# Leader Election Using NewSQL Database Systems

Salman Niazi[1(✉)], Mahmoud Ismail[1], Gautier Berthou[2], and Jim Dowling[1,2]

[1] KTH - Royal Institute of Technology, Stockholm, Sweden
{smkniazi,maism,jdowling}@kth.se
[2] SICS - Swedish ICT, Stockholm, Sweden
{gautier,jdowling}@sics.se

**Abstract.** Leader election protocols are a fundamental building block for replicated distributed services. They ease the design of leader-based coordination protocols that tolerate failures. In partially synchronous systems, designing a leader election algorithm, that does not permit multiple leaders while the system is unstable, is a complex task. As a result many production systems use third-party distributed coordination services, such as ZooKeeper and Chubby, to provide a reliable leader election service. However, adding a third-party service such as ZooKeeper to a distributed system incurs additional operational costs and complexity. ZooKeeper instances must be kept running on at least three machines to ensure its high availability. In this paper, we present a novel leader election protocol using NewSQL databases for partially synchronous systems, that ensures at most one leader at any given time. The leader election protocol uses the database as distributed shared memory. Our work enables distributed systems that already use NewSQL databases to save the operational overhead of managing an additional third-party service for leader election. Our main contribution is the design, implementation and validation of a practical leader election algorithm, based on NewSQL databases, that has performance comparable to a leader election implementation using a state-of-the-art distributed coordination service, ZooKeeper.

## 1 Introduction

One of the main difficulties when designing a replicated distributed system is to ensure that the nodes will reach agreement on the actions to take. Agreement protocols are complex to design and inefficient in terms of throughput and latency, for example, classical Paxos [1] in a failure-recovery model. As a result, most distributed systems rely on a unique leader node to coordinate the tasks running in the system. For this *leader* pattern to work correctly the nodes need to be able to solve the general agreement problem [2] in order to agree on which one of them is the leader. Solving this problem is the purpose of the leader election protocol.

Leader election protocols are a fundamental building block that play a central role in many scalable distributed systems such as stateful middleware [3],

© IFIP International Federation for Information Processing 2015
A. Bessani and S. Bouchenak (Eds.): DAIS 2015, LNCS 9038, pp. 158–172, 2015.
DOI: 10.1007/978-3-319-19129-4_13

distributed filesystems [4], and distributed databases [5]. The typical role of a leader is to propose global state updates and to disseminate them atomically among the nodes. Having a unique leader is imperative to avoid multiple leaders proposing conflicting updates that would compromise the integrity of the system. Additionally, the failure of the leader should not affect the system availability. Moreover, the detection of the leader failure and the election of a new leader should be low latency events (at most, in the order of seconds).

Implementing an algorithm that provides both uniqueness of the leader and low latency is very challenging. In order to avoid errors and to curtail development time, developers often rely on third-party, standalone coordination services such as Chubby [6] and ZooKeeper [7]. These services have the advantages of being widely used and well tested but they introduce additional operational costs and complexity as they must be kept running on at least three machines if the leader-election service itself is to be highly available.

Many existing distributed systems use highly available relational databases or key-value stores to manage their persistent data. Why not build the leader election service using the database as a shared memory? This would allow developers to exploit the leader pattern without paying the extra operational cost of a dedicated coordination service. Implementing the leader election using shared memory is not a new problem; Guerraoui [8] and Fernandez [9, 10] have shown that a leader election service can be implemented using shared memory. However, in partially synchronous systems, these solutions do not guarantee that there will be a unique leader while the system is unstable. They only guarantee that nodes will eventually agree on a leader once the system has stabilized. As a result, these solutions are not widely used in production systems.

In contrast, we propose an algorithm based on locking and transaction primitives provided by the database to guarantee that there is at most one leader in the system at any given time. Traditional highly available relational database management systems are not suitable for building our leader election service, since it can take long time for transactions to complete if a database node failure occurs, which would slow down the leader election process. However, NewSQL systems have emerged as a new class of distributed, in-memory databases that are optimized for on-line transaction processing (OLTP) and have low timeouts for transactions, thus, making them a viable platform for building our leader election service.

