2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

Quorum Selection for Byzantine Fault Tolerance

Leander Jehl
Department of Electrical Engineering and Computer Science
University of Stavanger
Norway
leander.jehl@uis.no

***Abstract*—Systems tolerating arbitrary failures use significant resources to constantly mask omission and timing failures from faulty processes.**

This paper presents Quorum Selection, a mechanism that allows to select well functioning processes for active participation in a system. Different from previous work, Quorum Selection not only excludes provably faulty processes that deviated from the protocol, but also takes omission and timing failures into account, even if they only affect individual links.

We present a system architecture for Quorum Selection, including a novel failure detector that uses expectations to detect omission and timing failures in a Byzantine environment.

We investigate how often an adversary may cause the quorum to changes. We show a quadratic lower bound for general Quorum Selection, but we also define a special case of Quorum Selection for leader based systems that requires trying at most $6f$ quorums.

I. Introduction

The continued success of blockchain technology has significantly increased both industrial and academic interest in systems tolerating arbitrary failures. However, these systems still incur a large performance overhead, compared to systems tolerating only benign failures. This high cost comes not only from the need to prevent that attackers corrupt the system state, but also from the constant masking of omission and timing failures from faulty processes.

This work introduces Quorum Selection, a technique that allows to avoid the masking of failures. Quorum Selection allows to select an active quorum of well functioning processes to run a byzantine fault tolerant (BFT) system. Omissions from or failures of processes outside of the active quorum have no effect on the system and therefore do not need to be masked.

Many BFT systems can profit from Quorum Selection. Systems like PBFT [1], Tendermint [2] or BFT-SMART [3] use $n = 3f+1$ replicas, broadcast messages to all replicas but require replies from only $n-f$ correct replicas. Similarly, BFT systems that use trusted components or similar assumptions to reduce the total number of replicas to $n = 2f + 1$, require replies from only $n - f$ replicas [4], [5].

If a quorum or subset of processes, containing $n - f$ correct processes can be selected, these systems can drop approximately $1/3$ or $1/2$ of the inter-replica messages. Distler et al. [6] have shown that this may results in respective performance improvements. Different from [6] Quorum Selection allows to maintain this benefit even in the presence of failures.

To allow Quorum Selection we present a modularized system architecture including a novel approach to failure detection for BFT systems. We define a failure detector that can detect omission and delay of individual messages and give an example how this failure detector can be integrated into a BFT algorithm. In Quorum Selection we take not only current suspicions into account, but also suspicions previously raised and canceled. Thus, processes that repeatedly delay messages will eventually be omitted from the quorum. Our solution for Quorum Selection is decentralized and does not require processes to agree on suspicions. Instead we use an eventually consistent data structure to record and distribute suspicions.

Two existing systems already apply Quorum Selection to reduce the cost of Byzantine fault tolerance. In BChain [7] processes in an active quorum communicate along a chain, thus drastically reducing the number of necessary intra-replica messages. XPaxos [8] uses Quorum Selection to implement state machine replication in the XFT model, requiring only $2f + 1$ processes to tolerate up to f arbitrary failures without relying on trusted hardware.

However the mechanisms used for Quorum Selection in BChain and XPaxos are unsatisfactory. Quorum Selection in BChain relies on replacing potentially faulty processes with new, external processes that are assumed to be correct. XPaxos on the other hand enumerates all possible quorums and tries them one after the other. Thus, even without false suspicions, an attacker may cause the quorum to change repeatedly over a long period, i.e. exponentially in the number of processes. In contrast, in absence of false suspicions, our solution for Quorum Selection ensures that faulty processes may cause at most $\mathcal{O}(n^2)$ many quorum changes. We show that this is optimal, but that it can be improved for leader centric systems with at least $3f + 1$ processes.

To summarize, this paper makes the following contributions:

- We introduce and specify the problem of Quorum Selection.
- We present a system architecture that can be used for Quorum Selection.
- We present an asymptotically optimal solution for Quorum Selection.
- We show that a variant of Quorum Selection called Follower Selection can be done in $\mathcal{O}(f)$ time, in systems with $3f + 1$ or more processes.

2575-8411/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDCS.2019.00213

II. Failure classification

In the following we give a short classification of failures and how they may be detected. This classification shows some fundamental limitations to our approach, since some failures cannot be detected. It also shows some important design choices we made, favoring flexible design over strict failure detection.

We assume that during an infinite execution, every process is expected to send infinitely many messages. This is the case in systems that use heartbeats to detect crash failures. We distinguish commission, omission and timing failures as listed below. We discuss below to what extent these failures can be detected.

- **Commission failures;** a faulty process committing a commission failure sends a correctly authenticated message that should not be sent according to the algorithm specification. This includes the creation of messages, but also a change in message parameters.
- **Omission failures;** a faulty process commits an omission failure if it does not send a message it should have sent.
- **Repeated omission failures;** a faulty process commits a repeated omission failure if it omits infinitely many messages.
- **Timing failures;** a faulty process commits a timing failure if it delays the sending or processing of a message.
- **Increasing timing failures;** a faulty process j commits increasing timing failures if there exists no upper bound Δ, such that j processes and responds to any received message within Δ time units.

Not all of the above failures can be detected. Further, if processes can suspect each other, and later cancel these suspicions there are different levels of failure detection. We say that process i *permanently detects* the failure of process j, if i raises a suspicion against j and never cancels that suspicion. We say that a process i *eventually detects* the failure of process j, if i raises and cancels infinitely many suspicions against j during an execution. Eventual detection is similar to the detection of "bad" processes in a failure recovery model, where "bad" processes may fail and recover infinitely often during an execution [9].

Commission failures may be detected permanently. Typically this is the case if one or several messages are sent, that could not be sent in the same execution, e.g., equivocation or wrongly formed messages. However, many commission failures cannot be detected. For example an algorithm may require process i to broadcast an alarm if it does not receive a heartbeat message from process j within Δ time units. If i does broadcast the alarm despite receiving a heartbeat, this failure may not be possible to detect.

