Programming Assignment # 4

TANZIR AHMED, SPRING 2021

Goal

To do the same things as in PA2, but much faster using threads

For instance, the following command asks you to download first 10K data points each for the first 10 patients (note, there are 15 patients each with 15K points):

- This will require 100K messages back-and-forth with the server
- Will take a long time

```
$./client –n 10000 –p 10
```

In addition, you have to put the collected data in 15 separate **histograms** for proper visualization

- Of course, you cannot just print them all in the screen

Like parallelizing anything other problem, the main technique is to:

- Divide the workload (i.e., of getting all these data messages) into smaller portions
- Dedicate each portion to a thread. Start all threads simultaneously. Also, make sure the threads access anything shared in a thread-safe manner (i.e., w/o race conditions)
- Wait for all threads to finish and then optionally, combine the results computed by the threads to a final result

Dividing the Work

Let us work with the given example: downloading 100K data points

We can simply create w threads (let us call them workers), and let each thread work on 100K/w points

The questions are:

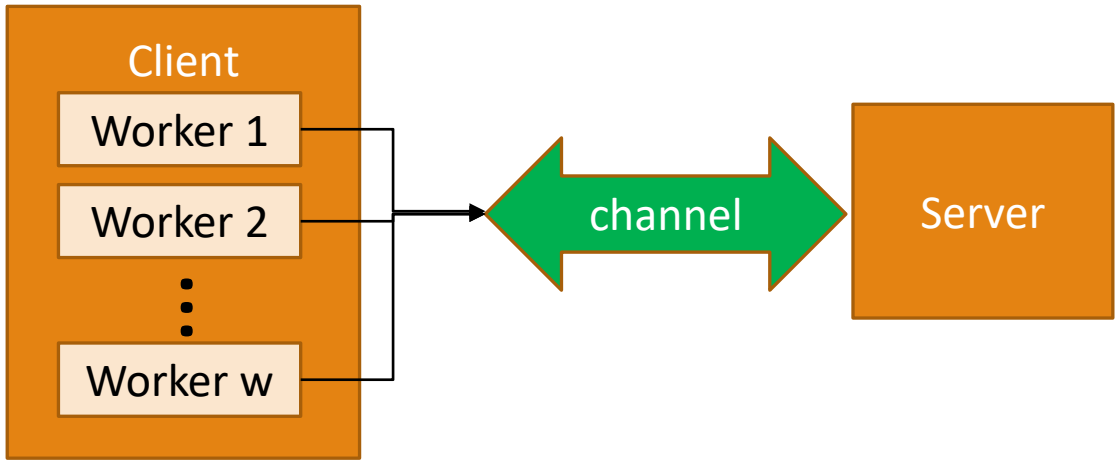
- Will this work?
- Will this give us w-fold speedup?

Answer to both questions is NO:

- The channel is not thread-safe (multiple threads cannot write to it concurrently)
- Even if it worked, the channel would be the main “bottleneck” making everything slow

Solution: Create ONE channel per ONE worker

- Create w new channels, 1 for each worker



Dividing the Work – One channel/worker

Will this achieve w-fold speed up?

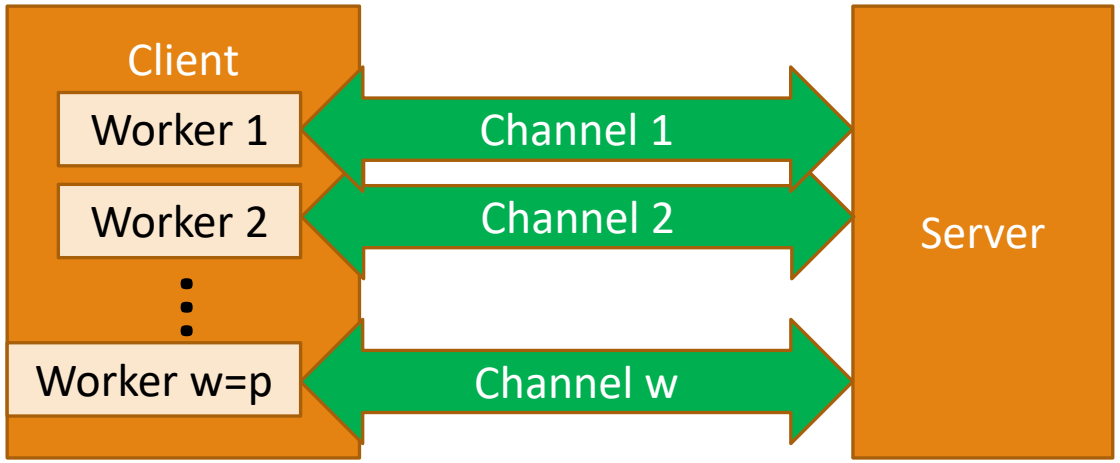
- Ideally, yes
- But, in practice, NO

The problem is in how the server responds to each request

- Remember, there is random sleep in each request
- As a result, some threads may take much longer than others

