

Discussion 9: Queueing Theory & Distributed (file)systems

April 24, 2026

Contents

1	Queueing Theory	2
1.1	Little's law.	2
1.2	M/M/1 queues.	3
2	Reliability	5
3	Transactions	6
3.1	Concept Check	6
4	Journaling	8
5	Distributed Data Processing	9
5.1	Concept Check	9
5.2	Leetcode golf	9
6	Distributed Systems	10
6.1	Concept Check	10
6.2	Two Phase Commit	10

1 Queueing Theory

In this question, we'll derive the formulas for an M/M/1 queue, and think about the properties as it relates to a real-world system. First, we will define the notation for a simple queueing system.

Let

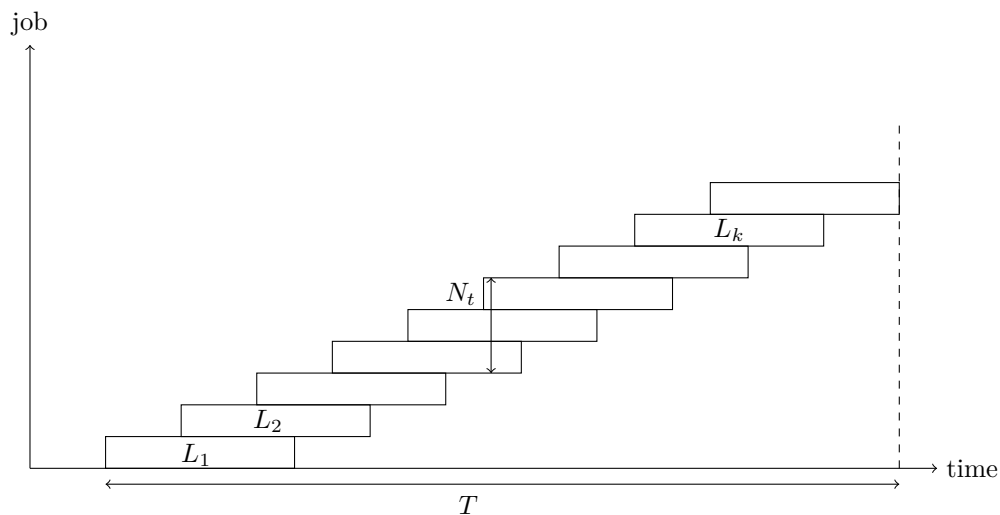
- λ be the arrival rate
- μ be the mean service rate
- T_q be the average time in queue
- T_s be the average service time

1.1 Little's law.

For this first part, we will derive Little's law. Recall that Little's law states that

$$\bar{N} = \bar{\lambda} \cdot \bar{L}$$

i.e., the average number of jobs in the system, \bar{N} is the product of the average arrival rate $\bar{\lambda}$ and the average response time \bar{L} .



1. Let L_i be the response time for job i . Write \bar{N} in terms of $\sum L_i$ and T .

$$\bar{N} = \frac{\text{total area under } N_t}{T} = \frac{\sum_i L_i}{T}$$

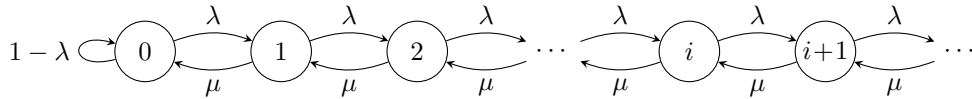
2. Derive Little's law, $\bar{N} = \bar{\lambda} \cdot \bar{L}$. It may help to introduce a factor of N .

Recall that $\bar{\lambda} = \frac{N}{T}$.

$$\bar{N} = \frac{N}{T} \cdot \frac{\sum_i L_i}{N} = \bar{\lambda} \cdot \bar{L}$$

1.2 M/M/1 queues.

Recall that the "M" in M/M/1 stands for Markovian, or memorylessness. We say this as the arrivals and services do not remember past events, and therefore can be modelled as an exponential distribution. Due to this memoryless property, we can model the number of jobs in the system as an infinite birth-death chain, where the forward transition probability (the arrival rate) of λ and the backward transition probability (the service rate) of μ .



