CS 7172
Parallel and Distributed Computation

Distributed Lock

Kun Suo

Computer Science, Kennesaw State University

https://kevinsuo.github.io/

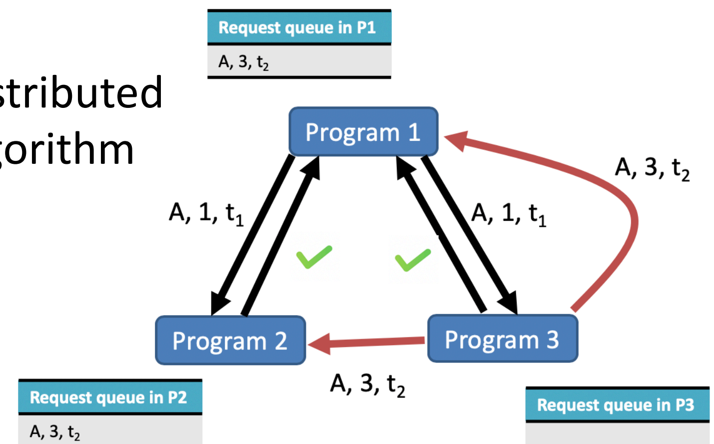
Outline

- Computer networks, primarily from an application perspective
- Protocol layering
- Client-server architecture
- End-to-end principle
- TCP
- Socket programming

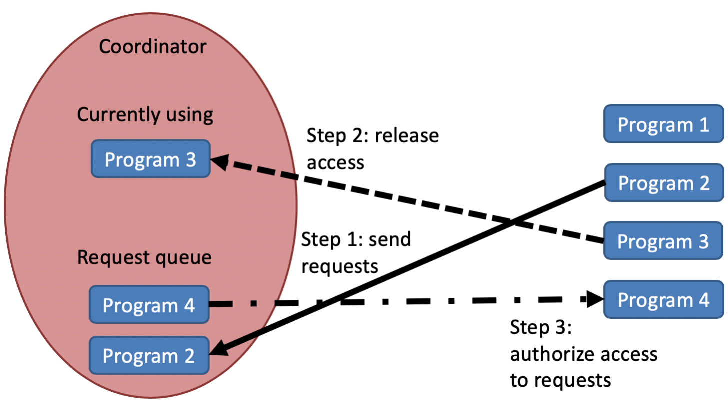
Revisit Distributed Mutex

- In a distributed system, the method to achieve access to exclusive resource is called ***Distributed Mutual Exclusion***

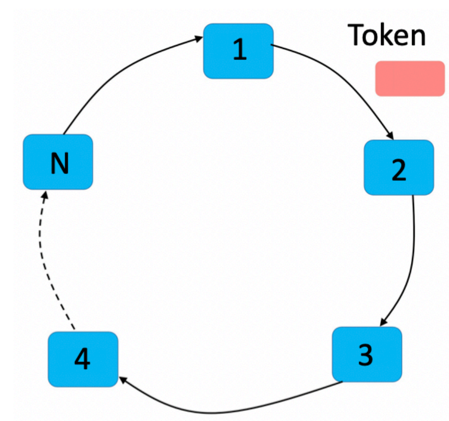
Distributed algorithm



Centralized algorithm



Token Ring Algorithm



Race condition

https://github.com/kevinsuo/CS3502/blob/master/race_condition.c

- A race condition occurs when two or more threads access shared data and they try to change it at the same time.
- The order in which the threads attempt to access the shared data makes the results unpredictable

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int counter = 0;

void *compute()
{
        int i = 0;
        while (i < 100) {
                counter = counter + 1;
                i++;
        }
        printf("Counter value: %d\n", counter);
}

int main()
{
        pthread_t thread1, thread2;

        pthread_create(&thread1, NULL, compute, (void *)&thread1);
        pthread_create(&thread2, NULL, compute, (void *)&thread2);

        pthread_exit(NULL);
        exit(0);
}
```

Race condition occurs for variable counter

```
pi@raspberrypi ~/Downloads> ./race_condition.o
Counter value: 100
Counter value: 200
```

Seem nothing wrong?

Race condition example

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int counter = 0;

void *compute()
{
        int i = 0;
        while (i < 10000) {
                counter = counter + 1;
                i++;
        }
        printf("Counter value: %d\n", counter);
}

int main()
{
        pthread_t thread1, thread2, thread3, thread4, thread5;

        pthread_create(&thread1, NULL, compute, (void *)&thread1);
        pthread_create(&thread2, NULL, compute, (void *)&thread2);
        pthread_create(&thread3, NULL, compute, (void *)&thread3);
        pthread_create(&thread4, NULL, compute, (void *)&thread4);
        pthread_create(&thread5, NULL, compute, (void *)&thread5);

        pthread_exit(NULL);
        exit(0);
}
```

Increase the loop number

Add more threads

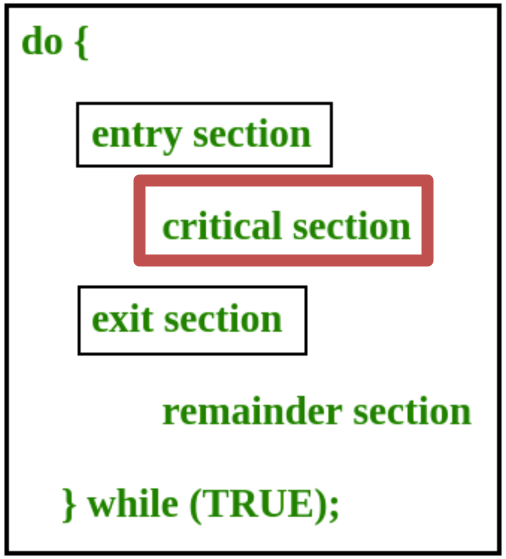
```
pi@raspberrypi ~/Downloads> ./race_condition.o
Counter value: 14467
Counter value: 10410
Counter value: 12080
Counter value: 22745
Counter value: 32725
```

Weird results!

Critical section

- A section of code in a concurrent task that modifies or accesses a resource shared with another task.

- Examples
  - A piece of code that reads from or writes to a shared memory region
  - Or a code that modifies or traverses a shared linked list.



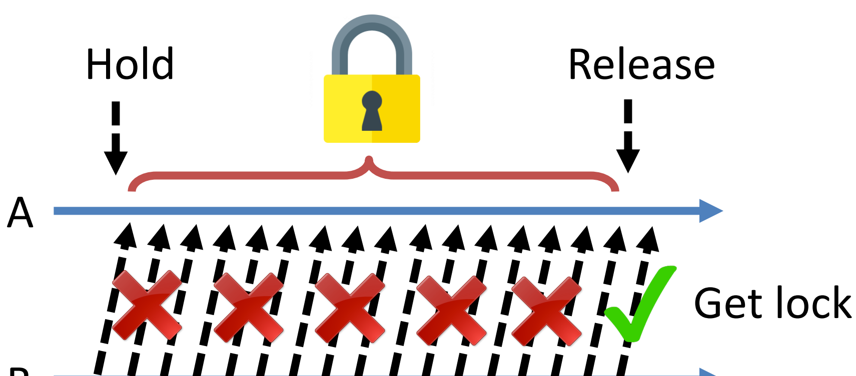
Critical section example

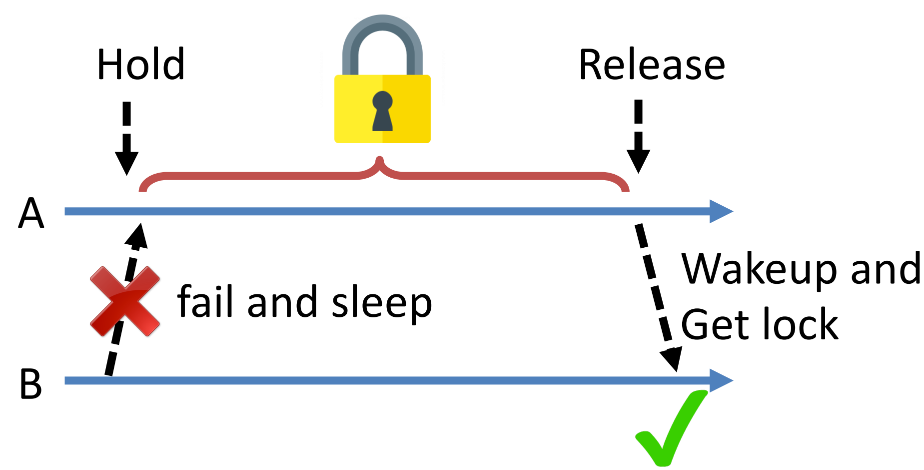
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int counter = 0;

