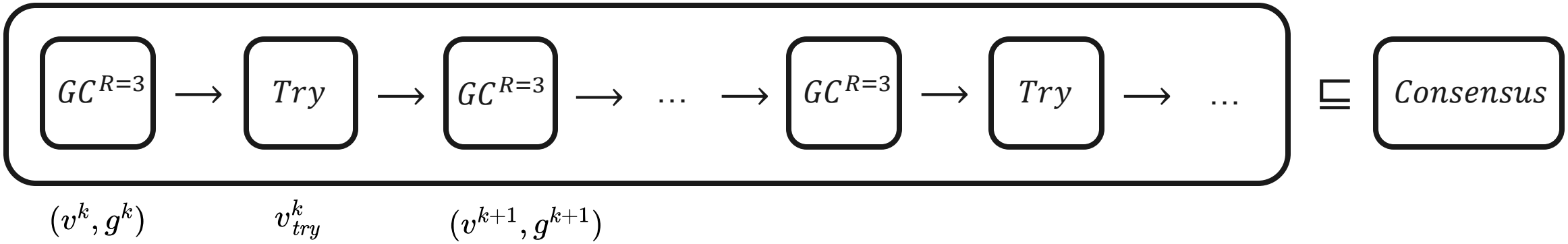
Consensus = stay safe + try, and retry again

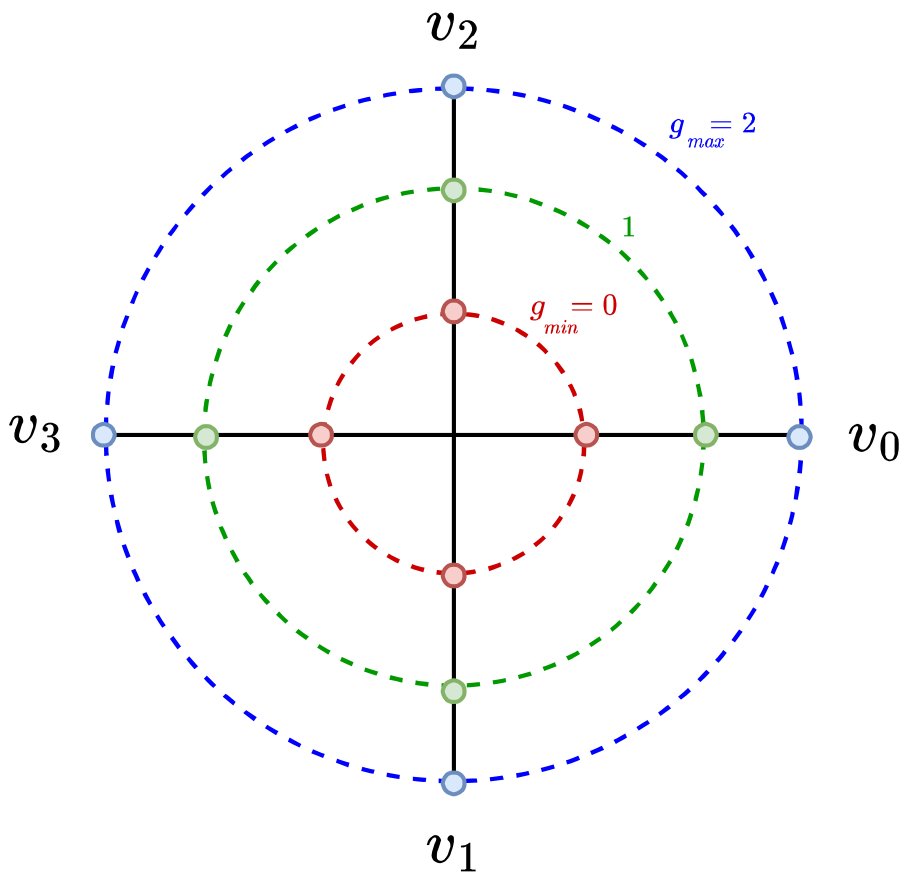
Distributed Algorithms



if $g^k = g_{max}$:
decide$(v^k) \wedge$
halt after $\mathcal{I}^{k+1}$

if $g^k > g_{min}$:
$v^k_{try} \leftarrow v^k$

$GC^{R=3}$



Byzantine Agreement (a.k.a. Consensus)

History

1978

The Implementation of Reliable Distributed Multiprocess Systems*

Leslie Lamport **
Massachusetts Computer Associates, Inc., 26 Princess Street, Wakefield, Mass. 01880, USA

A method is described for implementing any system by a network of processes so it continues to function properly despite the failure or malfunction of individual processes and communication arcs; where "malfunction" means doing something incorrectly, and "failure" means doing nothing. The system is defined in terms of a sequential "user machine", and a precise correctness condition for the implementation of this user machine is given. An algorithm to implement the user machine in the absence of malfunctioning, and a rigorous proof of its correctness, are given for a network of three processes with perfect clocks. The generalization to an arbitrary network of processes with imperfect clocks is described. It is briefly indicated how malfunctioning can be handled by adding redundant checking to the implementation and including error detection and correction in the user machine.

1. Introduction

1.1. Goals

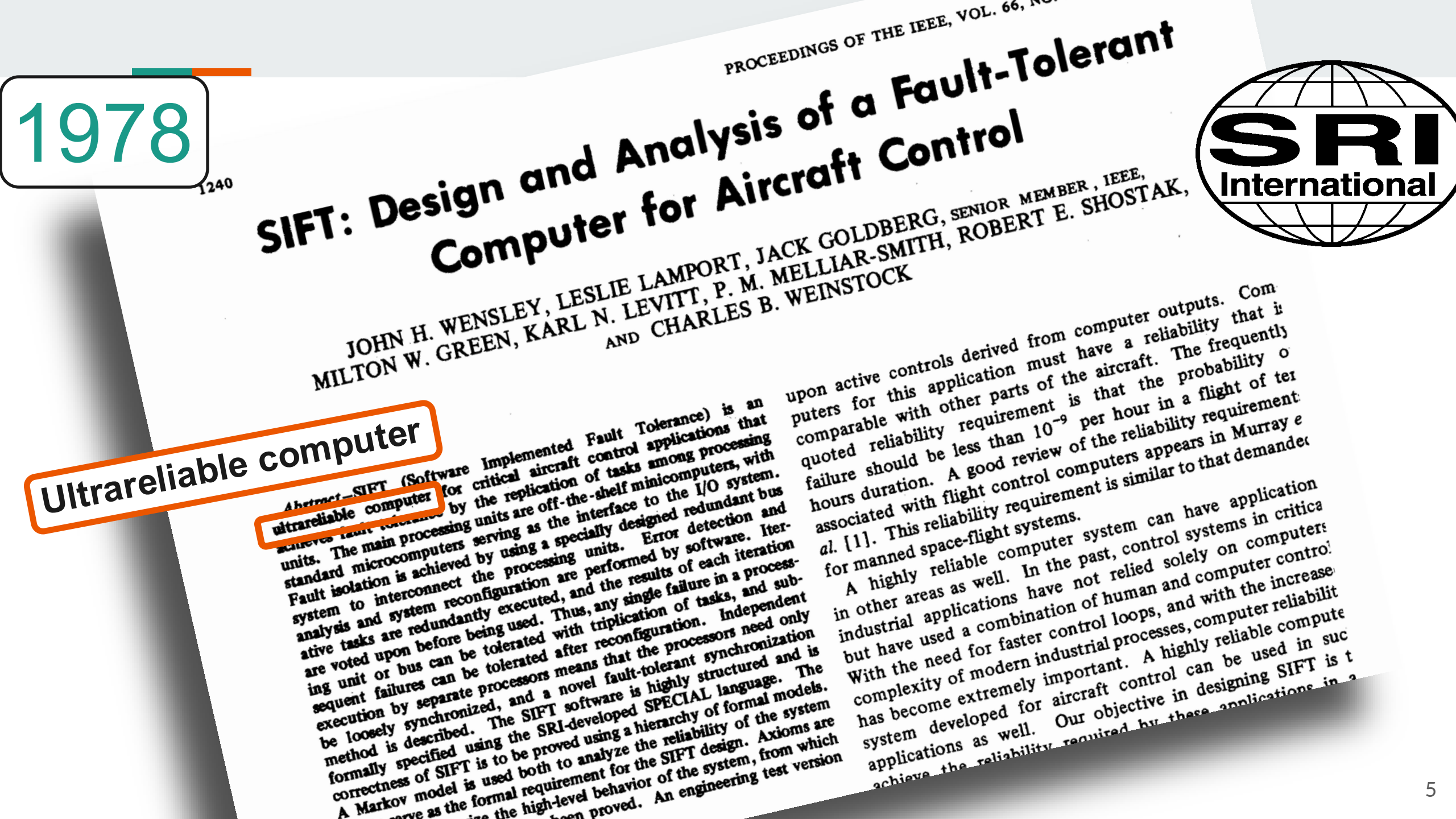
By a distributed multiprocess system, we mean a system composed of physically separated processes which communicate with one another by sending messages. We will describe an algorithm for implementing any system as a highly reliable distributed multiprocess system. The reliability of such a system involves two major goals.

Goal 1. To enable the system to continue functioning despite the failure of one or more processes or communication lines.

Goal 2. To enable the system to function correctly despite the malfunctioning of one or more processes or communication lines.

1978

Ultrareliable computer



PROCEEDINGS OF THE IEEE, VOL. 66, NO.

1240

SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control

JOHN H. WENSLEY, LESLIE LAMPORT, JACK GOLDBERG, SENIOR MEMBER, IEEE, MILTON W. GREEN, KARL N. LEVITT, P. M. MELLIAR-SMITH, ROBERT E. SHOSTAK, AND CHARLES B. WEINSTOCK

Abstract—SIFT (Software Implemented Fault Tolerance) is an ultrareliable computer for critical aircraft control applications that achieves fault tolerance by the replication of tasks among processing units. The main processing units are off-the-shelf minicomputers, with standard microcomputers serving as the interface to the I/O system. Fault isolation is achieved by using a specially designed redundant bus system to interconnect the processing units. Error detection and analysis and system reconfiguration are performed by software. Iterative tasks are redundantly executed, and the results of each iteration are voted upon before being used. Thus, any single failure in a processing unit or bus can be tolerated with triplication of tasks, and subsequent failures can be tolerated after reconfiguration. Independent execution by separate processors means that the processors need only be loosely synchronized, and a novel fault-tolerant synchronization method is described. The SIFT software is highly structured and is formally specified using the SRI-developed SPECIAL language. The correctness of SIFT is to be proved using a hierarchy of formal models. A Markov model is used both to analyze the reliability of the system as well as the formal requirement for the SIFT design. Axioms are used to prove the high-level behavior of the system, from which an engineering test version

upon active controls derived from computer outputs. Com
puters for this application must have a reliability that i
comparable with other parts of the aircraft. The frequently
quoted reliability requirement is that the probability o
failure should be less than 10^{-9} per hour in a flight of ten
hours duration. A good review of the reliability requirement
associated with flight control computers appears in Murray *e*
al. [1]. This reliability requirement is similar to that demanded
for manned space-flight systems.

A highly reliable computer system can have application
in other areas as well. In the past, control systems in critica
industrial applications have not relied solely on computers
but have used a combination of human and computer control
With the need for faster control loops, and with the increase
complexity of modern industrial processes, computer reliabilit
has become extremely important. A highly reliable compute
system developed for aircraft control can be used in suc
applications as well. Our objective in designing SIFT is t
achieve the reliability required by these applications in a

1980

Reaching Agreement in the Presence of Faults

M. PEASE, R. SHOSTAK, AND L. LAMPORT

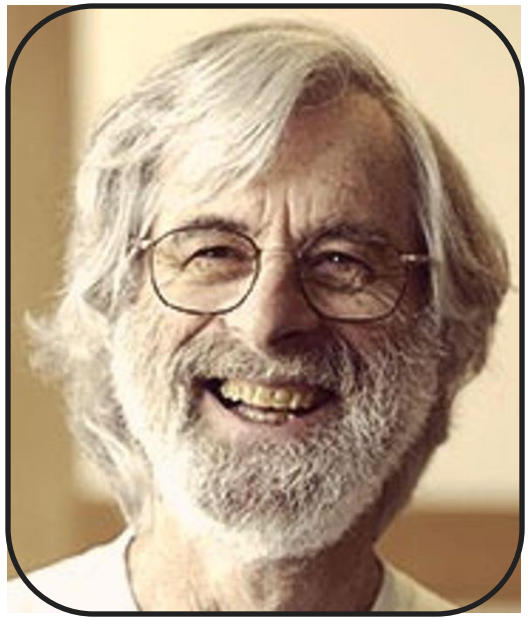
SRI International, Menlo Park, California

ABSTRACT. The problem addressed here concerns a set of isolated processors, some unknown subset of which may be faulty, that communicate only by means of two-party messages. Each nonfaulty processor has a private value of information that must be communicated to each other nonfaulty processor. Nonfaulty processors always communicate honestly, whereas faulty processors may lie. The problem is to devise an algorithm in which processors communicate their own values and relay values received from others that allows each nonfaulty processor to infer a value for each other processor. The value inferred for a nonfaulty processor must be that processor's private value, and the value inferred for a faulty one must be consistent with the corresponding value inferred by each other nonfaulty processor.