Similarly, if i omits the alarm, despite absence of the heartbeat, this omission may never be detected. In systems like [10] omissions are permanently detected to prevent selfish processes from omission failures done to save bandwidth. However, permanent detection of omission failures requires that all expected messages are actually sent. This significantly reduces the flexibility in algorithm design, since processes need to be aware of all messages that are expected from them. It also may require correct processes to send unnecessary messages. For example, after a short downtime, e.g., due to migration or software updates, a correct process would have to resend all messages it was expected to send during downtime.

We therefore believe omission failures should not be permanently detected, as long as these failures only endanger liveness. Instead we ensure that repeated omission failures are eventually detected. This allows to exclude processes from the quorum that systematically omit messages, while it allows correct processes to drop some messages, e.g., to catch up with the running system.

Timing failures are impossible to detect in an asynchronous system. However, if we assume an eventually synchronous system, increasing timing failures can be eventually detected.

III. Related work

To the best of our knowledge, the problem of Quorum Selection has not been studied by other works than XPaxos [8] and BChain [7], mentioned above.

However there exists a significant body of work on failure detection. Failure detection is a fundamental building block for many systems targeted to survive crash failures. See [11] for a survey on failure detection.

In the presence of arbitrary failures, failure detection cannot be implemented independently of the application [12]. Thus failure detection is less prominent in BFT systems.

Several systems detect commission failures, where processes detectably deviate from the protocol and exclude detected processes, when a proof of misbehavior exists [8], [10]. However this leaves processes with the ability to perform omission and timing failures, that may affect liveness or performance of the system.

Malkhi and Reiter [13] rely on a reliable broadcast primitive to mask all individual omissions. Thus every message is either delivered by all or none of the correct processes. Their failure detector detects *quiet* processes from which no messages are delivered.

Similarly [14] detects mute processes, that omit all messages. In [15] this muteness detector is extended with a certification module to detect commission faults. These works are similar to our approach since they detect omissions on individual links. However [14], [15] assume a leader-driven protocol and only monitor and detect failure of the current leader.

PeerReview [16] introduces a failure detector that aims to detect commission failures but also omission of individual messages. However, if omitted messages can be forwarded via a third party or arrive late, suspicions are canceled. In PeerReview, processes may raise and cancel a suspicions towards a specific process infinitely often during an execution, without taking further action.

IV. System model

We assume an asynchronous system equipped with a failure detector. That is a set $\Pi = \{p_1, p_2, ..., p_n\}$ of n processes,

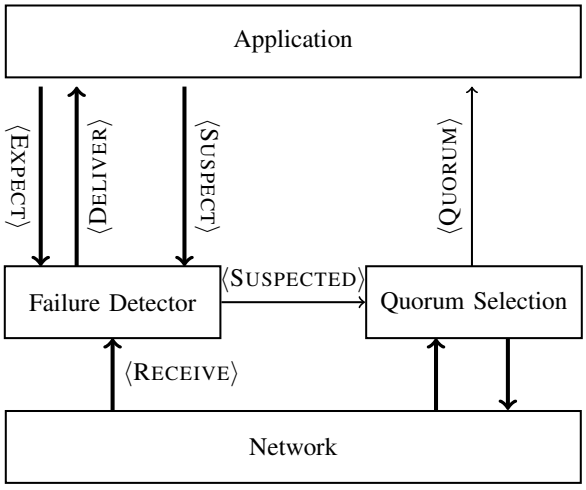


Fig. 1: Module composition in our system architecture

connected by reliable, asynchronous channels. We assume processes can be ordered by unique identifiers. We use i, j to refer to p_i and p_j.

Processes are subject to arbitrary failures. We assume however that the majority of processes is correct and we assume that cryptographic primitives cannot be broken.

We say that a *communication round* is the time it takes for messages between all correct processes to be delivered.

Figure 1 shows the different modules of our system. The system is composed of a failure detector, a module for quorum selection and an application, e.g., an algorithm for state machine replication or consensus, running on the selected quorum. Section IV-A presents the quorum selection module. The failure detector component is explained in Section IV-B. We assume that the events between different modules at one process are processed in the order they were produced.

A. Quorum Selection

We can now specify our quorum selection module. Quorum Selection at process i outputs quorums $\langle \text{QUORUM}, \mathcal{Q} \rangle_i$ with $\mathcal{Q} \subset \Pi$ and $|\mathcal{Q}| = n - f$.

We require the following properties from the quorum selection module:

1) **Termination** A correct process changes the quorum only finitely often.
2) **No suspicion** For every correct process j, eventually j is never part of the quorum, or eventually j never suspects any process in the quorum.
3) **Agreement** Eventually, correct processes always output the same quorum.

Termination and Agreement ensure that the correct processes actually agree on a single quorum. No suspicion ensures that this quorum is functioning.

Our properties do not require processes to agree on a quorum containing only correct processes. The reason is that a byzantine process may behave like a correct process, or only deviate from the protocol in undetectable ways.

The no suspicion property ensures that if only one process within the quorum suspect another process within the quorum, a new quorum must be issued. Note that, since we assume $n - f > f$ the quorum always contains at least one correct process.

Quorum Selection and Leader Election: The problem of Quorum Selection is similar to Leader Election. Given a solution for Quorum Selection it is trivial to elect a leader, e.g., electing the process with lowest identifier in the quorum.

The main difference between Quorum Selection and Leader Election is that the no suspicion property requires Quorum Selection to react to even a single suspicion. On the other hand, in many Leader Election algorithms, a leader is changed only if it is suspected by $f + 1$ processes, including at least one correct process [17]. Under favorable system conditions, Leader Election thus only changes a leader, if it actually did misbehave.

