IN4150 Distributed Algorithms Notes

Wenyu Gu

November 2018

1 Modeling Distributed System

1.1 Networks, Processors, and Processes

1.1.1 The Interconnection Structure

- **Complete network**: There is a link from every processor to every other processor.
- **Ring**: Every processor is connected to two other processors.
  - **Unidirectional Ring**: Every processor can only send message to one of its neighbors (**downstream neighbor**) and receive messages from its other neighbor (**upstream neighbor**).
  - **Bidirectional Ring**: Every processor can send to and receive messages from both its neighbors.
- **Two-dimensional grid**: processors are arranged in a rectangle, with processors internal to the rectangle having four connections (up, down, left, right), processors at the edges having three connections, and the four processors at the corners having two connections.

1.1.2 Synchronous and Asynchronous Systems

Two forms of synchrony are distinguished in DSs:
- Processors are synchronous when the ratios of their speeds are bounded.
- Message passing is synchronous when the message delays are bounded.

Synchronous Systems: All of its processes have access to a global common clock. Both processors and message passing are synchronous.
Asynchronous Systems: When there is no global common clock. Either processors or message passing, or both, are asynchronous.

1.1.3 Synchronous and Asynchronous Communication

Synchronous Communication: The two events of sending a message and receiving it occur (logically) simultaneously. Sending a message is now blocking, as the sending process has to wait for the receiving process to be ready for the reception of a message.
Asynchronous Communication: The events of sending a message and receiving the same message are truly separate events, with the latter occurring after the former. Then, the sending process does not have to be blocked for the receiving process to be ready to receive the message.

1.1.4 Configurations and States, Transitions and Events

State
- **State** of a process is defined as the set of values of all its relevant variables. For each process it contains **initial states** and a set of **terminal states**.
- **State** of a channel is the set of messages sent along the channel but not yet received. The **initial state** of a channel is the empty set.

Transition means a state change of a DS. Transitions are caused by events in one or more processes.

Three types of **events** are distinguished:
- **Internal events**: An internal event in a process only causes its own local state to be modified.
- **Message send events**: A message send event in a process modifies the state of one of its outgoing channels by adding a message to it.
- **Message receive events**: A message receive event in a process modifies the state of one of its incoming channels by removing one of the messages from the state of the channel, and may cause a modification of the state of the process.

1.1.5 Properties of Network Links

DA impose conditions on the communication links in order to function properly. Some of conditions are:
- No loss of messages: messages sent are guaranteed to be received.
- No damage of messages: messages received are guaranteed to be correct.
- The FIFO property: the messages sent along a single channel are received in the order sent.
- Bounded or unbounded buffers: the number of messages sent but not yet received along a link is or is not bounded.
- Finite delay: messages sent along a link (that are not lost) are always received within a finite (but possibly unbounded) amount of time.
- Bounded delay: there exists some upper bound on the delay experienced by messages that are not lost.

In systems with synchronous communication, only the first two properties are relevant.
In systems with asynchronous communication, all the above properties are relevant.

1.2 Distributed Algorithms

2 Synchronization

2.1 Time Concepts in Asynchronous Distributed Systems

In asynchronous distributed systems, the need may arise to reason about events based on their order of occurrence. Events within a single process(or) are assumed to have a total order. The difficulty of course lies in ordering events that occur in different processors. When no messages are sent between processors, nothing can nor has to be said about the order of events in different processors.

System consists of n processes Pi, $i = 1, \ldots, n$.
Ei: the set of events in Pi

2.1.1 The Happened-Before Relation

The happened-before (HB) or precedes relation or causality relation $\rightarrow$ on E is the smallest relation satisfying:

- Local order If a, b $\in Ei$ for some i and a occurred in Pi before b, then a $\rightarrow$ b.
- Message exchange If a $\in$ Ei is the event in Pi of sending message m and b $\in$ Ej is the event in Pj of receiving message m for some i, j with i $\neq$ j, then a $\rightarrow$ b;
- Transitivity If for a, b, c $\in$ E, a $\rightarrow$ b and b $\rightarrow$ c, then a $\rightarrow$ c.

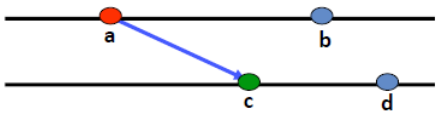

Figure 1: Happen Before Relation

Examples:
Local Order: a $\rightarrow$ b, c $\rightarrow$ d
Message exchange: a $\rightarrow$ c
Transitivity: a $\rightarrow$ d

Concurrency
Two events a, b $\in$ E are concurrent (written as a $\parallel$ b) when neither a $\rightarrow$ b nor b $\rightarrow$ a holds.