It is shown that the problem is solvable for, and only for, $n \geq 3m + 1$, where m is the number of faulty processors and n is the total number. It is also shown that if faulty processors can refuse to pass on information but cannot falsely relay information, the problem is solvable for arbitrary $n \geq m \geq 0$. This weaker assumption can be approximated in practice using cryptographic methods.

KEY WORDS AND PHRASES: agreement, authentication, consistency, distributed executive, fault avoidance, fault tolerance, synchronization, voting

CR CATEGORIES: 3.81, 4.39, 5.29, 5.39, 6.22



I have long felt that, because it was posed as a cute problem about philosophers seated around a table, Dijkstra's dining philosopher's problem received much more attention than it deserves. (For example, it has probably received more attention in the theory community than the readers/writers problem, which illustrates the same principles and has much more practical importance.) **I believed that the problem introduced in [Reaching Agreement in the Presence of Faults] was very important and deserved the attention of computer scientists**. The popularity of the dining philosophers problem taught me that the best way to attract attention to a problem is to present it in terms of a story.

https://lamport.azurewebsites.net/pubs/pubs.html

1982

The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement. It is shown that, using only oral messages, this problem is solvable if and only if more than two-thirds of the generals are loyal; so a single traitor can confound two loyal generals. With unforgeable written messages, the problem is solvable for any number of generals and possible traitors. Applications of the solutions to reliable computer systems are then discussed.

Categories and Subject Descriptors: C.2.4. [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.4 [**Operating Systems**]: Communications Management—*network communication*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*

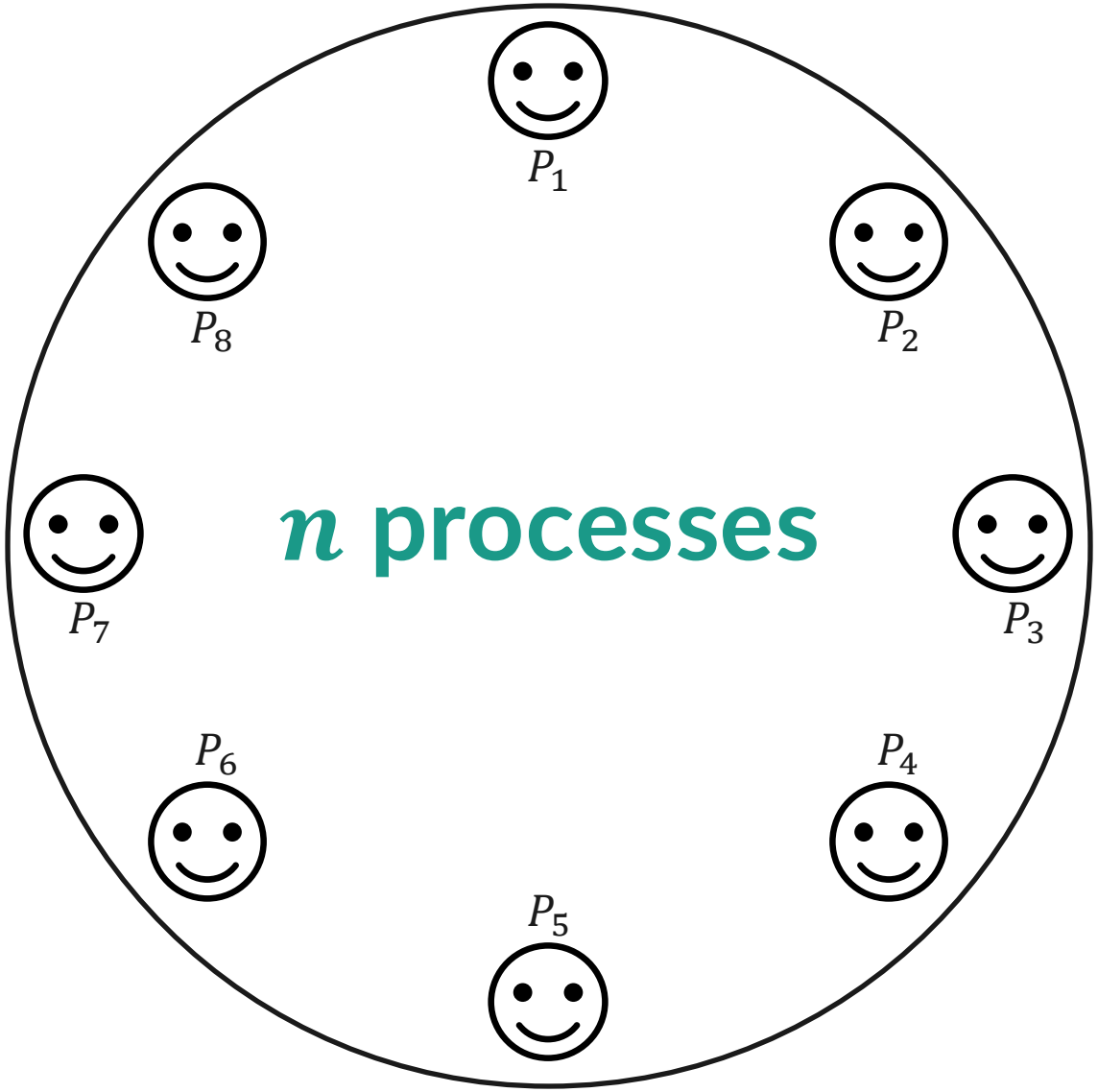
General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Interactive consistency

Consensus

Problem definition

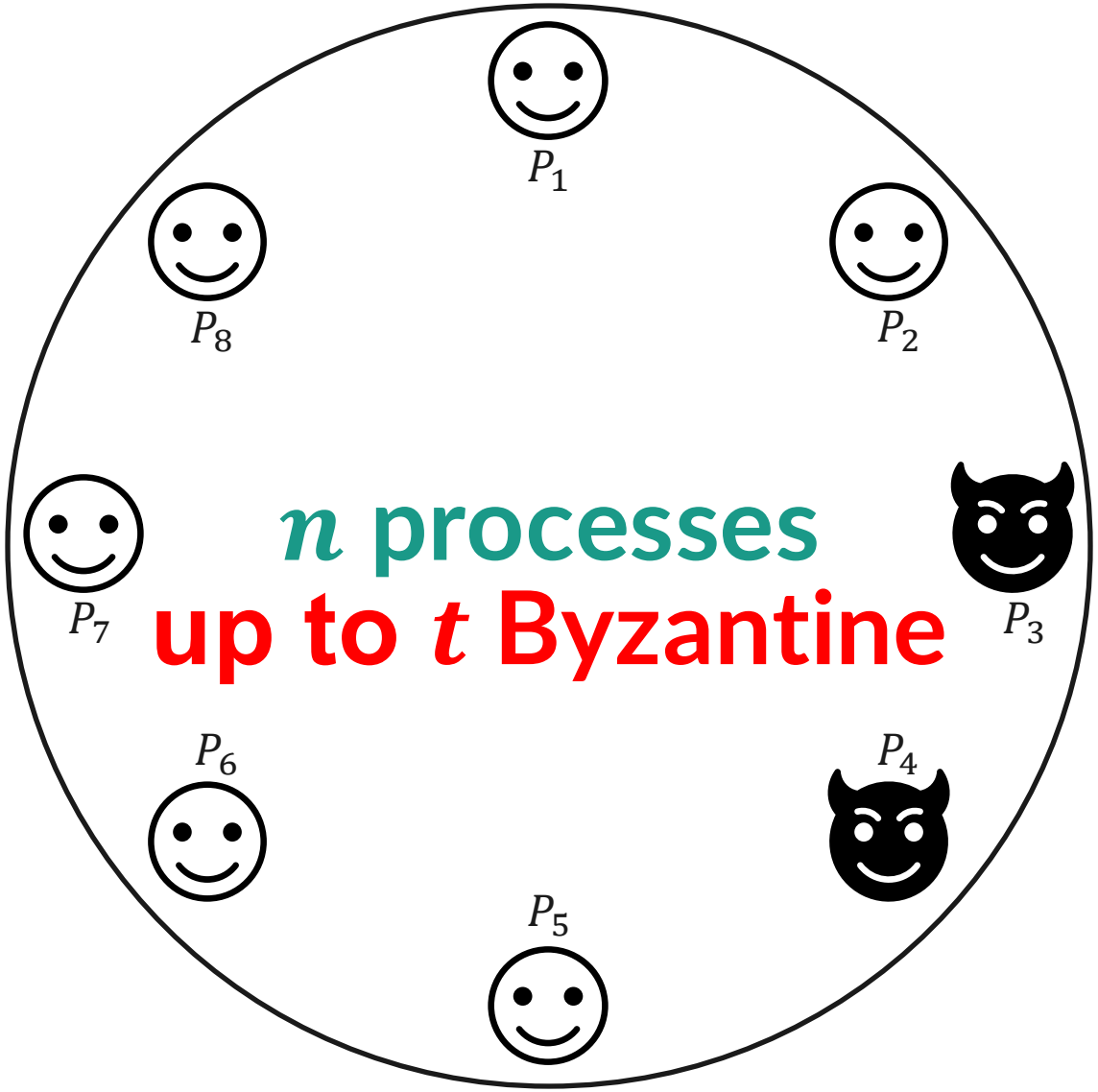
System



Static set of n publicly known identities

Message passing through reliable point-to-point channels

System



Static set of n publicly known identities

Message passing through reliable authenticated point-to-point channels

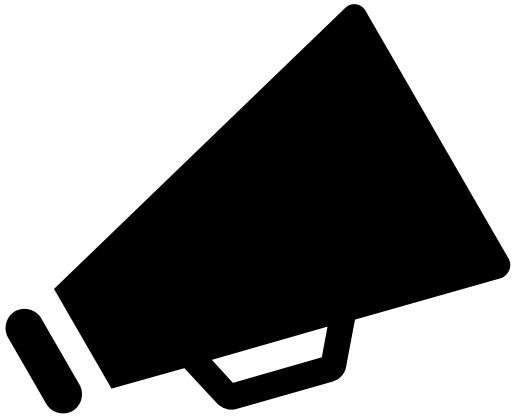
Asynchronous Model (Informal)

- No shared global clock: no shared notion of time
- Arbitrary (but finite) message delays
- Model is purely event-driven: reception → sending

Interface of Consensus

Propose

Operation



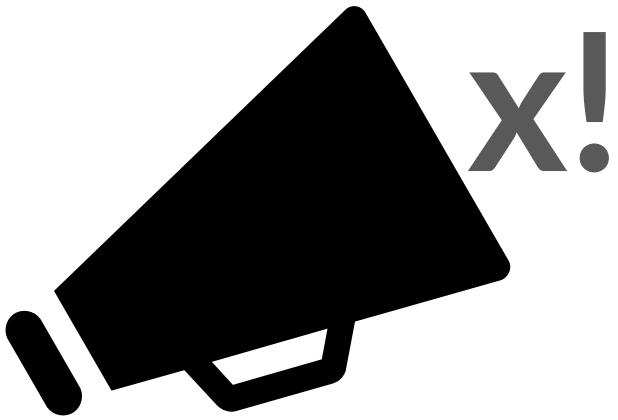
Decide

Callback

Interface of Consensus

Propose

Operation



Decide

Callback

Interface of Consensus

Propose
Operation

x!

