1.

a. TTL (Time to Live) is the number of times a descriptor is forwarded by Gnutella servants until the TTL reaches 0 (TTL decrement by 1 on every hop). When TTL = 0 the descriptor is no longer forwarded.

In this problem we are aware there are 15 processes with each process connected to its immediate three clockwise neighbors and immediate three anticlockwise neighbors, which is a total of six neighbors per process. While all links are bidirectional.

While we know that the Gnutella topology looks like a virtual ring:

TTL = 4: 0 processes apart from sender received the query.

TTL = 3: **6** neighbors processes of the sender received the query.

TTL = 2: 6 neighbor process of the sender + 6 new process received the query (since our topology models a virtual ring, each of the 6 neighbors explores/send only 1 new neighbor receives the query for the first time) = 12 processes

TTL = 1: 12 processes + 6 new process = 18 processes (all processes received the query in TTL = 3)

TTL = 0: 18 + 6 new processes = 24 processes

**Notice that in TTL 4 up to 24 processes could have received the query if there were 24 processes in the topology, in our case all 15 processes received the query in TTL 3.**

b. According to the model above the minimum TTL in a Query message to ensure all nodes in the system receive the Query is **TTL=3**

c. Since the 16 th new node neighbors with all the 15 nodes.

**Worst-case scenario:**

TTL =2: 0 processes apart from sender received the query.

TTL =1: 6 neighbors process of the sender received the query.

TTL =0: 6 neighbors process of the sender + 9 process that are left can receive the query from the $16^{th}$ process that neighbors with all process = 15 processes

**In conclusion TTL=2 if the new $16^{th}$ is not the sender.**

**Best-case scenario:**

TTL=1: 0 processes apart from sender received the query.

TTL= 0: 15 neighbors process of the sender received the query.

**In this scenario if the new $16^{th}$ process is the sender, TTL=1 is enough for all processes to receive the query**.

2.

a. BitTorrent focuses on fetching shard based on the rarity of the shard, to calculate which shard is fetched first we need to find the rarest shard in all planets.

- Shard A appeared on Mercury, Earth, Mars, Jupiter, Saturn, Uranus, and Neptune a total of 7 planets.
- Shard B appeared on Venus, Earth, Mars, Jupiter, Saturn, Uranus, and Neptune a total of 7 planets.
- Shard C appeared on Mercury, Earth, Mars, Jupiter, Saturn, and Uranus a total of 6 planets.
- Shard D appeared on Venus, Earth, Mars, Jupiter, and Saturn a total of 5 planets.
- Shard E appeared on Mercury, Earth, Mars, and Jupiter a total of 4 planets.
- Shard F appeared on Venus, Earth, and Mars a total of 3 planets.
- Shard G appeared on Mercury and Earth a total of 2 planets.

**In this case Shard G appeared the rarest of all shards and since it is the rarest Shard G will be fetched first.**

b. **According to the information above the order in which shards will be fetched by the querying node is G, F, E, D, C, B, a**

c. **One shard that could change the first fetched shard is shard G in the new planet DeGrasseTyson and the first Fetched shard can be F because shard F and G creates a tie where the order now could be changed to F, g, E, D, C, B, a making shard F the first fetched.**

**4.**

a. In the case where Cassandra doesn't use bloom filter:

**The advantage that could be conceived would be the high probability of false positive rate, where the key could be represented in the Bit Map by another key. This results in an added use of resources to check if the key is present in the Bloom Filter and in the database.**

**Disadvantage of not using Bloom Filter would be that with using Bloom Filter large sets of data elements in the database could be represented in the Bloom Filter using less amount of memory and checking a key in the bloom filter would be cheap than querying the key in the database. Another advantage would be that there would never be a False Negative, while searching if the key is present in the bloom filter. This results in no additional resource being used to query the database for a key that is not present in the database.**

b. In the case of Cassandra not using Memtable and want's to directly write into the latest SSTable:

**The advantage could be that there would be less memory used for storing recent data, which can be used as additional resource for the entire system and another advantage would be that time could be conserved from the process of flushing data to SSTable while using Memtable when the data is directly wrote to the latest SSTable**

