Parallel Framework

Parallel is a command-and-control distributed botnet. There are 3 characters that participate in this system.

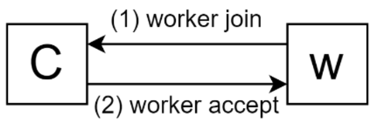
- Client - Contacts the system and receives result from the system.
- Chief – facilitates connection to client and distribution of resources in the botnet.
- Worker – reports to a Chief and completes tasks for the group based on given inputs.

The Design Philosophy

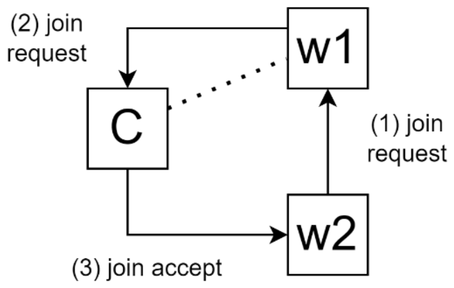
I want the capability to process data using “modules” on multiple computers that behave as an **asynchronous** distributed system. Modules contain instructions for data splitting, merging results, and processing data. Users will define their applications as a module package which is uploaded to this network.

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.



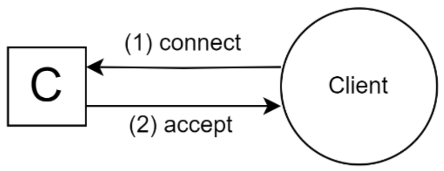
- (1), worker w sends a request to join the network directly to the chief C. (2), C accepts the request, adds the new worker to the network and assigns the worker’s supervisor as itself.



- (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

Instantiate a client process that connects to the chief. The client is only able to use the network through the chief and is not allowed to contact its subordinate workers. From this point the client can upload resources to the network. The client can also start jobs and access the user interface. Clients may only have direct contact to one chief, but chiefs can have multiple clients.



- (1), the client asks the chief to connect to the network. (2), the chief adds the clients to its contacts and the client adds the chief as its primary contact.

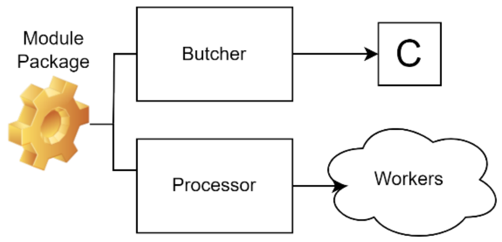
File Distribution with (SFTP)

One way of doing file distribution that we did not implement but thought about. Every process on this network operates its own sftp client/server. There should probably be some kind of standard directory in the protocol where the files are stored. When data fragments are available for processing, available workers are identified by the chief. The chief uses a distributor that autonomously allocates fragments to the list of available workers and delivers those fragments over sftp. When a fragment is delivered that worker is "locked" until they deliver a result back to its supervisor.

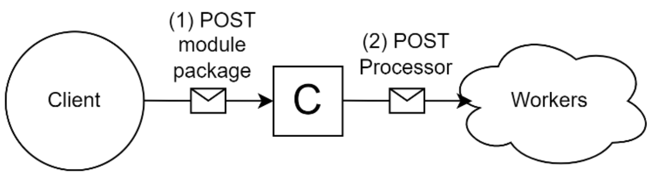
File Distribution with Web Sockets (http)

This is the method that we are using. Every process on this network operates its own http client/server. Each character's http client/server behaves differently. There should be some kind of standard directory in the protocol where the files are stored. When fragments are available for processing. The chief allocates fragments as it processes requests from workers and delivers them over its http server. 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 networks to larger groups of workers. This also allows us to potentially run more than 1 job at a time.

Modules



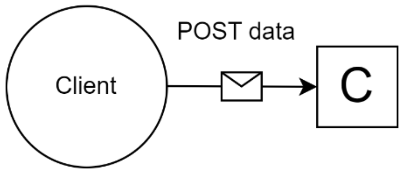
Modules are python scripts that are developed by the user for application. Every task requires a module package. A module package consists of 2 components, a **Butcher,** and a **Processor**. The Butcher is a pair of modules named Split.py and Merge.py used only by the chief. These files are defined by the user and must adhere to the constraints of this protocol to function properly. The 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.

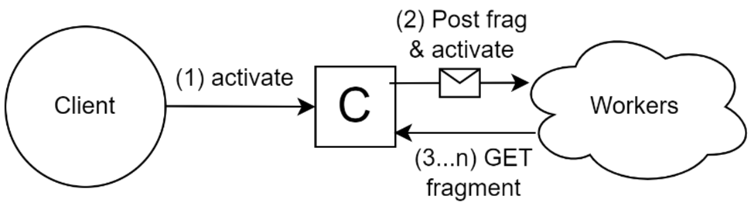


(1) The client posts the package to the chief. (2) The chief sends the processor to everyone in its directory. Modules are cached in workers so they can be used repeatedly without requesting the webserver between different jobs.

Data Distribution & Network Activation

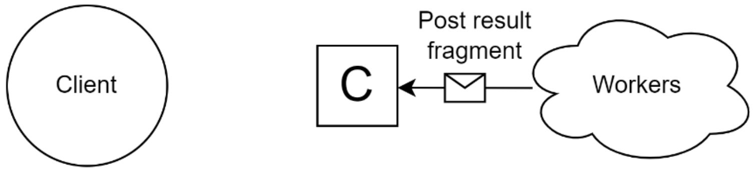
The data that is delivered across this network must be compatible with the modules defined by the user. To upload data to the network we post the whole dataset to the chief as the client. Once the chief receives the data **and has acquired the Butcher** it will proceed to create fragments with the dataset. We should be allowed to send components from the butcher to the workers if we want to recursively butcher our dataset because it's big or the fragmentation/recombination process is expensive.

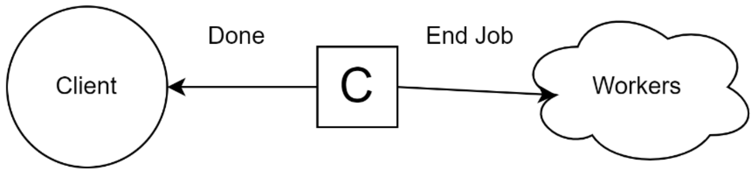




(1) triggers the activation, (2) Post fragment and activate provides the worker with the endpoint directory to retrieve fragments from, it also provides the worker with a fragment to start with.

As each worker completes their task it will notify the chief by posting their results and the chief will unlock the worker. 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 notify the client and add the merged results to its server. The chief will also deactivate every worker from the job.





Project Structure

This framework uses TCP RPCs for communication and http for uploading and downloading resources. The RPC's required to implement this framework follows.

1. Worker-join-request	Worker->Chief
2. Worker-join-accept.	Chief->Worker
3. Client-connect.	Client->Chief
4. Client-accept.	Chief->Worker
5. Done	Chief->Client
6. Activate	Client->Chief
7. End-job	Chief->Worker

This framework uses a rest API to implement the http client/server network. The following endpoints are required to implement this framework.

1. Client: (POST) - upload/module-package
2. Client: (POST) - upload/data
3. Worker: (POST)- upload/result-fragment
4. Chief: (POST) - processor
5. Chief: (POST) - activate/data-fragment
6. Worker: (GET) - data-fragment/(fragment #)
7. Client: (GET) - final-result