UNIT- V
DISTRIBUTED PROGRAMMING ALGORITHMS

DISTRIBUTED PROGRAMMING ALGORITHM

INTRODUCTION
Distributed algorithms are a **sub-type of parallel algorithm**, typically **executed concurrently**, with separate parts of the algorithm being **run simultaneously on independent processors**, and having limited information about what the other parts of the algorithm are doing.

DEFINITION
A **distributed algorithm** is an algorithm designed to run on computer hardware constructed from interconnected processors.

CHARACTERISTICS OF DISTRIBUTED ALGORITHM
- Concurrent execution
- Coordination often needed
- Communication frequent
- Hard to implement and debug
- Often escapes intuition
- Correctness often deceiving
- Failures of various types

APPLICATIONS OF DISTRIBUTED ALGORITHM
Distributed algorithms are used in many varied application areas of distributed computing
- Telecommunications
- Scientific computing
- Distributed information processing
- Real-time process control.

PROBLEMS SOLVED BY DISTRIBUTED ALGORITHMS
- Leader election
- Consensus,
- Distributed search

• Spanning tree generation

• Mutual exclusion

• Resource allocation.

CHALLENGE

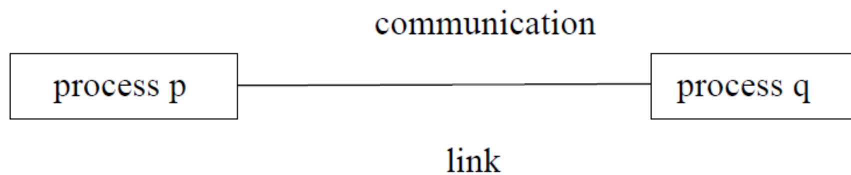
One of the major challenges in developing and implementing distributed algorithms is successfully coordinating the behavior of the independent parts of the algorithm in the face of processor failures and unreliable communications links.

The choice of an appropriate distributed algorithm to solve a given problem depends on both the **characteristics of the problem**, and **characteristics of the system the algorithm will run** on such as the type and probability of processor or link failures, the kind of inter-process communication that can be performed, and the level of timing synchronization between separate processes.

TYPES OF DISTRIBUTED ALGORITHM

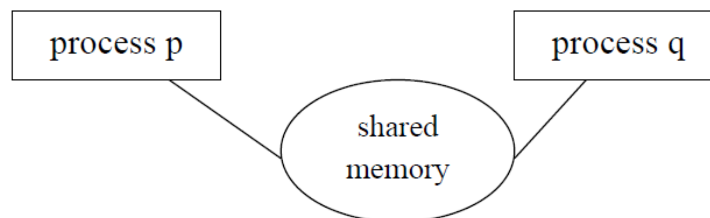
1. MESSAGE PASSING ALGORITHM

Processes communicates by sending and receiving messages.



