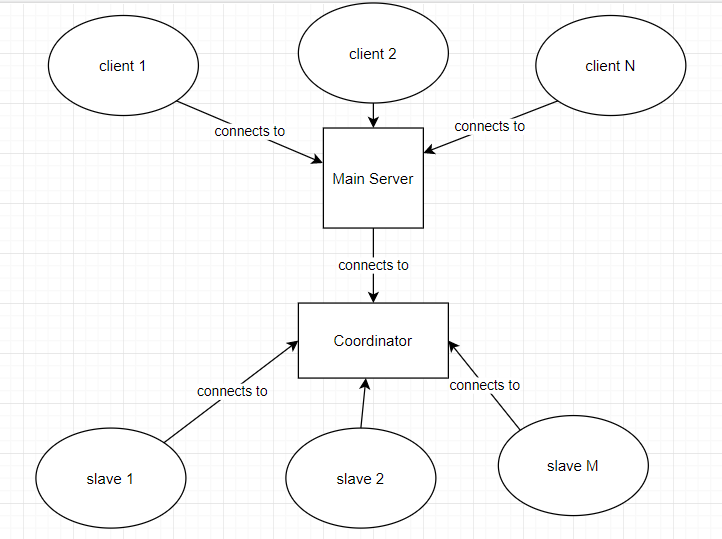
**Software Specification Scope for Distributed Prime Factorization**

Below is the high-level architecture for this project:



**Components**

1. Clients

The clients connect to the main server. The client is the entity which wants to factorize numbers. The client connects to the main server. The main server then asks the client which number they want to factorize. The client provides the number as input, which they send to the main server. The main server then forwards that request to the coordinator.

1. Main Server

Responsible for handling multiple client nodes. Forwards client requests for which numbers they want to factorize to the coordinator. Awaits a response from the coordinator. The coordinator will return a list of all prime factors of the number requested by the client.

1. Coordinator

The coordinator manages a network of slave nodes (workers). The coordinator maintains which prime factors were found for a given client’s number to factorize so far. Once all the prime factors have been found for the client’s requested number, the coordinator returns this information back to the main server, who then forwards it to the client.

1. Slaves

Perform the Pollards Rho algorithm in order to find prime factors of the clients requested number.

**Work Breakdown**

Brian: Coordinator and Documentation

Matt: Slave nodes

John: Main server and client’s (should be most like other projects we’ve worked on)

**Overview of Messages Sent Between Systems:**

A message sent in this system will be of the following format:

MESSAGE\_TYPE|param1 |param2|…|paramN

1. Valid message types: FACTOR\_REQ, FACTOR\_RESP, POLLARD\_REQ, POLLARD\_RESP, CANCEL\_REQ, CANCEL\_RESP, WHOAMI
2. Params can be any value, but should follow the specifics of the message you are sending (specified below…)

**Communication between Main Server and Coordinator (for John):**

When you connect, send the following message to the coordinator (coordinator will not respond to this):

WHOAMI|MAIN\_SERVER

1. Send a message of the following format to the coordinator:

FACTOR\_REQ|clientID|numberToFactor

1. clientID = unique identifier for a client; numberToFactor = self-explanatory…
2. Example: FACTOR\_REQ|34|690
3. For a given client, await a response of the following format from the coordinator:

FACTOR\_RESP|clientID|numberToFactor|prime1,prime2,…,primeM

1. clientId and numberToFactor are same info you sent in the request
2. prime1,prime2,…,primeM are the prime factors of numberToFactor (comma-separated)
3. Example: FACTOR\_RESP|34|690|2,3,5,23
4. Once received, send prime factors back to client who requested

**Communication between Slave Node and Coordinator (for Matt):**

When you connect, send the following message to the coordinator (coordinator will not respond to this):

WHOAMI|SLAVE

1. Coordinator will send a message of the following format to the slave node:

POLLARD\_REQ|slaveConnID|clientID|numberToFactor

1. You don’t need to do anything with ClientID, you will only care about numberToFactor (ClientID is pass thru info needed by coordinator)
2. Example: POLLARD\_REQ|34|690
3. Do Pollards Rho on numberToFactor
4. Return prime factors to coordinator with a message of the following format:

POLLARD\_RESP|slaveConnID|clientID|numberToFactor|prime1,prime2,…,primeN

1. slaveConnID and clientID is pass through info which you already have from the initial message
2. Prime1,prime2,…,primeN is a comma separated list of all prime factors found by pollards rho of numberToFactor (assessible via primes vector of DivFinder)
3. (will happen for some slave nodes) You may also receive a message from the coordinator that indicates the slave node should stop execution (because some other slave node found the prime factors first). Listen for this message, which will be of the following format:

CANCEL\_REQ|slaveConnID

You should then respond back to the coordinator with the following message:

CANCEL\_RESP|slaveConnID

1. In both these messages, slaveConnID is just pass through, you don’t do anything with it.
2. Once you receive the cancel message, stop running pollards rho on the current numberToFactor.