In Quorum Selection faulty processes can force a change in the quorum without "help" from correct processes. However, a faulty process can cause such interruptions only, when itself is part of the quorum. Once the correct processes have agreed on a quorum without faulty processes, no further interruptions can be caused by these faulty processes. For this reason, our goal for Quorum Selection is to minimize the number of quorum changes caused by faulty processes under favorable system conditions.

B. Failure Detection

As argued by Doudou et al. [12], Failure Detection of byzantine failures cannot be done independently of the application. Our Failure Detection module therefore relies on the application to determine which messages to expect. Additionally, we rely on the application to detect commission failures and report these to the failure detector. That is because this detection is highly application specific. Detection of some commission failures, e.g. equivocation may also require to inspect multiple messages.

Within the application, detection of commission failures could be done by a certification module, as in [15].

The failure detector module at a process j provides the following input and output events.

- $\langle \text{RECEIVE}, m, i \rangle$ is used by the network layer to receive a message m from process i.
- $\langle \text{DELIVER}, m, i \rangle$ is an output of the failure detector, used to deliver a correctly authenticated message from process i to the application or the quorum selection module.
- $\langle \text{EXPECT}, P, i \rangle$ is used by the application to inform the failure detector that it expects a message complying to predicate P from process i.
- $\langle \text{SUSPECTED}, \mathcal{S} \rangle$ publishes the set $\mathcal{S} \subset \Pi$ of currently suspected processes.
- $\langle \text{DETECTED}, i \rangle$ is used by the application to inform the failure detector that evidence, showing that process i is faulty has been found.
- $\langle \text{CANCEL} \rangle$ is used to cancel previously issued expectations.

1) Failure detector properties: Failure detectors typically fulfill a completeness property, stating that faults are actually detected, and an accuracy property, stating that (eventually) correct processes do not suspect each other to be faulty [18].

We say that the failure detector suspects process i if it invokes $\langle \text{SUSPECTED}, \mathcal{S} \rangle$ with $i \in \mathcal{S}$.

a) Completeness: Our completeness property ensures that processes, that do not send expected messages are suspected, and that failures detected by the application are processed.

- **Expectation completeness** If process j expects a message from process i and does not cancel that expectations then, the failure detector at j either delivers a message matching the expectation, or eventually suspects i.
- **Detection completeness** If failure of i is detected by the application at j, i is eventually forever suspected by the failure detector at j.

We require that expectation completeness and suspected completeness hold for the failure detector at a correct process.

Note that expectation completeness only requires processes to be suspected once. Thus it can only ensure eventual detection. Detection completeness on the other hand ensures permanent detection of commission failures.

b) Accuracy: We require that our failure detector fulfills eventual strong accuracy.

- **Eventual strong accuracy** Eventually no correct process ever suspects another correct process.

However, we note that eventual strong accuracy cannot be ensured by the failure detector alone, but also requires a correct implementation of the application. Especially, eventual strong accuracy can only hold, if correct processes eventually send all the messages that are expected by other correct processes. Clearly the following requirement is sufficient to implement the failure detector in an eventually synchronous system.

- **Accuracy requirements** The application at a correct process never detects a different correct process to be faulty and any message expected between correct processes is delivered within two communication rounds.

In Section V we show how our Failure Detection module can be utilized in an BFT algorithm and how the accuracy requirements can be achieved. The implementation of our failure detector in a eventually synchronous system is straightforward.

V. Failure Detection in XPaxos

As mentioned in the introduction, XPaxos [8] is a BFT algorithm that allows to tolerate up to half of the processes to fail arbitrarily without relying on trusted hardware components. This is achieved through the XFT model that combines characteristics of eventually synchronous and synchronous systems.

XPaxos uses only an active quorum to run normal operation. Upon failure, this quorum has to be changed. However the existing XPaxos algorithm only detects failures at the granularity of a quorum. Thus XPaxos can detect that the current quorum contains a faulty process, but does not identify the culprit.

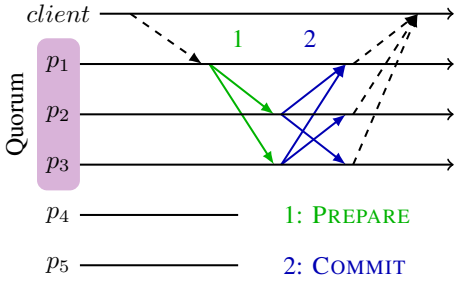


Fig. 2: XPaxos normal case message flow with $f = 2$.

Thus XPaxos solves Quorum Selection by iterating through all possible quorums.

In this section, we explain how the XPaxos algorithm can be adjusted to use our failure detector. This mainly requires to extend the XPaxos protocol with expectations issued to the failure detector and to ensure these expectations are met by correct processes.

A. Failure Detection in XPaxos normal operation

We now explain how our Failure Detection module can be integrated into the normal case operation of XPaxos. Figure 2 shows the normal case message pattern used in XPaxos.

1) The process in the active quorum with lowest id, i.e. the leader, sends a PREPARE message, including a client request to all processes in the quorum.
2) Processes in the quorum send a COMMIT message, including a hash of the client request from the prepare message to all other processes in the quorum.
3) On receiving a COMMIT message from every other process in the quorum, with matching hashes, processes commit and execute a client request.

To allow Failure Detection in XPaxos, processes have to issue expectations. Adding expectations is straightforward. When receiving or sending a PREPARE message, a process issues an expectation for a COMMIT message for every process in the quorum. However there are some subtleties that must be taken into account:

- First, a COMMIT message from process k may arrive before the PREPARE from leader l. In this case, no expectation should be issued for process k.
- Second, COMMIT messages may be malformed. We therefore require that a COMMIT includes the PREPARE message from the leader[1].
  Upon receiving a malformed COMMIT message, i.e. correctly authenticated by the sender, but not including a valid PREPARE, a process issues a $\langle$DETECTED$\rangle$ for the sender. Upon receiving a COMMIT including a valid PREPARE that differs from the one a process received from the leader, it issues a $\langle$DETECTED$\rangle$ event for the leader. Thus, equivocation is detected.