Defining events

- Casual Past(history) $P(a) = \{b \in E | b \rightarrow a\}$
- Concurrent Events $C(a) = \{b \in E | b \parallel a\}$
- Casual Future $P(a) = \{b \in E | a \rightarrow b\}$

2.1.2 Logical Clocks

Logical clocks will assign an element of some totally ordered set to each event — which we will call its timestamp— in a way that respects the HB relation.

S: a partially ordered set with partial order $\prec$
A logical clock is a function C: $E \rightarrow S, if a \rightarrow b, C(a) \prec C(b)$ (weak clock condition). It characterizes the HB relation if for any two events $a, b \in E, C(a) \prec C(b) if f a \rightarrow b$ (strong clock condition).

Scalar Clock
Scalar logical clock is a 1-dimensional vector logical clock. A scalar clock C can be constructed by having each process Pi maintain an integer counter Ci with initial value 0.

1. If a $\in$ Ei and if a is not a message-receive event, then Pi first increments Ci by 1, and then sets C(a) equal to the new value of Ci.
2. If a $\in$ Ei is the event in Pi of sending message m and b $\in$ Ej is the event in Pj of receiving message m for some i, j with i $\neq$ j, then Pi sends C(a) along with message m to Pj. On receipt of m, Pj first assigns Cj the value max(Cj +1,C(a) + 1), and then sets C(b) equal to the new value of Cj .

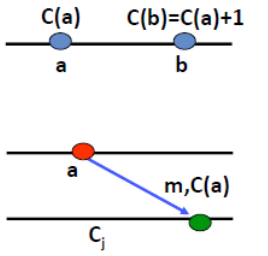

Figure 2: Scalar Clock Implementation

Vector Clock
A k-dimensional vector logical clock is a logical clock with S = N^k with the order like:
if v, w $\in N^k$, then:

$$
\begin{aligned}
\text{v} = \text{w} &\iff \text{v[i]} = \text{w[i]}, \text{ i} = 1,\ldots\text{k} \\
v \leq w &\iff v[i] \leq w[i], \text{ i} = 1,\ldots\text{k} \\
v < w &\iff v[i] < w[i], \text{ i} = 1,\ldots\text{k} \\
v \geq w &\iff v[i] \geq w[i] \text{ , i} = 1,\ldots\text{k} \\
v > w &\iff v[i] > w[i], \text{ i} = 1,\ldots\text{k}
\end{aligned}
$$

1. If a $\in$ Ei and if a is not a message-receive event, then Pi first increments $V_i[i]$ by 1, and then sets C(a) equal to the new value of V_i.
2. If a $\in$ Ei is the event in Pi of sending message m and b $\in$ Ej is the event in Pj of receiving message m for some i, j with i $\neq$ j, then Pi sends V(a) along with message m to Pj. On receipt of m, Pj first assigns Vj the value max(Vj + e_j,v(a)), and then sets V(b) equal to the new value of Vj .

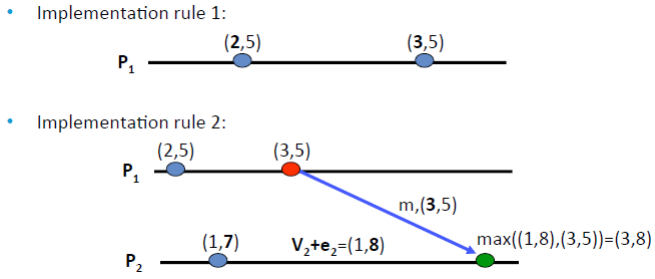

Figure 3: Vector Clock Implementation

2.2 Message Ordering

In asynchronous distributed systems, messages have arbitrary (but finite) delays. However, some applications may impose conditions on the order in which messages are received.

Message Types
Dest(m): set of destinations
Point-to-point: $|Dest(m)| = 1$
Multicast: $|Dest(m)| > 1$
Broadcast: $|Dest(m)| = \#\text{process}$
m(m): an event of multicasting a message m
$d_i(m)$: an event of delivering message m to process P_i

Implementation
There are two processes on each processor. One process receives messages, checks their order, and maintains a buffer with messages that cannot yet be delivered. The other process is the application process proper to which the messages are delivered in the correct order.

