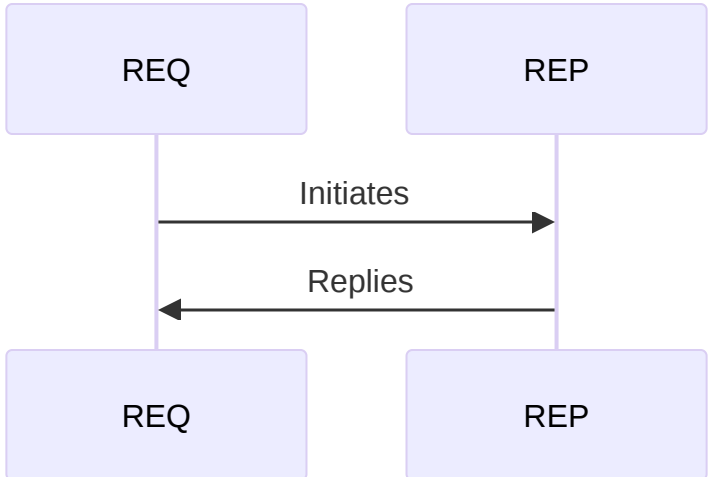
Master Thesis

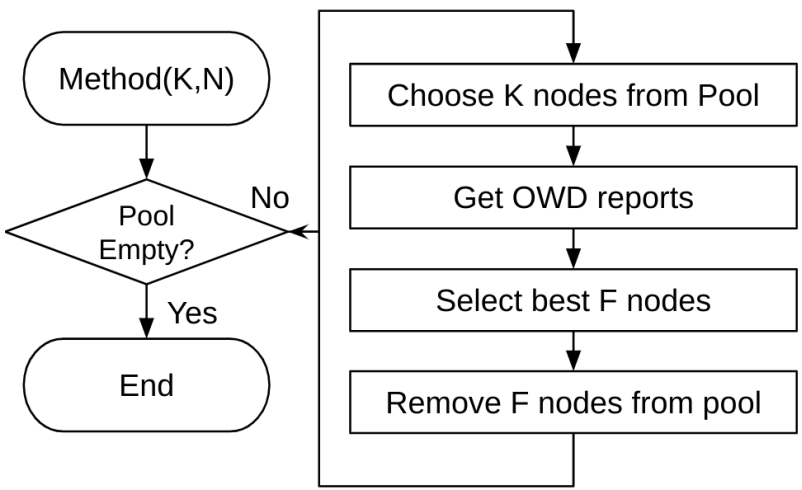
ZMQ Sockets

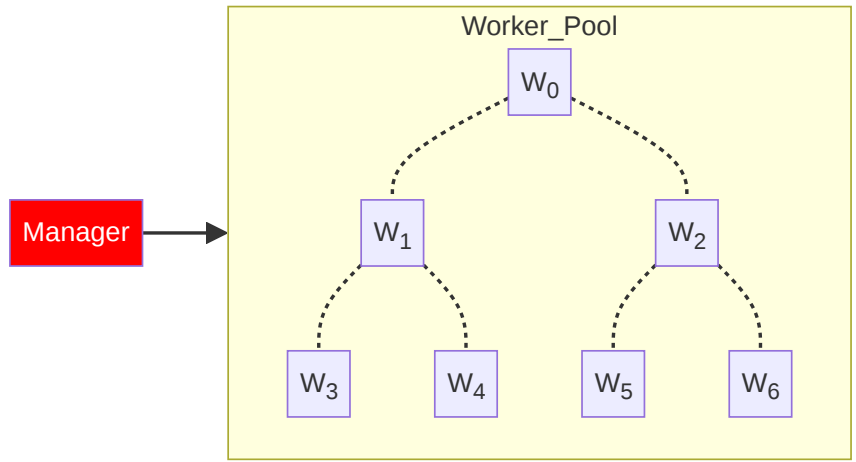
- ZMQ_REQ/ZMQ_REP
  - Send/Receiver order has to be respected
  - Reply remembers only last received address
- Other Sockets:
  - Push/Pull
  - Pub/Sub
  - Pair/Pair
  - Router/Dealer



Testbench and Heuristic

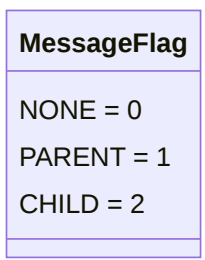
1. Allocate **N** VMs.
2. Run Jasper on Vanilla Setup
    1. Terminate
    2. Store Results
3. Apply Proposed Heuristic