2. SHARED MEMORY ALGORITHM

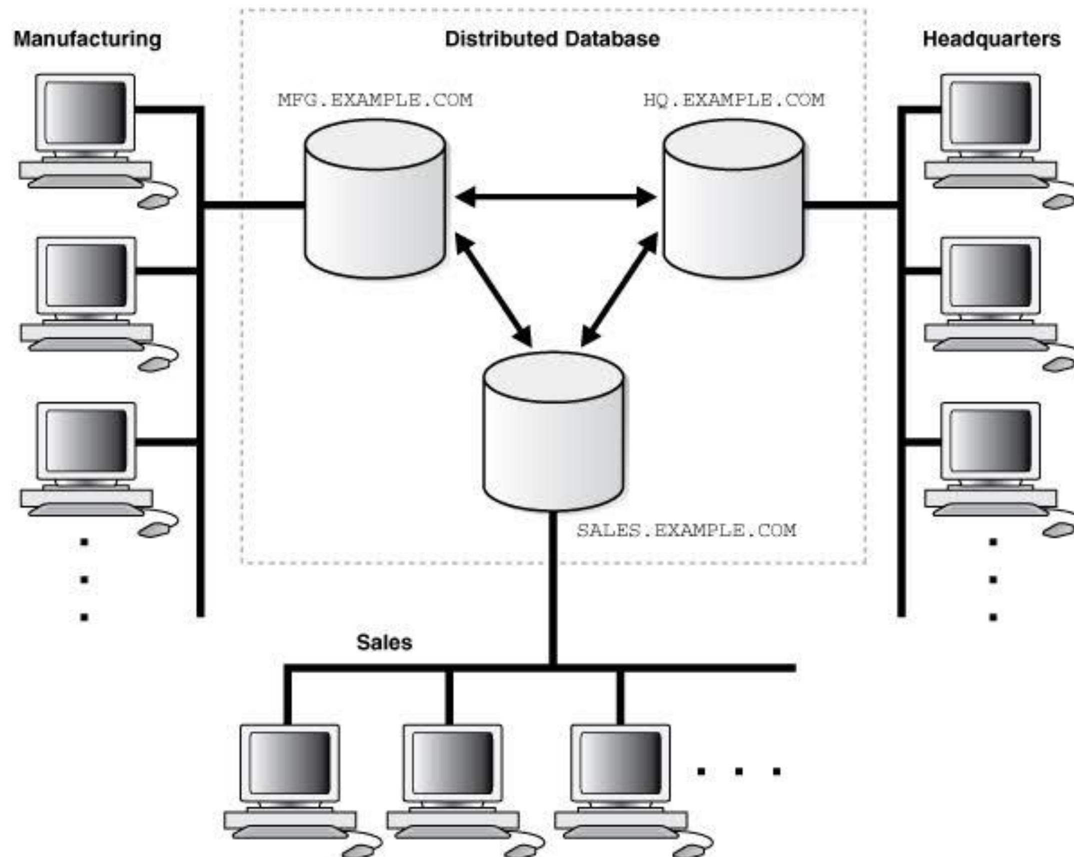
Processes communicate by writing/reading on shared memory.



Synchronization in Distributed Systems

Distributed System is a collection of computers connected via the high speed communication network. In the distributed system, the hardware and software components communicate and coordinate their actions by message passing. Each node in distributed systems can share their resources with other nodes. So, there is need of proper allocation of resources to preserve the state of resources and help coordinate between the several processes. To resolve

such conflicts, synchronization is used. Synchronization in distributed systems is achieved via clocks.



The physical clocks are used to adjust the time of nodes. Each node in the system can share its local time with other nodes in the system. The time is set based on **UTC (Universal Time Coordination)**. UTC is used as a reference time clock for the nodes in the system.

The clock synchronization can be achieved by 2 ways:

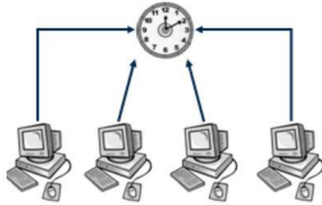
- External Clock Synchronization.
- Internal Clock Synchronization.

External clock synchronization is the one in which an external reference clock is present. It is used as a reference and the nodes in the system can set and adjust their time accordingly.

Internal clock synchronization is the one in which each node shares its time with other nodes and all the nodes set and adjust their times accordingly.

Clock Synchronization

External Clock Synchronization



Synchronize clocks with respect to an external time reference
Example: NTP

Internal Clock Synchronization

Synchronize clocks among themselves

Mutual exclusion in distributed system

Mutual exclusion is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process can not enter its critical section while another concurrent process is currently present or executing in its critical section i.e only one process is allowed to execute the critical section at any given instance of time.

Mutual exclusion in single computer system Vs. distributed system:

In single computer system, memory and other resources are shared between different processes. The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variable (For example: Semaphores) mutual exclusion problem can be easily solved.

