CLOUD / DISTRIBUTED SYSTEMS: HANDLING CONCURRENCY / SYNCHRONIZATION WITH DB LOCKING

Andrew Rembrandt,
andrew@3ap.ch

TechHub Swiss
24 February, 2022

3AP Lightnight talk / 'Brain snack'
14 April, 2022

WHEN & METHODS OF SYNCHRONISATION / LOCKING

- Why/when
  - Multiple instances of a service (cloud / HA / ...)
  - Exclusive locking scenario
  - Unable to implement ordered events / requests
    - Cloud pubsub Ordering Keys
    - Kafka Partitioning

MULTI-PROCESS LOCKING

- Datastore
  - DB (Postgres, Oracle ...)
  - NoSQL (Redis, Zookeeper, Hazelcast, MongoDB, ...)

DB LOCKING

- Pessimistic lock
  - Row / table level => contention / performance impact

```
@Lock(LockModeType.PESSIMISTIC_WRITE)
@QueryHints({@QueryHint(name = "javax.persistence.lock.timeout",
value = "1000")})
@Query("SELECT p FROM person p WHERE p.id = ?1")
Optional<PersonEntity> findByIdPessimisticLocked(Long id);
```

- Optimistic locking
  - Atomic compare and set with a version or timestamp

```
@Lock(LockModeType.OPTIMISTIC_FORCE_INCREMENT)
@Query("FROM shared_lock WHERE lockType = :lockType AND sourceId =
:sourceId")
Optional<SharedLockEntity> findByLockTypeAndSourceIdLocked(LockType
lockType, Long sourceId);
```

OPTIMISTIC LOCKING (CONT)

- Typical sql statement:

```
update
    item
set
    version=1,
    amount=10
where
    id='abcd1234'
and
    version=0
```

- Existing entity locking (version / lock timestamp column)
- Dedicated lock table
  - Decoupling provides more flexibility => 'multi-process mutex'

REACTIVE EXAMPLE

```
  public Mono<FolioEntity> createOrUpdateFolioTransaction(Event event)
{
    return doCreateOrUpdateFolioTransaction(event)
        .retryWhen(
            Retry.backoff(5, Duration.ofMillis(300))

.filter(OptimisticLockingFailureException.class::isInstance)
                .doBeforeRetry(
                    s ->
                        log.warn("Optimistic locking failure while
createOrUpdateFolioTransaction for ...", ...)
  }


  private Mono<FolioEntity> doCreateOrUpdateFolioTransaction(Event
event) {
    return Mono.defer(() -> {
            FolioEntity folioEntity = null;
            Optional<FolioEntity> folio =

folioRepository.findByFolioIdLocked(event.getObjectId());

            if (folio.isEmpty()) {
              folioEntity = mapper.createFolioEntity(event);
            } else {
              folioEntity = mapper.updateFolioEntity(event,
folio.get());
```

```
            }
            return folioRepository.save(folioEntity);
          })
      .flatMap(folioRepositoryService::addTransientFields);
  }
```

GENERIC DB SHARED LOCK TABLE

```
  public <T> Mono<T> acquireLock(Long sourceId, String reportableId,
    LockType lockType, Supplier<Mono<T>> callbackOnLockAcquisition) {
    return Mono.defer(
            () ->
                sharedLockRepository
                    .findByLockTypeAndSourceIdLocked(lockType,
sourceId)
                    .or(() -> Optional.of(

sharedLockRepository.save(SharedLockEntity.builder()

.lockType(lockType).sourceId(sourceId).failureCount(0)
                                    .expiryCount(0).build())))
                    .flatMap(
                        lock -> {
                          if (lock.getAcquiredLockTime() != null)
return Optional.empty();
                          lock.setAcquiredLockTime(Instant.now());
                          val updatedLock =
sharedLockRepository.save(lock);
                          return Optional.of(updatedLock);
                        }))
        .onErrorResume( // OpLock exeception for existing rows,
integrity violation for conflicting row inserts
            t -> t instanceof ObjectOptimisticLockingFailureException
|| t instanceof DataIntegrityViolationException,
            e -> Mono.empty()
```

```
            })
        .flatMap(lock -> callbackOnLockAcquisition.get()) // We have
acquired the lock, call the callback
        .flatMap(
            resultItem ->
                tenantContext.fromSupplier(
                    () -> {
                      log.info("Remove lock for {}, sourceId: {}",
reportableId, sourceId);
                      sharedLockRepository.unsetLock(lockType,
sourceId);
                      return resultItem;
                    }));
  }
```

5 MINS != EXPERT :)

- To lock or not to lock
  - Complexity trade-off
    - DB locks are simple until it's use grows
  - Idempotent updates
  - Event normalisation
- Audit trail / logging

QUESTIONS

- https://github.com/andrewrembrandt/distributed-locks-talk
- https://blog.mimacom.com/tag/concurrency/