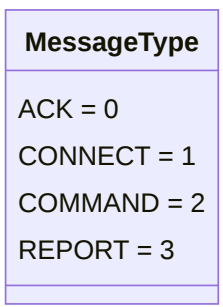


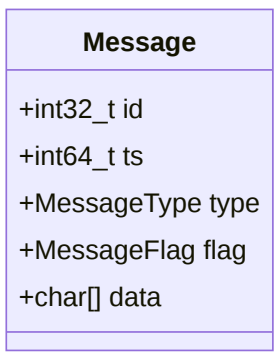


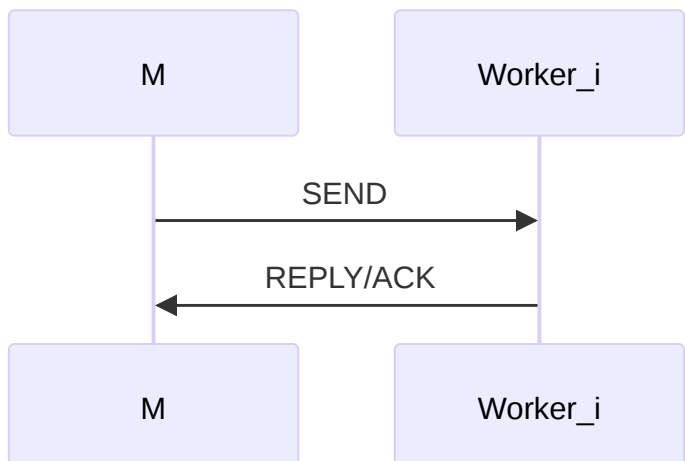
Manager x Worker: Communication

- ZMQ Sockets
- Pairwise send and reply initiated by Manager
- Manager: ZMQ_REQ
- Worker: ZMQ_REP





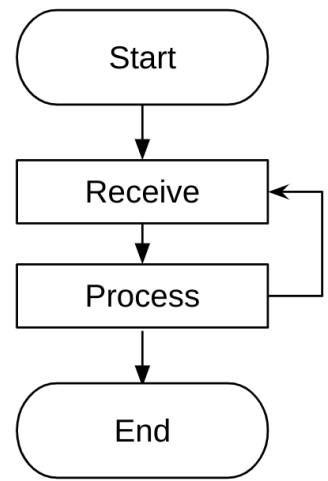




Worker State Machine

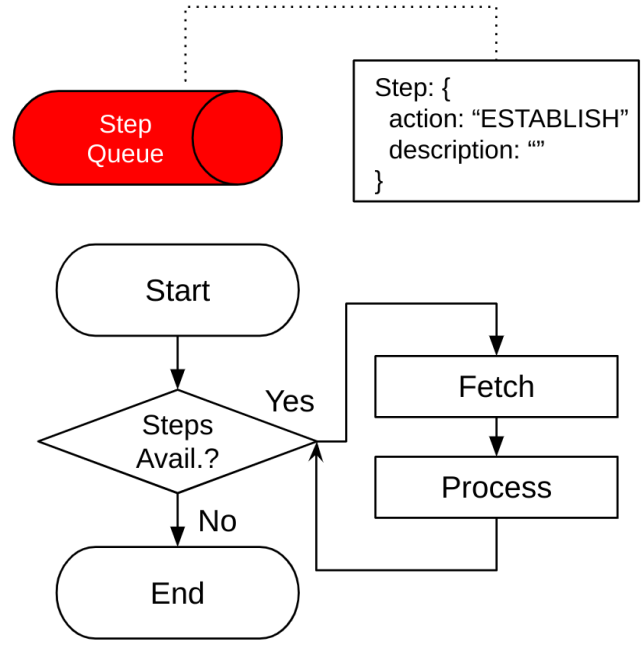
- Workers are reply sockets
- Bind and block on `recv()`
- Process message based on type

```
while(True):
    m = self.recv_message()
    match m.type:
        case CONNECT: self.connectACK(m)
        case COMMAND: self.commandACK(m)
        case REPORT:  self.reportACK(m)
        case _:       raise RuntimeError()
```



