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.

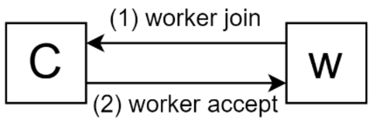
The Design Philosophy

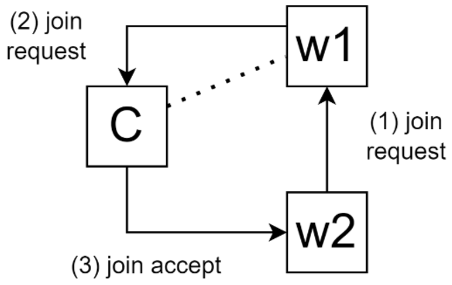
I want the capability to process data using *modules* that run in isolated environments on multiple computers and behave as an **asynchronous** distributed system. Modules contain instructions for data splitting, merging results, and processing. Users will define their applications as a module package 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.



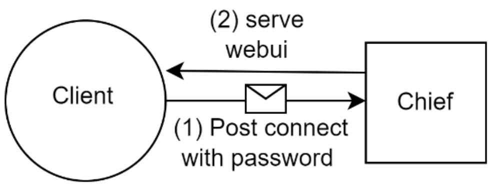


- (1), worker w sends a request to join the network directly to the chief C. (2), C accepts the request, adds the new worker to the network and assigns the worker's supervisor as itself.

- (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.

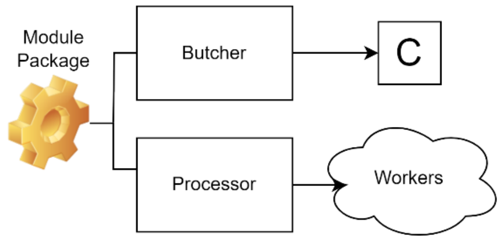


- (1), The client attempts to access the user-interface through a web-ui form. (2) the chief serves the web-ui or refuses the client

File Distribution with Web Sockets (http)

This is the method that we are using. Its practical, scalable and we can implement https as well. 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.

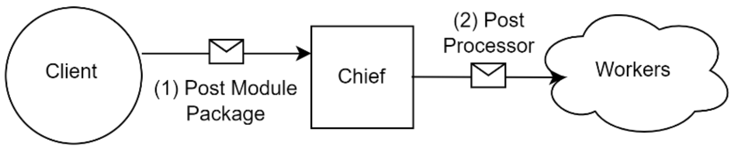
Modules



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.

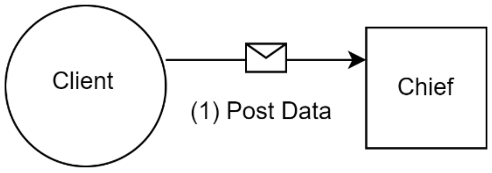
The Processor module consists of 1 or more python files used by the workers. To upload a module package containing required resources for a job we have the following procedure.

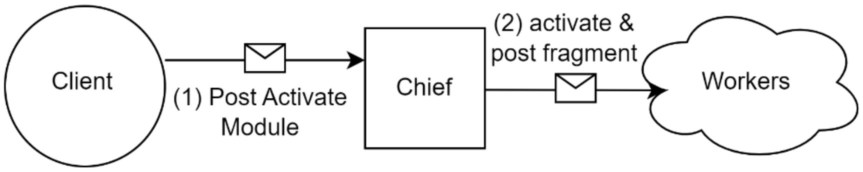


(1) The client posts the package to the chief. (2) The chief sends the processor to everyone in its directory. Modules are cached in workers so they can be used repeatedly without requesting the webserver between different jobs.

Data Distribution & Network Activation

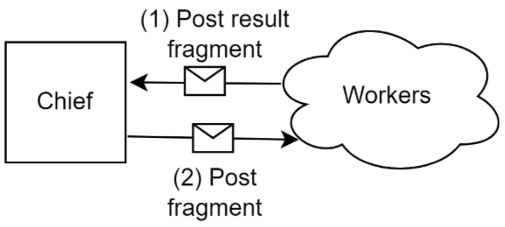
The data that is delivered across this network must be compatible with the modules defined by the user. To upload data to the network we post the whole dataset to the chief as the client. Triggering the chief to send a fragment causes it to be created on demand with the splitter.

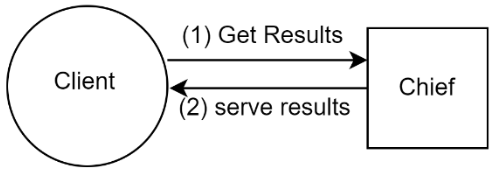




(1) triggers the activation, (2) provides the worker with the endpoint directory to retrieve fragments from, it also provides the worker with a fragment to start with.

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. Therefore, the goal is to maximize the size of the fragment. We are using a system that produces sparse fragment sizes, so this introduces extra complications, but we can solve these pretty easily. Let the size of fragment f_k in bytes be the function $bytes(k) = sizeof(Split_k)$ where $Split_k$ is the kth iteration of the generator function. We can make no assumptions about the range of the output except that it is less than the maximum fragment size. This means we need to handle sparse input spaces (not same fragment size for every fragment). Let's just also define the job size for k fragments be the expression $job_{size}(k) = \int_0^n bytes(k)dk$ $,\sum_{k=0}^{n} bytes(k)$

While our system is executing a job, $job_{size}(k)$ will increase as k increases, k being the number of inputs sent. We want to create an objective function that optimizes a target size. This will be used to batch fragments together. So, **we are going to find the best batch size for the user's inputs**. Which means we want to solve a least squares optimization problem over the current input sizes that optimizes a parameter α being the batch size. We can use linear algebra to do this **very** quickly.

We have identified the type of problem we are solving, now to solve least squares optimization we need to evaluate the expression $A^T Ax = A^T b$. This orthogonally projects b onto the column space of A which contains all possible vectors that solve the equation $Ax = b$. Now next we need to define what A, x, and b are in relation to this. A is a matrix containing the linear equations we want to solve, each data point is a 2-dimensional object (k, s). Each equation in the system is $batch_{size} = Mk + B + size_{max}$ so A's rows are of the form $[k, size_{max}]$, x is of the form $[M, B]$. The entries of b are each fragment size that we have so far. Our system will solve this equation every time it sends a new fragment with the new fragment added to the input set. This will give us a linear equation called a batch model that can be associated with each job. A new fragment whose distance to the max size lies above the linear equation will be saved for batching. If it lies below, it will be sent immediately. When another item is batched, we take the average size of the fragments in the batch and apply it to the model with the batch size that will determine if we send the batch or keep it. This model will significantly improve performance time in relation to the number of post requests sent.