Additionally, recall that when $\lambda < \mu$, this Markov chain has a stationary distribution. We can compute this stationary distribution by satisfying the local balance equations for a reversible chain,

$$\pi(i)P(i, i+1) = \pi(i+1)P(i+1, i)$$

1. What does state i represent? What about $\pi(i)$?

State i represents when the system has i total jobs, including queued and running jobs. $\pi(i)$ is the probability the system is in state i in the stationary distribution.

2. Let $\rho = \frac{\lambda}{\mu}$ be the system utilisation. Show that $\pi(0) = 1 - \rho$, and identify the distribution of $\pi(i)$.

Apply the local detailed balance equations.

$$\begin{aligned} \pi(i)\lambda &= \pi(i+1)\mu \implies \pi(i+1) = \rho\pi(i) \\ &\implies \pi(i) = \rho^i\pi(0) \end{aligned}$$

Any probability distribution must sum to 1, so

$$\sum_{i=0}^{\infty} \pi(i) = 1 \implies \sum_{i=0}^{\infty} \rho^i \pi(0) = 1 \implies \boxed{\pi(0) = 1 - \rho}$$

$$\pi(i) = \rho^i(1 - \rho) \sim \text{Geometric}(1 - \rho) - 1$$

3. What does the system look like when $\rho < 1$ is small? What does this look like in terms of the "burstiness" of the CPU?

The system is likely empty, as the system is able to service jobs faster than they arrive. The CPU is mostly idle with short bursts of work occasionally.

4. What about when $\rho \rightarrow 1$? What happens to the mean length of the queue in this case?

The mean length of the queue $\mathbb{E}[L] \rightarrow \infty$. The Markov chain is said to be "null-recurrent" when $\rho = 1$ and the expected time to reach back to an empty system (for the second time) is infinity. Intuitively, as the arrival rate and service rate are the same, there is no "restoring" force back to the empty state, so the queue length grows unboundedly.

5. Let L be the number of jobs in the system. Show that

$$\mathbb{E}[L] = \frac{\rho}{1 - \rho}$$

You may use the fact that $\sum_{n=0}^{\infty} n\rho^n = \frac{\rho}{(1-\rho)^2}$.

$$\mathbb{E}[L] = \sum_{n=0}^{\infty} n \cdot \pi(n) = (1 - \rho) \sum_{n=0}^{\infty} n\rho^n = \frac{\rho}{1 - \rho}$$

6. We can write $\mathbb{E}[L] = \mathbb{E}[L_q] + \mathbb{E}[L_s]$, the sum of the number of jobs queued, and the number of jobs *currently being serviced*. Show that

$$\mathbb{E}[L_q] = \frac{\rho^2}{1 - \rho}$$

and apply Little's law to show that the average queueing time

$$\mathbb{E}[T_q] = \mathbb{E}[T_s] \cdot \frac{\rho}{1 - \rho}$$

As L_s is an indicator variable, $\mathbb{E}[L_s] = \mathbb{P}[L_s = 1] = \rho$

$$\mathbb{E}[L_q] = \mathbb{E}[L] - \mathbb{E}[L_s] = \frac{\rho}{1 - \rho} - \rho = \frac{\rho^2}{1 - \rho}$$

Then by Little's law, $\mathbb{E}[L_q] = \lambda \cdot \mathbb{E}[T_q]$, so

$$\mathbb{E}[T_q] = \frac{1}{\lambda} \cdot \frac{\rho^2}{1 - \rho} = \frac{\rho}{\mu(1 - \rho)} = \mathbb{E}[T_s] \cdot \frac{\rho}{1 - \rho}$$

7. Explain why when $\rho \approx 1$, small increases in load lead to large increases in the delay. How does this apply to real world systems?

When $\rho \approx 1$, $1 - \rho \approx 0$ so the waiting time becomes very sensitive to changes in utilisation. This means that when a server is almost always busy, there is little capacity to handle additional small bursts in arrival. Real systems should never be operated close to 100% as they need headroom to handle bursts.