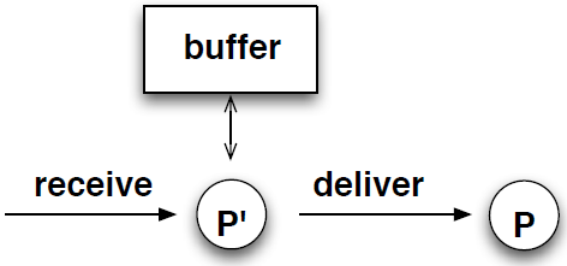

Figure 4: Implementation of message ordering

Message order
Message order is causal when for every two messages m1 and m2, if $m(m1) \rightarrow m(m2)$, then $d_i(m1) \rightarrow d_i(m2)$ for all i $\in Dest(m1) \cap Dest(m2)$.
Message order is total when for every two messages m1 and m2, $d_i(m1) \rightarrow d_i(m2)$ if and only if $d_j(m1) \rightarrow d_j(m2)$ for all i, j $\in Dest(m1) \cap Dest(m2)$.

2.2.1 Birman-Schiper-Stephenson algorithm

2.2.2 Schiper-Eggli-Sandoz algorithm

2.2.3

3 Coordination

3.1 Mutual Exclusion

Mutual exclusion has played an important role in the development of algorithms in central systems. The need for mutual exclusion arises when a resource can only be accessed by one process at a time. Such resources can be hardware components such as printers, or software components such as data structures stored in memory. To access the resource, a process executes a critical section (CS), and so the problem translates into guaranteeing at most one process to be in its CS at a time.
Mutual-exclusion algorithms in distributed system are divided into two kinds:

- **token-based algorithm** : The **token** is a single distinguished message. The possession of the token allows the possess to execute its CS. The main concerns for token-based algorithms are the prevention of starvation and of deadlock.
- **assertion-based algorithm** : A process has to request permission from all or part of the other processes. Based on their replies, it may conclude that it gets exclusive access to CS.

Name	Description
Lamport	basic algorithm (request-reply-release)
Ricart-Agrawala	small optimization (defer replies)
Maekawa	optimization with request sets
Generalized	generalizes 2. and 3
Suzuki-Kasami	basic algorithm (requests to everybody)
Singhal	optimization (requests only to potential token holders)
Raymond	optimization (use a spanning tree)

3.1.1 Lamport's mutex algorithm

Idea (data structure):

- All links have FIFO property
- All messages carry a scalar logical timestamp and the id of sending processor
- Each process maintains a request queue with all requests it receives, ordered according to their timestamps

Idea (operation):

- **Request for access** to CS:
  Broadcast REQUEST message to all processes, including itself.
- **Receive a REQUEST**:
  Enter the request into its request queue and send back a REPLY message.
- **Condition for entering CS**
  Receive a REPLY from every process and its own request is at the head of its request queue.
- **Leave CS**:
  Send a RELEASE message to all processes.
- **Receive a RELEASE**:
  Remove the request from sending process in its request queue.

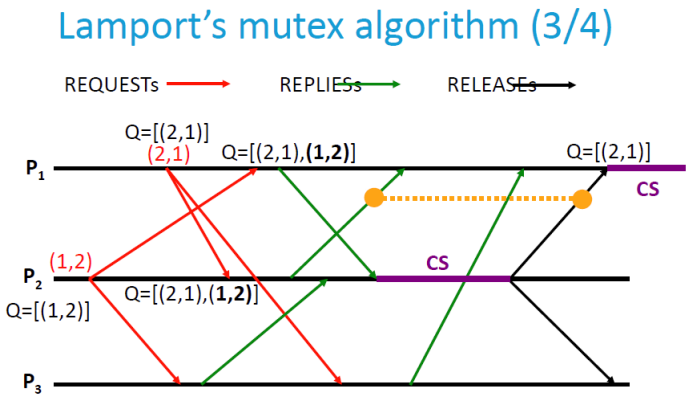

Figure 5: Lamport's mutual exclusion algorithm

Complexity: 3(n-1) messages for each CS invocation
n-1 REQUEST messages, n-1 REPLY messages, n-1 RELEASE messages

Optimization: a process can suppress unnecessary REPLYs.
When a process receives a CS request while it has a older CS request itself, it first sends a REPLY message. Later after it has finished its CS, it sends a RELEASE message. These two message can be combined into a single one.