In this paper, we present a practical leader election service based on shared memory in a NewSQL database. Our implementation uses the Network Database (NDB) storage engine for MySQL Cluster [5], but our approach is generalizable to all NewSQL databases. Our main contribution is to prove that two-phase commit can be used to implement practical leader election algorithm. We validate our algorithm and show that its performance is comparable to a leader election algorithm implemented using ZooKeeper [7] for cluster sizes of up to 800 processes.

## 2 NewSQL Database Systems

Although a database can be used as shared memory to implement leader election, some databases do not provide sufficient primitives to implement a reliable leader election service. Our leader election service requires a highly available database with support for transactions and locking primitives. Additionally, the database must ensure that database node failures and slow clients do not cause transactions to take too long to complete (commit or abort). We will now discuss different types of database systems and their suitability for leader election service.

**Highly Available Relational Databases** typically provide high-availability using either an active-standby replication protocol that provides eventually consistent guarantees for data (as used in SQLServer [11] and MySQL [12]) or a shared-state replication protocol, as used in Oracle RAC [13]. For the active-standby model, a crash of the active node will result in the leader election service being unavailable until the standby node takes over. There are no guarantees on how long this failover will take, and, in practice, it can take from seconds up to minutes to complete depending on the degree of lag at the standby node. Moreover, until the failover completes, the system remains vulnerable to failures as the standby node now becomes a single point of failure. For shared-state databases, it can take up to a minute for transactions to complete if a database node failure occurs (the default distributed lock timeout in Oracle RAC is 60 seconds [13]). For these reasons, traditional highly available relational databases are not suitable for building leader election services.

**NoSQL Systems** are highly available, but they only provide eventually consistent guarantees for data [14,15]. This make them unsuitable as the basis for a leader election service, as eventually consistent data may lead to multiple leaders in the system.

**NewSQL Systems** are a new class of highly available databases that can scale in performance to levels reached by NoSQL systems, but still provide ACID guarantees and a SQL-based declarative query interface [16, 17]. NewSQL systems achieve high performance and scalability by redesigning the internal architecture of traditional databases, often to a shared-nothing architecture, that take better advantage of modern multi-core hardware along with increasingly cheap in-memory storage. NewSQL systems can be scaled-out by adding additional nodes. What makes NewSQL systems a viable platform for building a leader election service is that they typically have low timeouts for locks and transactions. Some notable NewSQL systems are the NDB storage engine for MySQL Cluster [5], FoundationDB [18], and VoltDB [19].

## 3 System Model and Eventual Leader Election

**Processes.** The system consists of a time varying finite set of processes $p_1$, $p_2$, $p_3$ ... $p_n$. Each process has a unique id assigned by a function that returns monotonically increasing ids. All the processes are assumed to behave according to their protocol specification, that is, the processes are not Byzantine. A process can fail by crashing, but until a process crashes it will execute the protocol and it will not halt for an indefinite amount of time. When a process fails it stops executing all operations. A failed process can recover after the failure, but it is assigned a new id by the monotonically increasing function. There is no restriction on the number of processes that can fail or join during the execution of leader election protocol.

The underlying system is partially asynchronous, as it is impossible to develop a leader election service for purely asynchronous systems [20]. In partially synchronous systems there are positive upper and lower bounds on the communication and processing latency. These synchrony primitives of the system are eventually determined by the application. Before these time bounds are determined, a distributed application may not function as expected. The time after which the lower and upper time bounds hold is called global stabilization time (GST). The protocol proceeds in rounds. The duration of these rounds expand until the GST is reached. Moreover, each process' local clock drift is significantly smaller than the round time of protocol.

**Shared Memory.** All processes communicate through reliable atomic registers (shared memory) implemented using rows in a table in the database. A reliable atomic register is always available, moreover, if two read operations $r_1$ and $r_2$ return $w_1$ and $w_2$ respectively and $r_1$ precedes $r_2$ then $w_1$ precedes $w_2$. Atomic registers can easily be implemented in a relational database using a strong enough transaction isolation level. To be considered correct, a process must successfully read and write the register values within a heartbeat period, that can expand during the execution of the protocol.

**Leader Election Service.** Eventually a correct process with the lowest id in the system will be elected as the leader. The service ensures that a correct leader is elected again in the subsequent rounds. Our service provides stronger guarantees than $\Omega$ [2]. With $\Omega$ there could be multiple leaders if the GST has not been reached. With the help of transactions, our leader election guarantees at most one leader at any given time and guarantees following properties:

- **Integrity**: there should never be more than one leader in the system.
- **Termination**: a correct process eventually becomes a leader.
- **Termination**: all invocations of the primitive *getLeader()* invoked by a correct process should return the leader's id.