**The disadvantage of not using Memtable would be that fast read and writes can't be used for recently written data, another disadvantage would be that multiple writes to the same key is not consolidated before storing it in SSTable.**

c. In the case of Cassandra SStable being mutable:

**The advantage could be that we can remove the compaction process entirely since the recent key-value pairs are being updated to the SSTable, which is beneficial in terms of not having old data being present in the database in the case SSTable were Immutable.**

**The disadvantage could be that the process could take up a lot of resources to process and find a match for the key-value pairs. Which is also time consuming, such that the system cannot support multiple reads and writes in SSTable efficiently.**

6.

a.

Linearizability is part of the strong consistency model where in a key-value store with linearizability, every put operation on a key will be immediately visible to subsequent get operations on the same key. Sequential consistency is also part of the strong consistency model where in a key-value store with sequential, every put operation on a key will be ordered and reordered to maintain consistency for all keys in the get operation. Casual consistency marks the middle ground between strong and weak consistency models, where every put operation on a key will focus on the causality of other put operation on other keys and orders the operation to respect the causality. Eventual consistency is one of the weak consistency models, were multiple put operation on a key will stop to converge eventually and get operation on the same key may return stale values unless there are few periods of low put operations.

b.

Linearizability:

Client B -> Write (class, C)

Client A -> Write (amount, 150)

Client C -> Read(amount) = return 150

Client C -> Read(class) = return C

Client A and Client B writes are immediately available for both writes made by client C. As long as the reads are before write, the updated write is immediately available to read.

Sequential:

Client B -> Write (class, C)

Client C -> Read(class) = return C

Client A -> Write (class, A)

Client B -> Read(class) = return A

Client C -> Write (class, B)

Client B -> Read(class) = return B

In sequential consistency the operations are ordered in the order of operation. Which means the operations are processed in the order Client B -> Client C -> Client A -> Client B -> Client C -> Client B.

Casual:

Client C -> Write(amount, 150)

Client A -> Read(class)

Client B -> Write (class, C)

In casual consistency, Client B write operation is processed first so Client C read can be processed with recent value. Which means the operations are processed in the order Client B -> Client A -> Client C

Eventual:

amount = 10

Client A -> Write(amount, 120)

Client B -> Write(amount, 110)

Client B -> Write(amount, 190)

Client C -> Read(amount)

Client A -> Write(amount, 180)

Client B -> Write(amount, 170)

Client A -> Read(amount) -> return 170

In eventual consistency, Client C read operation will remain previous amount 10. Unless all write converges, which is not guaranteed but could happen in Client A read.

7.

Round-trip time for one round of synchronization messages is exactly 0.78 ms

**Server side**:

0.056 ms for a packet to get from an application to the network interface.

0.12 ms delay on the opposite path (network interface to application buffer).

**Client side:**

0.34 ms for a packet to get from network interface to the application.

X minimum time on the reverse path.