This is a general problem beyond just this PA

- Some threads finish much later than others
- Such threads are called “stragglers”

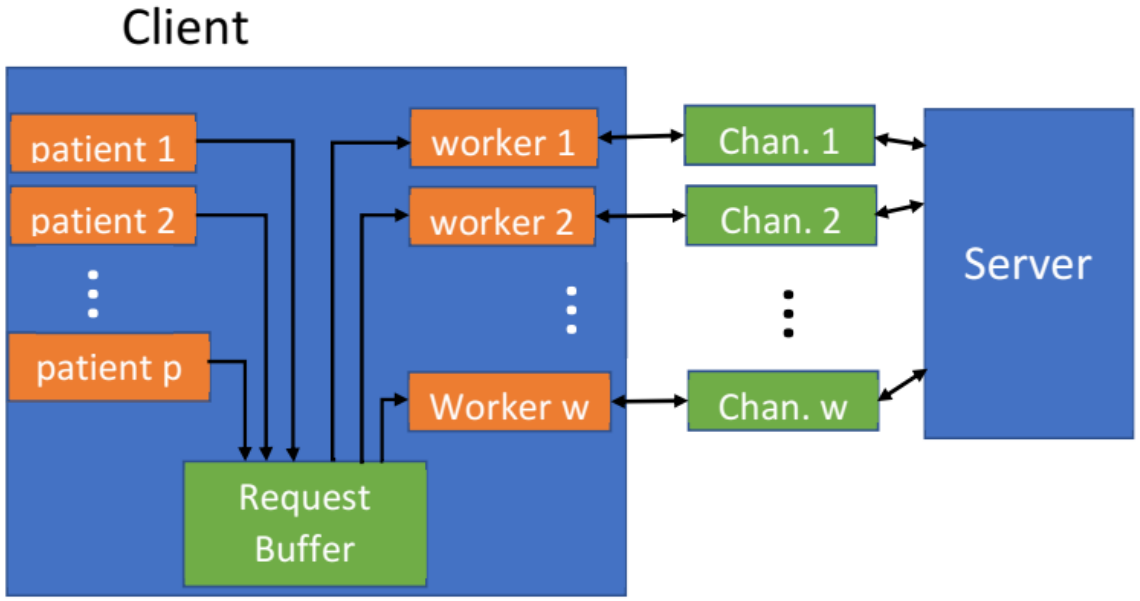



```
tanzir@marcelago:~/PA4/naivesol$ ./client
Thread 7: Took 31 seconds and 456611 micro seconds
Thread 3: Took 31 seconds and 680019 micro seconds
Thread 1: Took 31 seconds and 684867 micro seconds
Thread 2: Took 31 seconds and 743521 micro seconds
Thread 5: Took 31 seconds and 754048 micro seconds
Thread 9: Took 31 seconds and 787876 micro seconds
Thread 6: Took 31 seconds and 866526 micro seconds
Thread 4: Took 31 seconds and 963686 micro seconds
Thread 8: Took 32 seconds and 107974 micro seconds
Thread 10: Took 32 seconds and 793209 micro seconds
Histogram collection is empty
Took 32 seconds and 886772 micro seconds
```

Avoiding Stragglers Problem

Instead we will use something called “Work Stealing”

- The idea is to generating the work requests and push them in a buffer
- The worker threads then pops those requests 1 at a time, download the request, and then repeat as long as the buffer is not completely depleted
- The effect is that all threads then finish at about the same time
- By that time, some threads may have done more downloads than the others
  - That is fine, because then the workers do not stay idle by “stealing” work from the buffer



Eliminating Race Condition

Implementing “Work Stealing” requires a “BoundedBuffer” which needs to be:

- Thread-safe or Race condition free. Note that queue::push()/pop() are not thread-safe
- Bounded to a maximum size so that too many push()es to do not make it infinite in size
  - If the buffer is full, the push()es will have to wait
- Protected against underflow so that too many pop()s do not deplete it below size 0
  - If the buffer is empty, more pop()s will wait

Since each worker now access all p Histograms simultaneously, there is a race condition there as well, which must be eliminated