## 4 Leader Election in a NewSQL Database

Logically, all processes communicate through shared registers (implemented as rows in a table). Each process has its own counter that it updates periodically (in a transaction) to indicate that it is still alive. Each process maintains a local history of the all processes descriptors. Process descriptor contains id, counter, ip and port information. Using the local history a process is declared dead if it fails to update its counter in multiple consecutive rounds. A process declares itself to be the leader when it detects that it has the smallest id among all the alive processes in the system. The leader evicts failed processes, and it is also responsible for increasing the heartbeat *round time* to accommodate slow processes.

All processes run in parallel, concurrency control could be handled with a transaction isolation level set to *serializable*, ensuring that conflicting transactions will execute one after another. For example, if two processes, $P_a$ and $P_b$, want to become leader simultaneously then the transactions will automatically be ordered such that if $P_a$ manages to execute first then $P_b$ is put on hold. The transaction $P_b$ waits until transaction $P_a$ has finished.

However, due to poor performance [21], NewSQL systems typically do not provide serializable as the default transaction isolation level, if they even support it at all. The strongest isolation level supported by NDB is the *read committed* isolation level, guaranteeing that any data read is committed at the moment it is read. However, it is not sufficient for implementing a reliable leader election service. We use row-level locking to implement stronger isolation levels for transactions. Row-level locking complicates the design, but allows for more fine-grained concurrency control and thus, higher throughput.

**Algorithm 1.** Leader Election

**Require:** VARS ▷ Atomic Register. Holds max id,$T_p$ and evict flag
**Require:** DESCRIPTORS ▷ Set of atomic registers that stores all descriptors

```
1: id = ⊥, role = non_leader, leader = ⊥
2: procedure periodicHeartbeatTask
3:     while true do
4:         begin transaction                                  ▷ Begin new round
5:             if role = leader | id = ⊥ | forceExclusiveLock  then
6:                 acquire exclusive lock on VARS register
7:                 forceExclusiveLock = false
8:             else
9:                 acquire shared lock on VARS register
10:            read all DESCRIPTORS                           ▷ No locks needed
11:
12:            updateCounter()
13:            leaderCheck()
14:            DESCRIPTORS ≫ history                          ▷ Add to history
15:
16:            T_p = VARS.getTimePeriod()
```

```
17:             if role = leader & VARS.evictFlag = true then
18:                 T_p = VARS.updateTimePeriod(T_p + Δ)
19:                 VARS.evictFlag = false
20:             L_hbt = currentTime()                    ▷ Leader's lease start time
21:         commit transaction
22:         sleep(forceExclusiveLock ? 0 : T_p)  ▷ Immediately retry with higher locks

23: procedure updateCounter
24:     if id ∈ DESCRIPTORS then
25:         updateDescriptor(id, getCurrentCounter()+1)
26:     else
27:         if id != ⊥ then                                        ▷ Case: evicted
28:             if transaction lock mode is not exclusive then
29:                 forceExclusiveLock = true
30:                 VARS.setEvictFlag()
31:                 return
32:         id = VARS.incrementMaxID()
33:         insertDescriptor(id)

34: procedure leaderCheck
35:     P_s = history.getSmallestAliveProcess()
36:     if P_s.id = id then
37:         if transaction lock mode is not exclusive then
38:             forceExclusiveLock = true
39:             return
40:         role = leader
41:         removeDeadNodes()                                  ▷ Evict processes
42:     else
43:         role = non_leader
44:         leader = P_s                                       ▷ Possible leader

45: procedure isLeader
46:     if role = leader then
47:         elapsed_time = currentTime() - L_hbt                   ▷ Lease check
48:         if elapsed_time < (T_p * Max_mhb - μ) then
49:             return true
50:     return false

51: procedure getLeader
52:     if role = leader & isLeader() then
53:         return this
54:     else if role = non_leader then
55:         return leader
```

### 4.1 Shared Memory Registers

We implement shared memory registers using rows in tables. Transactions ensure atomicity of the registers. The atomic register VARS stores global parameters such as the maximum allocated process id, and the duration of heartbeat rounds. The maximum allocated process id is used in monotonic id generation. It also

stores a boolean flag that is used to change the heart beat round time to cater for slow processes. VARS is backed by single row in a table that contains all the global variables. DESCRIPTORS represents a set of registers that store information about all the alive process. It is backed by a table where each row contains a process descriptor.