**Error is at most (RTT-min2-min1)/2**

**min1 = 0.12 + 0.056 = 0.176 ms**

**min2 = 0.34 + X ms**

**RTT = 0.78 ms**

**0 = (0.78 – (0.34 + X) – 0.176)/2 ms**

**0 = (0.78 – 0.34 – X -0.176) ms**

**X = (0.78 – 0.34 – 0.176) ms**

**X = 0.264 ms**

**min2 = (0.34 + 0.264)ms = 0.604 ms**

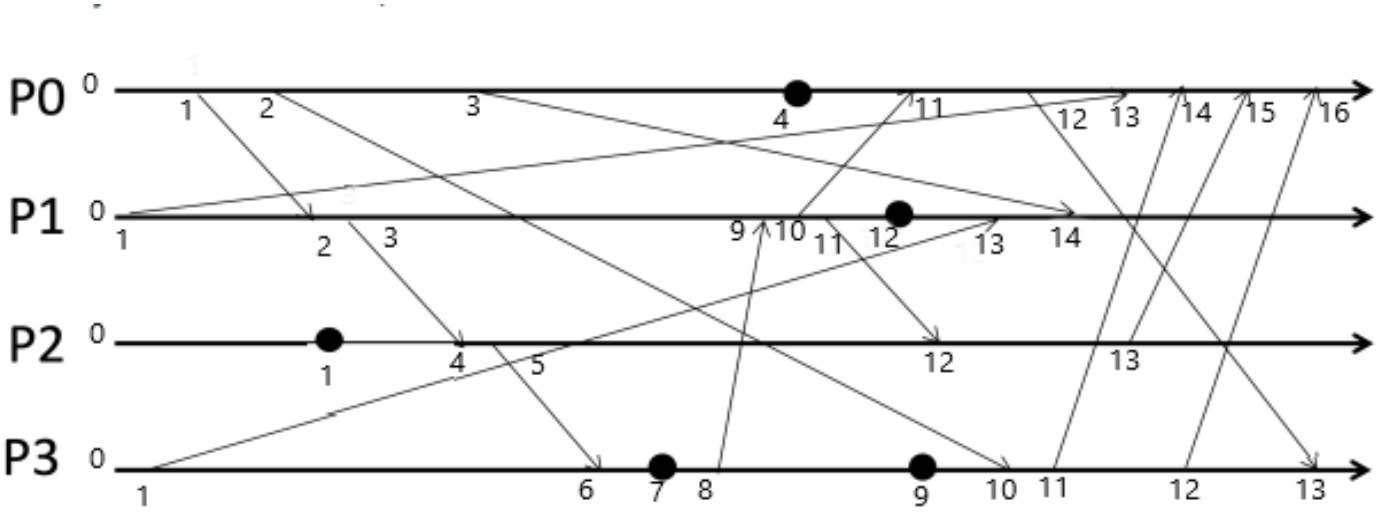
**interval [t+min2, t+RTT-min1] = interval [t+0.528ms , t+0.78ms-0.176ms]**

- ➔ **Interval [t+0.604ms , t+0.604ms]**

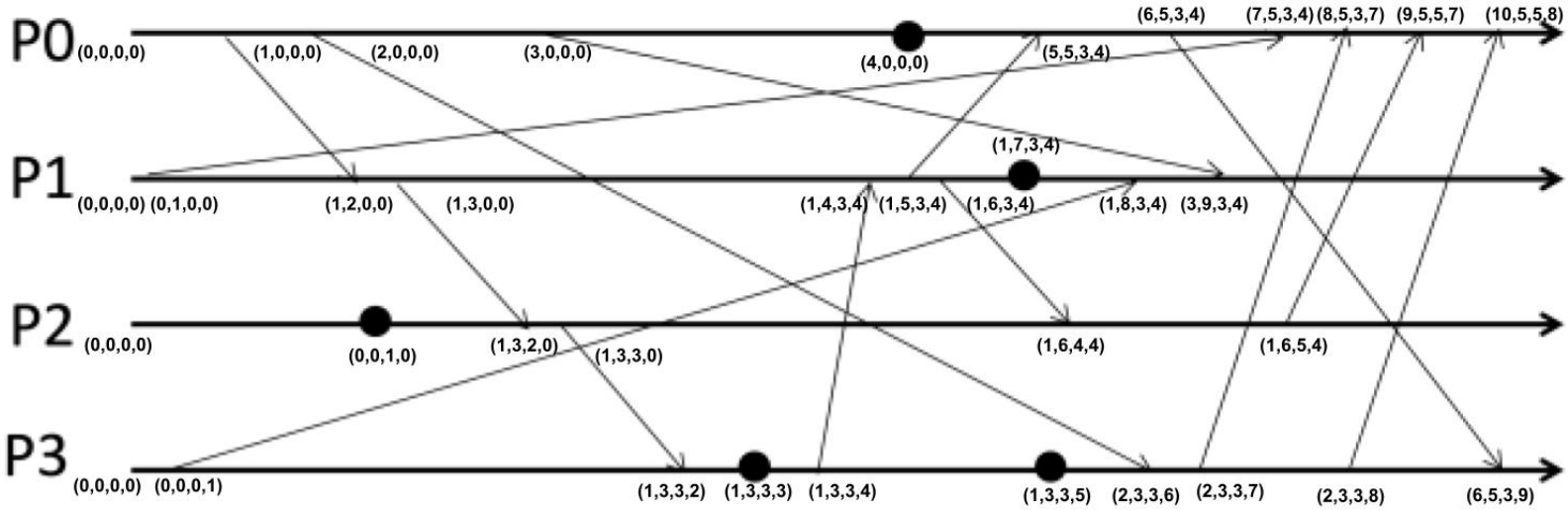
8.