[1]In XPaxos, also the PREPARE message only contains the hash of the client request and the actual request is only appended to the message. Thus this change does not significantly change the payload.

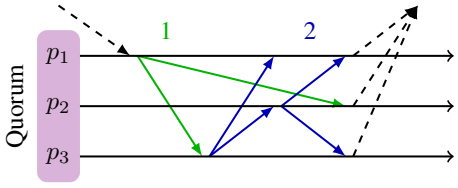


Fig. 3: XPaxos normal case with delayed PREPARE messages.

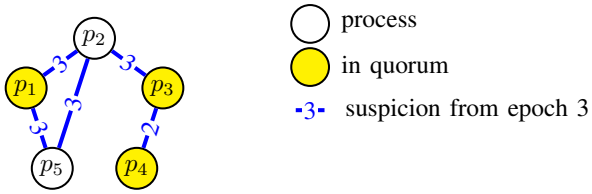


Fig. 4: Example for a graph created from suspicions. The edges are labeled with the epoch of their suspicions. In epoch 2, no independent set of size 3 can be found. If the epoch is increased to 3, the edge between p_3 and p_4 will be removed and $\{p_1, p_3, p_4\}$ and $\{p_3, p_4, p_5\}$ are independent sets.

- Third, when receiving a COMMIT message before the causally preceding PREPARE, the process sends a COMMIT and issues an expectation for the PREPARE. This scenario is shown in Figure 3.

We give a brief informal argument that the changed protocol fulfills the accuracy requirements necessary to implement our failure detector. It is clear that after sending a PREPARE, a correct leader receives a COMMIT message within 2 communication rounds. Similarly, due to our third change above, when sending a COMMIT message, the commit from other correct replicas is received within 2 communication rounds. Finally, since a correct leader will not sign two different PREPARE messages with the same view and slot number, it will not be detected for equivocation.

B. Installing the selected quorum

XPaxos starts with a default quorum. When the current quorum is suspected XPaxos moves to the next quorum in an enumeration of all $\binom{n}{f}$ possible quorums, using round robin if the list is exausted. We note that the procedure to change quorum, a view change is a complex protocoll.

We can use this view change in combination with Quorum Selection in the following way. If a process i receives $\langle \text{QUORUM}, Q \rangle$ from the Quorum Selection module, i suspects all quorums ordered before Q. Process i must also invoke $\langle \text{CANCEL} \rangle$ since during view change, processes may no longer send expected PREPARE and PROMISE messages.

VI. QUORUM SELECTION AND SUSPECT GRAPHS

In this section we give an implementation of our quorum selection module. Remember that each process receives a set of suspected processes in $\langle \text{SUSPECTED}, \mathcal{S} \rangle$ events from its failure detector. These sets may be inconsistent and the quorum selection module may have to react to a suspicion that is raised at only one process.

We therefore propagate these local suspicions as updates to an eventual consistent data-structure. This ensures that correct processes eventually agree on the same quorum.

A. Handling suspicions

Algorithm 1 Lines 9 to 23 shows how suspicions are handled. Processes maintain an epoch and associate current suspicions with this epoch. The set of processes we are currently suspecting is stored in the variable *suspecting*. Additionally, every process maintains a matrix *suspected* to keep track, which process is or was suspecting which other process. This is done by storing the last epoch in which a specific suspicion was issued.

An event $\langle \text{SUSPECTED}, \mathcal{S} \rangle_j$ issued by the failure detector at process j is handled by the function updateSuspicions(). This function updates *suspecting* with the processes currently suspected. Process j also marks that the processes have been suspected in the current epoch, updating *suspected*$[j]$. The updated vector showing j's suspicions are then signed and broadcast to all processes.

On receiving a correctly signed update from process l, process j updates the vector *suspected*$[l]$ with the maximum of the existing and new values. If the local state has been updated, the message is forwarded to all processes. This ensures that all correct processes receive the same suspicions.

Since local values in *suspected* are always set to the maximum values, correct processes produce the same results, even if updates are handled in a different order. This holds, even if faulty processes equivocate, sending different updates to different processes. Thus *suspected* can be seen as an eventual consistent shared data structure.

B. Finding a quorum

When the array *suspected* was updated, processes check whether a new quorum needs to be issued, invoking function updateQuorum() on Line 24 of Algorithm 1. To find a quorum, processes build an undirected graph whose nodes correspond to the processes in Π. The quorum is then found as an independent set of size q in that graph.

A process i with current epoch $epoch = e_i$ builds a **suspect graph** G_i by connecting nodes $l, k \in \Pi$, if one of these processes suspected the other in epoch e_i or later, i.e. *suspected*$[l][k] \geq e_i$ or *suspected*$[k][l] \geq e_i$.

If all suspicions issued in epoch e_i or later are accurate, i.e., correct processes do not suspect each other, then the correct processes do form an independent set of size $q = n - f$ in the simple graph G_i. If the failure detector is accurate an independent set can thus be found.

If no such independent set can be found, some correct process suspected another correct process in epoch e_i. In this case we move to the next epoch.

