DECS SERVER
Autograding Server
(CS744)

Siva prasad reddy 23m0747
David Tarun 23m0774

Coding choices

- In the initial we have used C as our language for the autograding version 1,2,3 but as there are many pred-defined functionalities present in the C++ for our convenience we have shifted to C++ for our final version of Asynchronous architecture.

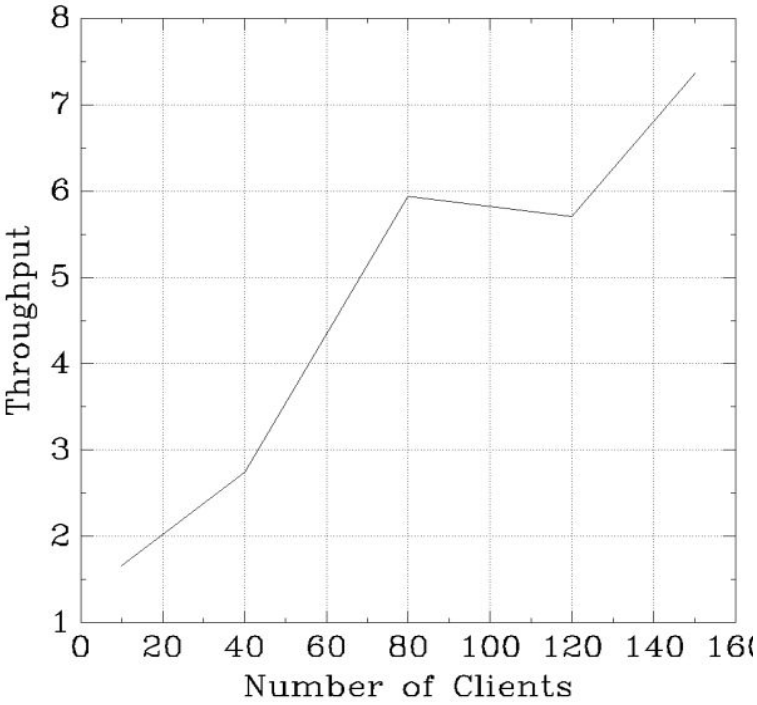
Version-1

Considered clients as 10,40,80,120,150.

The computer in which I ran only offers 8 cores.This version has single thread implementation so it as good as single core processor.

The throughput has increased linearly and we observe that

It became somewhat constant after 80 clients.