| | |
|---|---|
| Offset o = (tr1 – tr2 + ts2 – ts1)/2<br>– Child is ahead of parent by oreal<br>– Parent is ahead of child by -oreal<br>latency of Message 1 is L1<br>latency of Message 2 is L2<br>L1 = L2<br>tr1 = ts1 + L1 + oreal<br>tr2 = ts2 + L2 – oreal<br>tr1 - tr2 = (ts1 + L1 + oreal) - (ts2 + L2 - oreal)<br>tr1 - tr2 = ts1 + L1 + oreal - ts2 - L2 + oreal<br>tr1 - tr2 = - ts2 + ts1 + L1 - L2 + 2oreal<br>-2oreal = -tr1 + tr2 - ts2 + ts1 - L2 + L1<br>oreal = (tr1 – tr2 + ts2 – ts1)/2 + (L2 – L1)/2<br>since L2 = L1 -> (L2 – L1)/2 = 0<br>oreal = (tr1 – tr2 + ts2 – ts1)/2<br>oreal = o<br>oreal – o = 0<br>\|oreal – o\| = \|(L2 – L1)/2\| < \|(L2 + L1)/2\| | Offset o = (tr1 – tr2 + ts2 – ts1)/2<br>– Parent is ahead of child by oreal<br>– Child is ahead of parent by -oreal<br>latency of Message 1 is L1<br>latency of Message 2 is L2<br>L1 = L2<br>tr1 = ts1 + L1 - oreal<br>tr2 = ts2 + L2 + oreal<br>tr1 - tr2 = (ts1 + L1 – oreal) – (ts2 + L2 + oreal)<br>tr1 – tr2 = ts1 – ts2 + L1 – L2 -2oreal<br>2oreal = -tr1 + tr2 + ts1 – ts2 + L1 – L2<br>Oreal = (tr2 - tr1 + ts1 -ts2)/2 + (L1 – L2)/2<br>Since L1 = L2 -> (L1 – L2)/2 = 0<br>Oreal = (tr2 - tr1 + ts1 -ts2)/2<br>oreal = -o<br>oreal + o = 0<br>\|oreal + o\| = \|(L2 – L1)/2\| < \|(L2 + L1)/2\| |

9.


P0
P1
P2
P3
0
1
2
3
4
11
12
13
14
15
16
9
10
11
12
13
14
1
4
5
12
13
1
6
7
8
9
10
11
12
13

10.


P0
P1
P2
P3
(0,0,0,0)
(1,0,0,0)
(2,0,0,0)
(3,0,0,0)
(4,0,0,0)
(5,5,3,4)
(6,5,3,4)
(7,5,3,4)
(8,5,3,7)
(9,5,5,7)
(10,5,5,8)
(0,0,0,0)
(0,1,0,0)
(1,2,0,0)
(1,3,0,0)
(1,4,3,4)
(1,5,3,4)
(1,6,3,4)
(1,7,3,4)
(1,8,3,4)
(3,9,3,4)
(0,0,0,0)
(0,0,1,0)
(1,3,2,0)
(1,3,3,0)
(1,6,4,4)
(1,6,5,4)
(0,0,0,0)
(0,0,0,1)
(1,3,3,2)
(1,3,3,3)
(1,3,3,4)
(1,3,3,5)
(2,3,3,6)
(2,3,3,7)
(2,3,3,8)
(6,5,3,9)