In Distributed systems, user neither have shared memory nor a common physical clock and there for user can not solve mutual exclusion problem using shared variables. To eliminate the mutual exclusion problem in distributed system approach based on message passing is used.
A site in distributed system do not have complete information of state of the system due to lack of shared memory and a common physical clock.

Requirements of Mutual exclusion Algorithm:

1. **No Deadlock:**

Two or more site should not endlessly wait for any message that will never arrive.

2. No Starvation:

Every site who wants to execute critical section should get an opportunity to execute it in finite time. Any site should not wait indefinitely to execute critical section while other site are repeatedly executing critical section

3. Fairness:

Each site should get a fair chance to execute critical section. Any request to execute critical section must be executed in the order they are made i.e Critical section execution requests should be executed in the order of their arrival in the system.

4. Fault Tolerance:

In case of failure, it should be able to recognize it by itself in order to continue functioning without any disruption.

Solution to distributed mutual exclusion:
A shared variables or a local kernel cannot be used to implement mutual exclusion in distributed systems. Message passing is a way to implement mutual exclusion. Below are the three approaches based on message passing to implement mutual exclusion in distributed systems:

Token Based Algorithm:
1. A unique token is shared among all the sites.
2. If a site possesses the unique token, it is allowed to enter its critical section
3. This approach uses sequence number to order requests for the critical section.
4. Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
5. This approach insures Mutual exclusion as the token is unique

Example: Suzuki-Kasami's Broadcast Algorithm

Non-token based approach:
1. A site communicates with other sites in order to determine which sites should execute critical section next. This requires exchange of two or more successive round of messages among sites.
2. This approach use timestamps instead of sequence number to order requests for the critical section.

3. Whenever a site make request for critical section, it gets a timestamp. Timestamp is also used to resolve any conflict between critical section requests.

4. All algorithm which follows non-token based approach maintains a logical clock. Logical clocks get updated according to Lamport's scheme

Example: Lamport's algorithm, Ricart–Agrawala algorithm

Quorum based approach:

1. Instead of requesting permission to execute the critical section from all other sites, Each site requests only a subset of sites which is called a quorum.

2. Any two subsets of sites or Quorum contains a common site.

3. This common site is responsible to ensure mutual exclusion

Example: Maekawa's Algorithm

TERMINATION DETECTION

The basis of **termination detection** is in the concept of a distributed system process' state

- Termination detection. During the progress of a distributed computation, processes may periodically turn active or passive . A distributed computation termination

GLOBAL TERMINATION DETECTION:

Can't observe all processes at the same instant

• All processes observed so far (in set X) may be passive

• But a process q yet unobserved may send a message to a process turning active when it receives the message

but we have moved on, observing p was passive And then q can turn passive itself

GENERAL STRATEGY:

• And no process observed as passive will ever turn active Is detecting termination similar to detecting

• Only active processes can send messages Receipt of a message reactivates a passive process

• P0, P1, P2, Pn−1, P0   for termination messages, communication only between Pi and Pi−1

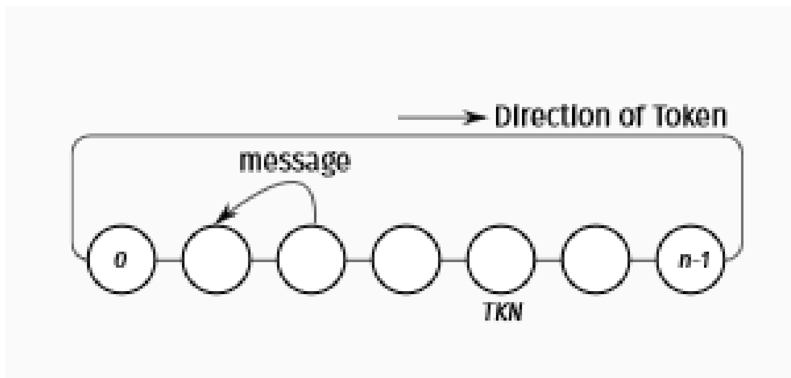
GENERAL IDEAS :