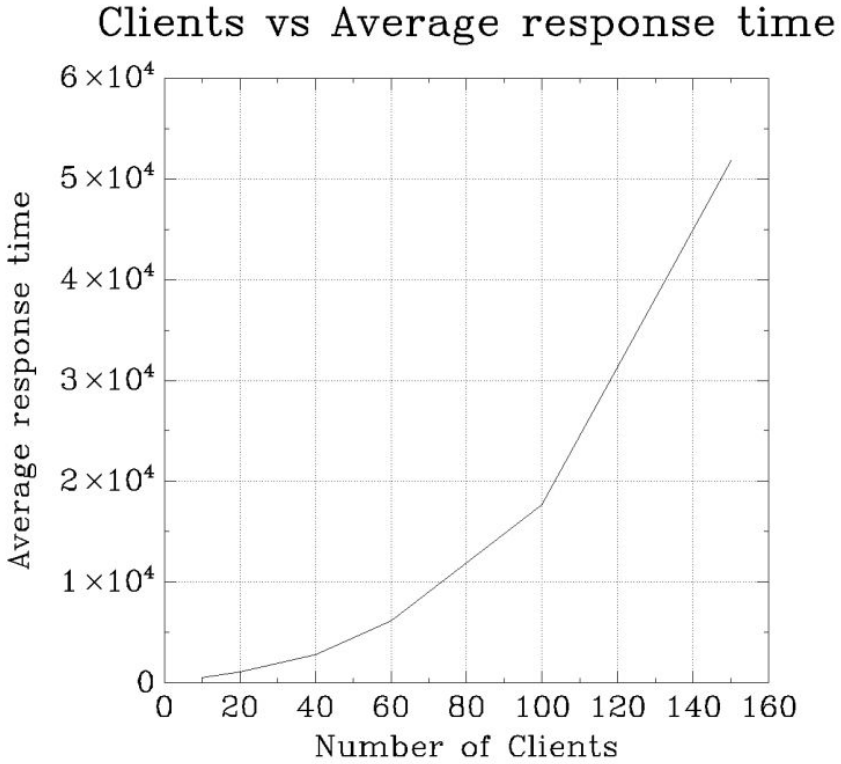
Version-1

Considered clients as 10,40,80,120,150.

The computer in which I ran only offers 8 cores.This version has single thread implementation so it as good as single core processor.

The response time is increasing with the increase in number of clients.

This can be seen because after enough requests some has to wait more time compared to when there is lesser number of clients.



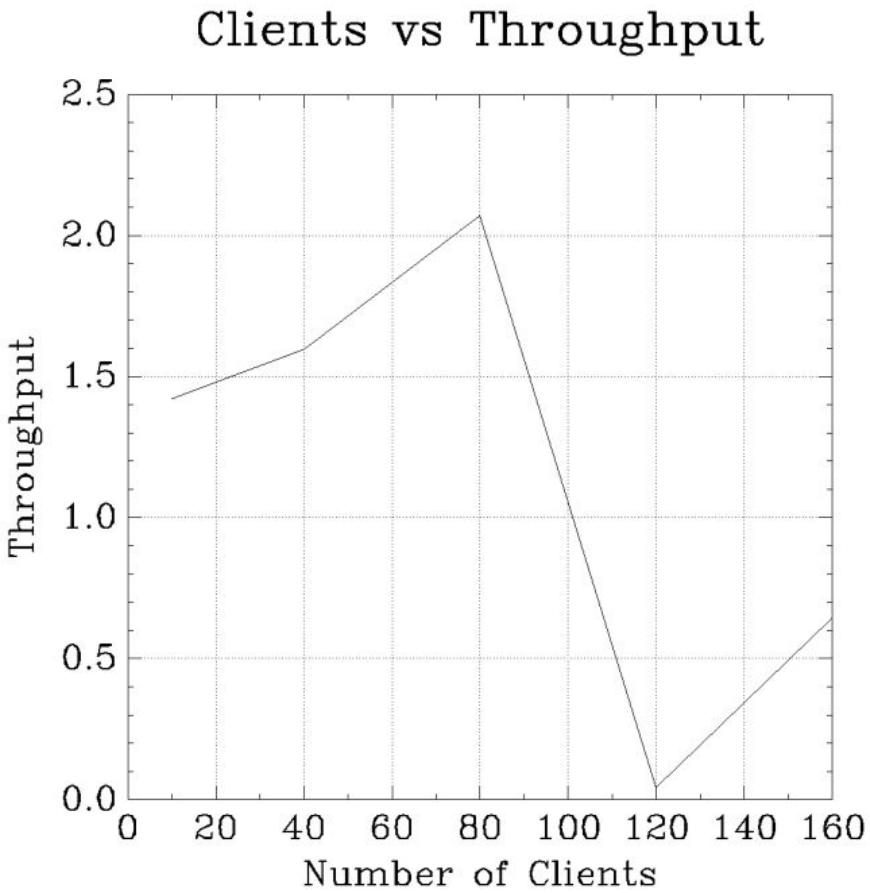
Version-2

Considered clients as 10,40,80,120,150.

The computer in which I ran only offers 8 cores.This version has single thread implementation so it as good as single core processor.

The response time is increasing with the increase in number of clients.

Then we can observe at certain number of clients i.e 80 the throughput has decreased drastically.