Our database, NDB, supports two main locking modes: *shared* (read) and *exclusive* (write) locks. Multiple transactions can concurrently obtain shared locks on an object. However, only one transaction can obtain an exclusive lock on an object.

Every processes is an element of one of two disjoint sets. The first set contains the majority of processes. These are non-leader processes that only update their counter in each round. The second set of processes contains the leader process, processes contending to become the leader, and processes that have not yet obtained a unique id. Usually this group is very small, and it depends upon the amount of churn in the system.

All the processes in the first set can run concurrently as they only update their own counters. However, the processes in the second set may take decisions or change the state of the global variables which can effect other processes. Therefore, all the transactions of the processes in the second set are serialized. For example, assume the leader wants to evict a slow process. By taking exclusive locks, the leader process prevents the slow process from committing any updates to shared state. When the slow process' transaction is scheduled, it will notice its id is missing and it will have to rejoin the system. Similarly, if two processes are contending to become the leader then their operations should be serialized to prevent a system state where there are multiple leaders. Moreover, the first round of new processes are also serialized to generate monotonically increasing ids.

### 4.2 Leader Election Rounds

Each round encapsulates its operations in a transaction that starts by taking a lock on the VARS register which acts as a *synchronization point*. Processes belonging to the first group acquire shared locks while the processes in the second group acquire exclusive locks on the VARS register, lines 5 – 9.

After acquiring locks on the VARS register all the processes descriptors are read without any locks (read committed). The processes update their counters and check if they can become the new leader. Each process maintains a history of process descriptors to identify dead processes, lines 12 – 14. Now, we explain these operations in more detail from the perspective of both groups of processes.

A new process starts by taking exclusive locks in the first round. It obtains a new monotonically increasing id and stores its descriptor, lines 32 –33. An exclusive lock is required to update the maximum process id in the VARS register. An evicted process will not find its descriptor, as it has been deleted. The evicted process cannot obtain a new process id if it does not hold an exclusive lock on the VARS register. In such a case, the transaction is immediately retried using exclusive locks, see lines 5, and 27 – 31. Additionally, the evicted process sets a flag to inform the leader that it was evicted prematurely, see line 30.

The service then checks for changes in the group membership. A process is declared dead if it fails to update its counter in multiple consecutive rounds. The threshold, $Max_{mhb}$, determines the number of rounds a process can miss before it is declared dead. The $Max_{mhb}$ is usually set to $\geq 2$. The process elects itself to be the leader if it has the smallest id among the alive processes. The leader process cleans the DESCRIPTORS register by removing the dead processes. If a non-leader process, that holds a shared lock, finds out that it can become the leader then it immediately retries the transaction with exclusive locks. It becomes the leader and removes the dead descriptors. If the process does not have the smallest id then it sets its role to *non_leader* and stores the descriptor of the process that has smallest id in a local variable, lines 34 – 44.

### 4.3 Global Stabilization Time (GST)

The time bounds for communication and processing latencies are not known in advance. For large systems the initial round time for periodically updating the counter may not be sufficiently long enough so that all processes mange to update their counters in a single round. Moreover, the round time must be automatically adjusted to cater for slow processes; otherwise, the system may not stabilize. In our implementation, only the leader process increases the round time by updating the VARS register (which is read by all processes).

Slow processes are evicted by the leader. When a process finds out that it was evicted, it obtains a new id and set a flag in the VARS register to notify the leader that it was wrongfully suspected. When the leader process finds that the evicted flag is set it increases the round time by a constant value $\Delta$, see lines 16 – 18.

### 4.4 Leader Lease

Our solution ensures that there is never more than one leader in the system. However, this *invariant* is difficult to enforce before the GST has reached. Additionally, in order to reduce contention on the registers, methods like $isLeader()$ and $getLeader()$ return information stored in the local variables. On a slow process these variables may contain stale values. For instance, assume a slow process, $L_a$, becomes the leader. After becoming the leader $L_a$ fails to update its counter in multiple consecutive rounds. Later, a new process becomes a leader and $L_a$ is evicted. However, $L_a$ will remain oblivious of the fact that it has been evicted, and its function *isLeader()* will keep on returning true until $L_a$ manages to read new values from the registers.