• Uses a token per detection

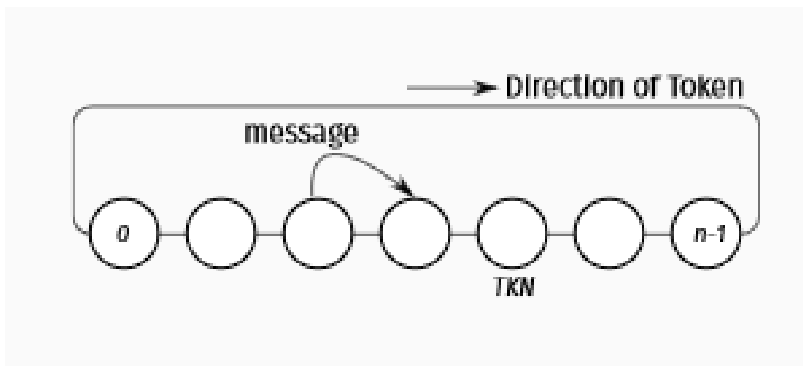
• Termination detection launched by a single process P0 Sends token when it is passive

• P0 sends token to Pn−1   Pn−1 sends token to Pn−2 only when it is passive

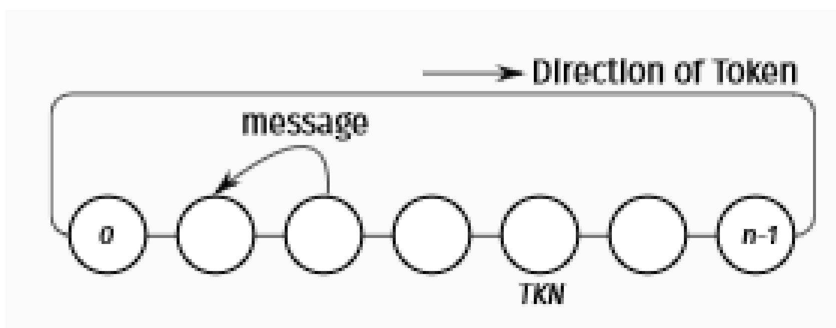
Complications - 1



Complications – 2



Complications – 3



HANDLING MESSAGES IN TRANSIT – 1

•All processes are also given a colour

•Initially, Consider a process P not visited by the send a message to Pk reactivating

TERMINATION DETECTION

- using Markers
- Useful for detecting loss of mutual exclusion token
- Useful for termination detection

Clocks and Event Ordering

Introduction

In this post, we will cover the notion of time in distributed systems. We rely a lot on time in our daily existence for ordering. One thing "happened before" another is a relation that is codified by time, using physical clocks as a mechanism to deliver time.Soordering of events is done using physical clocks in the real world. In a distributed system, this notion of time needs to be understood more carefully. Without relying on physical clocks that can be inaccurate, the central question becomes how do we define ordering of events in system of collection of nodes. Each node can establish their own local order of events. But how can one establish a coherent order of events across multiple nodes. This order is important for solving problems like distributed locking — a shared resource which can only be used by one node at a time and the node requesting the lock on the resource needs to be granted the lock as long as another node hasn't asked for it before. In such cases local order is not sufficient and some global order needs to be established across multiple nodes.