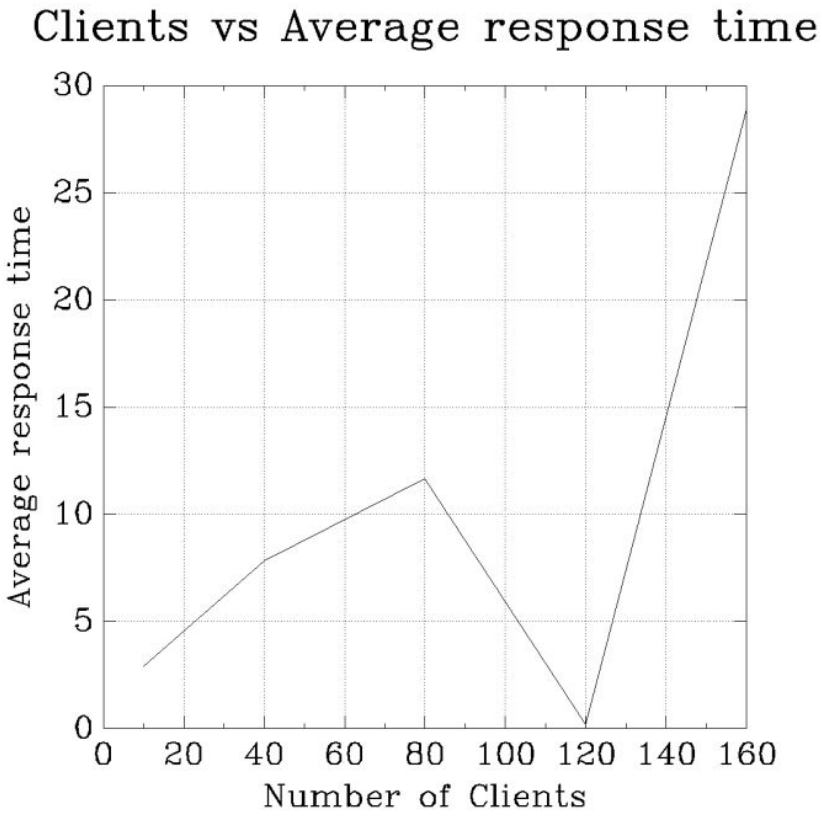
Version-2

Considered clients as 10,40,80,120,150.

The computer in which I ran only offers 8 cores.This version has single thread implementation so it as good as single core processor.

The response time is increasing with the increase in number of clients.

This can be seen because after enough requests some has to wait more time compared to when there is lesser number of clients.At clients=80 we can see the response time is decreasing drastically.

