And the **average** batch size is converging towards *e* for all input spaces.

If we take 2^{12} samples of randomly sized fragments between 0 and 2Kb we get a distribution that looks like this. It will end up helping our workload distribution.

Workload Management

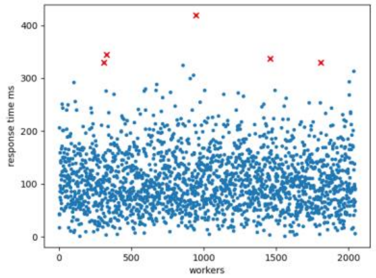
We know that the distribution of worker nodes in a network is somewhat a normal distribution of performance, performance being how fast they can return batches regardless of all other variables, we do not care about the network or miscellaneous errors. So now that we have control over the input's sizes, we can match batches to each worker. We can calculate the standard deviation of the input window and find out how far a point is away from the mean batch size. We also need to calculate the standard deviation for response times and find out how far a worker is from the mean performance. **(batching)**



Say our workers each have a standard performance deviation that is evaluated when they return a batch, the chief takes the amount of time it took to retrieve a batch from a worker. To find the right match we need to minimize the distance between performance and batch size. Now we calculate the squared differences between each worker and the standard size deviation of the next batch. The index of the minimum entry corresponds to the worker that the batch gets assigned to, if it is not the worker asking for data, we cache it. If the batch was already caught, then we don't need to do any of the calculations. We can formulate a stable matching problem using the size of the batch and the performance of a worker node to decide who gets what. We can use the *Gale-Shapely* algorithm to solve stable matching problems. There are *m* jobs in the cache and *w* free workers. Our optimality or "preference", in the textbook is calculated as the difference in z-scores of the works and the jobs in the cache. So how do we assign each incoming batch then, a batch being created on the return of a previous batch unless the job is starting which we are ignoring. At first Batches are sent randomly and the system waits for at least 2 workers to calculate the first round of z-scores. Workers that have not yet responded are not part of the stable matching problem yet. When the 3rd fastest worker returns a batch for the first time, we know the standard deviation, so we just get its z-score then we run the matching problem for the available workers. We can also assign jobs to workers every time a request for data is made to implement a prefetching strategy where workers only request more data if their assignment queue is empty.

Worker 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. If a worker has failed, in our context it's either fail-stop, or network lag, or misc. error. In this context I don't care at all if the node becomes useless to me once it has any of those properties. 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. **(worker elimination)**


response time ms
workers