3.1.2 Ricart-Agrawala

Idea:
A process defers sending a REPLY message to a request if it is currently having a request of its own that is older, until its own request has been satisfied. If a process does not have a request, it still sends a REPLY message immediately. RELEASE messages are not needed anymore.

Idea (operation):

- **Request for access** to CS:
  Broadcast REQUEST message to all processes, including itself.
- **Receive a REQUEST**:
  Send a REPLY message if the process does not have a request of its own, or its own request is later than the request received. Defer a request when its own request is older.
- **Condition for entering CS**
  Receive a REPLY from all other processes.

- **Leave CS**:
  Send a REPLY message to whom a REPLY has been deferred.

Complexity: 2(n-1) messages for each CS invocation
n-1 REQUEST messages and n-1 REPLY messages

3.1.3 Maekawa

Idea:
In previous algorithms a process sends its request to every other process. In order to reduce the message complexity, one may reduce the size of the request set of processes to which a process sends its request and from whom it needs permission to enter its CS. A requesting node did not need permission from every other node, but only from enough nodes to ensure that no one else could have concurrently obtained permission.

Features of **request set**:

- Every two request sets have a non-empty intersection. (Requirement)
- Every process is in its own request set. (Desirable 1)
- All request sets have the same size K. (D2)
- Every process appears the same number of times D in all request sets. (D3)

In order to reduce the message complexity as much as possible, the objective is to choose the sets in a way that K is minimal.
Lemma: the minimum size of the request set is $O(\sqrt{n})$

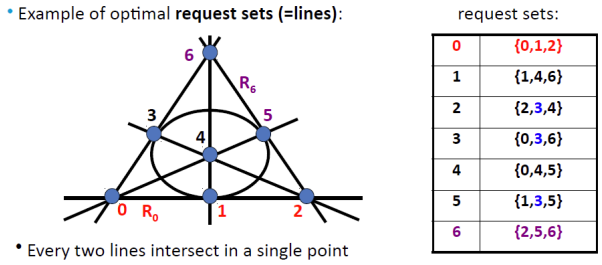

Figure 6: Example of optimal request sets

Need to handle deadlock:
Idea of handling deadlock:

- suppose a process P has granted permission to some process
- when P receives a new request, it checks whether the request is older than all other requests it knows about
- if not, the request is queued and a POSTPONED message is returned
- if so, P will try to retract its permission with an INQUIRE message to the process whose request

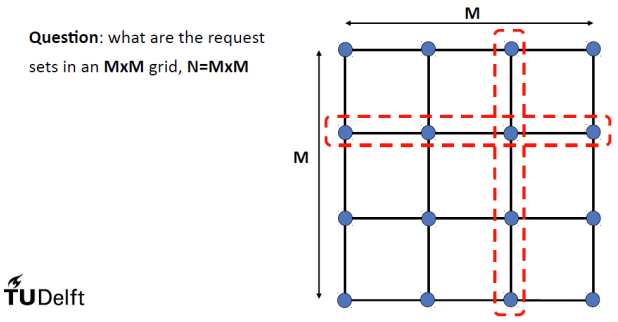

Figure 7: Example of optimal request sets

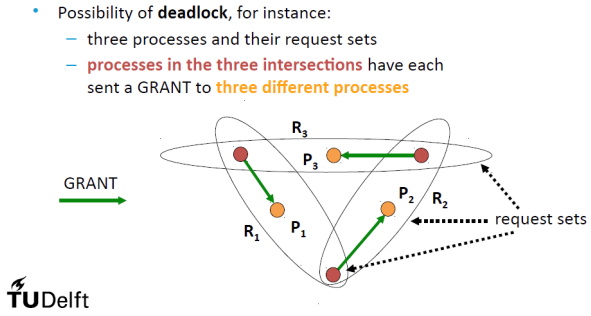

Figure 8: Possibility of deadlock

it has granted

- P may later receive a RELINQUISH message

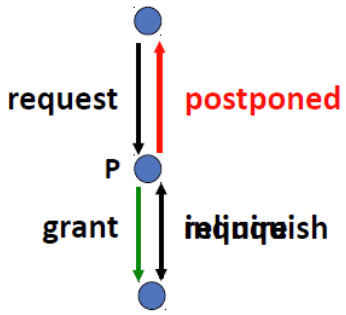

Figure 9: Possibility of deadlock

- **Request for access** to CS:
  Broadcast REQUEST message to all processes in its request set, including itself.
- **Condition for entering CS**
  Receive a GRANT from all processes in its request set.
- **Receive a REQUEST**:
  Give GRANT for that process if never give permission since last RELEASE.
  Else enter process into request queue. If that request is older than all, send INQUIRE to process that granted. Otherwise send POSTPONE mes-