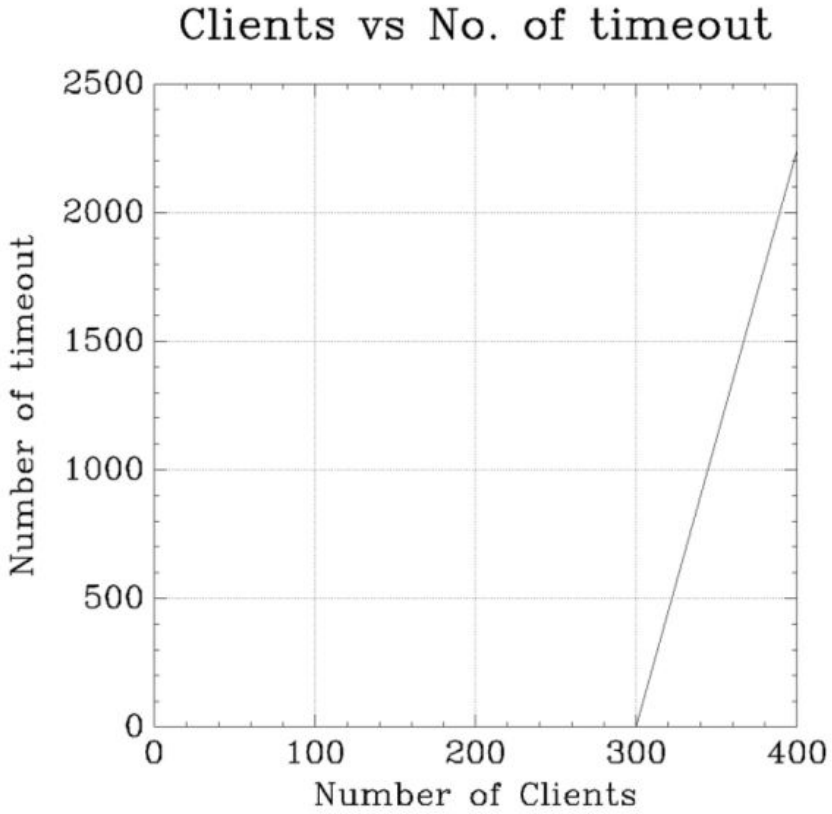


Version-3

Considered clients as 10,20,40,60,100,150,200,250,300,350,400.

The computer in which I ran only offers 8 cores.This version thread pool . I have given 10 as thread pool size.

There are 0 timeouts until 300, but once the number of clients are more than the threshold it suddenly increased after crossing number of clients=300.



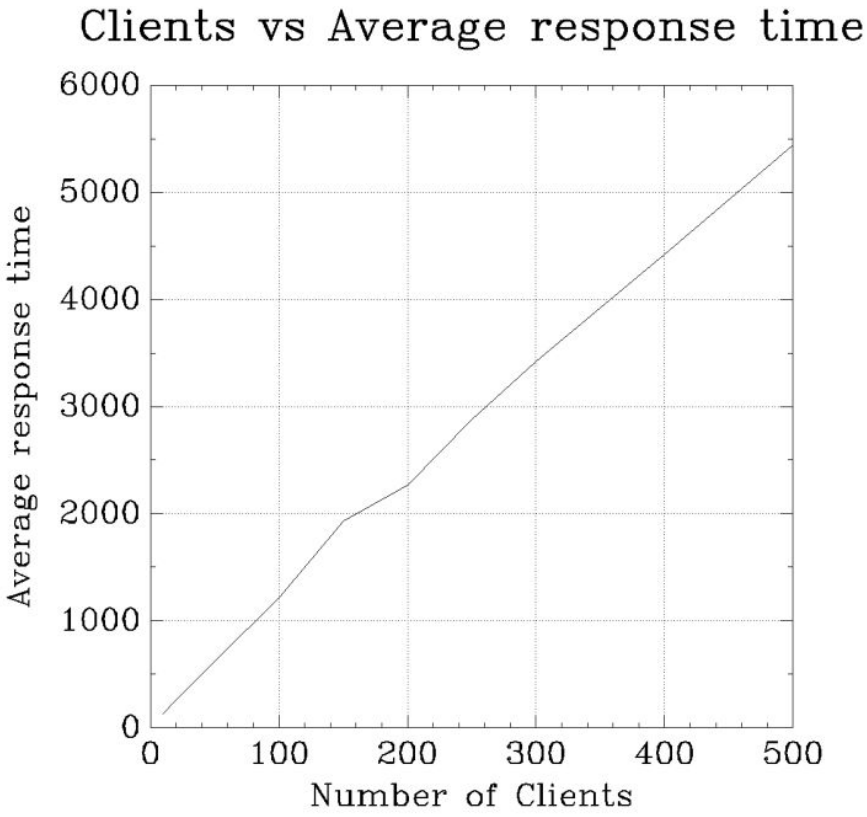
Version-3

Considered clients as 10,20,40,60,100,150,200,250,300,350,400.

The computer in which I ran only offers 8 cores.This version thread pool . I have given 10 as thread pool size.

The response time is increasing with the increase in number of clients.

This can be seen because after enough requests some has to wait more time compared to when there is lesser number of clients.

