CS 555: DISTRIBUTED SYSTEMS
Department of Computer Science
Colorado State University

FALL 2025
URL: http://www.cs.colostate.edu/~csx55
Professor: Shrideep Pallickara

PROGRAMMING ASSIGNMENT 2

DISTRIBUTED LOAD BALANCING OF COMPUTATIONAL TASKS USING THREAD POOLS

Version 1.0

DUE DATE: Wednesday, October 8th, 2025 @ 5:00 pm

1 Objective

The objective of this assignment is to get you comfortable with threads and synchronization mechanisms. Another objective of this assignment is to introduce the role that data structures, locking mechanisms, and synchronization play in designing concurrent programs.

Generative AI Use and Consequences
Use of AI tools such as ChatGPT, Claude, Github Co-Pilot, or anything of their kind to write or "improve" your code or written work at *any* stage is prohibited; this includes the ideation phase. It is your responsibility to ensure that you don't have the GitHub Co-Pilot extension installed in your IDE; assignment solutions generated by Co-Pilot aren't written by you. Turning in code or an essay written by generative AI tools will be treated as turning in work created by someone else, namely an act of plagiarism and/or cheating. At a minimum, this will result in a 100% deduction (i.e., you will receive a −10/10). To ensure fairness and maintain integrity, grading will also include code reviews, interviews, and on-the-spot code modifications.

Ultimately, you will get out of the class what you put in. Simply copying and pasting code from generative AI tools is not only unethical, it robs you of the chance to learn. Here are four reasons why these generative AI tools undercuts your own education:

1. They take away the struggle that leads to understanding. They rob you of the ability to think and learn the concepts for yourself. Solving problems yourself is how concepts stick. If the AI does the work, what's left for you to learn?

2. You will struggle with the in-classroom quizzes and exams where you will not have access to these tools.

3. Yes, AI tools will become an important part of a software engineer's workflow. But to use them effectively later, you first need solid expertise in the subject matter; and, that only comes from practicing *without* them.

4. These tools are prone to generating imperfect or even incorrect solutions, so trusting them blindly can lead to bad consequences.

2 Grading

This assignment will account for **10 points** towards your cumulative course grade. The components of this assignment, and the points breakdown are listed in the remainder of the text. This assignment is to be done individually. The lowest score that you can get for this assignment is 0. The deductions will not result in a negative score.

3 Setting

There are a set of N computational nodes in the system. These nodes are arranged in a ring topology. You are free to reuse code you developed in HW1 for the registry node and for setting up the overlay (in this case a ring topology). Each node produces a random number of tasks; the system is collectively responsible for ensuring that the tasks are completed, and that every node has performed "about" the same amount of work

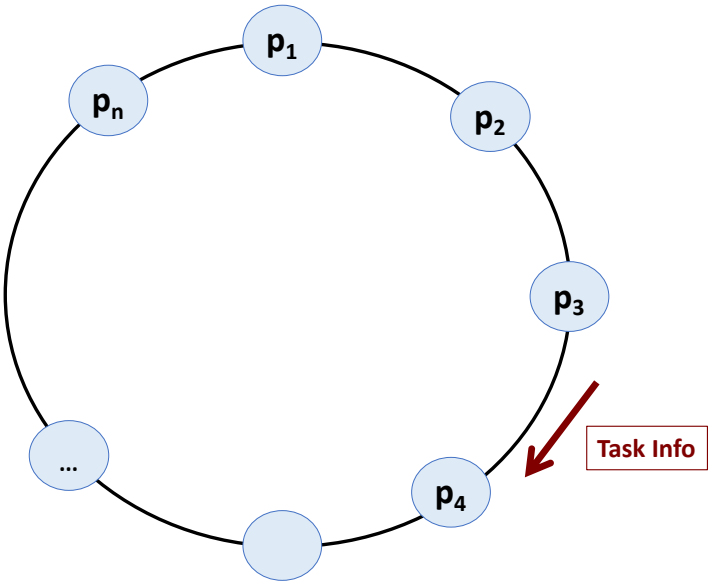

Figure 1: Computational nodes organized in a ring topology.

4 Components

There are two primary components in the system: the singular registry and multiple computational nodes. The registry helps with the construction of the overlay for the computational nodes. The registry is also where information about the execution of tasks at each node is collated and printed out. You have the freedom to design your own wire formats to facilitate communications between elements in the system.

4.1 The Registry

There is exactly one registry in the system. The registry provides the following functions:

A. Allows computing nodes to register themselves. This is performed when a computing node starts up for the first time.
B. Enables the construction of the overlay by orchestrating connections that a computing node initiates with other computing nodes in the system. Based on its knowledge of the computing nodes (through function A) the registry informs messaging nodes about the other computing nodes that they should connect to.