Manager State Machine

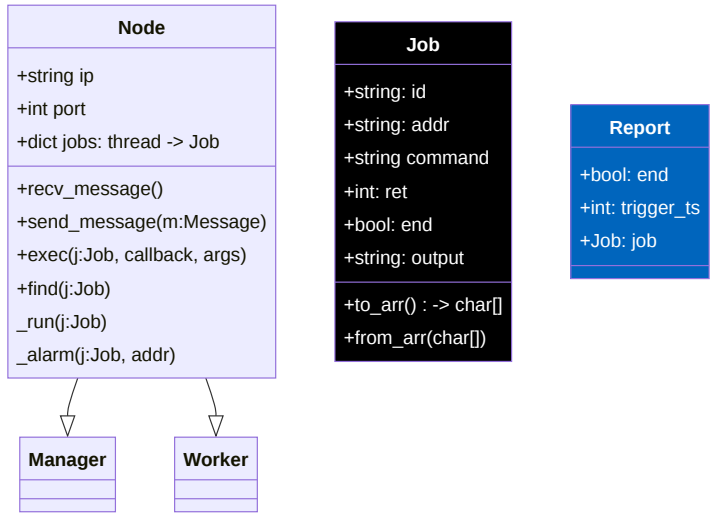
- Manager actively sends requests to workers
- Fetches steps from `step_queue`
- Process steps based on action type

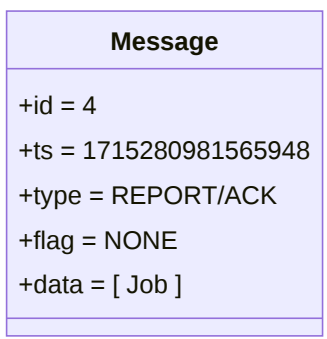


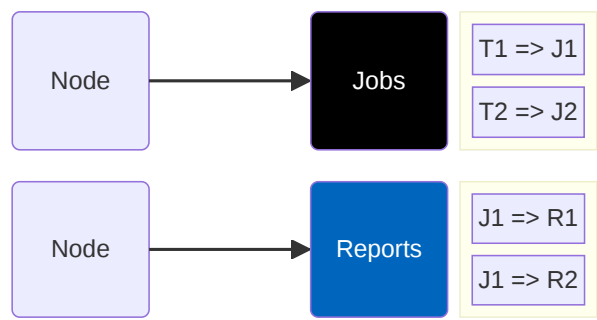

```
while(True):
    step = self.pop_step()
    if not step: break
        match step["action"]:
            case "CONNECT": self.establish()
            case "ROOT":    self.root()
            case "REPORT":  self.report()
            case _:         raise RuntimeError()
```

Manager x Worker: Jobs

- Manager and Workers inherit Node Class
- Nodes own jobs, mapped via a dictionary of threads
- `exec_job(j:Job)`:
  - Runs `j.command` in separate thread
  - stores thread handler in dict
  - thread ultimately modifies the overloaded Job



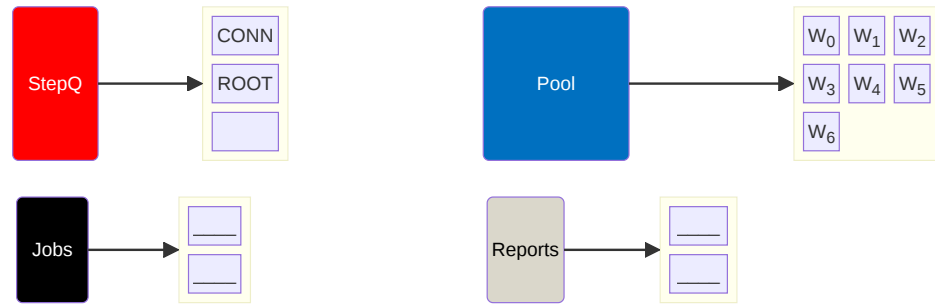


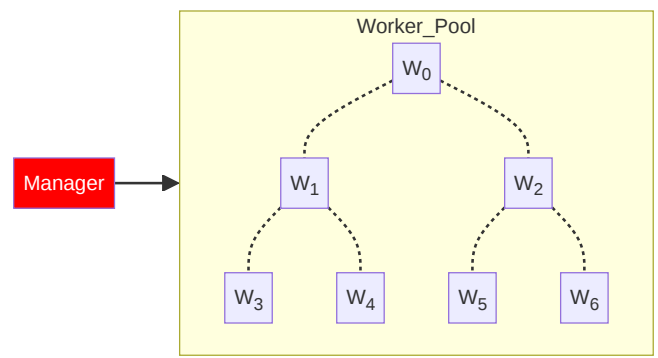


Manager x Worker: Workflow [i = 0]

- Manager reads in YAML *script*
- Populates step queue
- Fetches first step

```yaml
name: DEFAULT
hyperparameter: 0.5
rate: 10
duration: 10
addrs:
  - "localhost:9091"
  - "localhost:9092"
  - "localhost:9093"
  - "localhost:9094"
  - "localhost:9095"
  - "localhost:9096"
steps:
  - action: "CONNECT"
    description: "Establish connection workers."
    data: 0
  - action: "ROOT"
    description: "Choose root among worker nodes."
    data: 0
```