8. We also saw that in a real-world system, queueing delay can grow unboundedly when utilisation is not 1. Why is this?

Even when the average utilisation is less than 1, randomness in arrivals and service times can lead to bursts, so jobs can accumulate in the queue and waiting time grows. When there is no job present, the server idles, which we consider *wasted time*. This wasted time cannot be reclaimed later, as μ is constant, so the lost capacity of the system cannot offset later queues that come.

9. In contrast, let's consider a deterministic system with regular arrivals. How does this differ from the previous part? Can we achieve $\rho = 1$?

Since the arrivals and services are uniform with no bursts or idle time, it is possible to sustain $\rho = 1$ with a bounded response time as the server can keep servicing requests as they arrive, with no delay or idling time.

2 Reliability

Availability

Availability is the probability that the system can accept and process requests. This is measured in “nines” of probability (e.g. 99.9% is said to be “3-nines of availability”).

Durability

Durability is the ability of a system to recover data despite faults (i.e. fault tolerance). It’s important to note that durability does not necessarily imply availability.

When making a file system more durable, there are multiple levels which we need to concern ourselves with. For small defects in the hard drive, Reed-Solomon error correcting codes can be used in each disk block. When using a buffer cache or any other delayed write mechanism, it’s important to make sure dirty data gets written back to the disk. To combat unexpected failures or power outages, the computer can be built with a special, battery-backed RAM called non-volatile RAM (NVRAM) for dirty blocks in the buffer cache.

To make sure the data survives in the long term, it needs to be replicated to maximize the independence of failures. **Redundant Array of Inexpensive Disks (RAID)** is a system that spreads data redundantly across multiple disks in order to tolerate individual disk failures. RAID 1 will **mirror** a disk onto another “shadow” disk. Evidently, this is a very expensive solution as each write to a disk actually incurs two physical writes. RAID 5 will stripe data across n multiple disks to allow for a single disk failure. Each stripe unit consists of $n - 1$ blocks and one **parity block**, which is created by XOR-ing the $n - 1$ blocks. To recover from a disk failure, the system simply needs to XOR the remaining blocks.

Reliability

Reliability the ability of a system or component to perform its required functions under stated conditions for a specified period of time. This means that the system is not only up (i.e. availability), but also performing its jobs correctly. Reliability includes the ideas of availability, durability, and security.

One approach taken by FAT and FFS (in combination with `fsck`) is **careful ordering and recovery**. For instance, creating a file in FFS may be broken down into the following steps

1. Allocate data block. Write data block.
2. Allocate inode. Write inode block.
3. Update free map. Update directory entry. Update modify time for directory entry.

To recover from a crash, `fsck` might take the following steps.

1. Scan inode table.
2. If any unlinked files (not in any directory), delete or put in lost and found directory.
3. Compare free block bitmap against inode trees.
4. Scan directories for missing update/access times.

It’s important to note that this is not a foolproof method. While there are a few ways that failures can happen in spite of this method, the file system will be recoverable most of the times.

The other approach is to use **copy on write (COW)**. Instead of updating data in-place, new versions are written to a new location on disk, and the appropriate mappings and references to these data are subsequently updated. Since the mappings and references are updated last, this allows for easy recovery if the system crashes sometime in the middle of updating data since the old data and mapping will still be in tact. Furthermore, data is being only added, not modified, so batch updates and parallel writes can help improve performance.

3 Transactions

A more general way to handle reliability is through the use of **transactions**, indivisible units which execute independently from other transactions. Transactions typically follow the **ACID properties**.

Atomicity A transaction must occur in its entirety or not at all.

Consistency A transaction takes system from one consistent state (i.e. meets all integrity and correctness constraints) to another.

Isolation Each transaction must *appear* to execute on its own.

Durability A committed transaction's changes must persist through crashes.

The idea of a transaction is similar to that of a critical section with the newly added constraint of durability. Each entry in a transaction needs to be **idempotent**, which have the same effect when executed once or many times (i.e. $f(f(x)) = f(x)$).