4.2 The Computational Nodes

Each node performs a set of computational rounds. In each round, every node generates a set of tasks and adds them to its task queue. The number of tasks that that a node generates is based on a random number generator that is constrained to produce a number between 1-1000; consequently, each node is likely to produce a different number of tasks in each round.

Each node performs the following set of actions.

1. Relies on a random number generator to create between 1-1000 tasks.
2. Once the node completes its set of generated tasks, it starts a new round by repeating step 1.
3. Each node is expected to complete a configurable number of rounds (**`number-of-rounds`**).
4. Since each node generates a different random number, there will be skews in the number of generated tasks. Alleviating these imbalances is one of the primary goals of this assignment.
5. Leverages the task queue and the thread pool.
    a. Tasks are managed in a task queue. New tasks are added to the tail of the queue and retrieved from the head of the queue.
    b. Threads within the thread pool retrieve tasks one at a time. A particular thread retrieves a new task from the task queue only if the current task was successfully completed by that thread. If the thread pool size is 8; the number of tasks that execute concurrently can be 8.

5 Key Elements

5.1 Thread Pools

Each computational node will encapsulate a thread pool. Broadly a thread pool encapsulates a fixed set of threads. Threads within the thread pool are initialized exactly once. In particular, the thread pool will create a set of threads at start-up, and these threads will be used for the entire life cycle of the node.

Individual threads within the thread pool remain in the run state (i.e., they never exit their run() method) till such time that the process is ready to terminate. For the purposes of this assignment, the number of threads in the thread pool can be anywhere between 2 to 16.

Tasks that must executed at a node must be added to the task queue. All threads within the thread pool have access to the task queue. Tasks retrievals must be FIFO i.e., tasks would be added to the tail of the queue and retrieved from the head of the queue. Each task must be executed by one of the threads within the thread pool.

The task queue is backed by a data structure of your choosing. A key requirement is that your task queue must include synchronization and/or locking mechanisms to facilitate concurrent, thread-safe retrieval and completion of tasks. Incorrect synchronization primitives will result in disappearing/missed tasks (that will manifest as incorrect results) or stalls (programs may take an inordinately long time to wrap up the computations).

Thread pool sizes must be configurable during startup. As the number of threads in the thread pool increase you should see a commensurate increase in task throughput. Note that as you increase the number of threads beyond a certain threshold, the execution times will actually increase as context-switching overheads start to dominate. But you should be seeing a clear increase in task throughput as the number of threads in your thread pool increases from 2 to 16.

5.2 Task

The computation performed by each task is similar to how mining is performed in some cryptocurrencies. Note that since the task's purpose is to induce computation load, we provided the code that does this.

Mining bitcoins refers to the activity of adding valid blocks to the blockchain. This involves finding a *nonce* that, when combined with the block, produces a hash smaller than the *difficulty target*. SHA256 is a one-way cryptographic hashing function used in bitcoin. SHA256 takes a byte array of any size as input and produces an output byte array of 32 bytes (256 bits).

In the code that we have provided, the *difficulty target* is expressed as the number of leading zeros of the hash, and the block is constituted by the Task object. The *mine* method () finds a *nonce* so that SHA256(task.getBytes()) has at least 17 leading zeros. Finding the nonce becomes exponentially harder as the number of leading zeros increases, thus it constitutes the **proof of work**. Verifying the correctness of a block is fast because it requires a single computation of SHA256.

5.3 Load Migration

This assignment requires nodes to coordinate among themselves to load balance tasks. Note that since each node generates a random number of tasks, task skews are very likely. Depending on the distribution of tasks, the nodes should initiate pair-wise load balancing maneuvers to balance workload.

After a node generates a set of tasks, it sends a message around this ring (in the clockwise direction) identifying the number of tasks that it has generated. Each node uses this information to track the total number of tasks within the system.

1. Given that there are N nodes in the system, this allows each node to independently compute the total number of tasks in the system. Also, each node is able to estimate the work (number of tasks) that it should complete in a load balanced system.
2. Each node uses this information to reapportion workloads in a pair-wise fashion. Consider two nodes **A** (lightly loaded) and **B** (heavily loaded). **A** might either pull tasks from **B**; alternatively, **B** might push some tasks to **A**.
3. One of the goals of this assignment is to damp oscillatory behavior. That is, we cannot have workloads migrating back-and-forth between nodes. One way to accomplish this is that tasks that have migrated to another node are no longer eligible to be migrated to some other node as part of rebalancing maneuvers. Another refinement to achieve better load-balancing is to perform task migrations – be it a push or a pull – in small batches (minimum of 10 tasks).

5.4 Outputs

The registry node should be used to retrieve task completion statistics from each node. Here is an example of output with 10 nodes and 100 rounds.