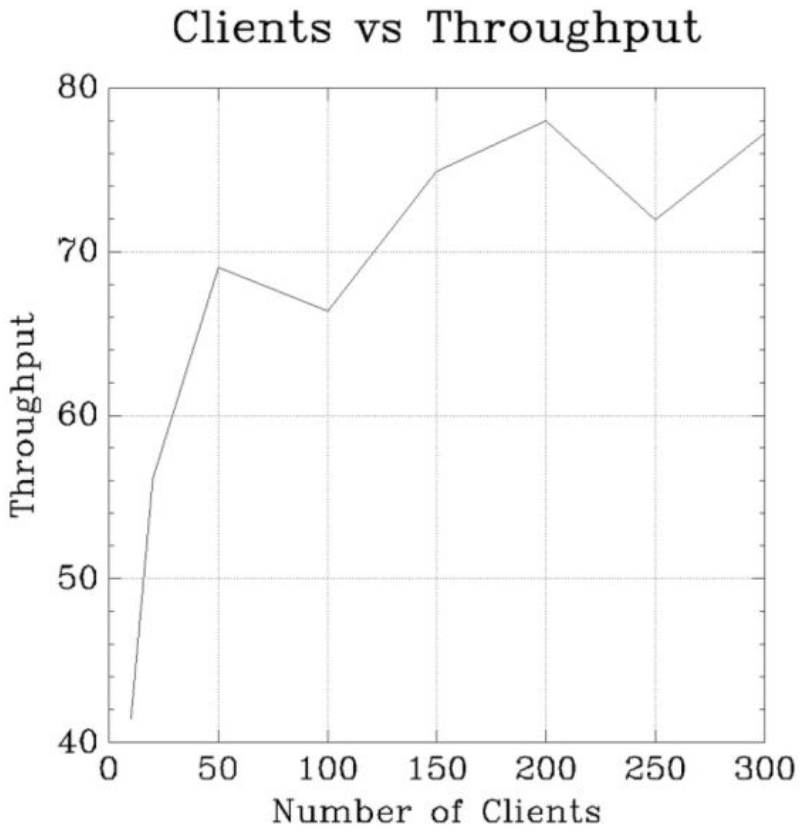


Version-3

Considered clients as 10,20,40,60,100,150,200,250,300,350,400.

The computer in which I ran only offers 8 cores.This version thread pool . I have given 10 as thread pool size.

The throughput is increasing with the increase in number of clients. But at a point of time it will be flattened. Here we can observe that there's sudden change in the rate of increase of throughput after 50.



Asynchronous Grading Architecture

- Clients need not to wait for entire compile-run-grade process to be done.
- The grading can be done asynchronously.
- Server queues all the requests and sends a response to client immediately.
- Client sends check-status requests periodically to server.
- Server checks the status and returns whether it is in queue, or being processed or has completed it’s grading.
- We will be using a unique requestID for every client to identify their requests.

Approach

- We have used file as our queue data structure and stored all the request file names with requestID in a file and we will we fetch a request status by checking these files.

Design for Request ID

- We are using time to create a unique ID every time.
- We are calculating number of seconds since the epoch (Jan1,1970).
- We are generating a random number and appending this number to no.of seconds calculated to bring more uniqueness.
- We will create a file with this requestID to store the data of request. 17014731579886.cpp