void *compute()
{
	int i = 0;
	while (i < 100) {
		counter = counter + 1;
		i++;
	}
	printf("Counter value: %d\n", counter);
}

int main()
{
	pthread_t thread1, thread2;

	pthread_create(&thread1, NULL, compute, (void *)&thread1);
	pthread_create(&thread2, NULL, compute, (void *)&thread2);

	pthread_exit(NULL);
	exit(0);
}
```

Critical section: All threads read and write the shared counter

Lock (mutual exclusion)

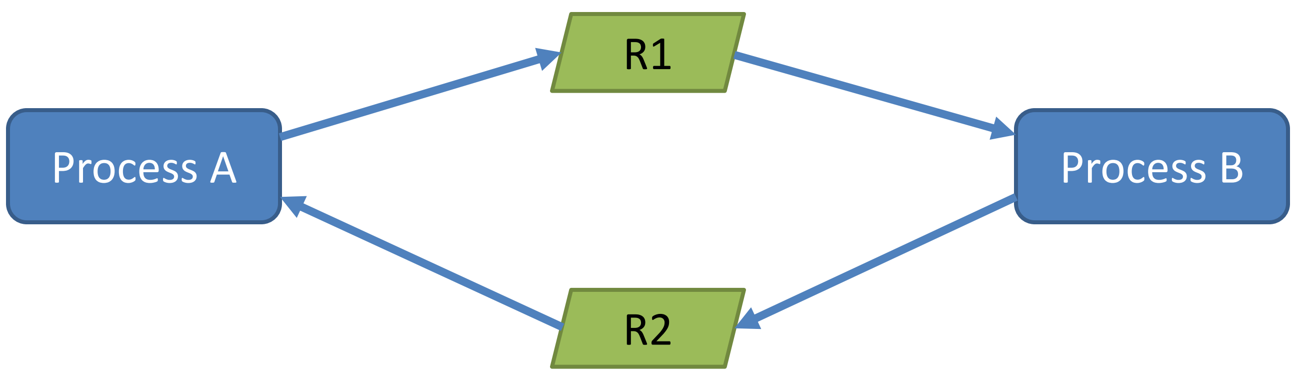
- A lock (mutual exclusion) is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution
- Types of mutual mechanism:
  - Busy-waiting, e.g., spinlock
  - Sleep and wakeup



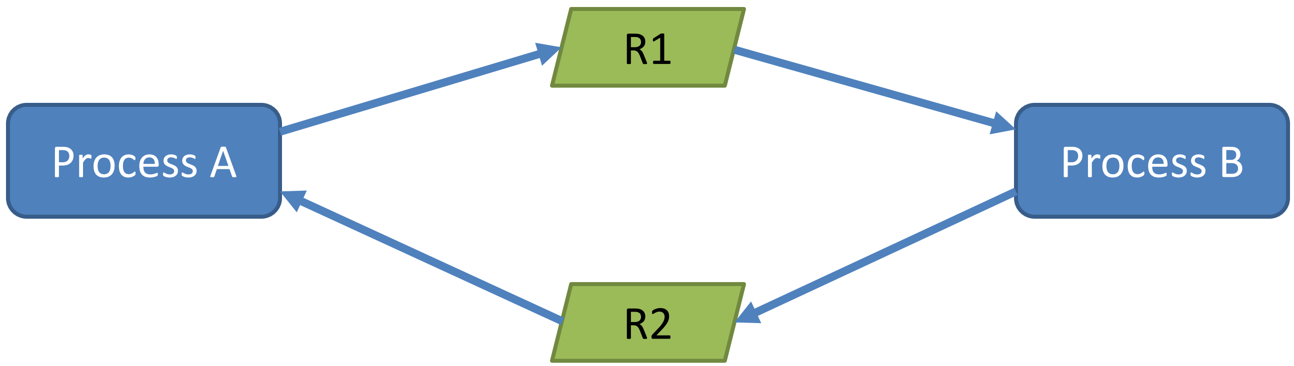


Deadlocks

- When two or more processes stop making progress indefinitely because they are **all waiting for each other** to do something.
  - If process A waits for process B to release a resource, and
  - Process B is waiting for process A to release another resource at the same time.
  - In this case, neither A not B can proceed because both are waiting for the other to proceed.



Deadlock example



Thread 1

```
pthread_mutex_lock(&m1);
/* use resource 1 */
pthread_mutex_lock(&m2);
/* use resources 1 and 2 */
do_something();
pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);
```

Stuck right here

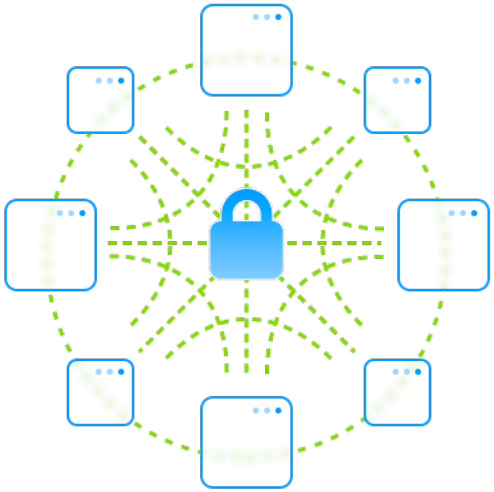
Thread 2