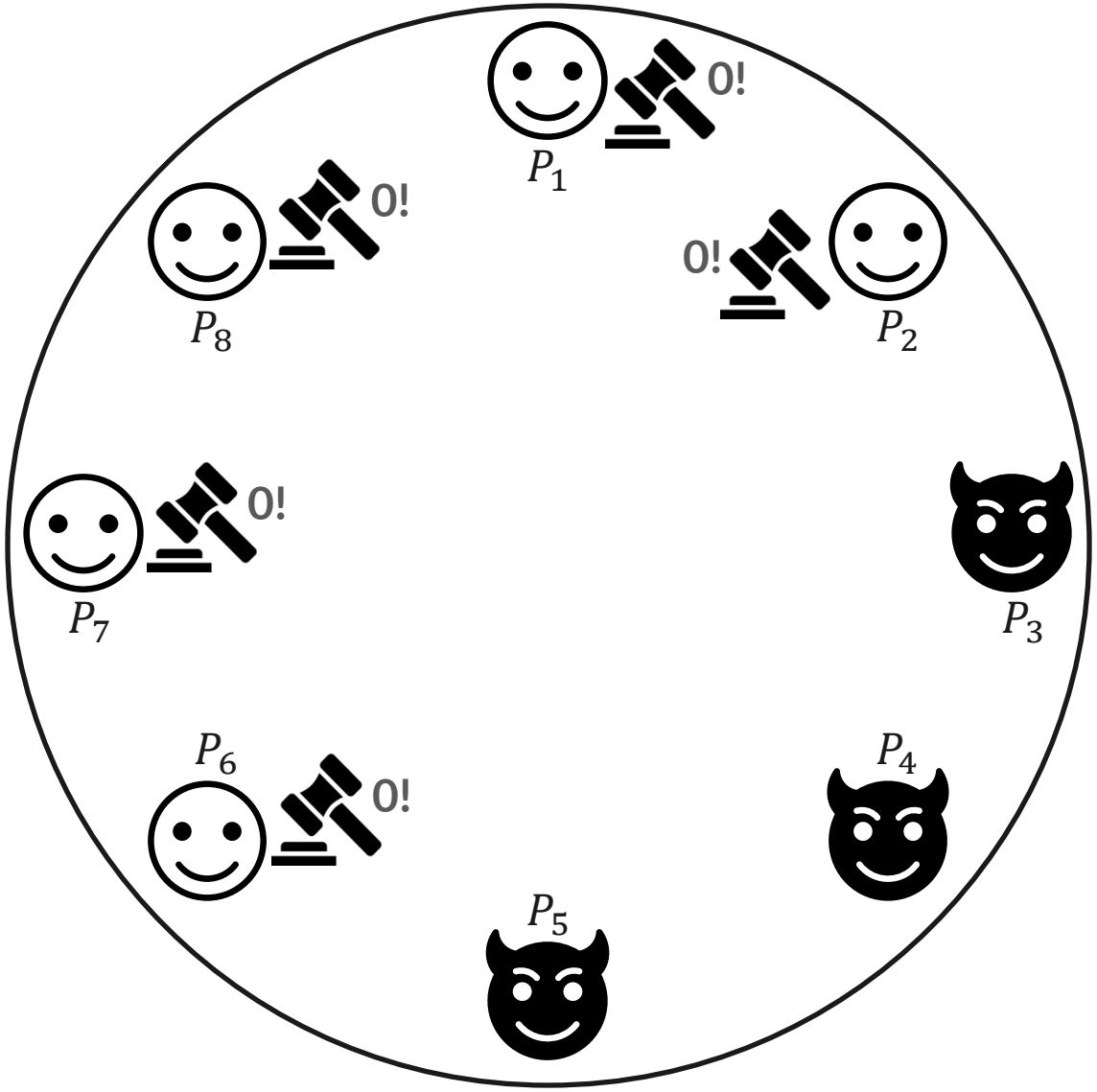
Decide
Callback

y!

Properties of Consensus

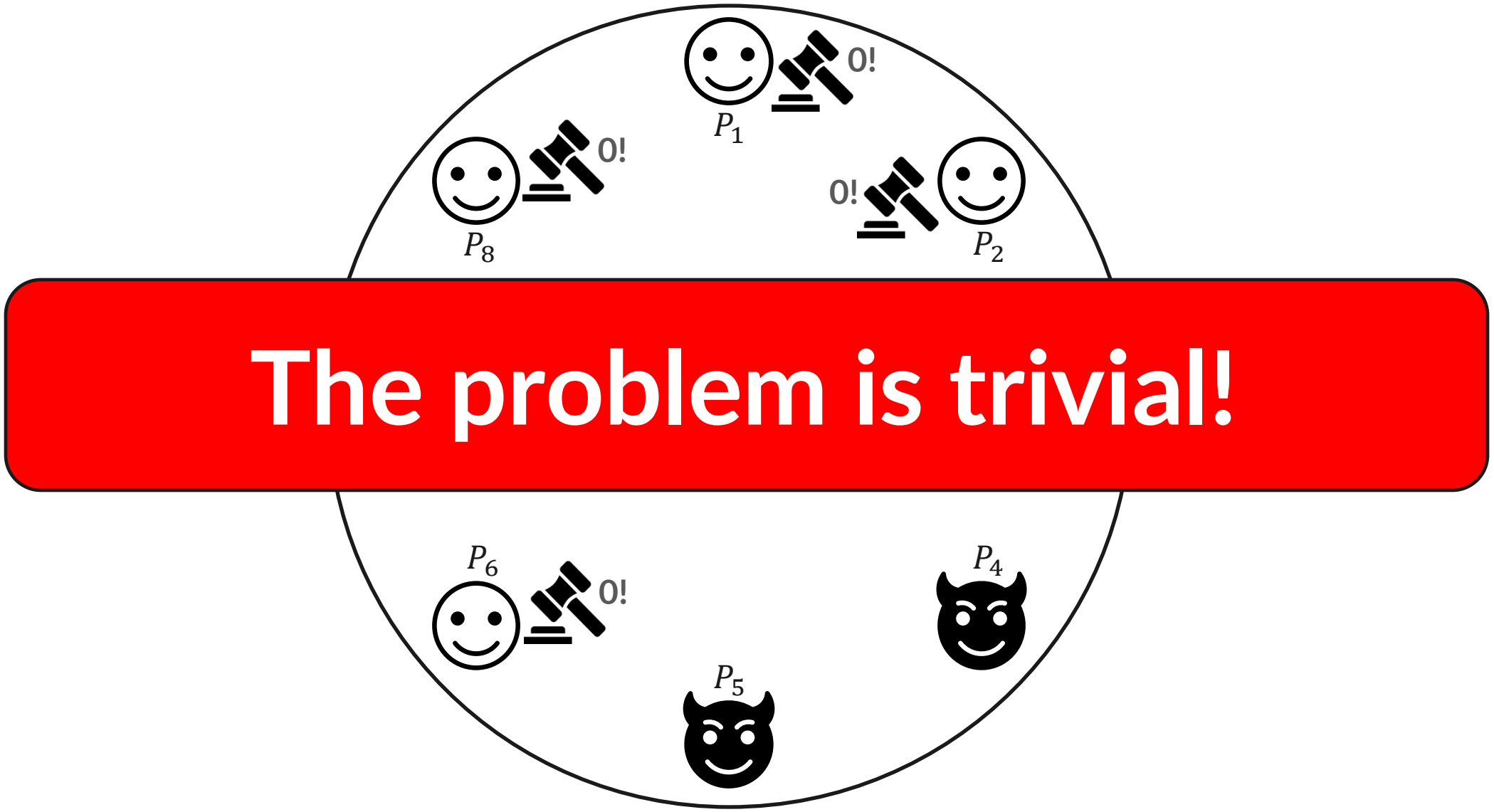
Agreement + **Termination**

Agreement + Termination



All correct processes eventually decide the same value.

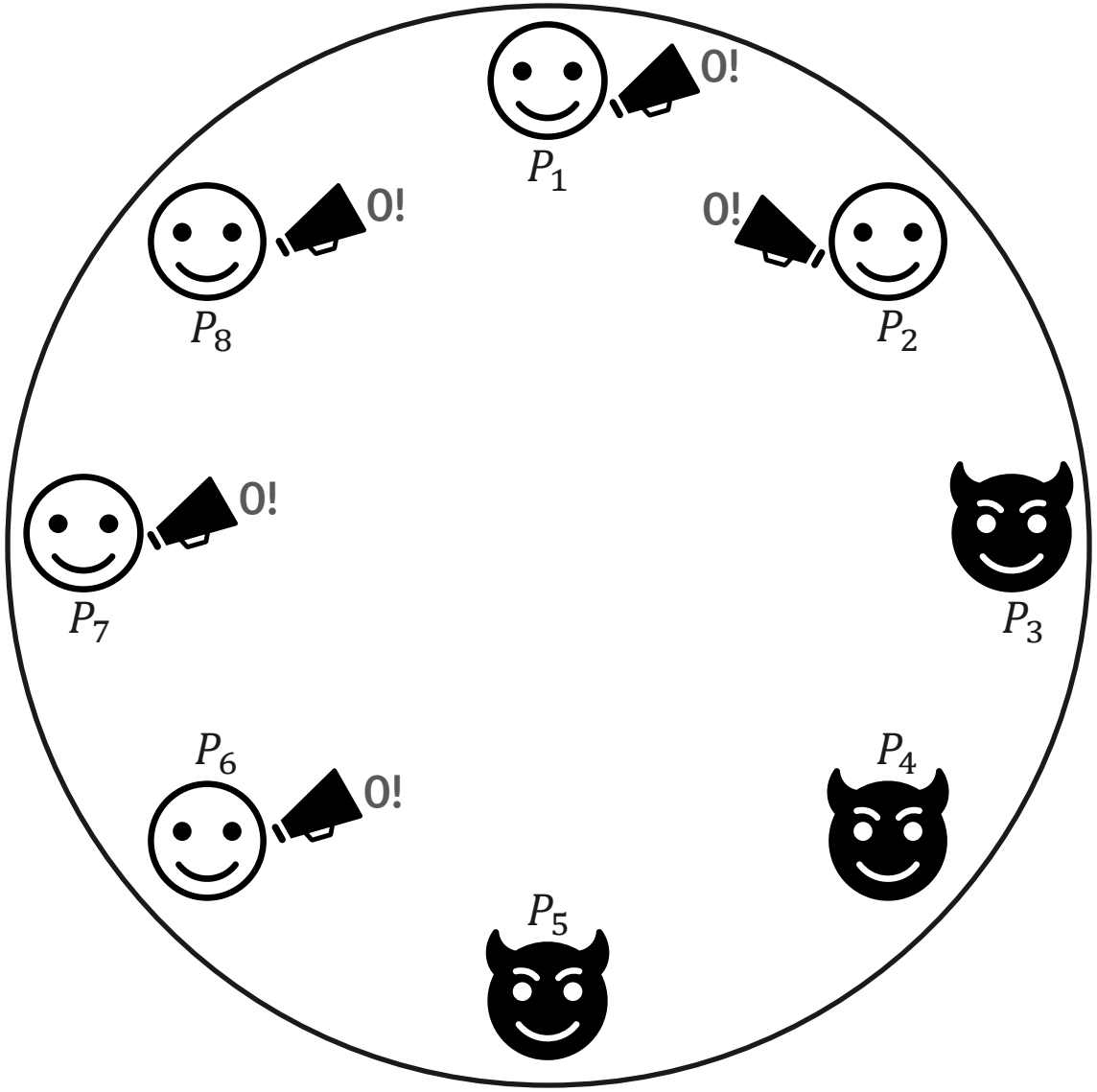
Agreement + Termination



The problem is trivial!