Partial Ordering of events

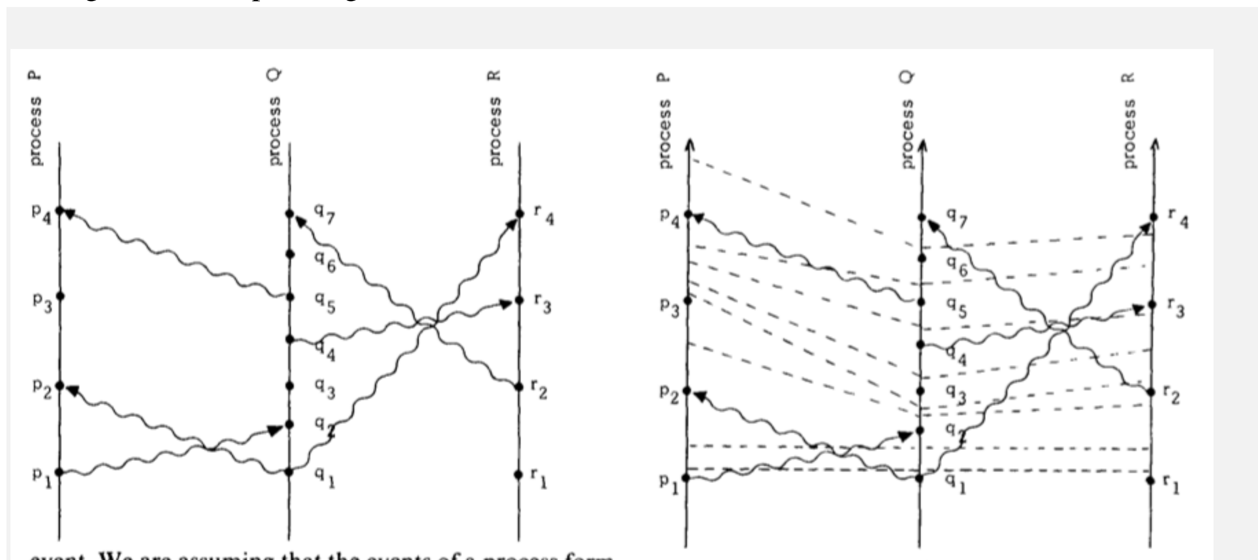
One way to define an order of events in a distributed system would be to have a physical clock. So "happened before" event can be described using $t1 < t2$. But clocks are not accurate, can

drift and are not necessarily synchronized in lock-step.So this paper takes another approach to define “happened before” relation. (The meaning of “partial order” will become clear later.)

A distributed system can be defined as a collection of processes. A process essentially consists of a queue of events(which can be anything — like an instruction, or a subprogram, or anything meaningful) and has apriori order. In this system, when processes communicate with each other, sending of a message is defined as an event. Let’s establish a more precise definition of “happens before” (⇢) in such a system.

1. In the same process, if event a comes before event b, then a ⇢ b
2. When process sends some message at a to process j and process j acknowledges this message at b, then a ⇢ b
3. Happens before is transitive. a ⇢ b and b ⇢ c, then a ⇢ c.
4. Two events a and b are described as concurrent when a hasn’t happened before b and b hasn’t happened before a. This condition generally reflects the fact that process may not have knowledge about all events that could have happened in process j. So we cannot establish authoritative order in the system — this also makes it clear that we have only a ***partial order*** in the system. There could be a lot of events in the system for which no meaningful order can be established without relying on physical time.

Taking some examples might be useful.



Partial ordering of events in the system. Vertical bars are processes. Wavy arrows are messages. A dot is an event. Time elapses bottom up.

As you can see in the figure on the left above, p1 ⇢ p2 as those are the events in the same process. q1 ⇢ p2 due to sending of the message from process Q to process P. If we consider that time is elapsing bottom up and q3 seems to happen before p3 if we were to consider physical time. But in this system, we will call these events as concurrent. Both processes don't know about these events — there is not causal relationship at this time. But what we can say is that q3 ⇢ p4, because q3⇢q5 and q5 ⇢ p4. Similarly, we can say that p1 ⇢ r3. Now having gone through some examples, it becomes clear that, another way to envisage a ⇢ b is that event a can causally affect event b. On a similar note, q3 and p3 mentioned above are not causally related.

Logical clocks

Lamport's paper introduces a new function, essentially a counter, in every process that can assign a number to an event. Let's call this function as Ci(a) as a counter in process for event. There are no physical clocks in the system. The function C(a) establishes the invariant that,event must have happened before b, then C(a) < C(b). Using this function, a partial ordering of events in the system can be established using the following two conditions:

C1: Ci(a) < Ci(b) if a happens before b in the same process i. This can be implemented using a simple counter in the given process.

