1 Introduction

Leader Election is one of the fundamental problems in distributed systems. A leader is a correct process that can be used to coordinate the work of a set of processes. An algorithm has to implement two properties to solve the Leader Election problem:

1. **Safety**: At any point in time there is at most one leader.
2. **Liveness**: Eventually there will be one leader forever.

We showed in our previous work how to solve a weaker form of Leader Election in an asynchronous system with local clocks [1]. But for electing a leader the system needs to stabilize: a leader is only guaranteed to be elected during periods in which the system behaves synchronously.

In this work we show that the stabilization property is not necessary for the Leader Election problem. We do this by examine the ability to solve Leader Election in the FAR model [2]. The FAR model makes four assumptions:

1. At most a minority of processes is allowed to crash. Crashed processes do not recover. We call all other processes correct processes.
2. Each process can schedule the execution of a sleep operation. The sleep defers the execution of all following operations for at least $G > 0$. $G$ is not known.
3. Each channel $SC_{p \rightarrow q}$ implements a rudimentary congestion control [3]: it only guarantees the delivery of message $m$ to correct process $q$ if $p$ defers all future messages (to $q$) until it receives an answer/acknowledgement for $m$ from $q$. We call this property stubborn. This stubborn channels prevent a "fast" process from flooding a "slow" process with messages.
4. The average response time for all – possibly infinitely many – messages send via $SC_{p \rightarrow q}$ convergence against a finite value. Note that convergence against a finite average does not establish an upper bound because there may always be a response time that is larger than any previous response time.

Our contribution is twofold: (1) we show that Leader Election is not solvable in the pure FAR model and (2) that it becomes solvable with local clocks with a bounded drift rate. Especially interesting is the implementation of the liveness property: we reuse an internal timeout originally introduced to provide liveness to a failure detector implementation.

2 Finite Average Response Time Model

The Finite Average Response Time Model (FAR) is built around the notion of response time. Each pair of processes $p$ and $q$ is connected by a pair of uni-directional channels $SC_{p \rightarrow q}$ and $SC_{q \rightarrow p}$. The response time for a message $m$ sent on $SC_{p \rightarrow q}$ is the time between $p$ sending $m$ and $p$ receiving an answer/acknowledgement for $m$ from $q$. The FAR model makes four assumptions:

1. At most a minority of processes is allowed to crash. Crashed processes do not recover. We call all other processes correct processes.
2. Each process $p$ can schedule the execution of a sleep operation. The sleep defers the execution of all following operation for at least $G > 0$. $G$ is not known.
3. Each channel $SC_{p \rightarrow q}$ implements a rudimentary congestion control [3]: it only guarantees the delivery of message $m$ to correct process $q$ if $p$ defers all future messages (to $q$) until it receives an answer/acknowledgement from $q$ for $m$. We call this property stubborn. This stubborn channels prevent a "fast" process from flooding a "slow" process with messages.
4. The average response time for all – possibly infinitely many – messages send via $SC_{p \rightarrow q}$ convergence against a finite value. Note that convergence against a finite average does not establish an upper bound because there may always be a response time that is larger than any previous response time.

We showed in [2] how to solve the Consensus problem (i.e., all correct process have to decide on a common value) in the FAR model. The details of the Consensus implementation are of interest because our Leader Election algorithm is built upon it. In the remainder of this section we
present the failure detector \( EA\text{-}FD \) which we used to implement Consensus in the FAR model and discuss why Leader Election is not solvable in the FAR model without further assumptions.

**Failure Detector \( EA\text{-}FD \)**  
Our Consensus implementation uses the fact, that Consensus can be solved with stubborn channels and a failure detector of class \( oS \) [3]. Because we already assume stubborn channels, we can safely restrict our focus on implementing a failure detector \( EA\text{-}FD \) sufficient to solve Consensus. In each step, each process may query its local \( EA\text{-}FD \) module for a set of processes suspected to have crashed. \( EA\text{-}FD \) provides two properties: (1) eventually, all correct processes will always suspect all crashed processes and (2) eventually, no correct process will suspect any correct process.

The implementation of \( EA\text{-}FD \) builds on an internal timeout. Therefore, a *weak clock* is implemented using the *sleep* operation: each time the clock is queried a *sleep* is scheduled, an internal counter is increased and its value is returned. A timeout \( t( SC_{p-q} ) \) for \( SC_{p-q} \) is maintained by \( p \). Whenever \( q \) does not respond to a message from \( p \) within \( t( SC_{p-q} ) \), \( q \) is added to the list of processes suspected by \( p \). There are two cases when \( p \) receives an acknowledgement \( m_a \) from \( q \) for message \( m \): \( m_a \) is a *fast message* if it is received before \( t( SC_{p-q} ) \) for \( m \) has expired – otherwise it is a *slow message*. Whenever a fast message is received by \( p \), the current timeout is *increased*. Whenever a slow message is received, \( p \) is taken from the list of suspected processes list and the timeout is potentially *decreased*. Please see [2] for more details and the correctness proof.

**Leader Election**  
Leader Election is impossible to solve in the FAR without any additional assumptions. Intuitively, this can be proven by contradiction. Let us assume an algorithm \( A \) exists that solves Leader Election in the FAR model. Now we take an arbitrary run \( R \) in which process \( p \) becomes leader at time \( t \). We derive two other runs from \( R \). In the first run, \( p \) crashes at time \( t \). Obviously, it will not become leader instead some other process \( q \) will become leader at time \( t' > t \). In the second run, \( p \) is "suspended" between \( t \) and \( t' \). Because \( p \) has no notion of this suspension, it will execute a step as leader at \( t' \). Process \( q \) will also take a step at \( t' \) as leader because both runs look the same to \( A \). This violates the requirements of Leader Election.

### 3 Timed Finite Average Response Time Model

The FAR model itself is not strong enough to solve Leader Election. That is why we substitute the *sleep* operation through the assumption that each process \( p \) has access to a local clock \( C_p \) with a bounded drift rate. We call this new model *Timed Finite Average Response Time Model*. Because the *sleep* operation can be implemented on top of these clocks, we have still access to \( EA\text{-}FD \).

We implement the *safety* property of Leader Election with leases. Each process \( p \) that wants to become leader asks all other processes to support it for a certain lease time \( t_l(p) \). Each process \( q \) supports at most one process \( p \) and only if the \( p \) has the smallest process id, that is not in \( q \)'s suspicion list. After a process \( p \) has received support messages from the majority of all processes it becomes leader until the end of the lease time. With the help of local clocks and the knowledge of their upper bound on their drift rate the safety property is enforced.

The liveness property is ensured with the help of \( EA\text{-}FD \). First, whenever a process \( p \) suspects all processes with a smaller process id and does not support any process (excluding itself), it asks for support to become leader itself. This ensures that eventually the correct process with the smallest process id tries to become leader. Second, \( t_l(p) \) is set to \( \max \{ t_l( SC_{p-q} ) | q \} \). This ensures that \( t_l(p) \) will be eventually large enough such that \( p \) will stay leader forever (if it has the smallest process id amongst the correct processes).

### 4 Conclusion

We have shown how to implement Leader Election in a system model with no bounds on upper communication and computation delays and that potentially never stabilizes. Our previously presented FAR model which is strong enough to solve Consensus is too weak to solve Leader Election. However, with the additional introduction of local clocks Leader Election becomes solvable. The implementation reuses an internal timeout used to implement a failure detector in the original FAR model.

### References