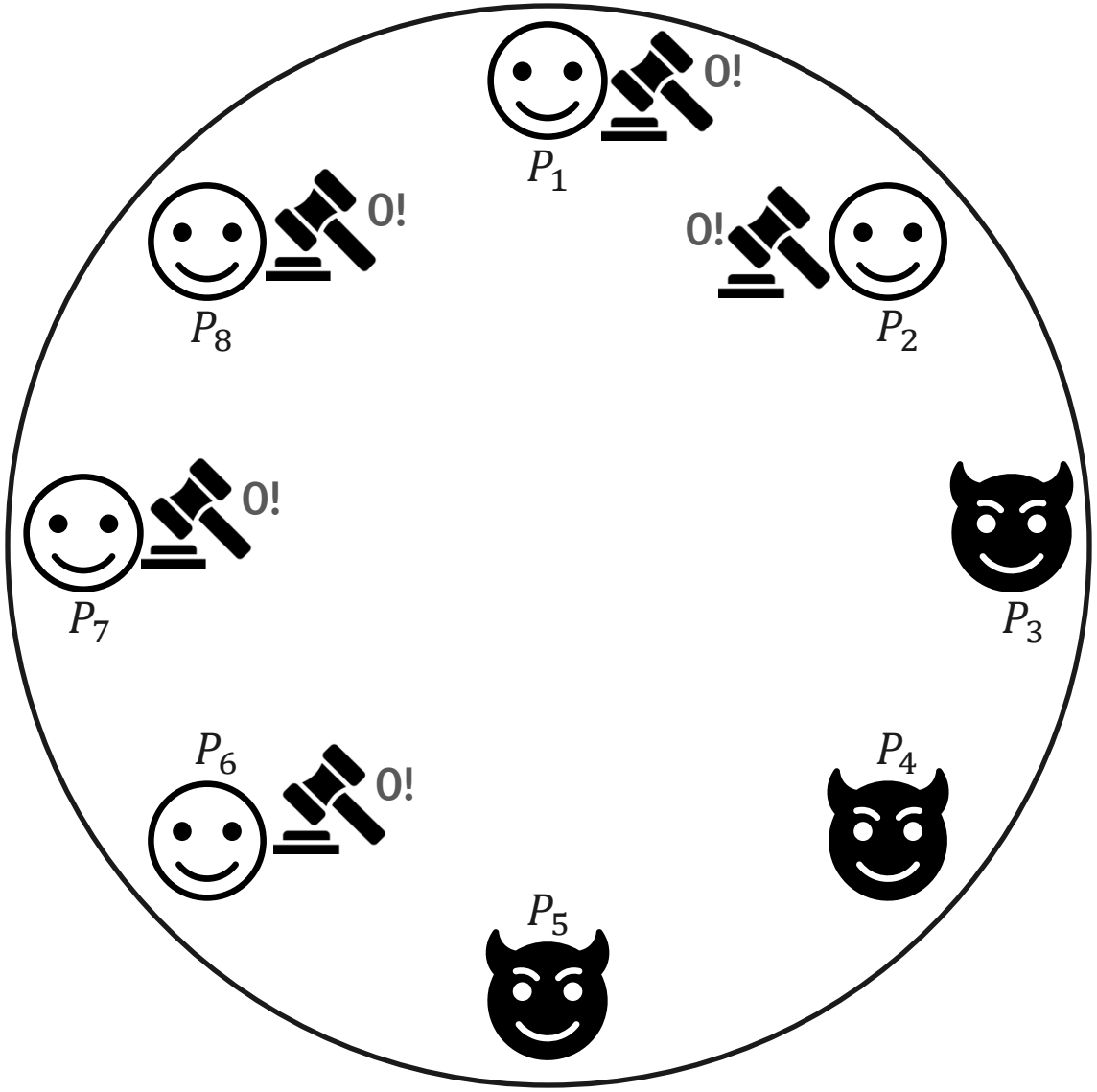
All correct processes eventually decide the same value.

Agreement + Termination + Validity



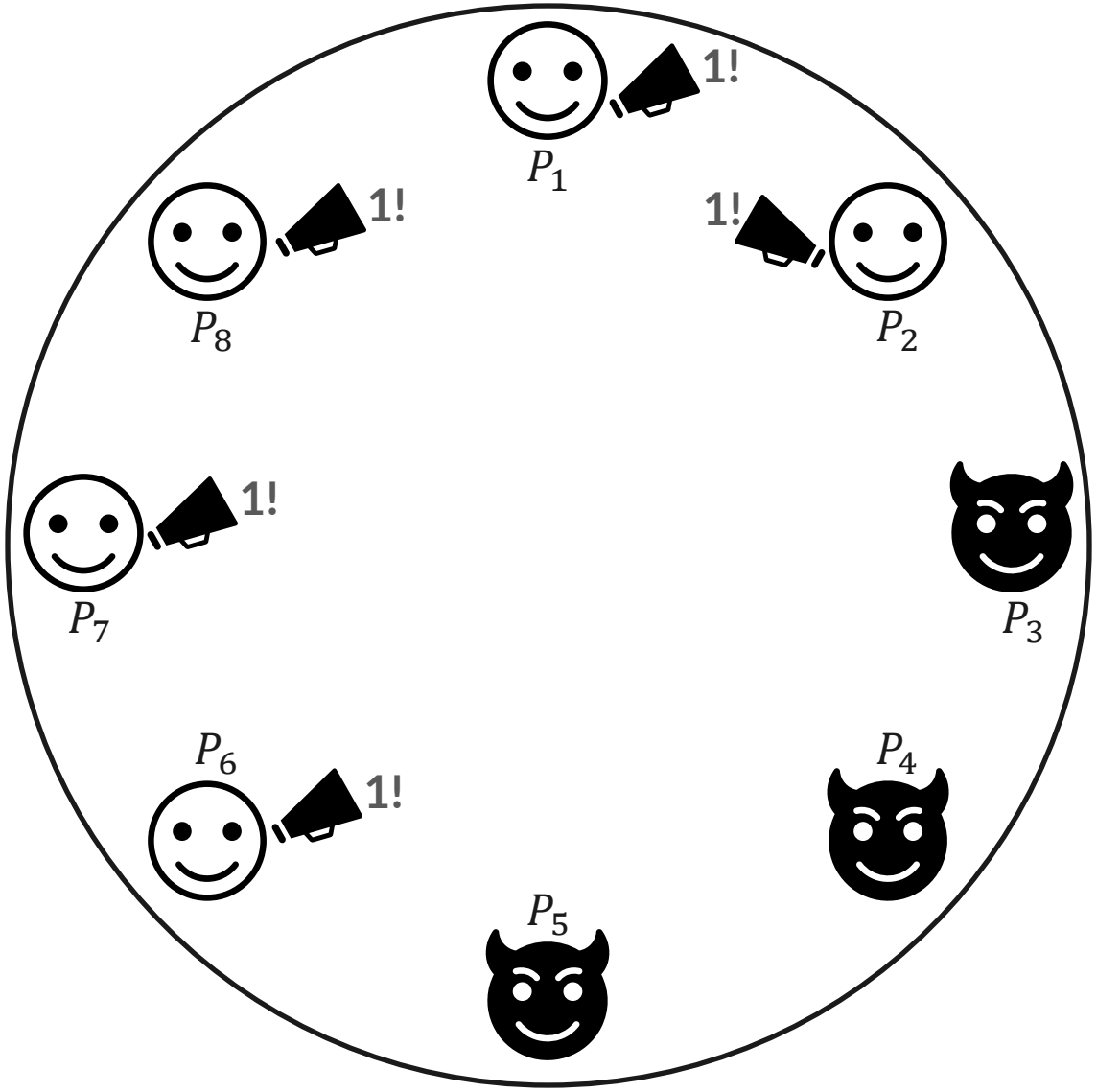
All correct processes eventually decide the same *admissible* value.

Agreement + Termination + Validity



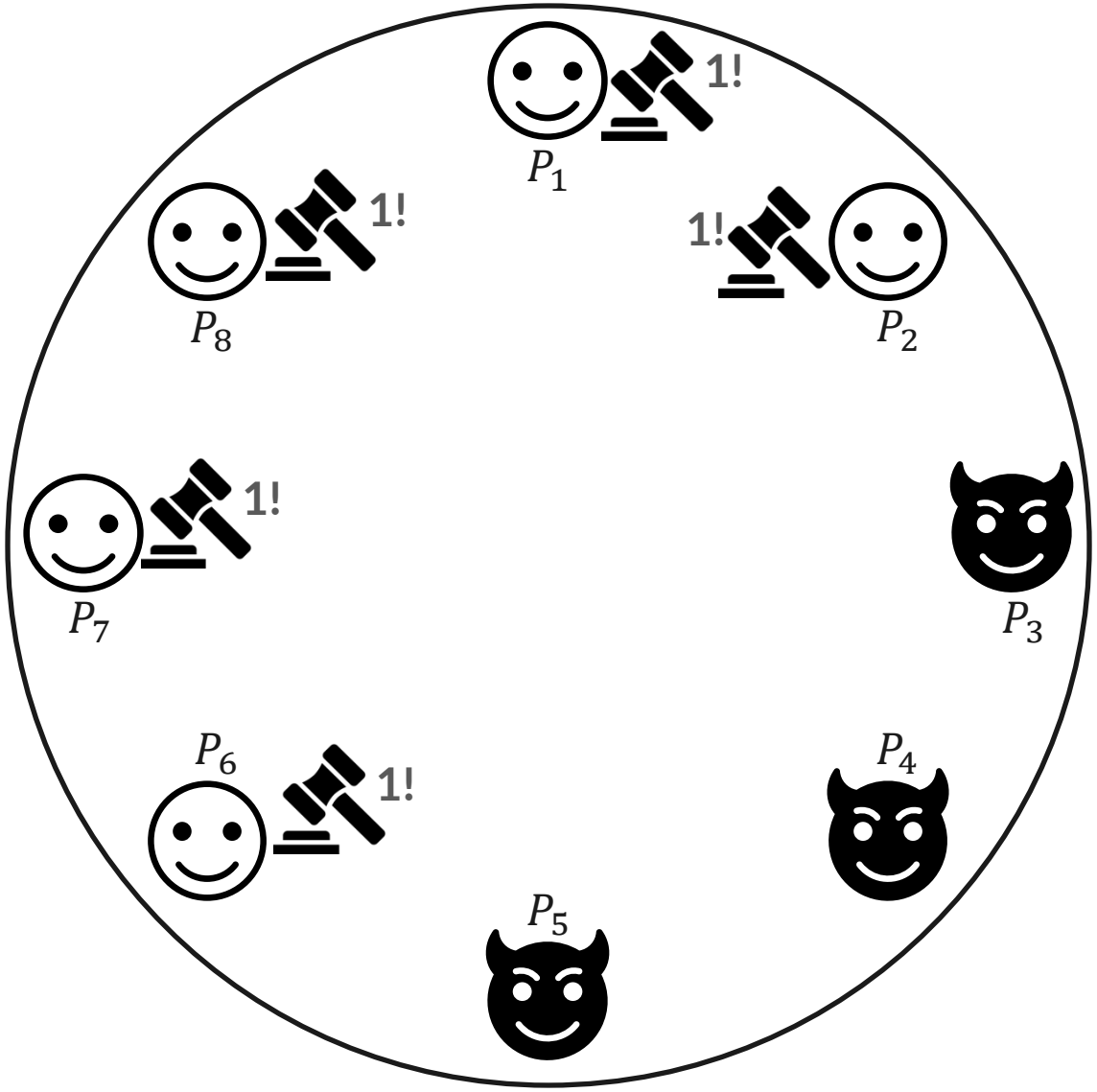
All correct processes eventually decide the same *admissible* value.

Agreement + Termination + Validity



All correct processes eventually decide the same *admissible* value.

Agreement + Termination + Validity



All correct processes eventually decide the same *admissible* value.

Traditional Validity Properties

- ***Strong Unanimity Validity:*** If all correct processes propose the same value, only that value is admissible.
- ***Weak Validity:*** If all processes are correct and they all propose the same value, only that value is admissible.
- ***Honest-Input Validity:*** Only values proposed by correct processes are admissible.

...

Traditional Validity Properties

- ***Strong Unanimity Validity:*** If all correct processes propose the same value, only that value is admissible.

- ***Weak Validity:*** If all processes are correct and they all propose the same value, only that value is admissible.

- ***Honest-Input Validity:*** Only values proposed by correct processes are admissible.

...

Traditional Validity Properties

- ***Strong Unanimity Validity:*** If all correct processes propose the same value, only that value is admissible.

Used as a building block for ``stronger'' (though ``computationally equivalent'') primitives:

Vector Consensus, Total-Order Broadcast (a.k.a., Atomic Broadcast), State-Machine-Replication, ``Blockchain'', ...