C2: When process sends message at event a and process j acknowledges the message at event b, then Ci(a) <Cj(b)

These clock functions can be thought of as ticks that occur regularly in a process. Between any two events, there needs to be at least one such tick. Each tick essentially increments the number assigned to the previous tick. This is illustrated in the right-hand side figure above. Similar numbered ticks are connected across process boundaries. In the same process one touches or crosses the tick boundary to land on the next event. Across processes when messages are sent, that message needs to cross or touch a tick boundary to define the "happens before" event.

This can be implemented in the following way:

IR1: This one obeys C1. This can be done by incrementing the Ci(a) between any two successive events in the system.

IR2: This is one implements C2. It can be done by sending Ci(a) as a timestamp to process j. When Process j acknowledges the receipt of this message at event b, it needs to set Cj(b). Cj(b) will be set to a value ≥ the current Cjand also greater than Ci(a)/timestamp.

Total ordering of events in the system

So far the system of logical clocks has established a partial order of events in the system. There are still events which are concurrent and it would be useful to break ties, specially for the locking problem described in the introduction — someone needs to get the lock. This can be done by introducing an arbitrary process priority. In case of a tie, the process with lower priority gets the event that happened before. More formally, a total order a ⇥ b (notice the new type of arrow)can be defined as:

1. Ci(a) <Cj(b).

2. If Ci(a) = Cj(b), then use Pi <Pj.

These two conditions imply that ⇥ completes the partial relationship ⇢. If two events are partially ordered then they are totally ordered already. While partial ordering is unique in the given system of events, total ordering may not be.

Distributed locks using total ordering

Consider the following problem which can be quite common in distributed systems. The central idea of the problem is to access a shared resource, but only one process can access it at any time. More formal conditions can be specified as:

1. A process which has been granted a resource, must release it before any other process can acquire it.

2. Resource access requests should obey the order in which requests are made

3. If every process releases the resource it asked for, then eventually access is granted for that resource.

One possible solution could be to introduce a centralized scheduler. While one issue is that it is a completely centralized, another is that ordering condition 2 may not work. Consider the scenario: Process i asking for a resource to the scheduler. Then it informs process j about the request. Process j now asks for the same resource, but its message reaches the scheduler before process i's. This means that event order was not obeyed. To address this issue, we can use total ordering

based off of IR1 and IR2. With this, every event is totally ordered in the system. As long as all the processes know about requests made by other processes, the correct ordering can be enforced. A decentralized solution can be designed such that each process keeps a queue of lock and unlock operations.

1. Process i asking for a resource lock, uses the current timestamp and puts lock(T,Pi) in the queue. It also sends this message to all other processes.

2. All other processes put this message in their queue and send the response back with a new timestamp Tn.

3. To unlock a resource, process i, sends unlock(T, Pi) message to all processes and removes the lock(T, Pi) message from its own queue.

4. Process Pj, upon getting unlock message, removes lock(T, Pi) message from its queue.

5. Process Pi is free to use the resource i.e. gets its lock request granted when: it has the lock(T, Pi) messages in its queue with T enforcing the ***total order*** such that T is before any other message in the queue. In addition, process Pi needs to wait until it has received messages from all the processes in the system timestamped later than T.

Conclusion

The notion of time/order-of-events is quite complicated in distributed system. Idea of logical clocks to ensure causal ordering, without having to rely on physical clocks, is quite useful in distributed systems. Lamport's logical clocks ensure that if $a \dashrightarrow b$ then, $C(a) < C(b)$. It is also good to understand that if $C(a) < C(b)$ then it doesn't necessarily mean that $a \dashrightarrow b$. If we just looked at C(j) and C(k) and C(j) happened to be less than C(k), then those could be concurrent events and processes may not have communicated with each other yet. This later property, of just looking at some timestamps and realizing the causality, is achieved by vector clocks.