If multiple independent sets of size q are found, the first in lexicographical order is chosen. This ensures that correct

Algorithm 1 Handling suspicions at process p_j

```
1:  Constants
2:      Π = {p_1, p_2, ...p_n}                                  {processes}
3:      q                                                       {quorum size, assume f + q = |Π|}
4:  State
5:      epoch = 1
6:      suspecting = {}                                         {who is currently suspected}
7:      suspected = [][]int                                     {n × n matrix, initially all 0}
8:      Qlast = {p_1, p_2, ..., p_q}                            {initial quorum}

9:  on ⟨SUSPECTED, S⟩ from failure detector
10:     updateSuspicions(S)

11: updateSuspicions(S)
12:     suspecting ← S
13:     for p_i ∈ suspecting do
14:         suspected[j][i] ← epoch                             p_j suspects p_i in current epoch
15:     bcast ⟨UPDATE, suspected[j]⟩_σj                         {To all including self}

16: on ⟨UPDATE, susted⟩_σl from p_l                             {correctly signed by p_l ∈ Π}
17:     changed ← false
18:     for i ∈ {1, ..., n} do
19:         if susted[i] > suspected[l][i] then
20:             suspected[l][i] ← susted[i]
21:             changed ← true
22:     if changed then
23:         bcast ⟨UPDATE, susted⟩_σl                           {forward to other processes}
24:         updateQuorum(suspected)                             {issue new quorum if necessary}

25: updateQuorum()
26:     G ← buildSimpleGraph(suspected, epoch)                  {build simple graph, as described in Section VI-B}
27:     if G contains no independent set of size q then
28:         epoch ← epoch + 1                                   {increase epoch if suspicions in current epoch are inconsistent}
29:         updateSuspicions(suspecting)
30:     else
31:         find Q as first independent set of size q in G      {in lexicographic order}
32:         if Q ≠ Qlast then
33:             issue ⟨QUORUM, Q⟩
34:             Qlast ← Q
```

processes will produce the same quorum, once they have received the same updates to their *suspected* matrix.

Figure 4 shows an example of how a suspect graph is used to find a quorum.

C. Discussion

For Algorithm 1 to implement Quorum Selection, Termination, No Suspicion and Agreement need to hold. In Section VII we proof an upper bound on the number of quorums issued by correct processes, when the failure detector is accurate. This implies Termination. No Suspicion follows since suspicions are represented as edges in the graph G and a quorum is selected as an independent set in G. Finally, agreement follows, since suspicions are propagated between correct nodes as proven in Section VII.

The eventually consistent updates to the matrix *suspected* require only a loose coupling between processes and avoid costly synchronization primitives like consensus. For example, we do not prevent faulty processes from equivocation, sending different updates to different processes. Instead such behavior will only cause Quorum Selection to terminate faster. Similarly, if a process experiences a benign crash failure, this process may be concurrently suspected by all correct processes. Once these suspicions are propagated, the crashed process will not be part of any quorum.

Line 27 of Algorithm 1 requires to solve the independent set decision problem, which is known to be NP-hard. However, for small graphs, e.g. including only tenth of nodes, it is easy to compute. Quorum Selection is thus suitable for the use in consortium or permissioned blockchains.

VII. Quorum Selection performance

In this section we show that our solution is asymptotically optimal with respect to how often faulty processes can interrupt the system after the failure detector has become accurate.

A. Upper bound

In the following let i and j denote correct processes. Local variables subscripted with i or j denote the value of that variable at process j, e.g., $suspected_j$.

We first present two lemmas that show how suspicions are propagated. Theorem 3 then shows an upper bound on how many quorums may be issued by correct processes, when the failure detector is accurate.

The proof of Theorem 3 establishes that at most $f \times (f+1)$ quorums are issued in one epoch. This is only an upper bound. Our simulations suggest that Algorithm 1 actually allows at most $\binom{f+2}{2}$ quorums in one epoch.

Lemma 1. *For two processes $k, l \in \Pi$, if $suspected_i[l][k] = m$ and $epoch_i = e$ holds at any time during an execution, then after one communication round $suspected_j[l][k] \geq m$ and $epoch_j \geq e$ holds.*

Proof: If $suspected_i[l][k] = m$ for $m > 0$, then i received an update $\langle \text{UPDATE}, susted \rangle_{\sigma_l}$ from process l including $susted[k] = m$. Within one communication round, process j will receive this update forwarded from process i and thus increase $suspected_j[l][k]$. After receiving all suspicions from j, i will advance its epoch to e or higher. ■

Lemma 2. *If process i issued $\langle \text{QUORUM}, Q \rangle$ in epoch e_i then i will only issue a new quorum $\langle \text{QUORUM}, Q' \rangle$ after adding an edge to G, connecting two processes in Q.*

Lemma 2 follows since a Quorum is found as an independent set, minimal in lexicographical order.

Theorem 3. *Assume that after time t_0 no correct processes suspect each other. Let $t' > t_0$ be a time, at least one communication round after t_0.*
After time t' correct processes issue at most $\mathcal{O}(f^2)$ quorums.

It can be shown that a correct process j issues at most $f \times (f+1)$ quorums in any epoch. The key observation for this is that since $|\Pi| = f + q$, a node of degree $f+1$ cannot be part of an independent set of size q. After time t' correct processes enter at most one new epoch.

B. A lower bound for selecting a quorum

Theorem 4 shows that Algorithm 1 cannot be improved asymptotically. The proof suggests that a good strategy for faulty processes is to concentrate false suspicions and faulty behaviour, e.g. omissions, on only two correct processes.

Theorem 4. *If $f + q = n$ holds, any deterministic algorithm for quorum selection may have to propose $\binom{f+2}{2}$ quorums.*

Proof: Choosing a quorum of $q = n - f$ processes is equivalent to choosing f processes that should be excluded.

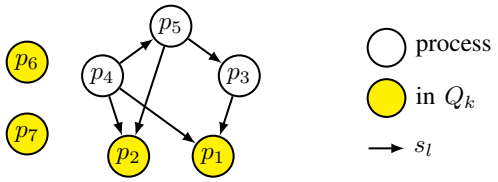


Fig. 5: Example to proof of Theorem 4: If $f = 3$ all suspicions could be cause by faulty processes $\{p_1, p_2, p_5\}$ or $\{p_3, p_4, p_5\}$.

