Bester Next Generation

Bester NG

Bester is a modular data processing network which allows nodes to submit jobs for processing by a set of message handlers for particular job/message types and then Bester also manages the distribution of those job results.

Terminology

Nodes are the network clients, they submit jobs and get job results sent to them.

Message handlers process jobs and can tell the daemon where the output should be directed (either to another handler) or to users/servers (remote users).

The Bester daemon's handle the movement of data from clients to message handlers and back to clients (and remote servers by virtue of remote clients).

Nodes

Nodes are the endpoints and starting points of jobs (the units of processing/work in the Bester network). Nodes have unique names on the local server they are connected to and are also authenticated with the server using a username-password pair.

Registration

TODO

Authentication

Authentication is the process of supplying your username and password and having the server verify your account. Your username, `<username>` is used to address you locally and remotely.

```
{
    "request" : "auth",
    "account" : {
        "username" : "<username>",
        "password" : "<password"
    }
}
```

If a node does not authenticate then it cannot use any other commands other then the authentication and registration ones.

Jobs

> Jobs are the core concept of Bester. Data goes in, is processed by message handlers for that job-type, then comes out

Job submission

Jobs are submitted with a `type` used to indicated which message handler should be responsible for processing the job and a `payload` - the data to be processed. There is also a priority field which can be used to express how urgent it is to run the job.

```
{
    "request" : "newJob",
    "job" : {
        "payload" : {
            "type" : "<type>",
            "data" : <payload>
        },
        "priority" : <priority>
    }
}
```

Once the above JSON is submitted then one will receive a reply with details about the submission and whether or not it was successful - indicated by the `<status>` field. If it is successful then the tracking identifier for the job will be in the `jobID` field.

```
{
    "reply"     : "jobSubmission",
    "submission" : {
        "status" : <status>,
        "jobID" : <jobID>
    }
}
```

See the first part as submitting the data for processing and the second as getting a tracking id (receipt) for it.

Job notifications

Jobs complete and you asynchronously get notifications for their completion, likewise for failures. Success or failure is indicated by the `status` field and for the specific job specified in the `jobID` field.

```
{
    "reply" : "jobNotification",
    "job" : {
        "status" : <status>,
        "jobID" : "<jobID>"
    }
}
```

See this as a notification, other than the job outputs (which are described below), that the job passed (successfully ran). Or on the other hand failed at some point.

Job outputs

It is worth pointing out that multiple clients are connected to the network, some submit jobs but then where do they go? Well, to other servers but they have to end up back at a client sometime. This is what an output is. A "job notification" is a otification of completion or error in processing a job but you need not (on the same client) necessarily receive a job output **too**. This is because the message handlers can direct the output to a client other than the one who submitted the job.

```
{
    "reply" : "jobOutput",
    "job" : {
        "jobID" : "<jobID>:<submittingUser>@<server>",
        "payload" : {
            "data" : ...,
            "type" : "<type>"
        }
    }
}
```

The output is in the `payload` object with both the `data` (generated by the message handler) and the type associated with that handler (`type`) (such that the nodes know how to interpret the data).

The unique job ID originally assigned to the job during the job submission process is placed in the `jobID` field. However, to disambiguate between jobs received with the same ID but generated from different users on the same server or a mix of the same server and remote jobs being ingested and presented as job outputs on a remote server, a change has to be made to

this field. The `jobID` is appended with a `:`, the user who submitted it, then an `@` and lastly the originating server.

Message handlers

Message handlers are what process job input payloads and spit them out to other clients (a mix of local and remote) or input to another message handler for further processing. This section describes the protocol used between the Bester daemon and a message handler process over a UNIX domain socket registered for the message type that handler is registered for.

Data in (`bester -> message handler`)

When the daemon receives a job submission it must then convert this into a format for the message handler which looks like this:

```
{
    "payload" : <payload>
}
```

It's pretty simple, just the `<payload>` data, from the original jobSubmission, is carried through.

Data out (`message handler -> daemon`)

The message handlers spit out this. Their processed payload is now `payload` and this can now either be redirected to a list of users (local and remote) or to another message handler for further processing *and then* to a list of users. A last type of director is a *handoff* which is s submission to a remote message handler but without awaiting a reply from it (the remote server deals with it as explained - recursively).

```
{
    "payload" : <payload>,
    "director" : {
        "type" : <type>,
        ....
    }
}
```

Directing message handler output to users (local and remote) (`output -> users`)

To direct the output to a set of users one would specify this by making the `director` field the following:

```
{
    "payload" : <payload>,
    "director" : {
        "type" : "userDelivery",
        "users" : ["user1", "user2@8.8.8.8:2222"]
    }

}
```

Directing message handler output to a local message handler (`output -> handler`)

To direct the output into another message handler one would specify this by making the `director` as follows (this allows recursive handling):

```
{
    "payload" : <payload>,
    "director" : {
        "type" : "handler",
        "handler" : "<handler>"
    }
}
```

This output from one message handler will then be sent in as input into the message handler specified in the `handler` field.

Directing a message handler output to a remote server / handoff (`output -> handler@remote`)

Another aspect is if you want to direct a handler's output to a remote server (whereby a handler there will process it). However, before we go any further, you must note that this is simplex and one way, there is no way to push the response back to the initiating server, hence why we call it *handoff*.

```
{
    "payload" : <payload>,
    "director" : {
        "type" : "handoff",
        "handler" : "handlerType1@10.0.0.1:8888"
    }
}
```

Network pipline a message through a set of message handlers ("symmetric handoff")

Bruh if you dare suggest a remote message handler (a true one, then I commit die pls - we never even had such a feature in Bester normal) - **fuckit - let's do it**

Server-to-server communication

Another aspect is how servers communicate with eachother. Whether it be a handler delivery the result of a job completed locally to a set of remote users indirectly via their home server or if a handoff is to occur. There needs to be a protocol for this.

Remote job delivery

If a handler sets the director type to be that of users and some of these users are on remote servers then the following will be sent to those servers:

```
{
    "request" : "jobOutputDelivery",
    "job" : {
        "jobID" : "<jobID>:<submittingUser>@<server>",
        "payload" : {
            "data" : ...,
            "type" : "<type>"
        }
    }
}
```

On the remote server this will be converted into a message for sending to the clients on that remote server in the form described in the **Job output** section.

Remote job handoff

If a handler sets the director type to be `handoff` then the following will be sent to the remote servers the handler output is meant to be sent to:

```
{
    "request" : "jobHandoff",
    "handoff" : {
        "type" : "<type>",
        "payload" : ...,
        "from" : "<user>@<remoteAddress>"
    }
}
```

Where the handler on the remote server that should handle the given `payload` is specified by `<type>`.

TODO: The server should append the jobID of this job or atleast who it caame from, the job ID thing is a bit weird as technically this is a new job I guess if we see it from the "ingestion" point of view but we also can see it as the job extending but that makes no sense as it is completed as soon as handoff is sent on the sending server (and the client gets such a notification).

TODO: Let's generate new job id with it and then append remote server and user (gathered from information done when sending (a.k.a. the above JSON). The `from` field.

TODO: From this information we ingest it as if it were a local user job submission except we notify no user (on the submitting server - hence the *"handoff"*), we do construct a job output though of course.

TODO: The notification happens after handoff **sending** is complete, not remote job completion.