LOCKING

Locking, often, isn't a good idea, and trying to lock something in a distributed environment may be more dangerous. But sometimes, we need to keep this risk and try to use a distributed lock for two main reasons:

- **Efficiency:** a lock can save our software from performing unuseful work more times than it is really needed, like triggering a timer twice.

- **Correctness:** a lock can prevent the concurrent processes of the same data, avoiding data corruption, data loss, inconsistency and so on.

We have two kinds of locks:

- **Optimistic**: instead of blocking something potentially dangerous happens, we continue anyway, in the hope that everything will be ok.
- **Pessimistic**: block access to the resource before operating on it, and we release the lock at the end.

To use **optimistic** lock we usually use a version field on the database record we have to handle, and when we update it we check if the data we read has the same version of the data we are writing.

Database access libraries, like Hibernate, usually provide facilities to use an optimistic lock.

The **pessimistic lock** instead will rely on an external system that will hold the lock for our microservices.

As for optimistic lock, database access libraries, like Hibernate usually provide facilities, but in a distributed scenario we would use more specific solutions that use to implement more complex algorithms like:

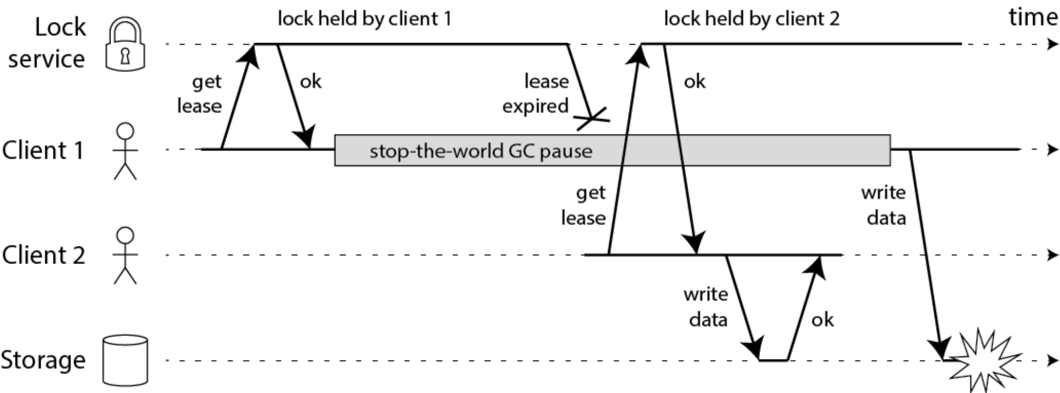
- Redis, using libraries that implements lock algorithm like ShedLock, andRedisson. The first one provides lock implementation using also other systems like MongoDB, DynamoDB, and more.
- Zookeeper, provides some recipes about locking.
- Hazelcast, offers a lock system based on his CP subsystem.

Implementing a pessimistic lock we have a big issue, what happened if the lock owner doesn't release it? If the lock owner dies? The lock will be held forever and we could be in a **deadlock**. To prevent this issue we will set an**expiration time** on the lock, so the lock will be **auto-released**.

But if the time **expires before the task handled by the owner isn't yet finished**, another microservice can acquire the lock, and both lock holders can now release the lock causing inconsistency. Remember, no timer assumption can be reliable in asynchronous networks.

We need to use a **fencing token** which is incremented each time a microservice acquires a lock. This token must be passed to the lock manager when we release the lock, so if the first owner releases the lock before the second owner, the system will refuse the second lock release. Depending on implementation we can also decide to let win the second lock owner.

The following diagram shows how you can end up with corrupted data:



In this example, the client that acquired the lock is paused for an extended period of time while holding the lock – for example because the garbage collector (GC) kicked in. The lock has a timeout (i.e. it is a lease), which is always a good idea (otherwise a crashed client could end up holding a lock forever and never releasing it). However, if the GC pause lasts longer than the lease expiry period, and the client doesn't realise that it has expired, it may go ahead and make some unsafe change.