Similarly, finding a independent set of size q is equivalent to finding a vertex cover of size $n - q$ [19].

The adversary may choose which processes are faulty, and may both cause a process to suspect one of the faulty processes, e.g. by omitting an expected message, or issuing a false suspicion at a faulty process.

An algorithm for Quorum Selection has to eventually output the same quorum on all correct processes. The no suspicion property of Quorum Selection requires that processes in this quorum do not suspect each other. Thus, if the adversary causes a suspicion between processes in the current quorum, a new quorum has to be issued.

We assume that the adversary follows the following strategy: It always waits until a quorum was output by all correct nodes. Then, if possible the adversary causes a single suspicion between nodes in that quorum. Further, we assume that the adversary selects 2 correct nodes and only causes suspicions between faulty nodes, or between a faulty node and one of the 2 selected correct nodes.

If the adversary behaves as described above, a run of any Quorum Selection algorithm can be described by a sequence $Q_1, s_1, Q_2, s_2, ...s_{k-1}, Q_k$ where $Q_l \subset \Pi$ is a quorum and $s_l = (s_l[0], s_l[1]) \in \Pi \times \Pi$ is a tuple representing a $s_l[0]$ suspecting $s_l[1]$. The following rules must hold:

$$\forall l < k : s_l[0] \in Q_l \wedge s_l[1] \in Q_l \tag{1}$$

$$\forall l < l' \leq k : s_l[0] \in Q_{l'} \rightarrow s_l[1] \notin Q_{l'} \tag{2}$$

Let τ be a sequence of suspicions and quorum as defined above of length $k < \binom{f+2}{2}$. We now show that the adversary may choose its faulty nodes such, that he can cause all suspicions s_l in τ and an additional suspicion s_k between nodes in Q_k.

Figure 5 shows an example for $f = 3$, where multiple quorums are possible, and every additional suspect removes at most one possible quorum. It is possible to follow the remaining part of the proof on the example of Figure 5 with $F^{+2} = \{a, b, c, d, e\}$ and resulting $F = \{a, b, e\}$.

By the assumption we have made about the strategy of the adversary, there exists a set F^{+2} containing $f+2$ nodes, such that all the tuples s_l in τ contain only nodes from F^{+2}. Since F^{+2} contains $f+2$ nodes there exist nodes $a, b \in F^{+2} \cap Q_k$ distinct from each other ($a \neq b$). Since $k < \binom{f+2}{2}$ and τ contains $k-1$ tuples, there exists additionally $c, d \in F^{+2}$, such that neither (c, d) nor (d, c) occurs in τ and that $a \notin \{c, d\}$. If the adversary chooses $F = F^{+2} \backslash \{c, d\}$ as faulty nodes, he may cause the suspicions in τ, since for every $s_l \in \tau$, $s_l[0]$

or $s_l[1]$ is in F. Since also a is in F, the adversary can cause another suspicion $s_k = (a, b)$, forcing Quorum Selection to issue another quorum. ■

VIII. Follower Selection

We now consider a variant of Quorum Selection that we call Follower Selection. Follower Selection can be used to select a quorum for applications that use a leader centric message pattern, e.g. where a single leader communicates with several followers, but followers do not directly communicate with each other. We show in the next section, that Follower Selection a different from Quorum Selection, Follower Selection can be solved after only $\mathcal{O}(f)$ quorum changes.

In Follower Selection, additionally to a set of processes $Q \subset \Pi$ a $\langle \text{QUORUM}, l, Q \rangle$ message contains a designated leader $l \in Q$. In leader centric applications, a followers omitting messages to another follower will not impact liveness of the application. We therefore change the **no suspicion** property from Quorum Selection to **no leader suspicion** which allows suspicions between followers:

- **No leader suspicion** Eventually no correct process in the quorum suspects the leader, and the leader, if correct, does not suspect any process in the quorum.

We show that, different from Quorum Selection, Follower Selection can be solved after only $\mathcal{O}(f)$ quorum changes under the following assumptions:

Assumption. *In the following, we assume that* $|\Pi| > 3f$. *We also assume that messages sent between correct processes arrive in FIFO order.*

Algorithm 2 shows our solution to Follower Selection. The rules to propagate suspicions, build a suspect graph G, and increase the *epoch* are the same as in Quorum Selection Algorithm 1, with the difference that after increasing the epoch, a new quorum with the default leader and followers is issued (Lines 12-14).

To find a Leader a process computes a maximal line subgraph $L \subset G$ and a leader l_L (see Definition 1). A maximal line subgraph is not unique, and two correct processes may compute different maximal line subgraphs. Further, Definition 1 does not ensure that a maximal line subgraph is maximal with regard to the number of edges or nodes it contains. However, maximality ensures that correct processes eventually agree on the same leader. If added suspicions do not result in a new leader, no new quorum will be issued (see Line 18).

Definition 1. *For a simple graph G, we define a* **line subgraph** *to be an acyclic subgraph with maximum degree 2, i.e. every node has at most 2 neighbors.*

A line subgraph L designates a leader l_L, i.e. the minimum node of degree 0.

$$l_L = \min\{i \in \Pi : \delta_L(i) = 0\}$$

Where $\delta_L(i)$ is the degree of node i in graph L. A **maximal line subgraph** *in G is a line subgraph L, such that for any other line subgraph $F \subset G$, $i < j$ holds for $p_i = l_F$ and $p_j = l_L$ (we write $l_F < l_L$).*

We rely on the leader to select followers and broadcast his choice (Line 25). The leader must select possible followers according to Definition 2. This ensures that any new suspicion between the leader and a follower will allow processes to select a new leader. Examples 1 and 2 below each show a suspect graph, a maximal line subgraph and possible followers.