**Example End-to-end Flow of Events and Messages:**

1. Client requests the number 437 to be factorized.
2. Main server receives request from client, and sends a request to the coordinator to factorize:

e.g. (John assigned clientID of 3 for this given client)

FACTOR\_REQ|3|437

1. Coordinator receives request from main server and sends a request to a set of available slave nodes for them to pollards rho factorize. Sends each the following message:

(e.g. slaveConnID assigned by coordinator is 4)

POLLARD\_REQ|4|3|437

1. Slave nodes each receive this request, and each try to factorize 437.
2. One of the slave nodes finds all the prime factors of 437 (437=19\*23), and sends the following message to the coordinator:

POLLARD\_RESP|4|3|437|19,23

1. Coordinator receives this response, and sends cancellation messages to all other slave nodes that it sent POLLARD\_REQ’s to for this given (ClientID, numberToFactor):

(e.g. coordinator assigned slaveConnID of 5 and 6 to slave nodes that need to stop)

CANCEL\_REQ|5

CANCEL\_REQ|6

1. Slave nodes receive cancellation requests, stop execution, and respond with the following messages back to the coordinator:

CANCEL\_RESP|5

CANCEL\_RESP|6

1. Coordinator sends the following message to the main server to respond to main server’s initial request to factorize:

FACTOR\_RESP|3|437|19,23

1. Main server sends prime factors back to the client who requested factorization of 437.

That is all you guys should need for your implementations, below is some descriptions of how the coordinator works….

**How the Coordinator Works**

The coordinator consists of the following key member variables:

1. Vector<int> slaveConns : vector to hold connection id’s of each slave node
2. Vector<int> deadSlaveNodes : vector to hold connection id’s of all slave nodes that died
3. Vector<slaveConnID, tuple<clientID, numberToFactorize, done, cancelled>> jobs : current jobs assigned to slave nodes (done is initially False; set to True when job either completed. Cancelled is initially false; set to true when one of the slave node’s finishes)
4. queue<tuple<clientID, numberToFactorize, vector<int> primes>> completedJobs : queue to hold all jobs that are complete and need to be sent to the main server

The coordinator consists of the following daemon services:

1. Completed Jobs Daemon (CJD)
   1. For all completed jobs in completedJobs queue, sends response back to main server
2. Job Management Daemon (JMD)
   1. Goes through jobs and checks for jobs that have slaveID=-1 (when a job is initially added to the list, it has this slaveID)
      1. Assigns the job a slave node that is available
      2. Sends job to assigned slave node
   2. If a slave node dies, updates entries with slaveID=failedSlaveId and sets it back to -1 (so it can be picked up by a working slave node)
   3. Goes through jobs and checks for jobs that have done = true. For these jobs, removes their entry from jobs table.

**Flow of Events for Coordinator after Initialization**

1. Main server sends request to factorize number:

Receive: FACTOR\_REQ|3|437

1. Upon receipt, add jobs to jobs vector (for now, based on preset # of jobs per client = 3)

Add: <-1, <3, 437, false>> to jobs 3 times

1. JMD looks through list of available slave nodes, and assigns available slave nodes (e.g. slaveID=4,5,6)
2. JMD sends POLLARD\_REQ for each assigned job

Send: POLLARD\_REQ|4|3|437

Send: POLLARD\_REQ|5|3|437

Send: POLLARD\_REQ|6|3|437

1. (slaves do pollards rho, and one returns with result)

Receive: POLLARD\_RESP|4|3|437|19,23

1. Set jobs entry with slaveConnID to done=true
2. Add entry to completedJobs
3. Send cancellation messages to other slave nodes (5 and 6)

Send: CANCEL\_REQ|5

Send: CANCEL\_REQ|6

1. Receive cancel responses from slave nodes:

Receive: CANCEL\_RESP|5

Receive: CANCEL\_RESP|6

1. Set jobs entry with slave id entries with slaveConnID’s 5 and 6 done=true
2. CJD sends message back to main server for completed job

Send: FACTOR\_RESP|3|437|19,23

1. Pop record from queue

Note: a race condition can occur between steps 5 and 7. Multiple nodes could respond with the prime factors, which would cause coordinator to send multiple messages back to main server. To get rid of this issue, do the following in the receive method:

Case POLLARD\_RESP:

If (this job isn’t cancelled):

1. Set jobs entry with slaveConnID to done=true
2. Get other slave node id’s also processing this (clientID, numberToFactor) and set their cancelled flags = true
3. Add record to completedJobs only if job entry’s cancelled flag = false
4. Send cancel requests to other slave nodes that didn’t finish first

^^^ will still work with threaded client’s, just need to mutex steps a-d