	Number of generated tasks	Number of pulled tasks	Number of pushed tasks	Number of completed tasks	Percent of total tasks performed
Node1	52989	21650	23510	51129	10.09917652
Node2	50218	23590	23110	50698	10.01404392
Node3	51949	22220	23100	51069	10.08732512
Node4	52257	22470	23800	50927	10.05927679
Node5	51557	22300	23140	50717	10.01779686
Node6	49142	22130	21170	50102	9.896319941
Node7	48545	22750	21000	50295	9.934441967
Node8	49483	22720	21790	50413	9.957749734
Node9	45173	25660	20620	50213	9.918245044
Node10	54956	20460	24710	50706	10.0156241
Total	506269	225950	225950	506269	100

- Number of generated tasks: This contains information about the number of tasks that were generated by each node across the specified rounds.
- Number of pulled tasks: This corresponds to the total number of tasks pulled by the node i.e., the total number of completed tasks that did not "originate" at the node in question.
- Number of pushed tasks: This corresponds to the number of tasks that were offloaded by this node to some other node.
- Number of completed tasks: This corresponds to the total number of tasks that were completed at any given node.
- Percent of total tasks performed: This identifies the percentage of ***all*** tasks that were completed at the node in question.

Note that several relationships hold across these variables. For example, the number of completed tasks is equal to the number of generated tasks plus the number of pulled tasks minus the number of pushed tasks.

All the mined tasks should also be printed by the node on which they are mined (do not print them on the Registry). Each task should be printed on a separate line. Use the toString() method of the Task class. We will use this during grading to verify correctness, recency (i.e., nothing was hardcoded), and the IP addresses where your blocks were generated. The total number of printed tasks should be equal to the total number of completed tasks that is reported in the table.

Note that like in HW1, the table must be printed in **space-separated form**. The header of the table is not required. The first column must contain <ip>:<port> for each node. For example, if Node1 has ip=192.168.1.10 and port=5001, then the first row of the table must be:

192.168.1.10:5001 52989 21650 23510 51129 10.09917652

6 Parameters and Program Execution

Here are the arguments that will be specified during program execution.

`java csx55.threads.Registry portNum`

Commands issued at the registry:
`setup-overlay thread-pool-size`
`start number-of-rounds`

`java csx55.threads.ComputeNode registry-host registry-portNum`

`thread-pool-size`	This parameter refers to the thread pool size and represents the number of threads that will be created upon start up. Once started, the threads must never exit their `run()` method till such time that the entire computation has been completed.
`number-of-rounds`	This corresponds to the number of rounds of task generation performed by each node in the system

7 Points distribution:

2 point	Initialization of the ring topology overlay. This means that the registry and computational nodes have exchanged messages that facilitates creation of the overlay.
2 points	Correct, non-stalling execution of program in a multi-threaded environment.
2 points	Load balancing of the workloads. Each node should have roughly equal number (within 10% of the average) of computational tasks that were performed.
2 points	Program executes correctly with thread pools of different sizes.
2 points	Program executes commensurately faster with 10 threads within each pool than it does for 1 thread.

Threads in a thread-pool must be created only once. There is a -1 point penalty for violating this rule.

8 Third-party libraries and restrictions:

The assignment must be implemented using the core packages in Java. However, you cannot use thread pool implementations that are available in the Java language library. You are not allowed to use *any* external jar files. You can discuss the project with your peers at the architectural level, but the project implementation is an individual effort.

9 Milestones:

You have 4 weeks to complete this assignment. The weekly milestones below correspond to what you should be able to complete at the end of every week.

MILESTONE 1 [WEEK 1**]**: The ring topology is fully set up and includes support for configuration of a thread pool of specific size is complete.

MILESTONE 2 [WEEK 2]: Robust prototype implementation of the thread pool is complete, and individual task executions are completed and confined to a single thread.

MILESTONE 3 [WEEK 3]: Identification of load skews across different nodes is complete. Preliminary support for load balancing maneuvers via a pairwise pull or push is complete. Checks to see if the system load exhibits oscillatory behavior is complete.

MILESTONE 4 [WEEK 4]: Iron out any wrinkles that may preclude you from always getting the correct outputs.

10 What to Submit

Use using **CANVAS** to submit a single .tar file that contains:

- all the Java files related to the assignment (please document your code)
- the `build.gradle` file you use to build your assignment
- a README.txt file containing a manifest of your files and any information you feel the GTA needs to grade your program.

Filename Convention: You may call your support classes anything you like. All classes should reside in a package called **`csx55.threads`**. The archive file should be named as <FirstName>-<LastName>-HW2.tar. For example, if you are Cameron Doe then the tar file should be named Cameron-Doe-HW2.tar.

11 Change History

Version	Date	Change
1.0	9/10/2025	First public release of the assignment.