sage.
- **Receive a INQUIRE**:
  If receives a POSTPONE, send REPLINQUISH back.
  If the number of GRANTS received equals to the size of request set, wait.
- **Leave CS**:
  Send a RELEASE message to all processes in its request set.
- **Receive a RELEASE**:
  Remove the request in its request queue. And send GRANT to process at head of request queue.

3.1.4 Suzuki-Kasami

3.1.5 Singhal

3.1.6 Raymond's token-based algorithm

3.2 Election

Election is an important part in the design of distributed algorithms as some applications require one processor be endowed with a special privilege. To solve the problem, all processors should cooperate to elect one from among them to get this privilege. To model the election problem, it is often assumed that each process has a unique integer processor id. In this case, the system is said to be **non-anonymous**, and **anonymous** otherwise.
Following properties effect the complexity of election algorithms.

- The topology of the network, e.g. unidirectional and bidirectional rings, complete network.
- Whether the system is synchronous or asynchronous.
- Whether the system is anonymous or not. In an anonymous ring, no deterministic solution to the election problem exists.
- Whether or not the size of the network is known ahead of time.
- Whether or not an algorithm is **comparison based**. Comparison based algorithm only allows operation of receiving, copying, and sending processor ids.

3.2.1 Asynchronous Bi-directional Rings

Hirschberg's and Sinclair's election algorithm

Idea:
In bidirectional ring, a process could send message in both directions. In first round, a process will check if its own id is the largest among two neighbors. If so, it continues messaging. Otherwise, the node stops participating in election. In every subsequent round k, it tries to establish it has the largest id among the processors in its $2^k + 1$-neighbourhood. In order to do so, it sends a PROBE message with its id and a hop counter, which is initialized to 2^{k-1} in each direction. When a processor receives a PROBE message with a smaller id than its own, it discards it. Otherwise, it forwards the message after decrementing the hop counter by one, or when the hop counter is equal to zero, it sends an OK message with the id of the initiating processor back to that processor. If a processor receives OK messages with its own id from both sides of one phase, it initiates the next phase.
If the ring size is known, a processor knows it has been elected in phase k_0 when it receives two PROBE messages, with k_0 the lowest number satisfies $2^{k_0} + 1 \geq n$.
If the ring size is not known, a process knows it has been elected when it receives a PROBE message from the wrong slides. Alternatively, if one processor receives a PROBE message from either side with the same id larger than its own, it can conclude that the corresponding processor is the one to be elected.

Message Complexity: $O(n \log n)$

An election algorithm in a bidirectional ring

Improvement:
In above algorithm, PROBE messages are sent in neighborhoods of statistically defined increasing sizes. The algorithm could be more dynamic in that in a subsequent phase neighborhoods are bounded by still active processes.

Idea:
In the first phase, a processor exchange its id with with two neighbours. When a process detects its id is larger than those of two neighbors, it remains active and initiates the next phase. Otherwise, it becomes passive. In every subsequent phase, only the active process execute exchanging ids in a virtual ring of processes that are still active. Passive processes only replies messages. When a processes receives its own id, it has been elected.

Message Complexity: $O(n \log n)$

3.2.2 Unidirectional rings

A non-comparison-based algorithm in synchronous unidirectional rings
In this algorithm, a process with minimum id is chosen. The ring size is n, and it is assumed that this size is known to all processes. In the first round, when a process finds its id is equal to 1, it knows it has the smallest id and will be elected. It sends its id to neighbour in round 1. Every process relays the message immediately after receiving it. If a process receives

nothing in the first n round, it knows that id 1 does not exist.In general, if a processor has id equal to k and it has received nothing in rounds 1 through $(k-1)n$, it knows that it will be elected, and it sends its id along the ring in round $(k-1)n+1$.

Chang's and Roberts's election algorithm

Idea:
At least one processor spontaneously starts the algorithm by sending its id to its neighbor. Upon receipt of a message, if the id is equal to its own id, then the process is elected. If the id is smaller, the message is discarded and the process sends its own id to its neighbor if it has not already done that. If the id is larger, the message will be replayed.