Transactions are recorded on **logs/journals** which are stored on disk for persistence. Writes to a Log are assumed to be atomic. Logs will typically use a circular buffer as its data structure, which maintains a head and tail pointer.

Transactional file systems use the idea of transactions and logs to make the file system more reliable. There are two main types of transactional file systems: journaling and log structured.

Journaling file systems use **write-ahead logging (WAL)** where log entries are written to disk *before* the data gets modified. During the preparation phase, all planned updates are appended to the log. Each log entry is tagged with the transaction id. During the commit phase, a commit record is appended to the log, indicating the transaction must happen at some point. Next, the write back will take place *asynchronously* after the commit phase where the transaction's changes will be applied to persistent storage. In the background, garbage collection will take place to clean up fully written back transactions. When recovering from a crash, only the committed transactions are replayed to restore the state of the file system.

Log structured file systems (LFS) use the log as the storage. The log becomes one contiguous sequence of blocks that wrap around the whole disk.

3.1 Concept Check

1. Why does each entry in a transaction need to be idempotent?

On recovery, all the entries from a committed transaction are replayed. There is a possibility that the data corresponding to an entry was already modified prior to a crash. A non-idempotent operation would put us in an undesired state.

2. Consider a file system with a buffer cache. Your program creates a file called `pintos.bean`. While the changes have been logged with a commit entry, the tail of the log still points to before the start of the log entry corresponding to creating `pintos.bean`. Assuming no further disk reads or writes have happened, if your program wants to read `pintos.bean`, will it need to scan through the logs?

No. The buffer cache will still hold the blocks corresponding to `pintos.bean`.

3. What is the purpose of a commit entry in a log?

The commit entry makes each transaction atomic. These changes to the file system's on-disk structures are either completely applied or not applied at all. For instance, the creation of a file involves multiple steps (e.g. allocating data blocks, setting up the inode) that are not inherently atomic, nor is the action of recording these actions in the journal, but we want to treat these

steps as a single logical transaction. Appending the final commit entry to the log (i.e. a single write to disk) *is* assumed to be an atomic operation and serves as the “tipping point” that guarantees the transaction is eventually applied.

4 Journaling

You create two new files, F_1 and F_2 , right before your laptop's battery dies. You plug in and reboot your computer, and the operating system finds the following sequence of log entries in the file system's journal.

1. Find free blocks x_1, x_2, \dots, x_n to store the contents of F_1 , and update the free map to mark these blocks as used.
2. Allocate a new inode for the file F_1 , pointing to its data blocks.
3. Add a directory entry to F_1 's parent directory referring to this inode.
4. *Commit*
5. Find free blocks y_1, y_2, \dots, y_n to store the contents of F_2 , and update the free map to mark these blocks as used.
6. Allocate a new inode for the file F_2 , pointing to its data blocks.

You may assume a single write to disk is an atomic operation.

1. What are the possible states of files F_1 and F_2 *on disk* at boot time?

File F_1 may be fully intact on disk, with data blocks, an inode referring to them, and an entry in its parent directory referring to this inode. There may also be no trace of F_1 on disk outside of the journal if its creation was recorded in the journal but not yet applied. F_1 may also be in an intermediate state (e.g. data blocks may have been allocated in the free map, but there may be no inode for F_1 , making the data blocks unreachable)

F_2 is a simpler case. There is no *Commit* message in the log for F_2 , so we know these operations have not yet been applied to the file system.

2. Say the following entries are also found at the end of the log.
 7. Add a directory entry to F_2 's parent directory referring to F_2 's inode.
 8. *Commit*

How does this change the possible states of file F_2 on disk at boot time?

The situation for F_2 is now the same as F_1 : the file and its metadata could be fully intact, there could be no trace of F_2 on disk, or any intermediate between these two states.

3. Say the log contained only entries (5) through (8) shown above. What are the possible states of file F_1 on disk at the time of the reboot?

We can now assume that F_1 is fully intact on disk. The log entries for its creation are only removed from the journal when the operation has been fully applied on disk.

4. When recovering from a system crash and applying the updates recorded in the journal, does the OS need to check if these updates were partially applied before the failure?