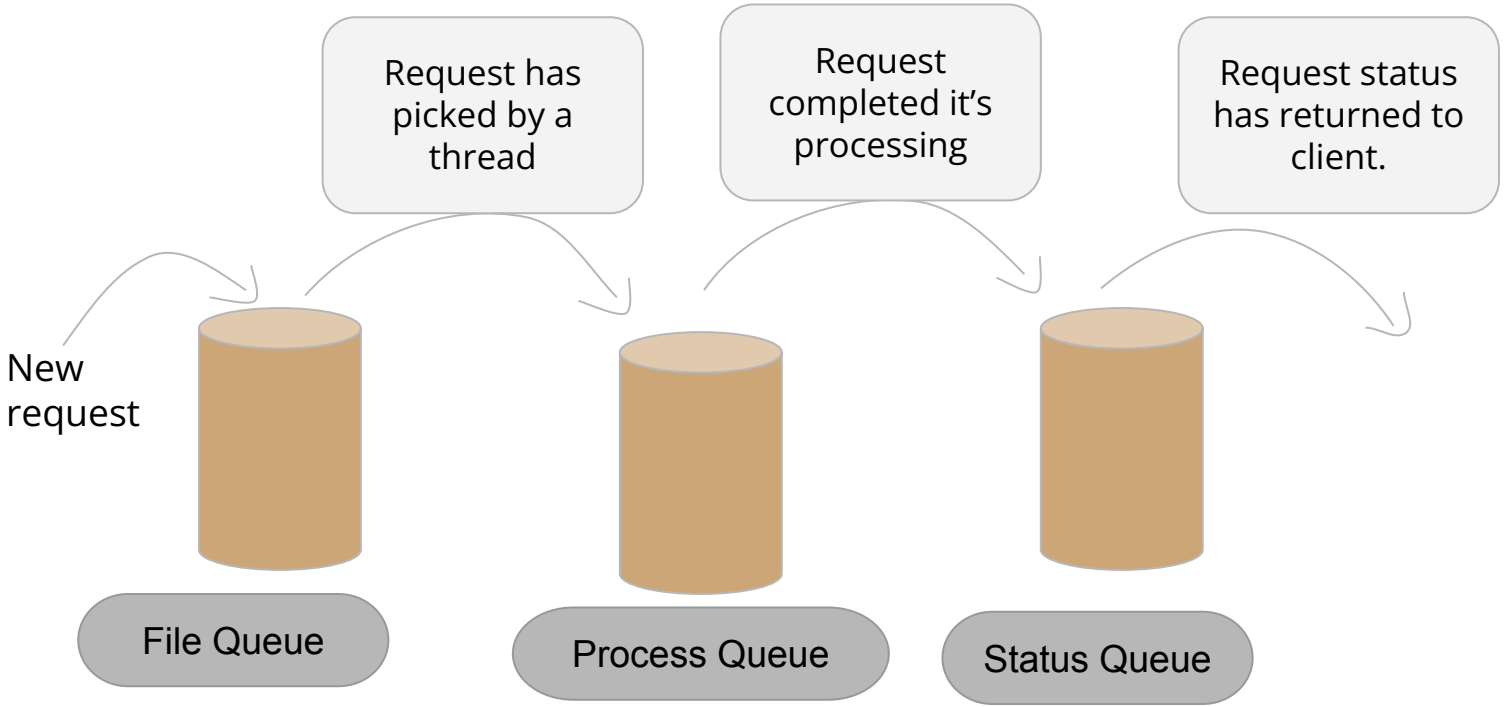
Maintaining Queue

- Maintained the Queue in the form of Files.
- We have created 3 File Queues

    1) File Queue - Initial requests which are waiting for processing

    2) ProcessQueue - Requests which are being processed

    3) Status Queue - Status of all the requests which have completed their processing with output filename of their output.


Request has picked by a thread
Request completed it’s processing
Request status has returned to client.
New request
File Queue
Process Queue
Status Queue

How client get it’s status?

- Client checks it’s status with the requestID given to it at the initial request.
- The requests will be stored with the name of the file generated using unique it’s request ID generated.
- If request is in
  1) fileQueue: return request is still waiting with it’s position in the file.
  2) ProcessQueue : return that request is being processed with it’s position in the ProcessQueue file.
  3) Status Queue: return the output of the request and remove the entry from Status Queue.

  4) If it is not present in any of these then return request not found.

How results are stored?

- The results will be stored in the Status Queue with

  1) Name of the file

  2) Output message of that request i.e. compiler error,program ran etc.

  3) File name where the actual output of this request has stored.

- We will search the request with requestID in filename.
- First the output message will be send to the client following by a file having actual output data.

Thank You