```
pthread_mutex_lock(&m2);
/* use resource 2 */
pthread_mutex_lock(&m1);
/* use resources 1 and 2 */
do_something();
pthread_mutex_unlock(&m1);
pthread_mutex_unlock(&m2);
```

What is a distributed lock?

- Distributed lock refers to a type of lock in which the system is deployed on multiple machines to achieve mutual exclusion between multiple processes in distributed system.

- In order to ensure that multiple processes can see the lock, such the lock is stored in the public storage (such as Redis, Memcache, database and other three-party storage), so that multiple processes can concurrently access the same critical resource, and only one process can access the shared resource at the same time.



Example

- Our online store sells two diamonds
- 5 users (A, B, C, D, E) order at the same time

- If you design the online system, who will get the diamond?

Example

- Different e-commerce companies adopt different strategies:
  - Some e-commerce companies determine who can make a successful purchase based on the order time, First-come-first-serve
  - Some e-commerce companies determine who can make a successful purchase based on the payment time
  - ...
- No matter which policy, the online store must make sure every diamond cannot be sold to two users

Example

- Different e-commerce companies adopt different strategies:
  - Some e-commerce companies determine who can make a successful purchase based on the order time, First-come-first-serve
  - Some e-commerce companies determine who can make a successful purchase based on the payment time
  - …
- No matter which policy, the online store must make sure every diamond cannot be sold to two users. In large-scale distributed systems, we use ***distributed locks*** to realize that.

Distributed Locks

- 3 ways to implement distributed locks:
  1. Distributed locks based on databases, here databases refer to relational databases;