No. The operation for each log entry is assumed to be idempotent. This greatly simplifies the recovery process, as it is safe to simply replay each committed transaction in the log, whether or not it was previously applied.

5 Distributed Data Processing

5.1 Concept Check

1. True or False: Completed map tasks are re-executed when their worker crashes.

True: Intermediate results are placed in the local storage of the worker, so they will be lost if the worker crashes.

2. True or False: User tasks must be idempotent for MapReduce.

True: If it is not deterministic or free of side effects, re-executing tasks will lead to different results.

3. How does a reduce worker find and read the relevant KV pairs on workers' disks?

The coordinator forwards the locations of the map results to the reduce workers. The reduce worker invokes an RPC to read the data.

4. What are some performance benefits of lazily evaluating transformations?

First, you can manage compute resources more efficiently. It allows you to avoid unnecessary computation. It also allows you to optimize the overall query, since the entire computation graph is known ahead of time.

5.2 Leetcode golf

Suppose we downloaded a file of all solution statistics from Leetcode. Each file is a comma-separated list of entries formatted as {key: PROBLEM_NAME, value: CODE_LENGTH} (e.g. {"2SUM", 162}). Implement the following MapReduce program to find the code golf champion for each problem!

Note: Code golfing is a competition to solve a given problem with the shortest source code possible.

```
map(key problem_name, value code_length) {
    Emit(_____, _____);
}

reduce(key problem_name, List<value> length_lists) {
    lowest = _____;
    for (value length : length_lists) {
        _____;
    }
    Emit(_____, _____);
}
```

```
map(key problem_name, value code_length) {
    Emit(problem_name, code_length);
}

reduce(key problem_name, List<value> length_lists) {
    lowest = 0;
    for (value length : length_lists) {
```

```

    lowest = min(lowest, length);
  }
  Emit(problem_name, lowest);
}

```

6 Distributed Systems

6.1 Concept Check

1. The vanilla implementation of 2PC logs all decisions. How could 2PC be optimized to reduce logging?

Abort decisions can be ignored and not logged with the idea of presumed abort. On recovery, if there is nothing logged, then we can simply assume that the decision made was to abort.

2. 2PC exhibits blocking behavior where a worker can be stalled until the coordinator recovers. Why is this undesirable?

If a worker is blocked on this coordinator, then it may be holding resources that other transactions may need.

6.2 Two Phase Commit

Consider a system with one coordinator (C) and three workers (W_1, W_2, W_3). The following latencies are given for each worker.

Worker	Send/Receive (each direction)	Log
W_1	400 ms	10 ms
W_2	300 ms	20 ms
W_3	200 ms	30 ms

You may assume all other latencies not given are negligible. C has a timeout of 3 s, log latency of 5 ms, and can communicate with all workers in parallel.

1. What is the minimum amount of time needed for 2PC to complete successfully?

For each phase, the worker needs to receive the message, log a result, and send back a message. Since each worker can operate in parallel, only the longest latency matters. Using the latencies given, W_1 has the longest round trip time latency with $400 + 10 + 400 = 810$ ms. Therefore, the two phases will require 1620 ms. However, we also need to add in the time that it takes for the coordinator to log the global decision, so the minimum amount of time is 1625 ms. The reason this is the minimum amount of time is because this assumes no failures.

2. Consider that all three workers vote to commit during the preparation phase. The coordinator broadcasts a commit decision to all the workers. However, W_2 crashes and does not recover until immediately after the coordinator's timeout phase. Does this transaction commit or abort? What is the latency of this transaction, assuming no further failures?

The transaction still commits since a commit decision was made by the coordinator. The preparation phase will take 810 ms as calculated before, and the commit decision needs to be logged which takes 5 ms. The first try of the commit phase will take 3000 ms (i.e. the timeout). The second try will only take $300 + 20 + 300 = 620$ ms because W_2 is the only worker that needs

to commit; all other workers succeeded on the first try. In total, the latency of this transaction is $810 + 5 + 3000 + 620 = 4435$ ms.