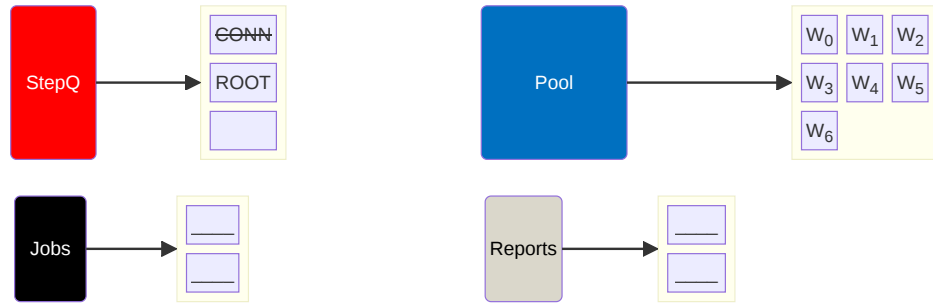




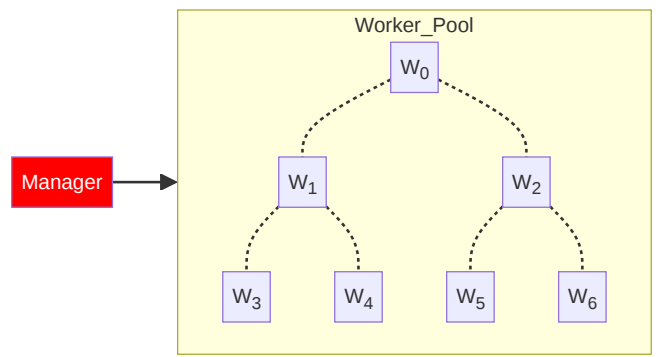
Manager x Worker: Workflow [i = 1]

- ACTION: CONNECT

1. Loops through all workers
    1. Establishes connection
    2. `Send()` CONNECT Messages
    3. `Recv()` ACK Messages
    4. Disconnects



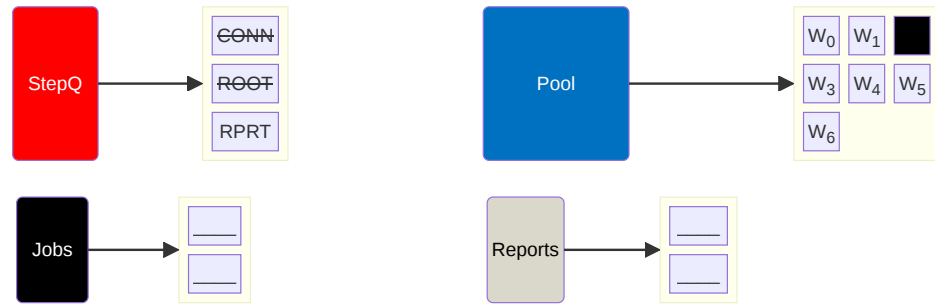
Message
+id = 0
+ts = 1715280981565948
+type = CONNECT
+flag = NONE
+data = [worker_addr_i, host_addr]



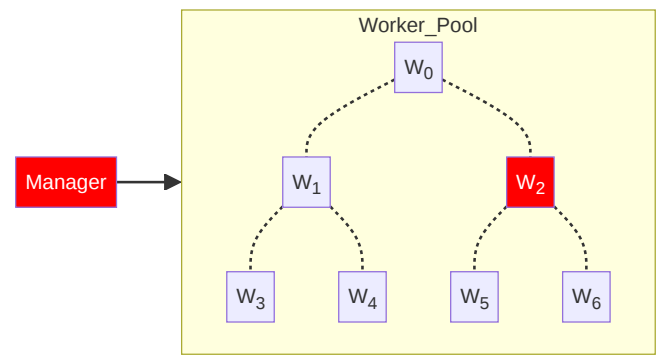
Manager x Worker: Workflow [i = 2]

- ACTION: ROOT

1. Select root from pool *(idx=2)*
2. Commands root to be *Parent*
3. Creates/Pushes: `Step=REPORT`
4. Creates/Pushes: `Report`