  2. Distributed locks based on caches;
  3. Distributed locks based on ZooKeeper.

Some Key Questions for Distributed Locks

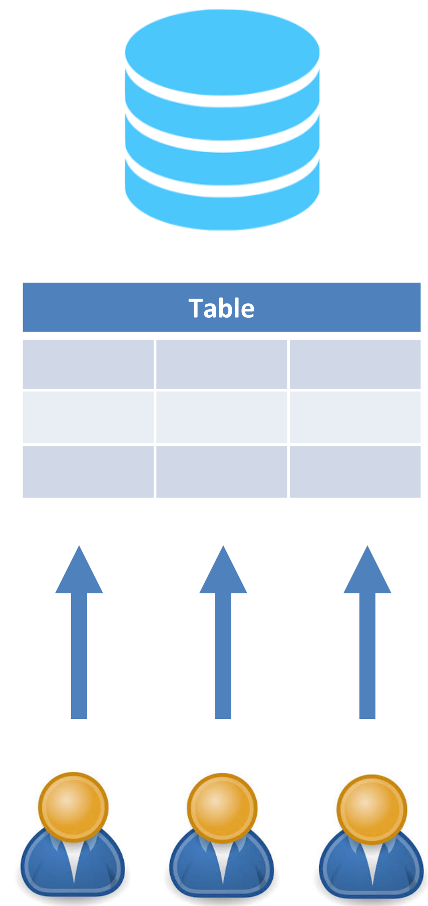
- Mutual exclusion:
  - A distributed lock should be able to guarantee that a resource or a method can only be operated by one process of one machine at a time.
- Preventing deadlock:
  - Even if one process does not actively unlock due to crash while holding the lock, it can guarantee that other processes can obtain the lock.
- Reentrancy:
  - A critical resource can be accessed multiple times when the process has not released the lock.
- Performance:
  - Have a high availability for lock acquisition and release.

1. Distributed locks based on databases

- The easiest way is to create a ***lock table***, and then manipulate the data in the table to achieve distributed locks

- Lock a resource → add a record to this table; Release the lock → delete this record.

- The database has consistent constraint on shared resources. If multiple requests are submitted to the database at the same time, the database will ensure that **only one** operation will be successful, and the thread that successfully operates will obtain a lock.

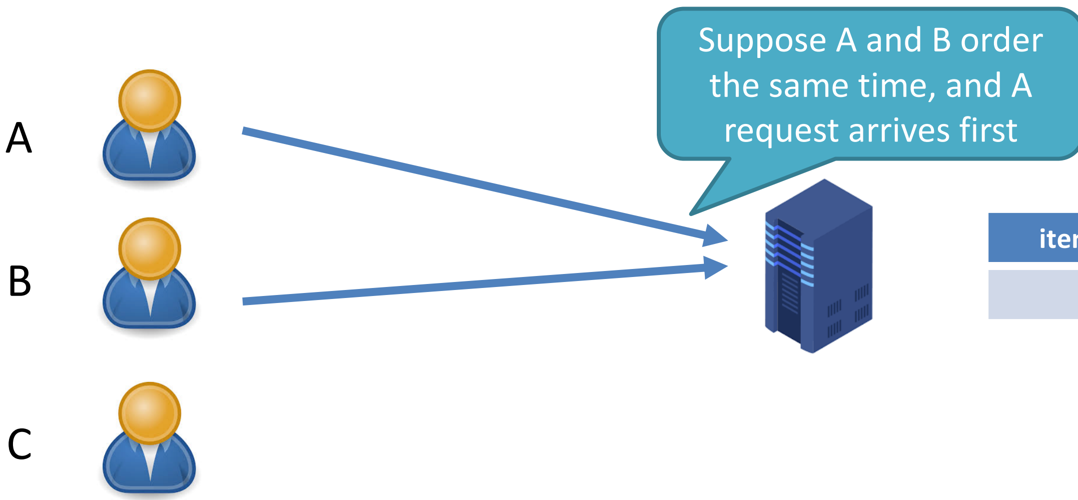


1. Distributed locks based on databases

- Problem for database:
  - The database needs to read/write the hard disk. Frequent interaction with the database will cause large I/O overhead
  - Only suitable for scenarios with low concurrency and low performance requirements

1. Distributed locks based on databases

- Example:
  - Our online store sells 2 diamonds
  - 5 users (A, B, C, D, E) order at the same time
  - A orders 1 diamond, B orders 2 diamonds, C orders 1 diamond

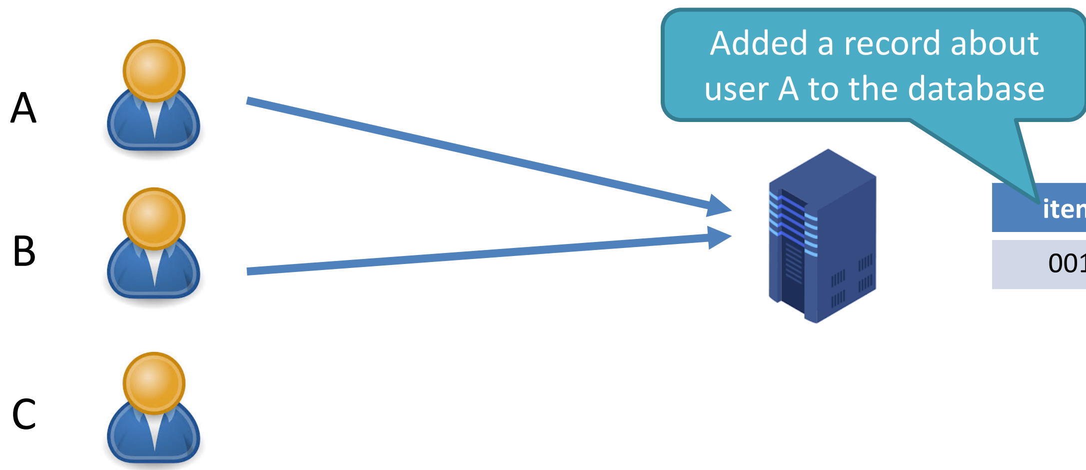


item	user	number

Diamond: 2

1. Distributed locks based on databases

- Example:
  - Our online store sells 2 diamonds
  - 5 users (A, B, C, D, E) order at the same time