Graded Consensus

« Stay safe »

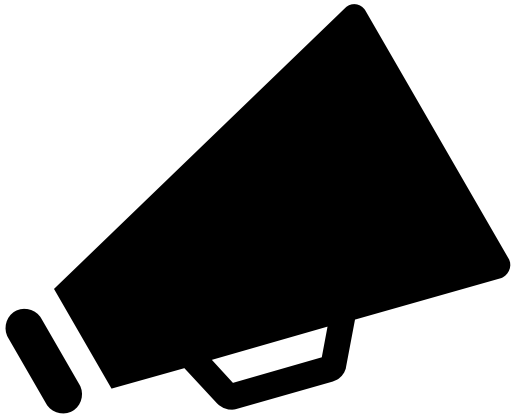
Graded Consensus

Specification

Interface of graded consensus

Propose

Operation

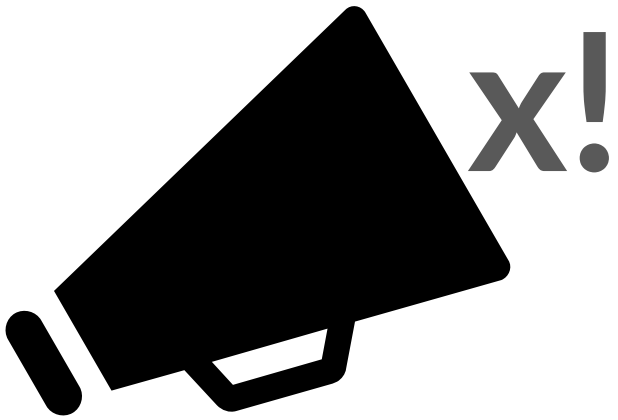


Decide

Callback

Interface of graded consensus

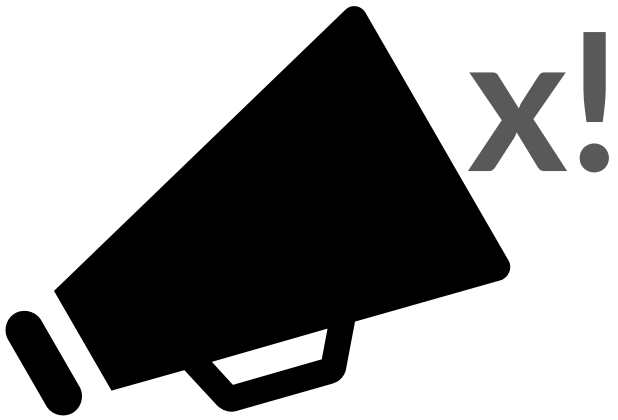
Propose
Operation



Decide
Callback

Interface of graded consensus

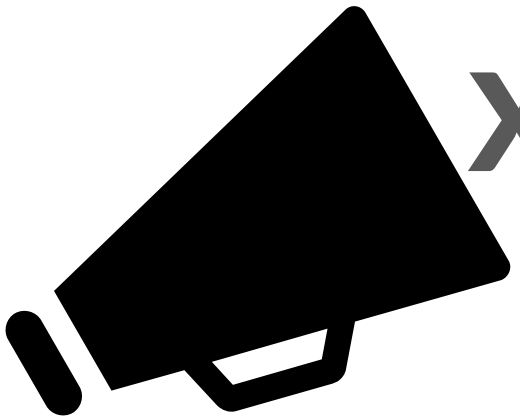
Propose
Operation



Decide
Callback

Interface of graded consensus

Propose
Operation



x!

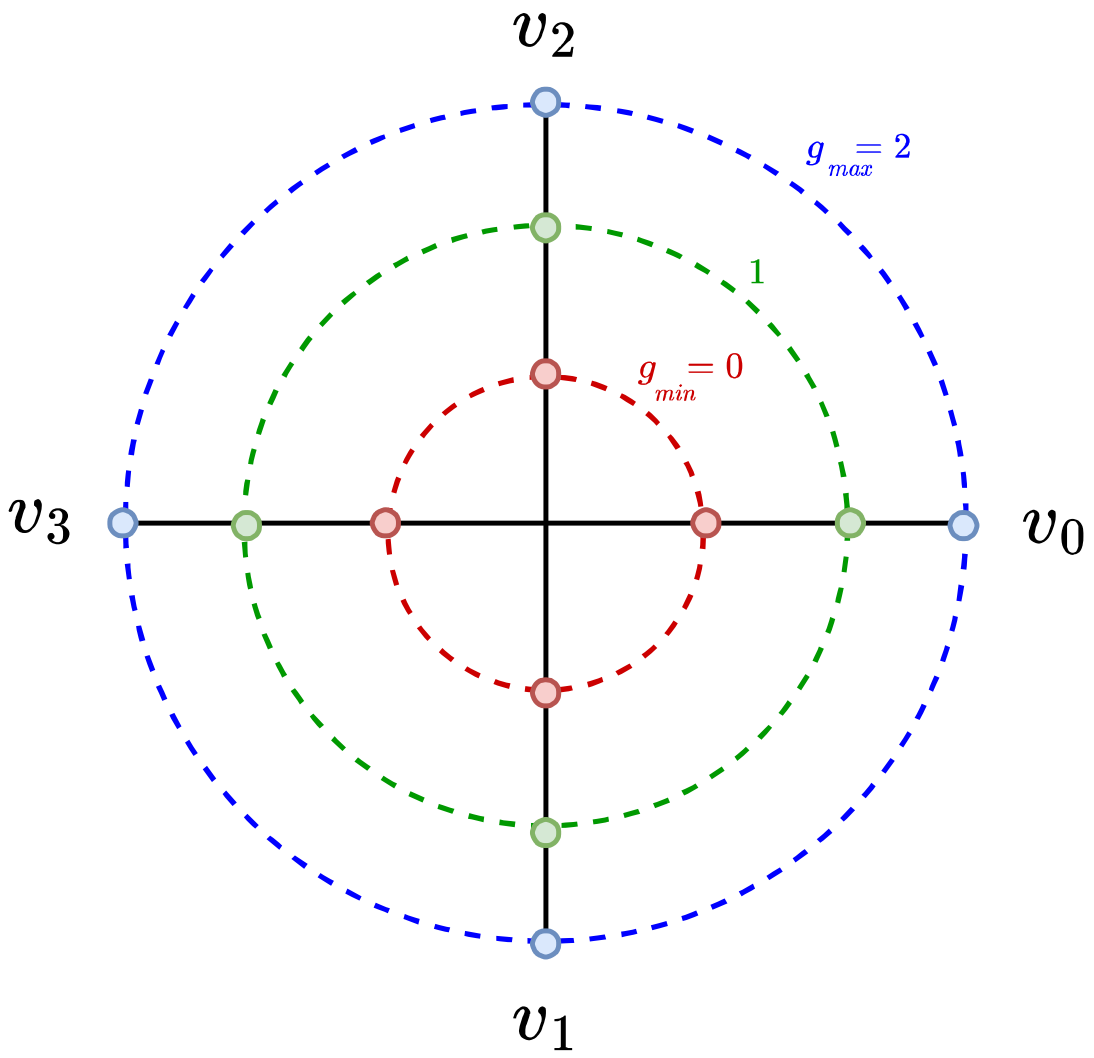
Decide
Callback

(y,g)!

Grade = ‘confidence’

$g_{min}, \cdots, g_{max}$

Interface of graded consensus



Decide

Callback

(y,g)!

Grade = ‘confidence’

$g_{min}, \cdots, g_{max}$

Properties of Graded Consensus

- **Termination:** All correct processes decide.

Properties of Graded Consensus

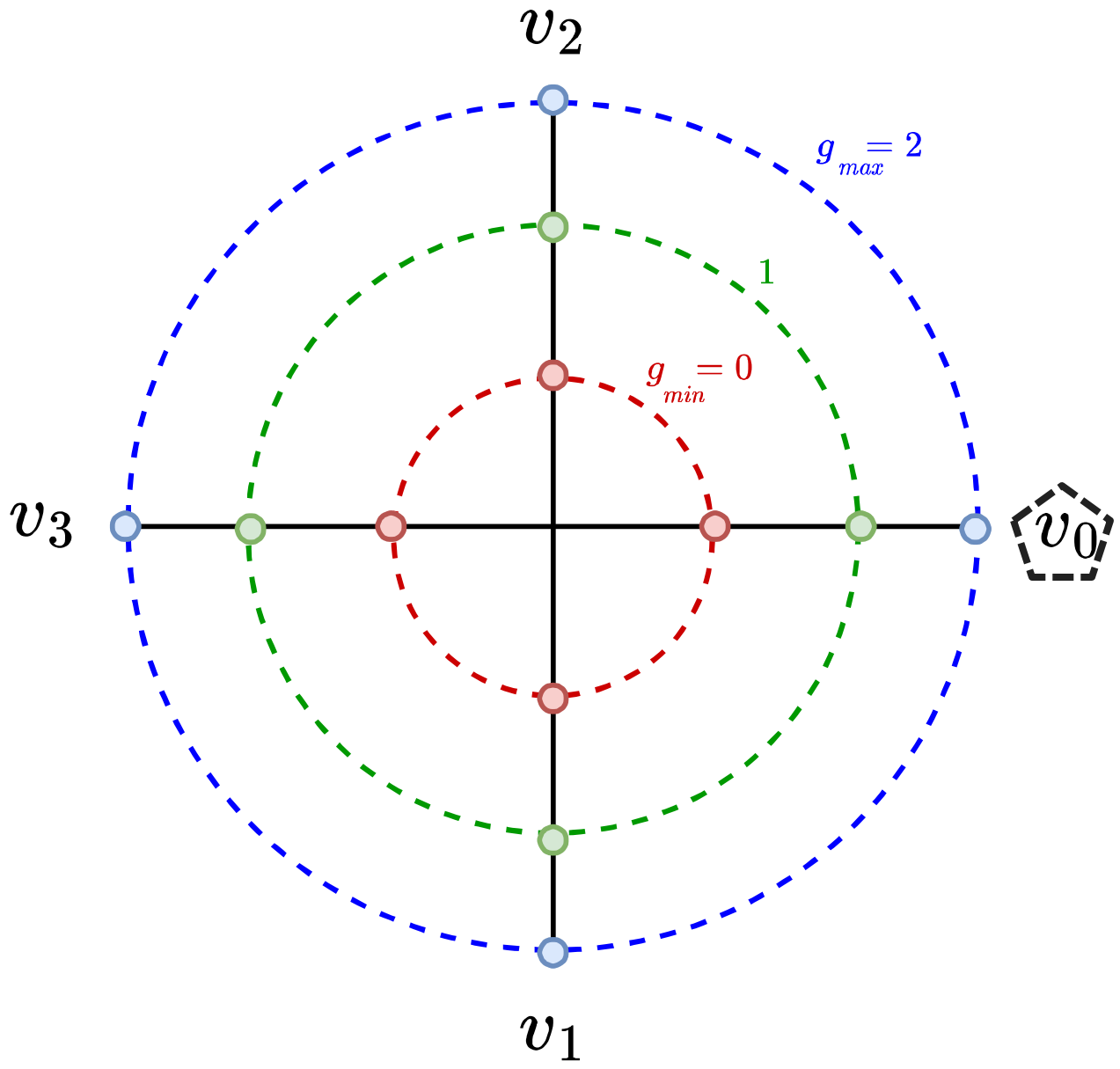
- **Termination:** All correct processes decide.
- **Unanimity:** If only v is proposed, then only …

Properties of Graded Consensus

- **Termination:** All correct processes decide.
- **Unanimity:** If only v is proposed, then only (v, g_{max}) can be decided.

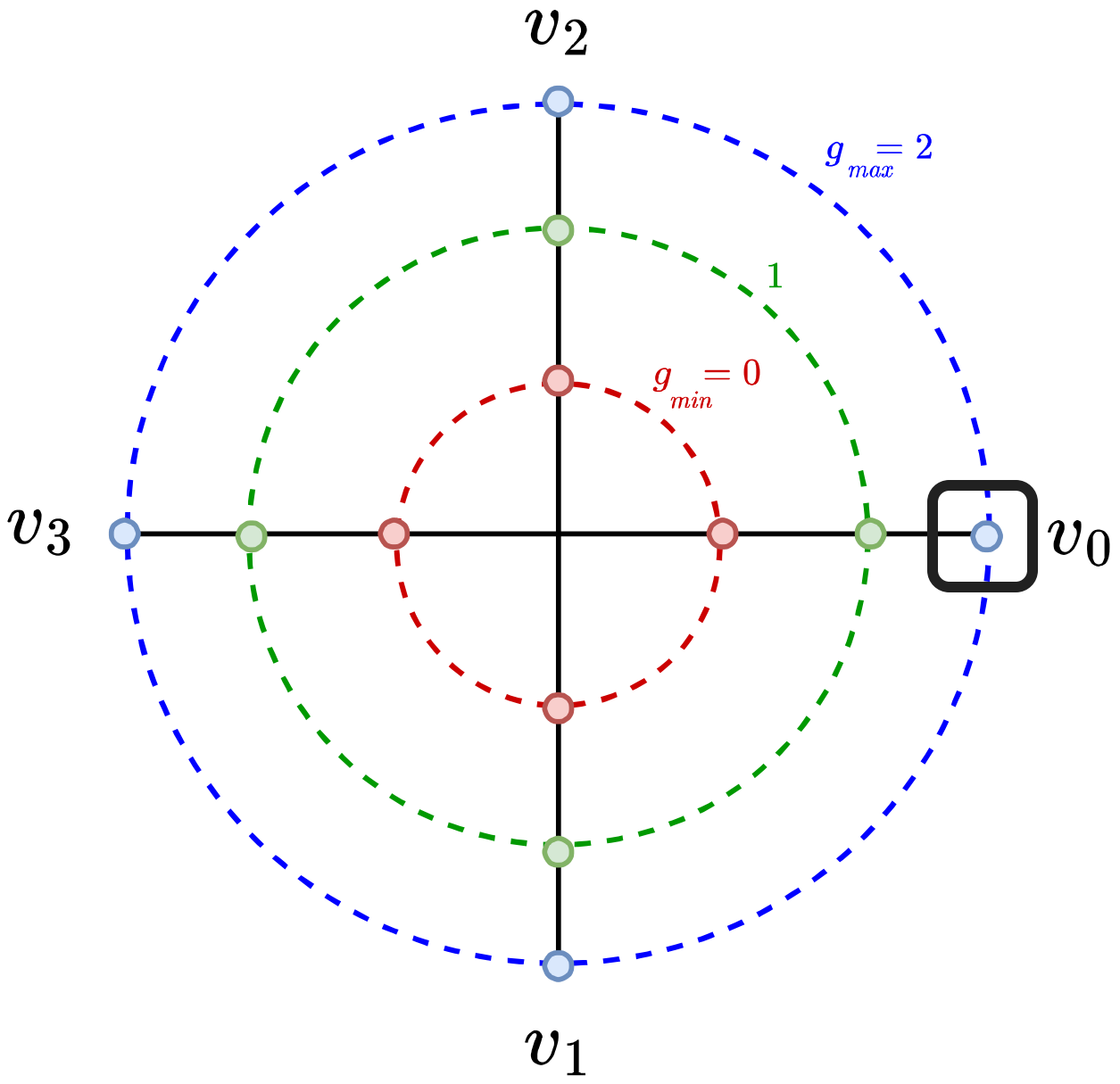
Unanimity

$$GC^{R=3}$$



Unanimity

$$GC^{R=3}$$



Properties of Graded Consensus

- **Termination:** All correct processes decide.
- **Unanimity:** If only v is proposed, then only (v, g_{max}) can be decided.
- **Consistency (~~Agreement~~):** Assume two correct processes decide (v, g) and (v', g'). We have …