In order to ensure *integrity* of the leader election service each leader process stores a local lease. Whenever the leader process updates its counter, it acquires a lease for the duration of $(T_p * Max_{mhb} - \mu)$. The constant, $\mu$, is to accommodate for clock drifts. Before committing the transaction, a timestamp is stored in $L_{hbt}$, which indicates the start of the leader lease time, line 20. The lease is the the maximum time during which the leader cannot be evicted by other processes. If the leader is slow and it fails to update its counter then the lease will eventually

expire and the process will voluntarily yield its leader role, line 45 – 50. The election of a new leader will happen after the lease of previous leader has expired, see theorem 1 for more details.

### 4.5 Dealing with Failures

Note that the transactions only guarantee the atomicity of the registers. Read committed isolation ensures that, during transaction execution, partial results (changes in the registers) are not be visible to other transactions until the transaction has committed. When a transaction fails the database rollbacks only the partial changes in the registers. However, it is the responsibility of the application to rollback all the local variables, such as $role$, $L_{hbt}$, $T_p$, and $id$. For clarity reasons we do not show code listing to rollback local variables.

## 5 Proof

In this section we prove the safety (at most one leader invariant) and the liveness (termination) properties of our leader election algorithm.

**Theorem 1.** *There is never more than one leader in the system.*

*Proof.* In order to prove that there cannot be two leaders, $L_s$ and $L_n$, in the system at the same time we will prove that (I) two processes cannot declare themselves as leader simultaneously (II) a process cannot become leader while another process still sees itself as leader.

**Case I:** In order to become a leader both the processes, $L_s$ and $L_n$, need to acquire exclusive locks at the beginning of the transaction. As a result the transactions for $L_s$ and $L_n$ will be serialized. If $L_s$ manages to acquire the exclusive lock first, it will update the counter and elect itself as a leader (assuming the transaction commits). $L_n$ will wait until $L_s$ releases the lock. $L_n$ will acquire the locks after $L_s$ commits the transaction, and it will find out that $L_s$ has already became the leader. As a result $L_n$ will not declare itself the leader.

In a case where $L_s$ halts after acquiring the exclusive lock, the database will timeout $L_s$'s transaction and release the lock. The database will rollback the transaction and $L_s$ will have to re-acquire the exclusive lock in order to become a leader. $L_s$ has to reset its local role variable to $non\_leader$.

**Case II:** When a process becomes the leader it acquires a lease that is valid for $(T_p * Max_{mhb} - \mu)$. The process voluntarily gives up the leader role if it fails to renew the lease before it expires. In order to ensure that a process $L_n$ cannot become the leader while a slow leader $L_s$ still has a valid lease, the protocol needs to ensure that the time needed by $L_n$ to suspect $L_s$ is higher than the time duration of $L_s$'s lease.

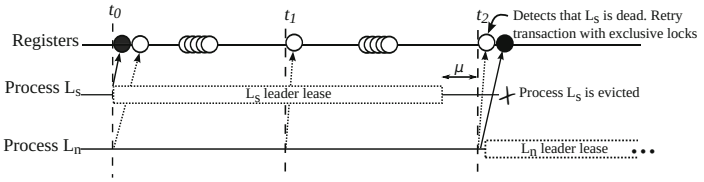



**Fig. 1.** Black and white circles represent exclusive and shared locks, respectively. Process $L_s$ is a slow leader that does not update the counter after $t_0$. Process $L_n$ becomes leader after the lease for $L_s$ expires.

Assume the processing and network latencies of the process $L_n$ are *zero*. Furthermore, the process $L_n$ performs a heartbeat (read and update the registers) soon after $L_s$ commits an update. The process $L_n$ will find out that $L_s$ is alive. After that, it will have to read the registers $Max_{mhb}$ times before it can suspect $L_s$. $Max_{mhb}$ heartbeat rounds will take $(T_p * Max_{mhb})$ seconds, assuming that $L_n$'s clock drift is negligible. Thus, the minimum time that $L_n$ needs to suspect $L_s$ and elect itself as leader is $(T_p * Max_{mhb})$, which is strictly more than the lease time of $L_s$. The assumption that $L_n$ does not have any processing and network latencies represents a worst case scenario. In a real system the latencies will always have some positive value which will increase the time needed by $L_n$ to declare $L_s$ as dead. The constant, $\mu$, should be configured to be higher than the upper bound on clock drift for any process in the system. In practical systems, NTP, GPS, atomic clocks are used to ensure low bounds on clock drift.