  - A orders 1 diamond, B orders 2 diamonds, C orders 1 diamond



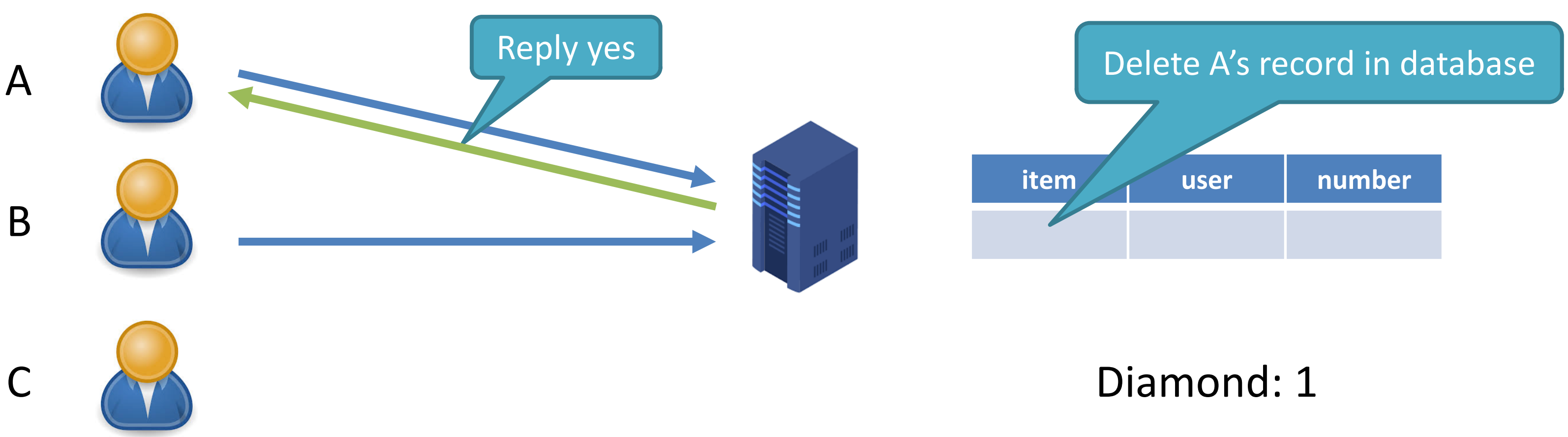
item	user	number
001	A	1

Diamond: 1

2->1

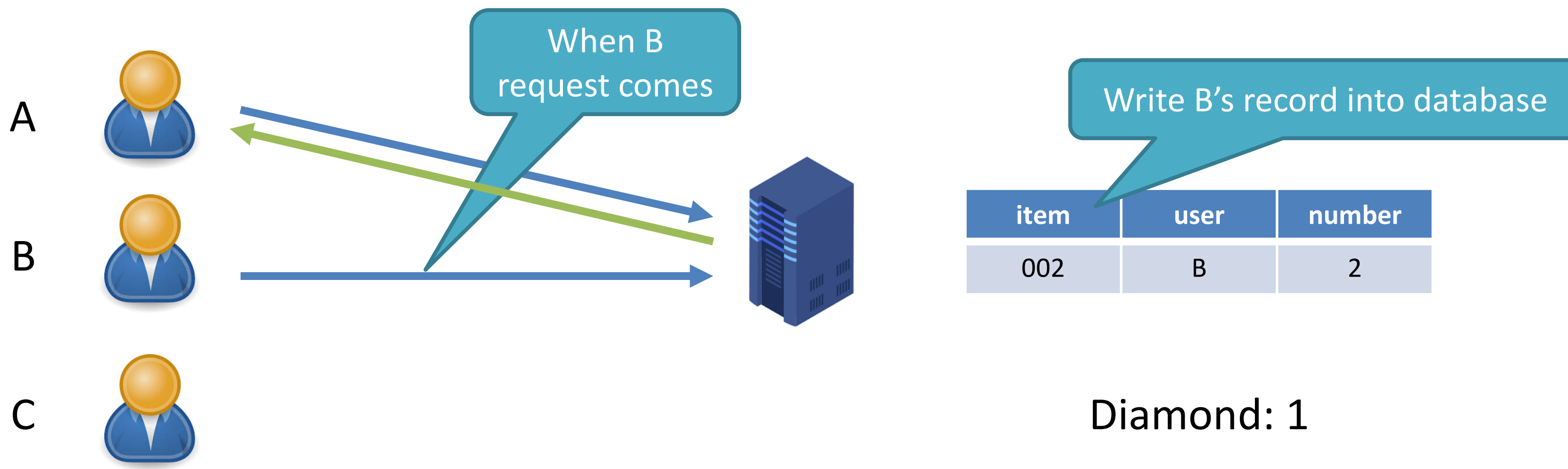
1. Distributed locks based on databases

- Example:
  - Our online store sells 2 diamonds
  - 5 users (A, B, C, D, E) order at the same time
  - A orders 1 diamond, B orders 2 diamonds, C orders 1 diamond



1. Distributed locks based on databases

- Example:
  - Our online store sells 2 diamonds
  - 5 users (A, B, C, D, E) order at the same time
  - A orders 1 diamond, B orders 2 diamonds, C orders 1 diamond

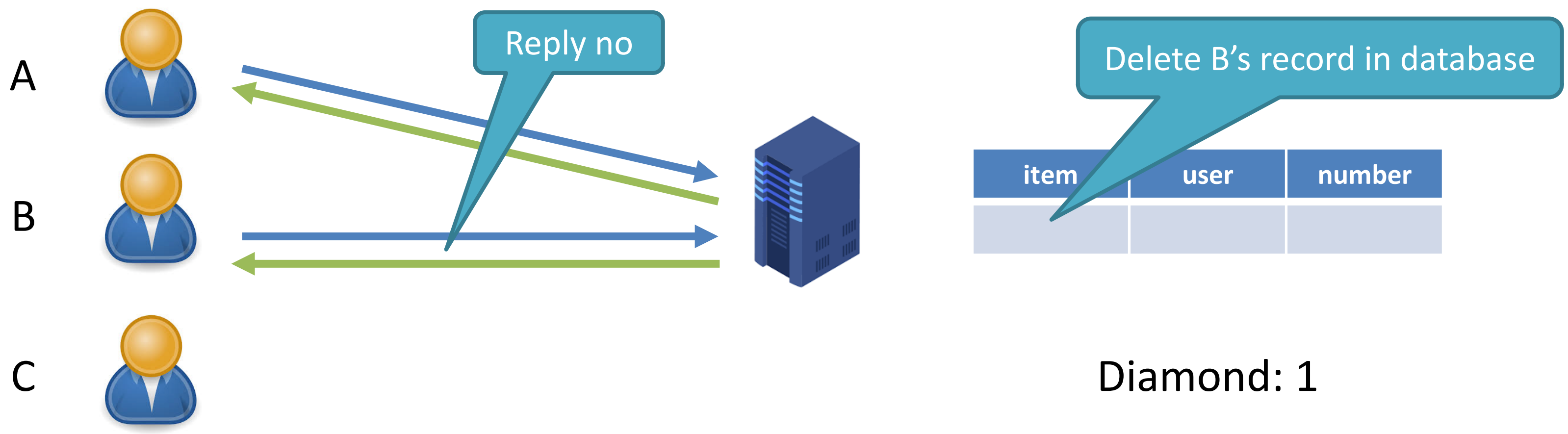


item	user	number
002	B	2

Diamond: 1

1. Distributed locks based on databases

- Example:
  - Our online store sells 2 diamonds
  - 5 users (A, B, C, D, E) order at the same time
  - A orders 1 diamond, B orders 2 diamonds, C orders 1 diamond

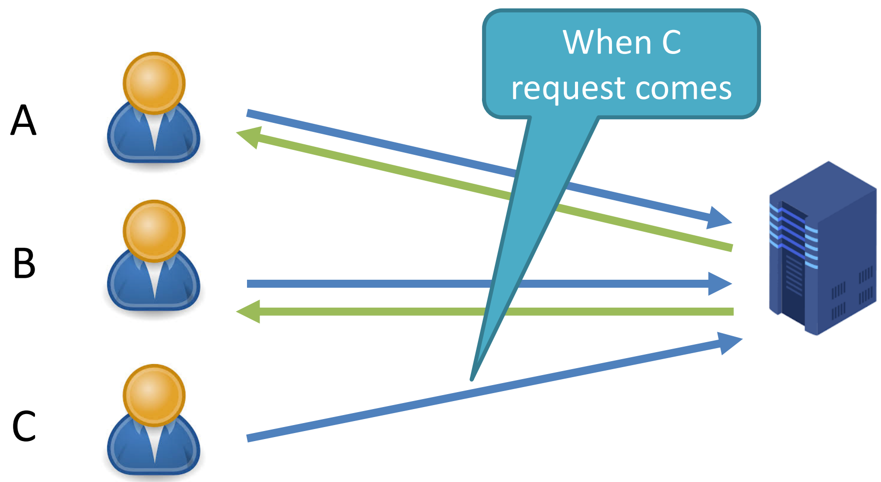


item	user	number

Diamond: 1

1. Distributed locks based on databases

- Example:
  - Our online store sells 2 diamonds
  - 5 users (A, B, C, D, E) order at the same time
  - A orders 1 diamond, B orders 2 diamonds, C orders 1 diamond



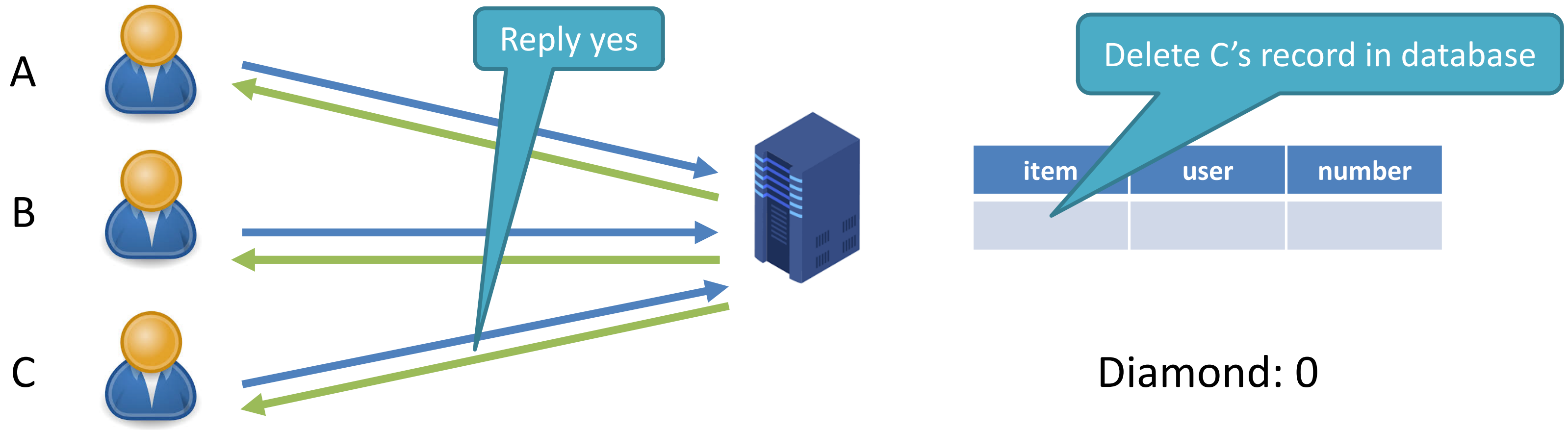
Write C’s record into database

item	user	number
003	C	1

Diamond: 0

1. Distributed locks based on databases

- Example:
  - Our online store sells 2 diamonds
  - 5 users (A, B, C, D, E) order at the same time
  - A orders 1 diamond, B orders 2 diamonds, C orders 1 diamond



1. Distributed locks based on databases

- Advantages:
  - Easy to implement a distributed lock, just create a lock table and create a record in the lock table for the applicant: get the lock when the record is successfully established, release the lock when the record is deleted

1. Distributed locks based on databases

- Disadvantages:
  - Single point of failure. Once the database is unavailable, the entire system will crash.