Complexity: least equal to n, is atmost equal to $n(n+1)/2$, and is on average of order $O(n \log n)$.

Peterson's election algorithm

Idea:
This algorithm is a simulation of solution 2 of bidirectional rings. Every process first sends its id to its downstream neighbor, and subsequently sends a max of its id and the value just received from its upstream neighbour. If the first value it received is at least as large as the other two, the process remains active. Otherwise, it turns passive. In every subsequent round, the process is repeated in a virtual ring only containing active processes. A process is elected when it receives its own id.

Complexity:
The number of rounds is at most equal to $\log n$ because in every round the number of nodes is at least cut half. The number of messages is at most equal to $2n \log n$ because in every round exactly two messages are sent along one link.
Time complexity: 2n-1

3.2.3 Complete Networks

Afek's and Gafni's synchronous algorithm
Idea:
A node that wants to be elected successively sends its id to a ever larger subset of nodes. A process will return ACKNOWLEDGEMENT if the id received is larger than the largest id it currently knows. When the number of ACKNOWLEDGEMENT received is the smaller than the number of nodes in the subset, it ceases to be a candidate. The size of subset id is initially equal to 1, and is powered by 2 in every next round. Nodes keep track of nodes of the remaining nodes they have not sent id to. Nodes adopt the largest id it have seen.
The algorithm proceeds in rounds, and any number of processes could spontaneously start the algorithm in different rounds. A node spawns two processes, a candidate process and an ordinary process. An ordinary process only replies messages. Candidate process keep track of their level, which is the number of rounds since start. These processes send candidate messages containing a pair of the current level of node and the node id to the ordinary processes in other nodes.
In every round, ordinary processes order messages by lexicographical ordering(level first). When the maximum is larger than its own current identifier, they adopt the maximum and send an acknowledgement back to corresponding candidate process. In fact, the node elected is the node with the largest id among processes start earliest.

Complexity:
Maximum number of rounds is equal to $\log n$. In each round every node sends at most one acknowledgement, leading to a maximum of $n \log n$ messages. The size of sets which a candidate process sends to candidate processes is 2^{k-1}. The number of candidate processes remaining alive is $\frac{n}{2^{k-1}}$. So the total number of candidate messages does not exceed $n \log n$.

Afek's and Gafni's asynchrounous algorithm

Idea:
It does not make sense in an asynchronous system that a candidate process wait for all messages that it have sent messages to. It should react immediately upon receipt of messages.
Candidate messages consist pairs of (level, id), but now the level indicates the number of nodes it has captured. If a candidate process captures a node that had already previously been captured by a lower pair, the level of previous owner is not correct. As the previous owner will not be elected anymore, in order to reduce the number of messages, the node should also capture its previous owner. In order to do so, captured processes maintain a father pointer and a potential-father pointer to their current and potential owner.
Because of the nature of asynchronous system, the previous owner may already been captured by another process.

Complexity:
Time complexity: n Message complexity: $n \log n$

3.3 Minimum weight spanning trees

A **spanning tree** of a undirected graph involves all nodes in the graph with minimum number of edges. A

weighted graph is a graph where each edge has an associated weight. A **complete graph** is a graph where each vertex is connected to other vextexes by an edge. A **Minimum-weight spanning tree** is a spanning tree of a weighted graph with minimal weights.

Lemma 1. *A weighted connected undirected graph in which all weights are different has a unique MST.*

Proof. Suppose there exist two MSTs T_1 and T_2 of a weighted complete Graph G. There must be some edges appear in one MST but not in both. Let's pick up one with lowest weight e. We assume e exists in T_1 but not in T_2. Adding e to T_2 yields a cycle. As there is no cycle in T_1, T_2 must contain an edge e_2 that is not part of T_1. Because of the choice of e, we have $w(e) < w(e_2)$. So T_2 with e and without e_2 has a smaller sum of weight than orginal T_2. Therefore we can conclude that T_2 is not a MST, contradicting the hypothesis. □

Lemma 2. *If F is a fragment and e is its MOE, then F with e and the node incident with e not in F is also a fragment.*

Proof. Suppose F with e is not a fragment, then in the final MST there must exist a path from P to Q. Adding e to the MST creates a cycle. At least one edge in MST is an outgoing edge of F. As e is the MOE of F, e has the minimum weight among all outgoing edges of F. We have $w(e) < w(x)$. So replacing x by e creates a spanning tree with smaller weight, which is a contradiction. □