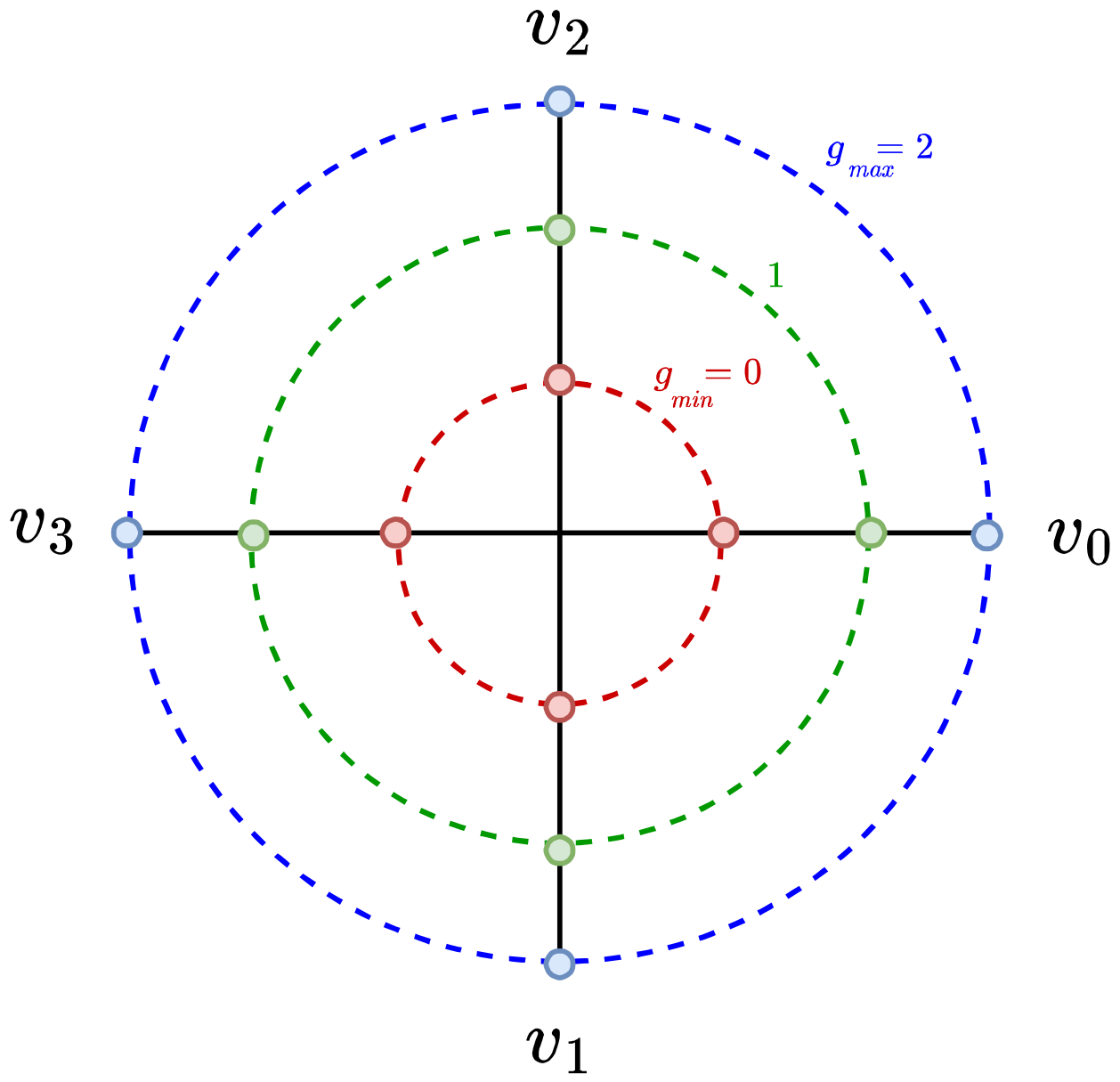
Properties of Graded Consensus

- **Termination:** All correct processes decide.
- **Unanimity:** If only v is proposed, then only (v, g_{max}) can be decided.
- **Consistency (~~Agreement~~):** Assume two correct processes decide (v, g) and (v', g'). We have **either** (i) g=g'=0 **or** (ii) …

Properties of Graded Consensus

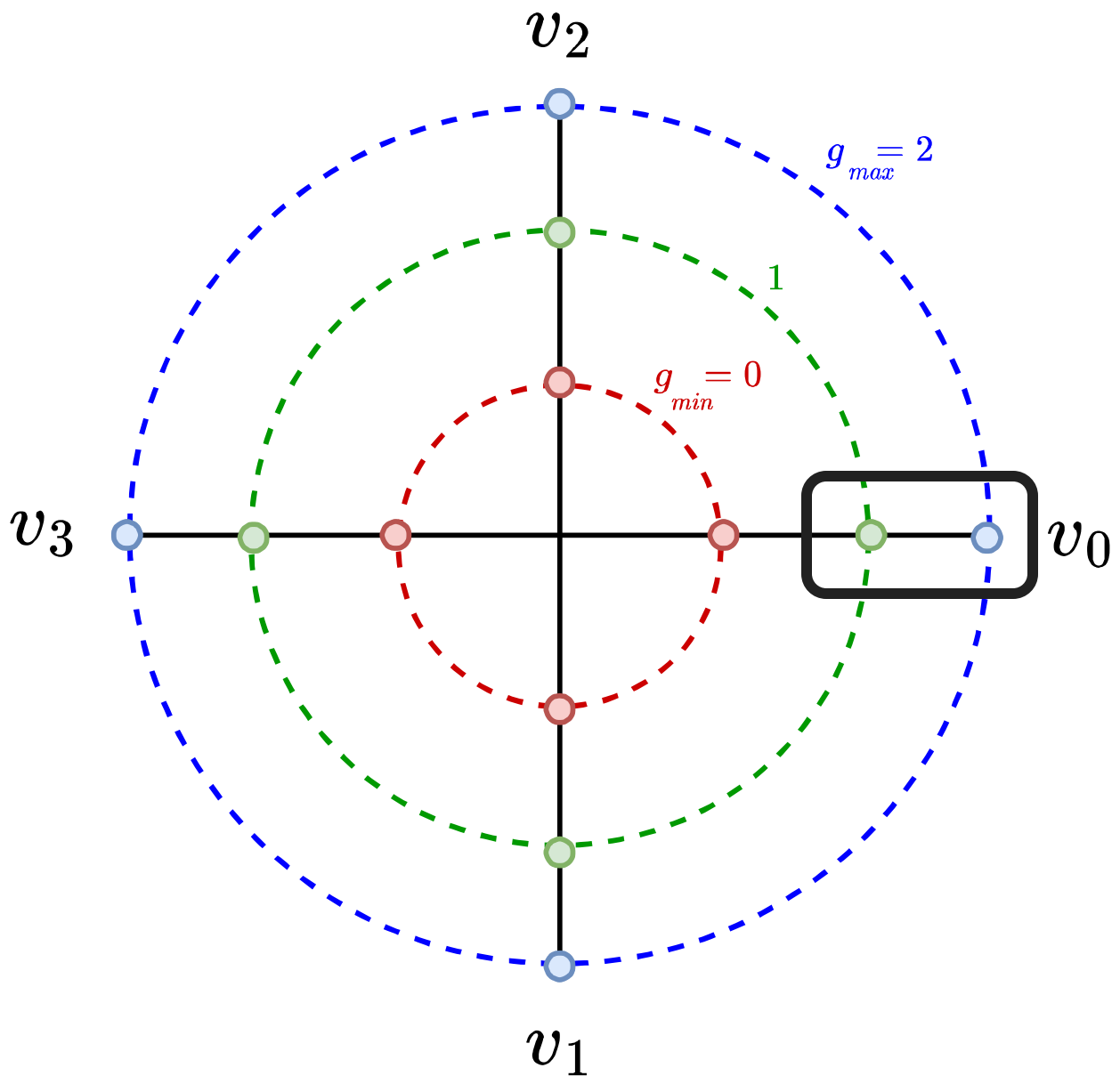
- **Termination:** All correct processes decide.
- **Unanimity:** If only v is proposed, then only (v, g_{max}) can be decided.
- **Consistency (~~Agreement~~):** Assume two correct processes decide (v, g) and (v', g'). We have **either** (i) g=g'=0 **or** (ii) |g-g'|≤1, and v=v'.