  - Deadlock problem. The lock based on database has no expiration time and the process that has not acquired the lock has to wait.
    - If the lock holder falls into loop or the unlock operation fails, the lock record will always exist in the database, and other processes cannot acquire the lock.

2. Distributed locks based on caches

- Based on database → introduce lots of I/O operation on hard disk

- Cache-based means that the data is stored in the memory and does not need to be written to disk, which reduces I/O reading and writing overhead.

2. Distributed locks based on caches

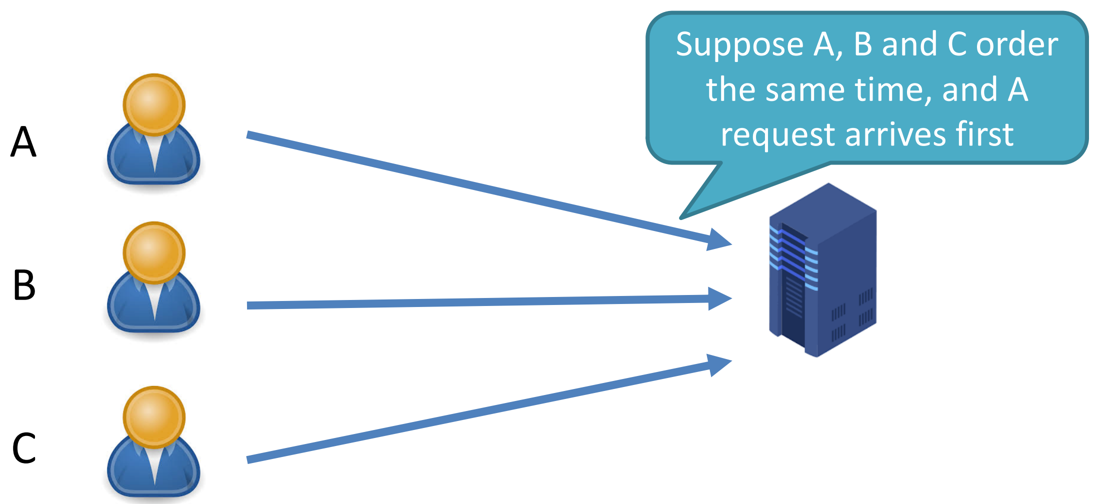
- How the cache-based lock works?

- Key-value store:
  - Key: lock ID
  - Value: currentTime + timeout, after the process gets the lock, the lock will be released by the system when the time is out
  - Function setnx(key, value) works on the table
    - Return 1 means getting the lock and writing one record into the table
    - Return 0 means the lock is used by other processes

Key	Value
1	
2	
3	
4	
…	

2. Distributed locks based on caches

- Example:
  - Our online store sells 2 diamonds
  - 5 users (A, B, C, D, E) order at the same time
  - A orders 1 diamond, B orders 2 diamonds, C orders 1 diamond

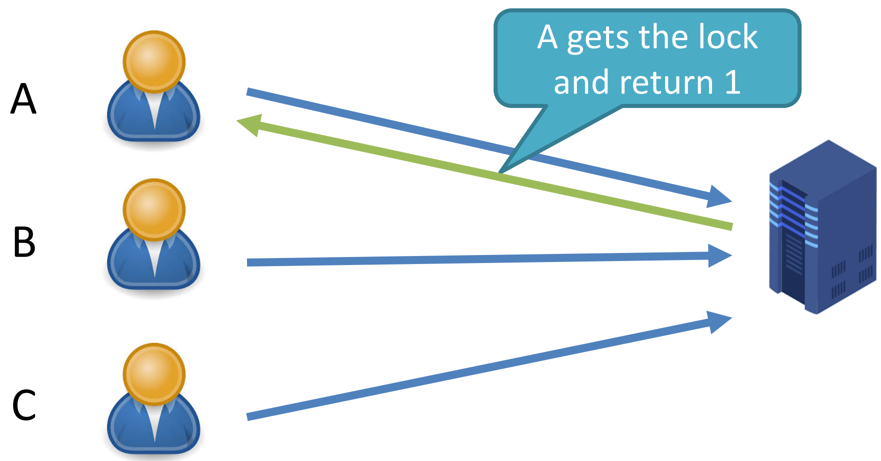


Key	Value
1	T_1+T_{out}
…	

Diamond: 2

2. Distributed locks based on caches

- Example:
  - Our online store sells 2 diamonds
  - 5 users (A, B, C, D, E) order at the same time
  - A orders 1 diamond, B orders 2 diamonds, C orders 1 diamond

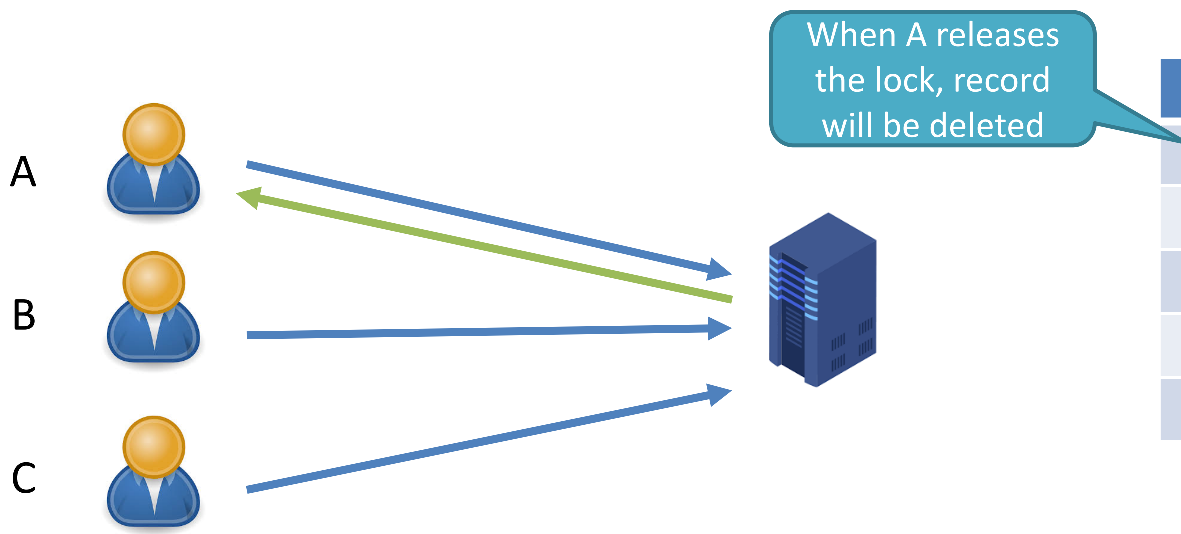


Key	Value
1	T_1+T_{out}
...	

Diamond: 2

2. Distributed locks based on caches

- Example:
  - Our online store sells 2 diamonds
  - 5 users (A, B, C, D, E) order at the same time
  - A orders 1 diamond, B orders 2 diamonds, C orders 1 diamond

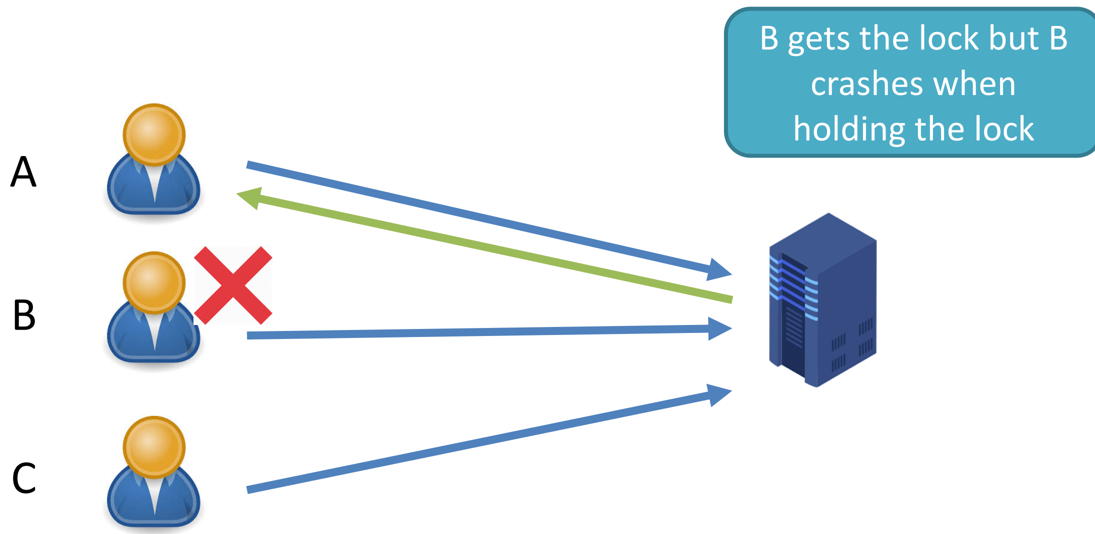


Key	Value
…	

Diamond: 1