Definition 2. *A node in a line subgraph L is a* **possible follower** *unless it is connected to two nodes of degree 1 in L.*

Reliance on a leader, to select followers opens a possibility for omission or commission failures. Namely a faulty leader may omit a FOLLOWERS message, send a malformed message or equivocate, sending different message to different processes.

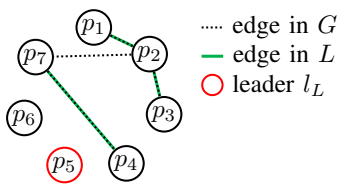
We rely on the Failure Detection module to detect this misbehavior. Thus on Line 23 processes tell the failure detector to expect a FOLLOWERS message from the leader. Before issuing this $\langle \text{EXPECT} \rangle$ previous expectations are cancelled on Line 21. If the leader does not send this message in a timely manor it will be suspected. Additionally, processes detect if a FOLLOWERS message is not well formed (see Definition 3) or if multiple follower messages are sent (Lines 29-32).

Definition 3. *A message $\langle \text{FOLLOWERS}, Fw, L', e \rangle_{\sigma_j}$ received at process i from process j is* well formed*, if properties a)-d) hold. In b) G_i is the suspect graph created at process i.*

a) $l \notin Fw \wedge |Fw| = q - 1$
b) $L' \subset G_i$ *and* L' *is a line subgraph*
c) $l_{L'} = j$
d) *all nodes* $fw \in Fw$ *are possible followers for* L'

We note that if i and j are correct, updates to the suspect graph G_j are forwarded to i in an UPDATE message, before a new line subgraph is computed, FIFO delivery ensures that $L' \subset G_i$ will hold.

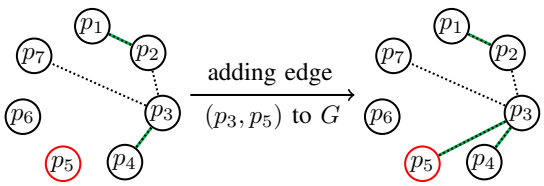
Example 1. *A graph G on 7 nodes and maximal line subgraph. Process p_2 is not a possible follower. Note that a new edge (p_2, p_5) added to G would not change the maximal line subgraph L.*



Algorithm 2 Selecting followers at process p_j

1: **State**
2: *epoch*, *suspecting*, *suspected*, *QLast* as in Algorithm 1
3: *leader* $= p_1$ {initial leader}
4: *stable* $=$ **true**
5: G, L {graphs created to find Quorum}
6: propagate suspicions as shown in Algorithm 1
7: updateQuorum()
8: $G \leftarrow$ buildSimpleGraph(*suspected*, *epoch*) {build simple graph, as described in Section VI-B}
9: **if** G contains no independent set of size q **then**
10: *epoch* $\leftarrow$ *epoch* $+ 1$
11: **issue** $\langle$CANCEL$\rangle$ {cancel expectations at failure detector }
12: *leader* $\leftarrow p_1$
13: *QLast* $\leftarrow \{p_1, p_2, ..., p_q\}$ {use default leader and Quorum}
14: **issue** $\langle$QUORUM, *leader*, *QLast*$\rangle$
15: updateSuspicions(*suspecting*)
16: **return**
17: $L \leftarrow$ buildMLineSubgraph(G) {find maximal line subgraph in G, see Definition 1}
18: **if** *leader* $\neq l_L$ **then** {l_L leader determined by L, see Definition 1}
19: *stable* $\leftarrow$ **false**
20: *leader* $\leftarrow l_L$
21: **issue** $\langle$CANCEL$\rangle$ {cancel expectations from Failure Detection }
22: **if** *leader* $\neq p_j$ **then** {I'm not the leader}
23: **issue** $\langle$EXPECT, $P_{Fw,epoch}$, *leader*$\rangle$ {expect signed FOLLOWERS message from leader in epoch}
24: **else**
25: $Fw \leftarrow$ selectFollowers(L) {select $q-1$ possible followers in L (see Definition 2)}
26: **bcast** $\langle$FOLLOWERS, $Fw, L, epoch\rangle_{\sigma_j}$ {broadcast signed FOLLOWERS message}
27: **on** $\langle$FOLLOWERS, $Fw, Ls, e\rangle_{\sigma_i}$ from process p_i
28: **if** $p_i =$ *leader* **and** $e =$ *epoch* **then**
29: **if** message is not well formed **then** {see Definition 3}
30: **issue** $\langle$DETECTED, $p_i\rangle$
31: **else if** *stable* $=$ **true and** $Fw \cup$ *leader* $\neq$ *QLast* **then**
32: **issue** $\langle$DETECTED, $i\rangle$ {equivocation detected}
33: **else if** *stable* $=$ **false then**
34: $stable \leftarrow$ **true**
35: *QLast* $\leftarrow Fw \cup$ *leader*
36: **bcast** $\langle$FOLLOWERS, $Fw, L, e\rangle_{\sigma_i}$ {forward FOLLOWERS}
37: **issue** $\langle$QUORUM, *leader*, *QLast*$\rangle$

Example 2. *An edge* (p_3, p_5) *is added to the graph* G *which changes the leader and maximal line subgraph* L*. Note that the line subgraph on the left side is maximal, even though it could be extended by additional edges.*



IX. Proofs for Follower Selection

In this section we proof that our Follower Selection, as described in Algorithm 2 and the definitions in Section VIII, is correct and allows the faulty processes to interrupt normal operation at most $\mathcal{O}(f)$ times.

We first proof that Algorithm 2 allows the failure detector to be accurate, i.e. that expected FOLLOWERS messages from correct processes arrive in time (Lemmas 5, 6) and that correct processes are not detected (Lemma 7).

We say that a quorum is issued in epoch e if *epoch* $= e$ at a process, when issuing this quorum. Theorem 9 shows that Algorithm 2 issues only a $\mathcal{O}(f)$ quorums in one epoch. As a corollary, we proof that only $\mathcal{O}(f)$ quorum are issued when

the failure detector is accurate.