$GC^{R=3}$



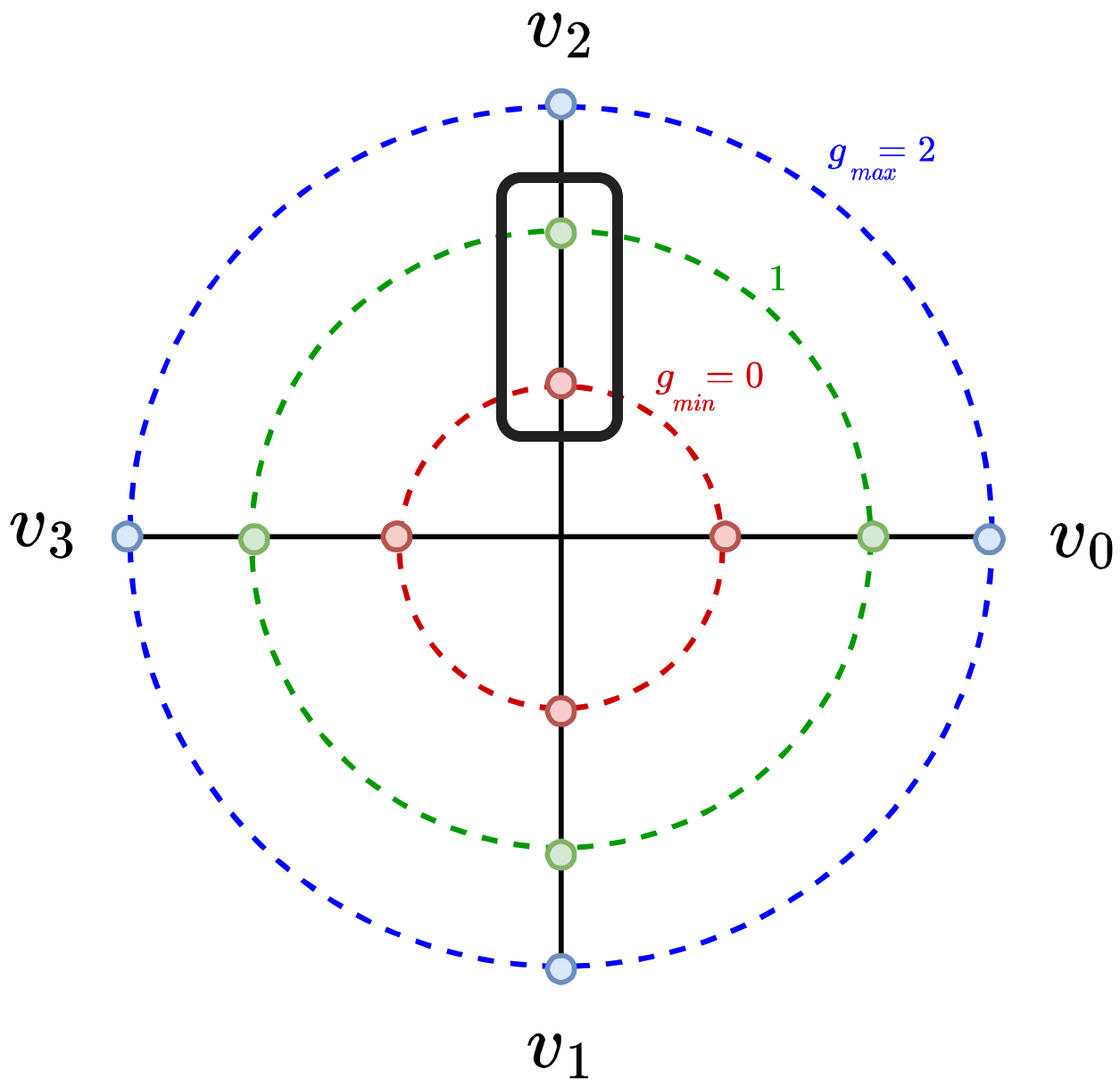
Consistency

$$GC^{R=3}$$



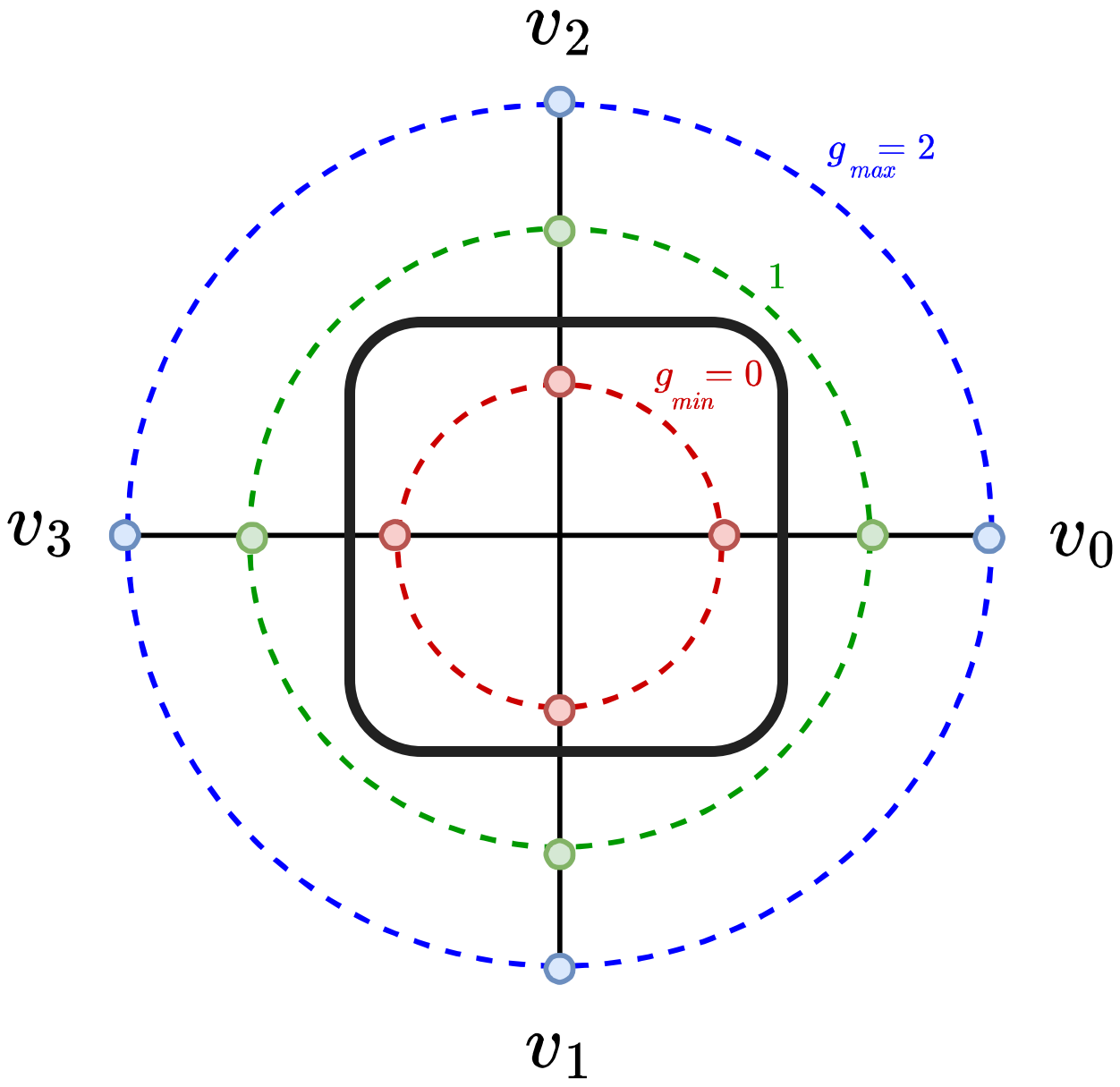
Consistency

$$GC^{R=3}$$



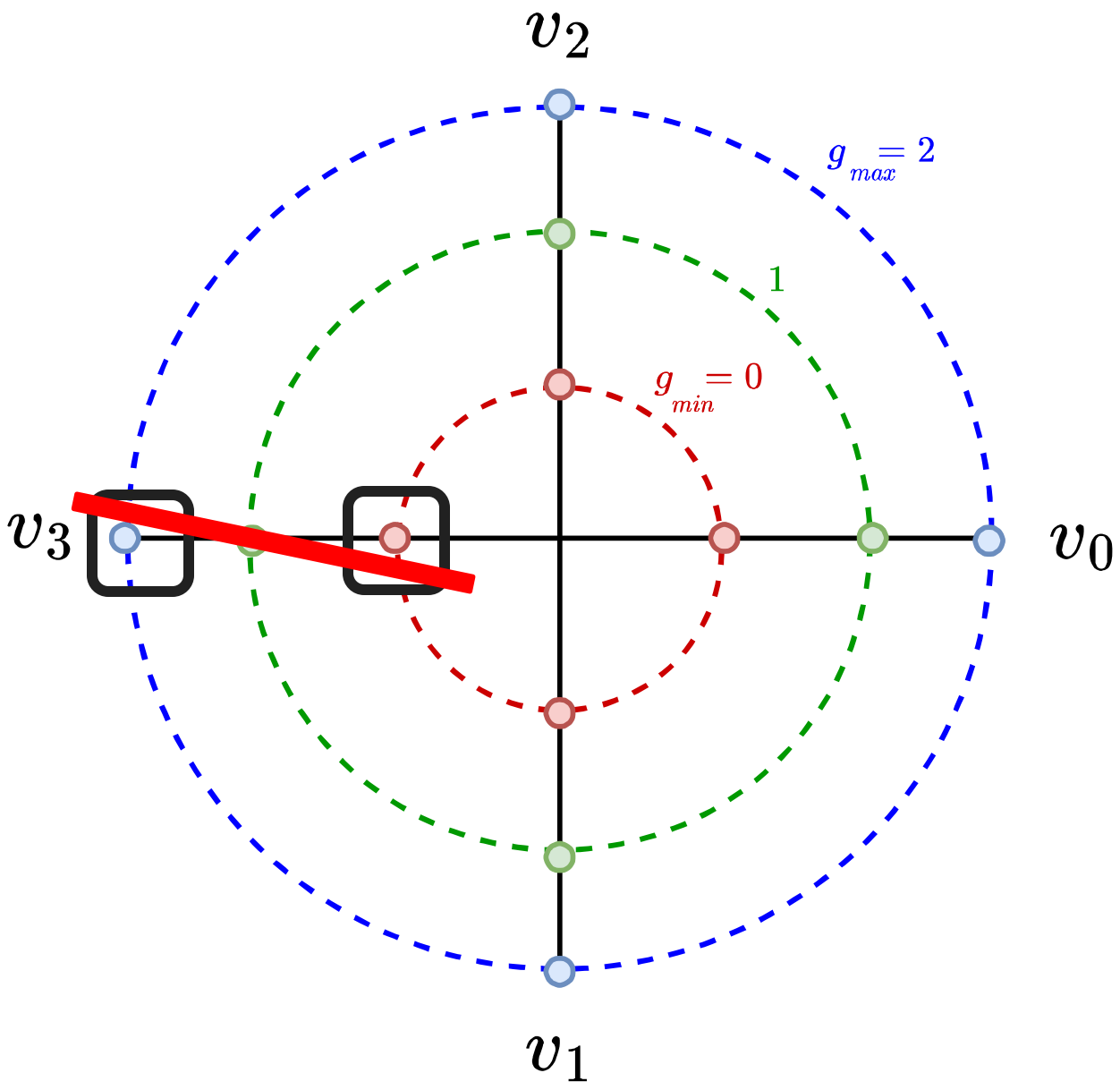
Consistency

$$GC^{R=3}$$



Consistency

$$GC^{R=3}$$



Graded Consensus

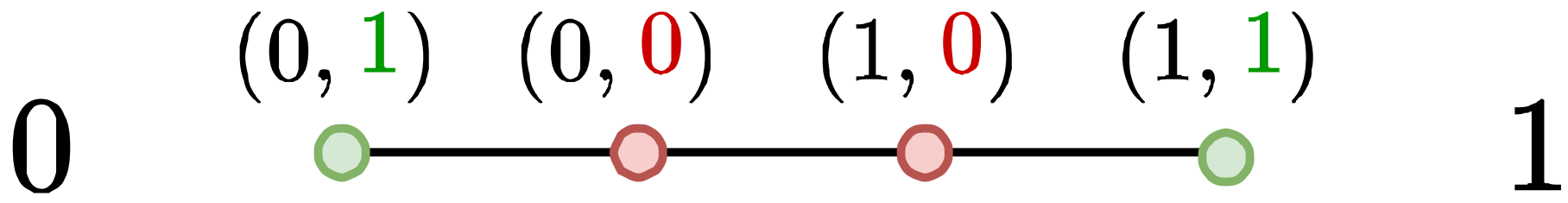
Asynchronous implementation with t<n/7 Byzantine failures

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose($v_i \in$ Binary_Value):
2: **broadcast** $\langle$PROPOSAL, $v_i\rangle$
3: **upon** $\langle$PROPOSAL, $\cdot\rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in$ Binary_Value, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide(v'', 1)
6: **else**:
7: **trigger** decide(v^*, 0), where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n - t)/2$

$BGC^{R=2}$

$g_{max} = 1$ $g_{min} = 0$ $g_{max} = 1$



Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose($v_i \in$ Binary_Value):
2: **broadcast** $\langle$PROPOSAL, $v_i\rangle$
3: **upon** $\langle$PROPOSAL, $\cdot\rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in$ Binary_Value, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide(v'', 1)
6: **else**:
7: **trigger** decide(v^*, 0), where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n - t)/2$

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose($v_i \in$ Binary_Value):
2: **broadcast** $\langle$PROPOSAL, $v_i\rangle$
3: **upon** $\langle$PROPOSAL, $\cdot\rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in$ Binary_Value, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide(v'', 1)
6: **else**:
7: **trigger** decide(v^*, 0), where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n - t)/2$

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

PROOF.

- **Termination:** Every correct process eventually triggers a proposal. Thus, every correct process eventually broadcasts a PROPOSAL message and receives $(n - t)$ PROPOSAL messages, before triggering a decision.

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose($v_i \in$ Binary_Value):
2: **broadcast** $\langle$PROPOSAL, $v_i\rangle$
3: **upon** $\langle$PROPOSAL, $\cdot\rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in$ Binary_Value, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide(v'', 1)
6: **else**:
7: **trigger** decide(v^*, 0), where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n - t)/2$

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

PROOF.

- **Termination:** Every correct process eventually triggers a proposal. Thus, every correct process eventually broadcasts a PROPOSAL message and receives $(n - t)$ PROPOSAL messages, before triggering a decision.

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose($v_i \in$ Binary_Value):
2: **broadcast** $\langle$PROPOSAL, $v_i\rangle$
3: **upon** $\langle$PROPOSAL, $\cdot\rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in$ Binary_Value, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide(v'', 1)
6: **else**:
7: **trigger** decide(v^*, 0), where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n - t)/2$

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

PROOF.

- **Termination:** Every correct process eventually triggers a proposal. Thus, every correct process eventually broadcasts a PROPOSAL message and receives $(n - t)$ PROPOSAL messages, before triggering a decision.