An illustration of this worst case scenario is presented in Figure 1 where $Max_{mhb} = 2$. The leader $L_s$ is faulty and it does not update the counter after $t_0$. At time $t_2$ the process $L_n$ detects that $L_s$ is faulty and it can become the new leader. As $L_n$ does not hold the exclusive lock, it immediately retries the transaction, acquires the exclusive lock and becomes the leader. The lease of $L_s$ expires after $T_p * 2 - \mu$, which is less than the time $L_n$ must wait to detect the failure of the process $L_s$.

**Theorem 2.** *A correct process eventually becomes the leader.*

*Proof.* Assume a system configuration of $p_1, p_2, p_3 ... p_k ... p_n$ processes. Additionally, assume $p_k$ is the only correct process that repeatedly manages to update its counter every $Max_{mhb}$ rounds. All the other processes are incorrect such that these processes do not always manage to update the counter within $Max_{mhb}$ rounds. A correct process is never suspected by any process in the system. We show that the process $p_k$ eventually becomes a leader and retains the leader role in the subsequent rounds.

Assume all the processes have just started and the history of each process is empty. The process $p_1$ will declare itself to be the leader and it will retain the role for $T_p * Max_{mhb} - \mu$ seconds. During the first $Max_{mhb}$ rounds no process will be evicted. If $p_1$ is an incorrect process which fails to update the counter, its lease for the leadership will expire. In the round $(Max_{mhb} + 1)$ a process

with least id that managed to update the counter, while $p_1$ was the leader, will become the new leader. The new leader will evict $p_1$ along with other suspected processes, if any. An evicted process might rejoin the system, but it is will be assigned with a new id by the monotonically increasing function. The eviction of incorrect processes will continue until $p_k$ becomes the process with the least id in the system. The process $p_k$ will elect itself as the new leader. As the process $p_k$ is correct it will not miss any heartbeats and it will retain the leader role in subsequent rounds.

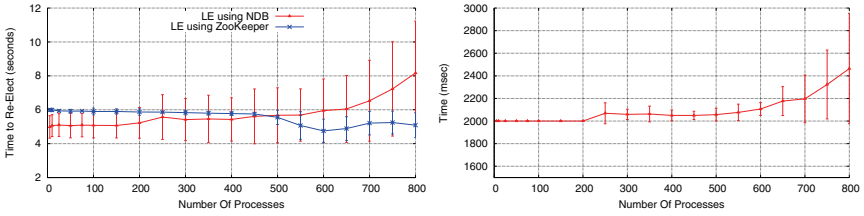
## 6 Evaluation

We have implemented the leader election using in-memory, highly-available, distributed database called NDB (Version 7.4.3), the storage engine for MySQL Cluster [5]. NDB is a real-time, ACID-compliant, relational database with no single point of failure and support for row-level locking. We use the native Java API for NDB, ClusterJ, as it provides lower latency and higher throughput than the SQL API that uses the MySQL Server.

All the experiments were performed on nodes behind a single 1 Gbit switch, where the network round trip time between any two nodes is in single digit millisecond range. The NDB setup consisted of six data nodes (6-core AMD Opteron 2.6 GHz, 32GB RAM) with replication factor of 2. We compare our solution with a leader election solution implemented using ZooKeeper. The ZooKeeper setup consisted of three quorum nodes (6-core AMD Opteron, 32GB RAM). We used the leader election library for ZooKeeper (Version 3.4.6) from the Apache Curator project (Version 2.7.1). Each ZooKeeper client creates a sequential ephemeral node in predetermined directory. Each node registers a *watch* (callback request) for its predecessor. Upon a node failure its successor is notified. The successor checks if there are any nodes with smaller sequential number. If there are no smaller nodes available then it elects itself as the new leader; otherwise, it registers a new watch for its new predecessor.