2. Distributed locks based on caches

- Example:
  - Our online store sells 2 diamonds
  - 5 users (A, B, C, D, E) order at the same time
  - A orders 1 diamond, B orders 2 diamonds, C orders 1 diamond

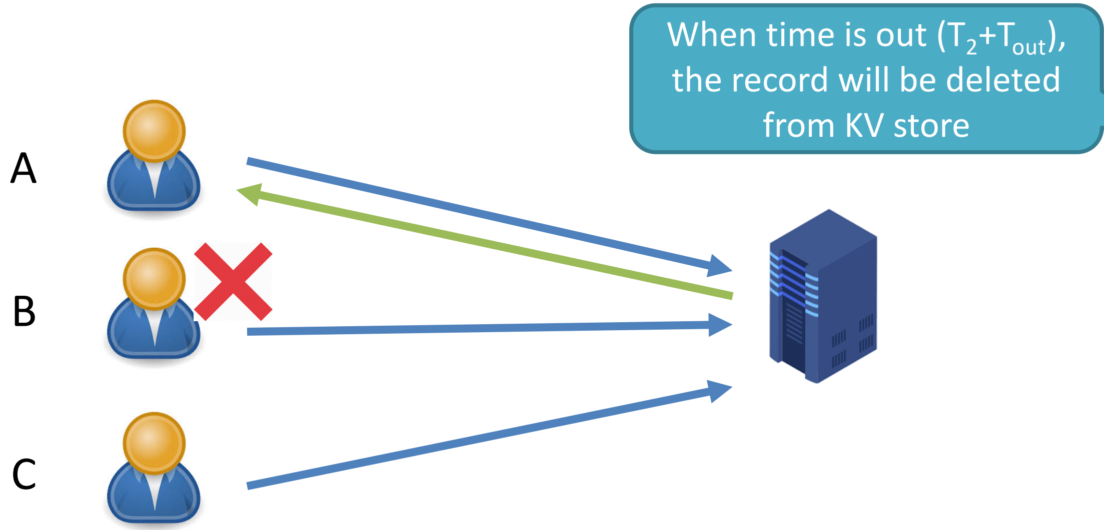


Key	Value
1	T_2+T_{out}
…	

Diamond: 1

2. Distributed locks based on caches

- Example:
  - Our online store sells 2 diamonds
  - 5 users (A, B, C, D, E) order at the same time
  - A orders 1 diamond, B orders 2 diamonds, C orders 1 diamond

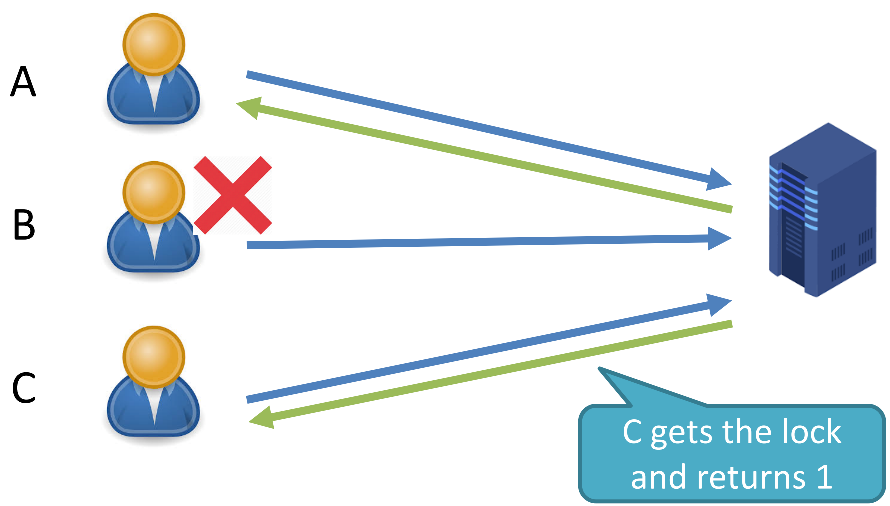


Key	Value
...	

Diamond: 1

2. Distributed locks based on caches

- Example:
  - Our online store sells 2 diamonds
  - 5 users (A, B, C, D, E) order at the same time
  - A orders 1 diamond, B orders 2 diamonds, C orders 1 diamond



Key	Value
1	T_3+T_{out}
…	

Diamond: 0

2. Distributed locks based on caches

- Advantages:
  - Better performance. Data is stored in memory instead of disk, avoiding frequent IO operations.
  - Many caches can be deployed across clusters, avoiding single points of failure.
  - It can directly set the timeout to control the release of locks. More flexible.

2. Distributed locks based on caches

- Disadvantages:
  - Simply using timeout might cause failure when releasing locks
    - One process executes longer (100ms) than it expects (50ms), while the lock is timeout (in 50ms). It will cause error when releasing locks earlier.

3. Distributed locks based on ZooKeeper

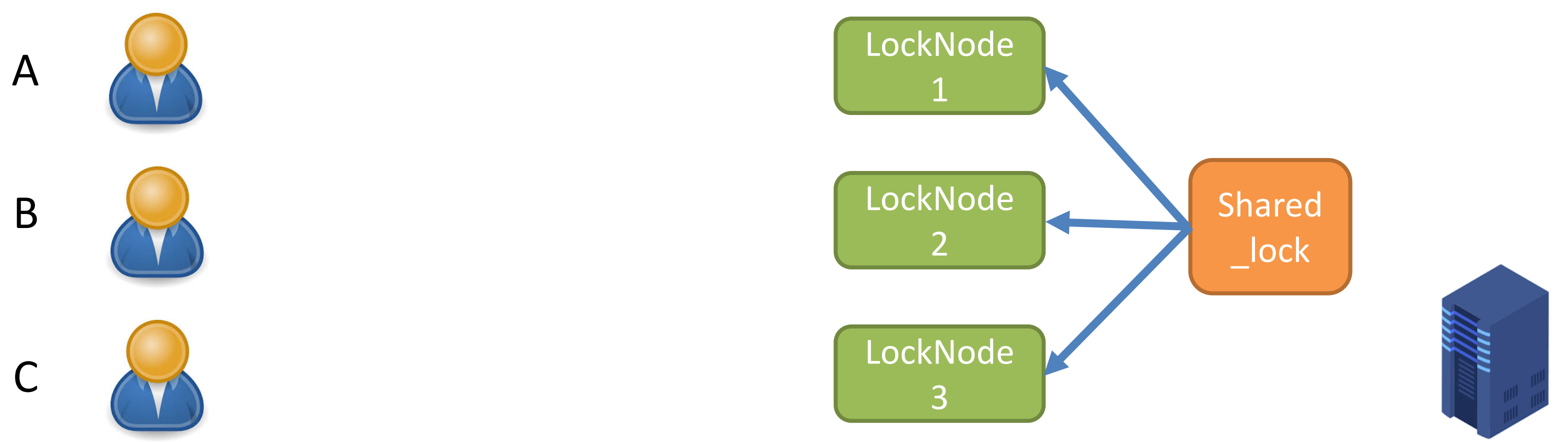
- ZooKeeper implements distributed lock based on tree data storage structure

- ZooKeeper consists 4 nodes:
  - Persistent node. This will always exist in ZooKeeper.
  - Persistent order node. When a node is created, ZooKeeper assigns numbers to the nodes according to when the node is created.
  - Temporary node. This will be deleted when not used.
  - Temporary sequence node: When a node is created, ZooKeeper assigns numbers to the nodes according to when the node is created.

