Parallel Framework

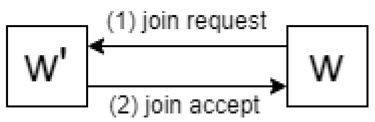
Parallel is a command-and-control distributed botnet. There are 3 characters that participate in this system.

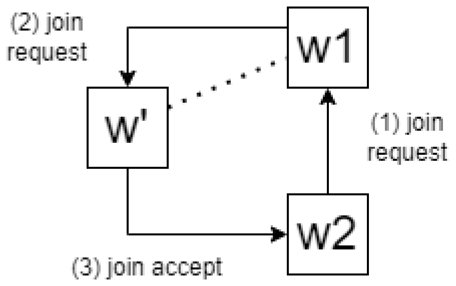
- Client - Contacts the system and receives result from the system.
- Chief Worker – facilitates connection to client and distribution of resources in the botnet.
- Worker – reports to a Chief Worker and completes tasks for the group based on given inputs.

The design philosophy is to have the capability to process multimodal data using "modules". Modules contain instructions for data distribution, merging results, and processing data. Any application that uses this framework requires a module package that provides "split" and "merge" modules along with your processing module.

Data distribution is implemented using a webserver that can be accessed by all characters. The webserver stores data in (tasks), module packages in (modules) and results in (results).

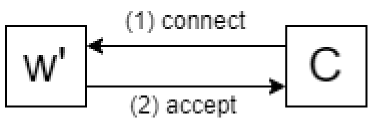
The network is built via TCP socket connections, this protocol uses port 11030. A parallel network is started by instantiating a chief, which will also start the webserver. To add workers, we can connect to any process currently in the network that may or may not forward that machine's request to the chief.





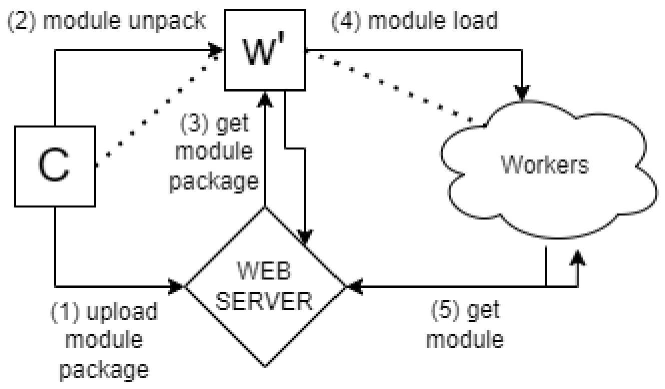
This figure demonstrates the 2 different ways that worker processes can connect to the parallel network. Either directly requesting the chief or request via proxy through another worker. All workers including the chief keep a directory of who they believe is in the network.

To access the network as a client we must instantiate a client process that must connect to the chief worker.



This figure demonstrates how a client connects to the network. Once connected the chief logs the client. The client gains access to the webserver and web-ui.

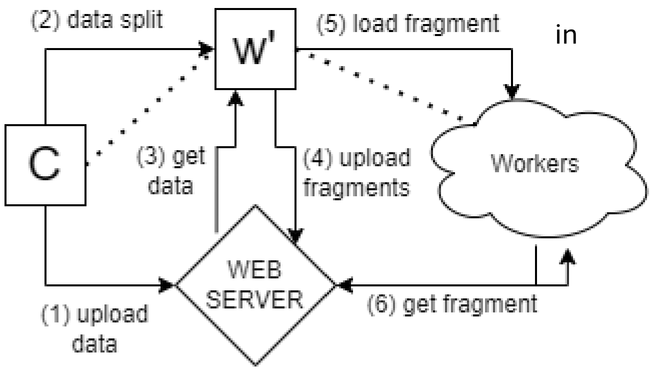
Modules are defined as python scripts that accept arrays as inputs and return arrays as results. To upload a module package containing required resources for a job we have the following procedure.



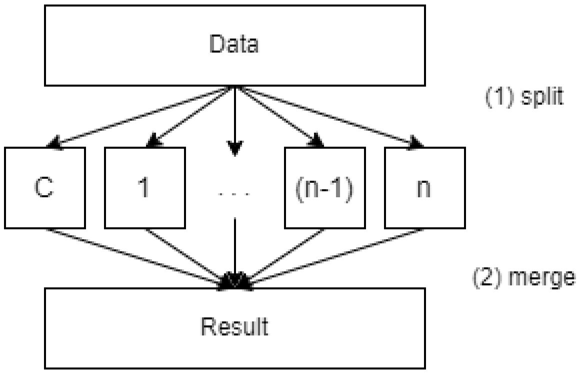
This figure demonstrates **how modules are distributed on a parallel network**. Modules are cached in workers so they can be used repeatedly without requesting the webserver. The chief is the only process that uses the split and merge modules, normal workers only need the processing module.

The data that is delivered across this network may be multimodal and it must also be compatible with the modules defined by the user. To upload data to the network we use the following procedure.

This figure demonstrates **how data is distributed in a parallel network**. It is similar structure to the method used to upload modules. In this procedure the chief worker chops the data provided with the split module. The data fragments are allocated and uploaded to the webserver.

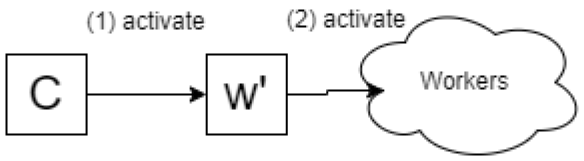


in

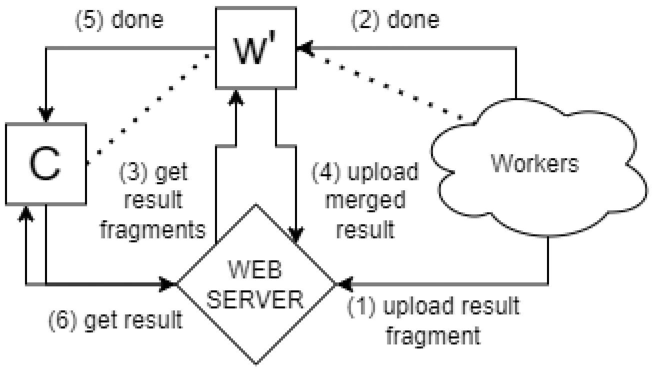


This figure is a representation of the **data processing pipeline for this framework**.

Once your network has finished gathering and allocating resources necessary to complete your application we can **activate the network** with the following procedure. This can be done repeatedly if needed for batch processing.



As each worker completes their task it will notify the chief. When the chief knows that all tasks have been completed it will download the result fragments and use the split module to merge them. The chief will then notify the client and post the merged results to the webserver. The following procedure is used to process results.



This figure demonstrates **how results are processed and gathered** in the parallel network.

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
2. Worker-join-accept
3. Client-connect
4. Client-accept
5. Unpack-module
6. Load-module
7. Split-data
8. Load-data
9. Done
10. Activate

This framework uses a rest API to implement the webserver. The following endpoints are required to implement this framework.

1. (POST) - upload/module-package
2. (GET) - module-package
3. (GET) - module
4. (POST) - upload/data
5. (GET) - data
6. (POST) - upload/data-fragments
7. (GET) - data-fragment
8. (POST) - upload/result-fragment
9. (GET) - result-fragments
10. (POST) - upload/final-result
11. (GET) - final-result