In the experiments the initial heartbeat round time was set to 2 seconds and $Max_{mhb}$ was set to 2. To accurately determine the failover time all the clients were run on a single machine (12-core Intel Xeon 2.8 GHz, 40 GB RAM). All experiments were performed fifteen times and the graphs show the average results, with the error bars showing the standard deviation of the results. In each experiment $N$ processes are started. When all processes have joined the system, the round time is continuously monitored for changes. If it does not change for a certain time (three minutes) then the system is considered stable. After the system has stabilized the leader process is repeatedly killed 50 times to measure failover time.

Figure 2a shows the relation between network size and the time to elect a new leader. Up to 200 processes the service consistently elects a new leader in around five seconds. However, when the network sizes increases beyond 200 nodes the time to elect new leader also increases. This can also be observed in figure 2b which shows the relationship between round time and the network




(a) Average time to elect a new leader. (b) Heartbeat round time after GST.

**Fig. 2.** Performance of leader election service with the default configuration settings for NDB (MySQL Cluster). Figure 2a shows the average time to elect a new leader when the current leader process fails. Figure 2b shows the increase in the heartbeat round time when the leader detects contention on the registers.

size. For network sizes up to 200 processes, all the processes manage to update the counter before they are suspected by the leader process. However, when the network size increases beyond 200, contention on the registers prevents some processes from writing to the shared register for consecutive heartbeats. The leader processes detects contention on the registers when an evicted process raises the evict flag. The leader process increases the heartbeat delay to release the contention on the registers, which has the side-effect of also increasing the leader failover time. In the experiments, the heartbeat delay increment ($\Delta$) was set to 50 milliseconds.

In the implementation of leader election using ZooKeeper, the time to elect a new leader is determined by two configuration parameters: *tick time* and *session timeout*. We set these values as low as possible to quickly elect a new leader in case of a leader failure. The lowest allowable values for *tick time* is 2 seconds, and *session timeout* is 4 seconds. In order to accurately determine the fail over time all leader election processes were run on one (12-core Intel Xeon 2.8 GHz, 40 GB RAM) machine. Up to 400 processes ZooKeeper constantly elects a new leader in six seconds. However the time to elect new leader starts to drop if we increase the number of clients on the same machine. This is because of the contention on the CPU and main memory because of which the processes slowed down. When a leader is killed it may have already skipped a heartbeat. This results in quicker reelection of a new leader. Due to memory limitations we could not add more than 800 processes in the experiment.

## 7 Related Work

Leader election is a well studied problem. All the related research can be classified into two broad categories: shared memory and message passing based leader election protocols.

Guerraoui et al. presented the first failure detector, $\Omega$, that was implemented using shared memory for an eventually synchronous system [8]. The protocol is write optimal, only the leader process writes to the shared memory, and all other non-leader processes only read shared memory. Fernandez et al. further investigated the problem in systems where all processes are not eventually synchronous [9, 10]. In [9], they propose solutions for systems which require only one process to eventually behave synchronously. All other process can behave fully asynchronously provided that their timers are well behaved. In [10], two t-resilient protocols are presented that require a single, eventually synchronous, process and $t - f$ processes with well behaved timers, where $t$ is the maximum number of processes that may fail, and $f$ is maximum number of processes that can fail in a single run. For synchronous systems, a leader election algorithm using shared memory is presented in [22], where a semaphore is used to prevent multiple writers from concurrently updating the counter.

The first leader election protocols using message passing are timer-based. Processes send messages to each other to indicate they are alive. A process is suspected if it fails to send a heartbeat message within a time bound. If a heartbeat is received from a suspected process the timer is increased to accommodate for slow processes. Eventually time bounds for processing and communication latencies are determined for the given system by successively increasing the timer upon receiving a message from a suspected process. Some notable leader election protocols in the message passing paradigm using timers are [23–25].

Mostefaoui et al. presented a time-free implementation of failure detectors [26]. It allows the communication and processing times to always increase. The protocol assumes that the query-response messages obey a certain pattern. The protocol requires a correct process $p$ and $f + 1$ processes from a set $Q$ such that if processes repeatedly wait to receive messages from $n - f$ processes, then eventually the messages from $p$ are always among the first $n-f$ messages received by each process in $Q$. Here $n$ is the system size and $f$ is the maximum number of processes that can fail. The protocol works for any value of $f$ (i.e., $1 \leq f < n$).