Lemma 5. *If some correct process i detects leader $= j$ in epoch $= e$, then within one communication round, every correct process will have epoch $> e$ or epoch $= e$ and leader $\geq j$.*

Proof: According to Lemma 1 suspicions and epoch are propagated from i to other correct processes within one communication round. The lemma follows from maximality of the line subgraph used to determine the leader. ■

Lemma 6. *If some correct process i detects a correct process j as leader then within two communication rounds i either detects a new leader, changes its epoch or receives a* FOLLOWERS *message from j in the current epoch.*

Proof: According to Lemma 5, leader and epoch are forwarded from i to j within one round. Process j then either has detected a different leader, a higher epoch, or it sends a FOLLOWERS message. Any of these propagates back to i within one communication round. ■

Lemma 7. *All* FOLLOWERS *messages sent between correct processes are well formed.*

Proof: It is clear that a FOLLOWERS message created by a correct process, meets the criteria a), c) and d) of Definition 3. Criterium b) follows from the assumption that communication obeys FIFO order, since UPDATE messages are forwarded before sending a FOLLOWERS message. ■

In the following lemmas we say that a line subgraph L contains a node i, if i has non-zero degree in L.

Lemma 8. *a) If G contains a line subgraph containing $3f$ nodes, then G contains at most one unique independent set of size q. If it exists, the independent set contains the leader and all possible followers.*
b) If G contains a line subgraph containing $3f + 1$ nodes, then G contains no independent set of size q.

Proof: The lemma follows easily if one considers that the nodes not part in an independent set must form a vertex cover [19]. ■

Theorem 9. *A correct process issues at most $3f + 1$ quorums in one epoch.*

Proof: For a given leader l and epoch e a correct process issues at most one quorum in epoch e with leader l. If process with number $3f + 1$ is leader, there exists a maximal line subgraph with $3f$ nodes. According to Lemma 8, b) another suspicion between a possible follower and the leader will trigger a new epoch. ■

Corollary 10. *After time t' as in Theorem 3, correct processes will issue at most $6f + 2$ quorums.*

Proof: Let e_c be the highest epoch, such that a suspicion between correct processes is issued in epoch e_c. According to Lemma 6 all correct processes will have epoch e_c or higher at time t'. According to Theorem 9 at most $6f + 2$ quorums are issued in epochs e_c and e_c+1. After $3f+1$ quorums have been issued in epoch $e_c + 1$, the process with number $3f + 1$ is the leader and there exists a line subgraph containing $3f$ nodes. Lemma 8 a) implies that the leader and possible followers are exactly the correct nodes. ■

X. CONCLUSION

We have presented Quorum Selection and shown how our failure detector and quorum selection module can be integrated into the XPaxos protocol.

We showed a lower bound on the number of quorum changes that may be caused by an adversary and a loosely coupled algorithm for Quorum Selection. Finally we presented Follower Selection, a restricted variant of Quorum Selection that allows to circumvent the lower bound.

Future work will investigate how best to integrate Quorum Selection in different BFT algorithms or other special cases of Quorum Selection, e.g. when processes are communicating along a chain.

REFERENCES

[1] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[2] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," http://arxiv.org/abs/1807.04938, 2018.

[3] A. N. Bessani, J. Sousa, and E. A. P. Alchieri, "State machine replication for the masses with BFT-SMART," in *DSN '14*, 2014, pp. 355–362.

[4] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo, "Efficient byzantine fault-tolerance," *IEEE Trans. Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[5] S. Duan, K. N. Levitt, H. Meling, S. Peisert, and H. Zhang, "Byzid: Byzantine fault tolerance from intrusion detection," in *SRDS 2014*, 2014.

[6] T. Distler, C. Cachin, and R. Kapitza, "Resource-efficient byzantine fault tolerance," *IEEE Trans. Computers*, vol. 65, no. 9, pp. 2807–2819, 2016.

[7] S. Duan, H. Meling, S. Peisert, and H. Zhang, "Bchain: Byzantine replication with high throughput and embedded reconfiguration," in *OPODIS 2014*, 2014.

[8] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic, "XFT: practical fault tolerance beyond crashes," in *OSDI*, 2016.

[9] M. K. Aguilera, W. Chen, and S. Toueg, "Failure detection and consensus in the crash-recovery model," *Distributed Computing*, vol. 13, no. 2, pp. 99–125, 2000.

[10] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth, "Bar fault tolerance for cooperative services," in *SOSP '05*. ACM, 2005.

[11] F. C. Freiling, R. Guerraoui, and P. Kuznetsov, "The failure detector abstraction," *ACM Comput. Surv.*, vol. 43, no. 2, Feb. 2011.

[12] A. Doudou, B. Garbinato, and R. Guerraoui, "Encapsulating failure detection: From crash to byzantine failures," in *Reliable Software Technologies - Ada-Europe 2002*, 2002, pp. 24–50.

[13] D. Malkhi and M. Reiter, "Unreliable intrusion detection in distributed computations," in *IEEE Workshop on Computer Security Foundations*, ser. CSFW '97, 1997.

[14] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper, "Muteness failure detectors: Specification and implementation," in *EDCC-3*. Springer-Verlag, 1999, pp. 71–87.

[15] R. Baldoni, J. Hélary, and M. Raynal, "From crash fault-tolerance to arbitrary-fault tolerance: Towards a modular approach," in *DSN '00*, 2000, pp. 273–282.

[16] A. Haeberlen, P. Kuznetsov, and P. Druschel, "PeerReview: Practical accountability for distributed systems," in *SOSP'07*, Oct 2007.

[17] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, 2nd ed. Springer Publishing Company, 2011.

[18] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[19] Wikipedia contributors, "Vertex cover — Wikipedia, the free encyclopedia," 2018.