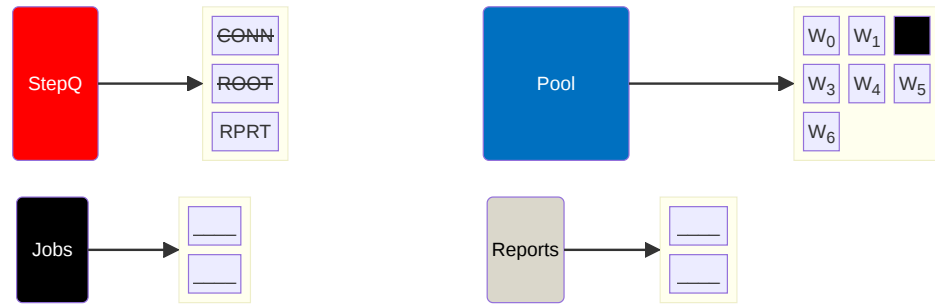
Message
+id = 1
+ts = 1715280981565948
+type = COMMAND
+flag = PARENT
+data = [rate, dur, w_addr_0, w_addr_1, w_addr_3]



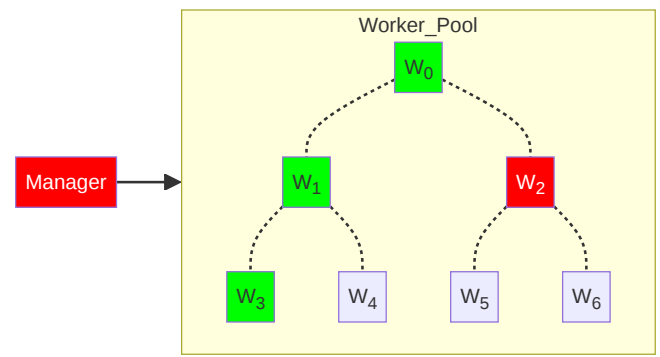
Manager x Worker: Workflow [i = 2.1]

- ACTION: ROOT

1. Connects to workers/children
2. Commands worker to be *Child*
    1. Starts Job: `./child <args`
3. Starts Job: `./parent <args`



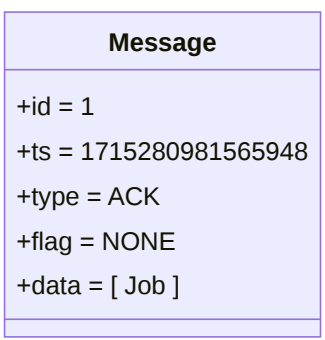
Message
+id = 1
+ts = 1715280981565948
+type = COMMAND
+flag = CHILD
+data = [child_addr_i, host_addr]

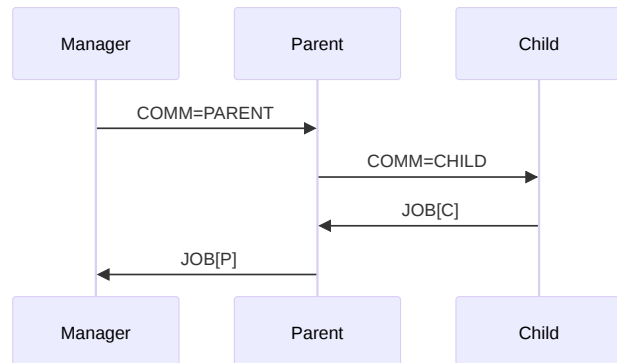


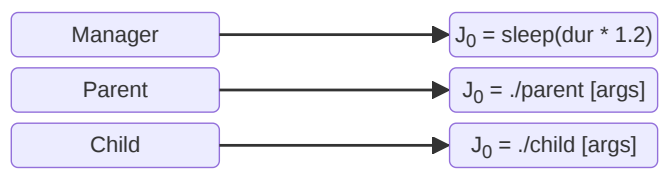
Manager x Worker: Workflow [i = 2.2]

- ACTION: ROOT

1. Manager: Tells root to be *Parent*
2. Parent:
    1. Creates required Parent Job
    2. Contacts Children
    3. Append their Jobs as dependencies
    4. Execs and Replies with Parent Job
3. Manager: Starts Timer Job *(1.2 * duration)*







Manager x Worker: Workflow [i = 3]

- ACTION: REPORT

1. Looks for available jobs
2. Either:

- If jobs still running: Pushes `Step=REPORT`
- Else:
  - Pop Job from dict
  - Looks at Job dependencies
  - Contact owners and ask for report

Manager x Worker: Workflow [i = 3.1]

- ACTION: REPORT

1. Looks for available jobs
2. Either:

- If jobs still running: Pushes `Step=REPORT`
- Else:
  - Pop Job from dict
  - Looks at Job dependencies
  - Contact owners and ask for report