## 8 Conclusions

We have shown that a reliable leader election service can be implemented using two phase commit transactions in the NDB storage engine, a NewSQL database. Our solution ensures that there is never more than one leader, and the time taken for leader election is comparable to ZooKeeper for clusters of up to 800 processes. Our algorithm enables distributed systems that already use NewSQL databases to save the operational overhead of deploying a third-party service, such as ZooKeeper, for leader election, as our algorithm can easily be re-implemented for other NewSQL databases.

**Acknowledgement.** This work funded by the EU FP7 project "Scalable, Secure Storage and Analysis of Biobank Data" under Grant Agreement no. 317871, and by Swedish e-Science Research Center (SeRC) as a part of the e-Science for Cancer Prevention and Control (eCPC) project.

## References


1. Lamport, L.: Paxos made simple. ACM Sigact News 32(4), 18–25 (2001)
2. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM 43(4), 685–722 (1996)
3. Junqueira, F.P., Reed, B.C.: The life and times of a zookeeper. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, p. 4. ACM (2009)
4. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Mass Storage Systems and Technologies, pp. 1–10 (May 2010)
5. Ronström, M., Oreland, J.: Recovery Principles of MySQL Cluster 5.1. In: Proc. of VLDB 2005, pp. 1108–1115. VLDB Endowment (2005)
6. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI 2006, pp. 335–350. USENIX Association, Berkeley (2006)
7. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC 2010, p. 11 (2010)
8. Guerraoui, R., Raynal, M.: A Leader Election Protocol for Eventually Synchronous Shared Memory Systems, pp. 75–80. IEEE Computer Society, Alamitos (2006)
9. Fernandez, A., Jimenez, E., Raynal, M.: Electing an eventual leader in an asynchronous shared memory system. In: Dependable Systems and Networks, DSN 2007, pp. 399–408 (June 2007)
10. Fernandez, A., Jimenez, E., Raynal, M., Tredan, G.: A timing assumption and a t-resilient protocol for implementing an eventual leader service in asynchronous shared memory systems. In: ISORC 2007, pp. 71–78 (May 2007)
11. Sqlserver @ONLINE (January 2015), http://www.microsoft.com/en-us/server-cloud/products/sql-server/
12. Mysql :: The world's most popular open source database @ONLINE (January 2015), http://www.mysql.com/
13. White paper: Xa and oracle controlled distributed transactions @ONLINE (June 2010), http://www.oracle.com/technetwork/products/clustering/overview/distributed-transactions-and-xa-163941.pdf
14. Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 35–40 (2010)
15. Riak, basho technologies @ONLINE (January 2015), http://basho.com/riak/
16. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era (it's time for a complete rewrite). In: Proceedings of the 33rd International VLDB Conference, pp. 1150–1160 (2007)
17. Özcan, F., Tatbul, N., Abadi, D.J., Kornacker, M., Mohan, C., Ramasamy, K., Wiener, J.: Are we experiencing a big data bubble? In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 1407–1408 (2014)
18. Multi-model database – foundationdb @ONLINE (January 2015), https://foundationdb.com/
19. In-memory database, newsql and real-time analytics – voltdb @ONLINE (January 2015), http://voltdb.com/
20. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32, 374–382 (1985)

21. Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., Abadi, D.J.: Calvin: Fast Distributed Transactions for Partitioned Database Systems. In: Proc. of SIGMOD 2012, pp. 1–12. ACM (2012)
22. Sanz-Marco, V., Zolda, M., Kirner, R.: Efficient leader election for synchronous shared-memory systems. In: Proc. Int'l Workshop on Performance, Power and Predictability of Many-Core Embedded Systems (3PMCES 2014). Electronic Chips and Systems Design Initiative (ECSI) (March 2014)
23. Aguilera, M., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega in systems with weak reliability and synchrony assumptions. Distributed Computing 21(4), 285–314 (2008)
24. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Communication-efficient leader election and consensus with limited link synchrony. In: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, pp. 328–337. ACM, New York (2004)
25. Larrea, M., Fernandez, A., Arevalo, S., Carlos, J., Carlos, J.: Optimal implementation of the weakest failure detector for solving consensus, pp. 52–59. IEEE Computer Society Press
26. Mostefaoui, A., Mourgaya, E., Raynal, M.: Asynchronous implementation of failure detectors. In: Proceedings of the 2003 International Conference on Dependable Systems and Networks, pp. 351–360 (June 2003)