3. Distributed locks based on ZooKeeper

- Example:

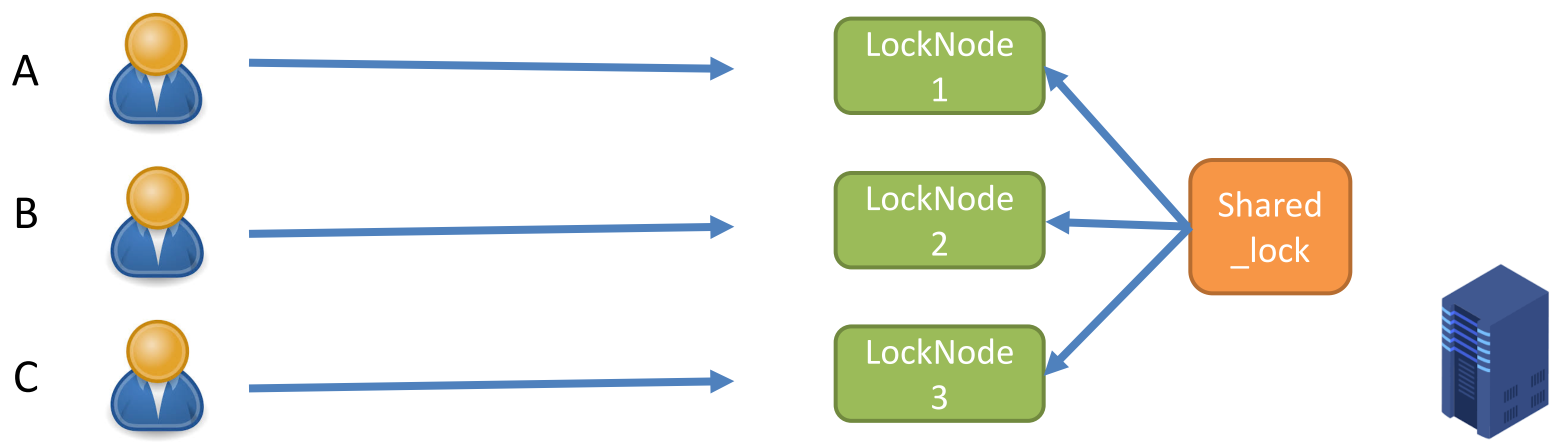
Step 1: Under the persistent node shared_lock, create a temporary sequential node for each process (LockNode i).



3. Distributed locks based on ZooKeeper

- Example:

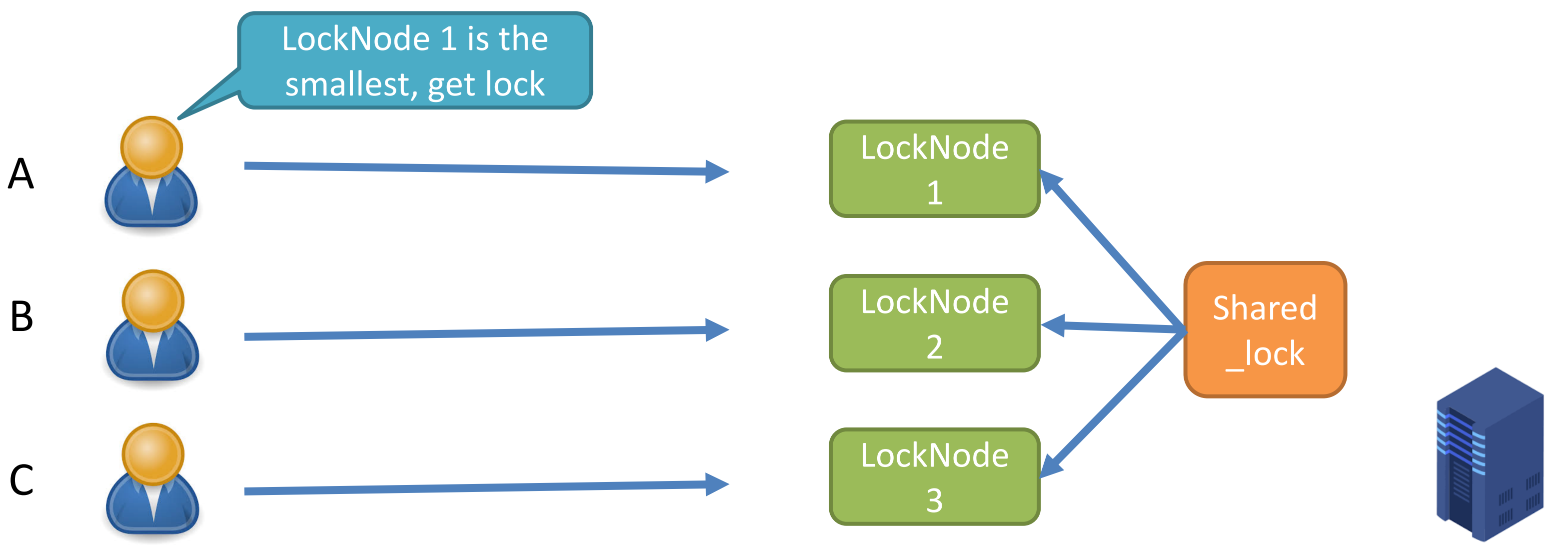
Step 2: Each process gets a list of all temporary nodes in the shared_lock directory



3. Distributed locks based on ZooKeeper

- Example:

Step 3: Each node determines whether its own number is the smallest of all nodes under shared_lock. If it is the smallest, just acquires the lock.

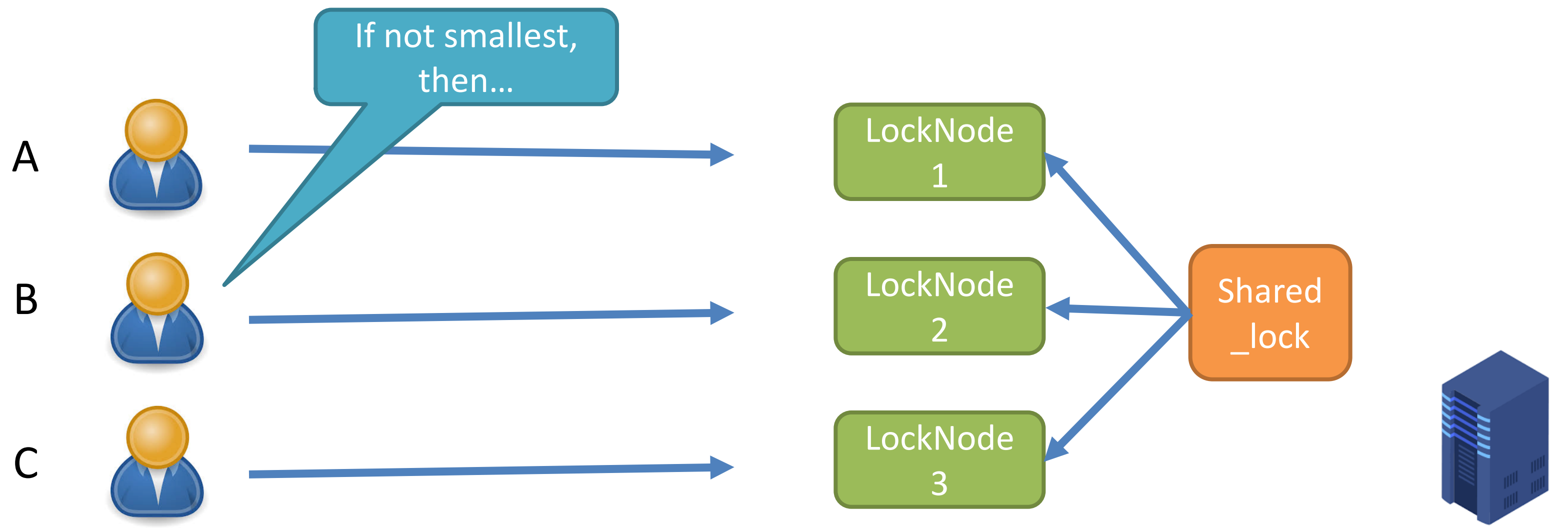


3. Distributed locks based on ZooKeeper

- Example:

  Step 4: If the temporary node number to this process is not the smallest:

  - If the request is to read: check whether there exist write requests with smaller number, if so, wait;
  - If the request is to write: check whether there exist read requests with smaller number, if so, wait;

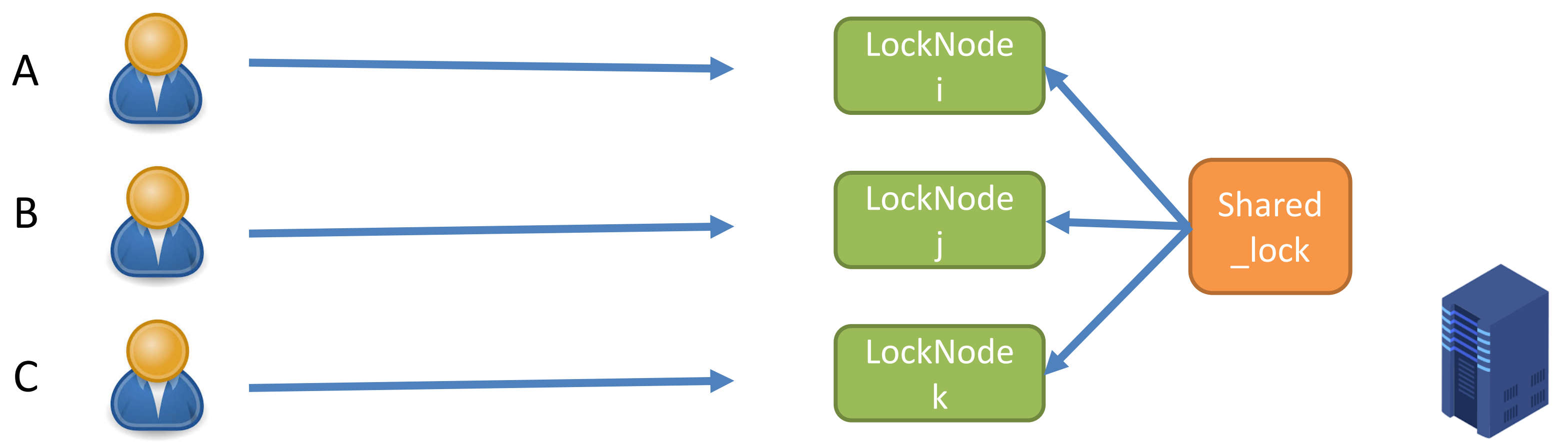


3. Distributed locks based on ZooKeeper

- Advantages:
  - ZooKeeper locks can solve problems such as single points of failure, non-reentrant, deadlock, etc.
  - ZooKeeper provides APIs for distributed locks, easy to implement

3. Distributed locks based on ZooKeeper

- Disadvantages:
  - Need to add and delete temporary nodes frequently
  - Low performance compared to Cache locks



Comparison

	Distributed locks implementation
Design idea (from easy to hard)	Databased based locks > Cached based locks > ZooKeeper based locks
Implementation difficulty (from low to high)	ZooKeeper based locks < Cached based locks < Databased based locks
Performance (from low to high)	Databased based locks < ZooKeeper based locks < Cached based locks
Reliability (from low to high)	Databased based locks < Cached based locks < ZooKeeper based locks