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose($v_i \in$ Binary_Value):
2: **broadcast** $\langle$PROPOSAL, $v_i\rangle$
3: **upon** $\langle$PROPOSAL, $\cdot\rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in$ Binary_Value, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide(v'', 1)
6: **else**:
7: **trigger** decide(v^*, 0), where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n - t)/2$

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

PROOF.

- **Termination:** Every correct process eventually triggers a proposal. Thus, every correct process eventually broadcasts a PROPOSAL message and receives $(n - t)$ PROPOSAL messages, before triggering a decision.

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

```
1: upon propose(v_i ∈ Binary_Value):
2:     broadcast ⟨PROPOSAL, v_i⟩
3: upon ⟨PROPOSAL, ·⟩ is received from n − t processes:
4:     if ∃v'' ∈ Binary_Value, s.t. at least n − 2t PROPOSAL messages contain value v'':
5:         trigger decide(v'', 1)
6:     else:
7:         trigger decide(v*, 0), where v* denotes the value with the highest frequency among the PROPOSAL messages.   ▷ #(v*) > (n − t)/2
```

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

PROOF.

- **Termination:** Every correct process eventually triggers a proposal. Thus, every correct process eventually broadcasts a PROPOSAL message and receives $(n - t)$ PROPOSAL messages, before triggering a decision.
- **Unanimity Property:** Suppose all correct processes propose the same value v. Then, each correct process broadcasts a PROPOSAL message with value v. Consequently, each process eventually receives $(n - t)$ PROPOSAL messages, including at least $(n - 2t)$ with value v and at most t with value $1 - v$. Given that $n > 3t$, we have $(n - 2t) > t$, and consequently decides on $(v, 1)$.

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

```
1: upon propose(v_i ∈ Binary_Value):
2:     broadcast ⟨PROPOSAL, v_i⟩
3: upon ⟨PROPOSAL, ·⟩ is received from n − t processes:
4:     if ∃v'' ∈ Binary_Value, s.t. at least n − 2t PROPOSAL messages contain value v'':
5:         trigger decide(v'', 1)
6:     else:
7:         trigger decide(v*, 0), where v* denotes the value with the highest frequency among the PROPOSAL messages.   ▷ #(v*) > (n − t)/2
```

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

PROOF.

- **Termination:** Every correct process eventually triggers a proposal. Thus, every correct process eventually broadcasts a PROPOSAL message and receives $(n - t)$ PROPOSAL messages, before triggering a decision.
- **Unanimity Property:** Suppose all correct processes propose the same value v. Then, each correct process broadcasts a PROPOSAL message with value v. Consequently, each process eventually receives $(n - t)$ PROPOSAL messages, including at least $(n - 2t)$ with value v and at most t with value $1 - v$. Given that $n > 3t$, we have $(n - 2t) > t$, and consequently decides on $(v, 1)$.

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose($v_i \in$ Binary_Value):
2: **broadcast** $\langle$PROPOSAL, $v_i\rangle$
3: **upon** $\langle$PROPOSAL, $\cdot\rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in$ Binary_Value, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide(v'', 1)
6: **else**:
7: **trigger** decide(v^*, 0), where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n-t)/2$

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

PROOF.

- **Termination:** Every correct process eventually triggers a proposal. Thus, every correct process eventually broadcasts a PROPOSAL message and receives $(n - t)$ PROPOSAL messages, before triggering a decision.
- **Unanimity Property:** Suppose all correct processes propose the same value v. Then, each correct process broadcasts a PROPOSAL message with value v. Consequently, each process eventually receives $(n - t)$ PROPOSAL messages, including at least $(n - 2t)$ with value v and at most t with value $1 - v$. Given that $n > 3t$, we have $(n - 2t) > t$, and consequently decides on $(v, 1)$.

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose($v_i \in$ Binary_Value):
2: **broadcast** $\langle$PROPOSAL, $v_i\rangle$
3: **upon** $\langle$PROPOSAL, $\cdot\rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in$ Binary_Value, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide(v'', 1)
6: **else**:
7: **trigger** decide(v^*, 0), where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n-t)/2$

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

PROOF.

- **Termination:** Every correct process eventually triggers a proposal. Thus, every correct process eventually broadcasts a PROPOSAL message and receives $(n - t)$ PROPOSAL messages, before triggering a decision.
- **Unanimity Property:** Suppose all correct processes propose the same value v. Then, each correct process broadcasts a PROPOSAL message with value v. Consequently, each process eventually receives $(n - t)$ PROPOSAL messages, including at least $(n - 2t)$ with value v and at most t with value $1 - v$. Given that $n > 3t$, we have $(n - 2t) > t$, and consequently decides on $(v, 1)$.

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose($v_i \in$ Binary_Value):
2: **broadcast** $\langle$PROPOSAL, $v_i\rangle$
3: **upon** $\langle$PROPOSAL, $\cdot\rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in$ Binary_Value, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide(v'', 1)
6: **else**:
7: **trigger** decide(v^*, 0), where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n-t)/2$

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

PROOF.

- **Consistency:** Assume a correct process p_i decides $(w, 1)$ (otherwise, consistency is immediate). Process p_i must have received PROPOSAL messages with value w from a set Q_i of $|Q_i| = n - 2t$ distinct processes. Let another correct process, p_j, decide on some value $(w', \cdot)$. We aim to show that $w' = w$.
  Process p_j's decision was based on receiving PROPOSAL messages from a set Q_j with $|Q_j| = n - t$ processes. The overlap between Q_i and Q_j is $|Q_i \cap Q_j| = |Q_i| + |Q_j| - |Q_i \cup Q_j| \geq (n - 2t) + (n - t) - n = n - 3t$, so $|Q_i \cap Q_j \cap \text{Corrects}| \geq n - 4t$. Therefore, process p_j receives at least $n-4t$ PROPOSAL messages with value w, which ensures that $w' = w$ if $n-4t > (n-t)/2$, i.e., if $n > 7t$.

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose($v_i \in$ Binary_Value):
2: **broadcast** $\langle$PROPOSAL, $v_i\rangle$
3: **upon** $\langle$PROPOSAL, $\cdot\rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in$ Binary_Value, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide(v'', 1)
6: **else**:
7: **trigger** decide(v^*, 0), where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n-t)/2$

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

PROOF.

- **Consistency:** Assume a correct process p_i decides $(w, 1)$ (otherwise, consistency is immediate). Process p_i must have received PROPOSAL messages with value w from a set Q_i of $|Q_i| = n - 2t$ distinct processes. Let another correct process, p_j, decide on some value $(w', \cdot)$. We aim to show that $w' = w$.

  Process p_j's decision was based on receiving PROPOSAL messages from a set Q_j with $|Q_j| = n - t$ processes. The overlap between Q_i and Q_j is $|Q_i \cap Q_j| = |Q_i| + |Q_j| - |Q_i \cup Q_j| \geq (n - 2t) + (n - t) - n = n - 3t$, so $|Q_i \cap Q_j \cap \text{Corrects}| \geq n - 4t$. Therefore, process p_j receives at least $n-4t$ PROPOSAL messages with value w, which ensures that $w' = w$ if $n-4t > (n-t)/2$, i.e., if $n > 7t$.

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose($v_i \in$ Binary_Value):
2: **broadcast** $\langle$PROPOSAL, $v_i\rangle$
3: **upon** $\langle$PROPOSAL, $\cdot\rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in$ Binary_Value, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide$(v'', 1)$
6: **else**:
7: **trigger** decide$(v^*, 0)$, where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n-t)/2$

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

PROOF.

- **Consistency:** Assume a correct process p_i decides $(w, 1)$ (otherwise, consistency is immediate). Process p_i must have received PROPOSAL messages with value w from a set Q_i of $|Q_i| = n - 2t$ distinct processes. Let another correct process, p_j, decide on some value $(w', \cdot)$. We aim to show that $w' = w$.
  Process p_j's decision was based on receiving PROPOSAL messages from a set Q_j with $|Q_j| = n - t$ processes. The overlap between Q_i and Q_j is $|Q_i \cap Q_j| = |Q_i| + |Q_j| - |Q_i \cup Q_j| \geq (n - 2t) + (n - t) - n = n - 3t$, so $|Q_i \cap Q_j \cap \text{Corrects}| \geq n - 4t$. Therefore, process p_j receives at least $n-4t$ PROPOSAL messages with value w, which ensures that $w' = w$ if $n-4t > (n-t)/2$, i.e., if $n > 7t$.

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose($v_i \in$ Binary_Value):
2: **broadcast** $\langle$PROPOSAL, $v_i\rangle$
3: **upon** $\langle$PROPOSAL, $\cdot\rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in$ Binary_Value, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide(v'', 1)
6: **else**:
7: **trigger** decide(v^*, 0), where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n-t)/2$

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

PROOF.

- **Consistency:** Assume a correct process p_i decides $(w, 1)$ (otherwise, consistency is immediate). Process p_i must have received PROPOSAL messages with value w from a set Q_i of $|Q_i| = n - 2t$ distinct processes. Let another correct process, p_j, decide on some value $(w', \cdot)$. We aim to show that $w' = w$.

  Process p_j's decision was based on receiving PROPOSAL messages from a set Q_j with $|Q_j| = n - t$ processes. The overlap between Q_i and Q_j is $|Q_i \cap Q_j| = |Q_i| + |Q_j| - |Q_i \cup Q_j| \geq (n - 2t) + (n - t) - n = n - 3t$, so $|Q_i \cap Q_j \cap \text{Corrects}| \geq n - 4t$. Therefore, process p_j receives at least $n-4t$ PROPOSAL messages with value w, which ensures that $w' = w$ if $n-4t > (n-t)/2$, i.e., if $n > 7t$.

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose$(v_i \in \text{Binary_Value})$:
2: **broadcast** $\langle \text{PROPOSAL}, v_i \rangle$
3: **upon** $\langle \text{PROPOSAL}, \cdot \rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in \text{Binary\_Value}$, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide$(v'', 1)$
6: **else**:
7: **trigger** decide$(v^*, 0)$, where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n - t)/2$

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

PROOF.

- **Consistency:** Assume a correct process p_i decides $(w, 1)$ (otherwise, consistency is immediate). Process p_i must have received PROPOSAL messages with value w from a set Q_i of $|Q_i| = n - 2t$ distinct processes. Let another correct process, p_j, decide on some value $(w', \cdot)$. We aim to show that $w' = w$.

  Process p_j's decision was based on receiving PROPOSAL messages from a set Q_j with $|Q_j| = n - t$ processes. The overlap between Q_i and Q_j is $|Q_i \cap Q_j| = |Q_i| + |Q_j| - |Q_i \cup Q_j| \geq (n - 2t) + (n - t) - n = n - 3t$, so $|Q_i \cap Q_j \cap \text{Corrects}| \geq n - 4t$. Therefore, process p_j receives at least $n - 4t$ PROPOSAL messages with value w, which ensures that $w' = w$ if $n - 4t > (n - t)/2$, i.e., if $n > 7t$.

Algorithm 1 Asynchronous Binary Graded Consensus with refinement $R = 2$, and $t < n/7$: Pseudocode (for process p_i)

1: **upon** propose($v_i \in$ Binary_Value):
2: **broadcast** $\langle$PROPOSAL, $v_i\rangle$
3: **upon** $\langle$PROPOSAL, $\cdot\rangle$ is received from $n - t$ processes:
4: **if** $\exists v'' \in$ Binary_Value, s.t. at least $n - 2t$ PROPOSAL messages contain value v'':
5: **trigger** decide(v'', 1)
6: **else**:
7: **trigger** decide(v^*, 0), where v^* denotes the value with the highest frequency among the PROPOSAL messages. ▷ $\#(v^*) > (n - t)/2$

LEMMA 2.1. *Algorithm 1 implements graded consensus with the refinement parameter $R = 2$ and $n/7$-resiliency.*

$$Alg_1 \sqsubseteq BGC^{R=2}$$

2025

$O(nL + n^2 \log(n))$

$n > 3t$

ABEL: Perfect Asynchronous Byzantine Extension from List-Decoding*

Ittai Abraham
Intel Labs
ittai.abraham@intel.com

Gilad Asharov†
Bar-Ilan University
Gilad.Asharov@biu.ac.il

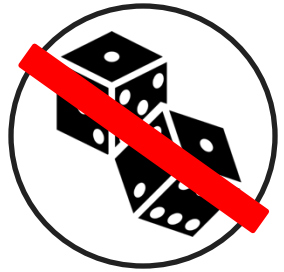
August 17, 2025

Abstract

Asynchronous byzantine agreement extension studies the message complexity of L-bit multivalued asynchronous byzantine agreement given access to a binary asynchronous Byzantine agreement protocol.

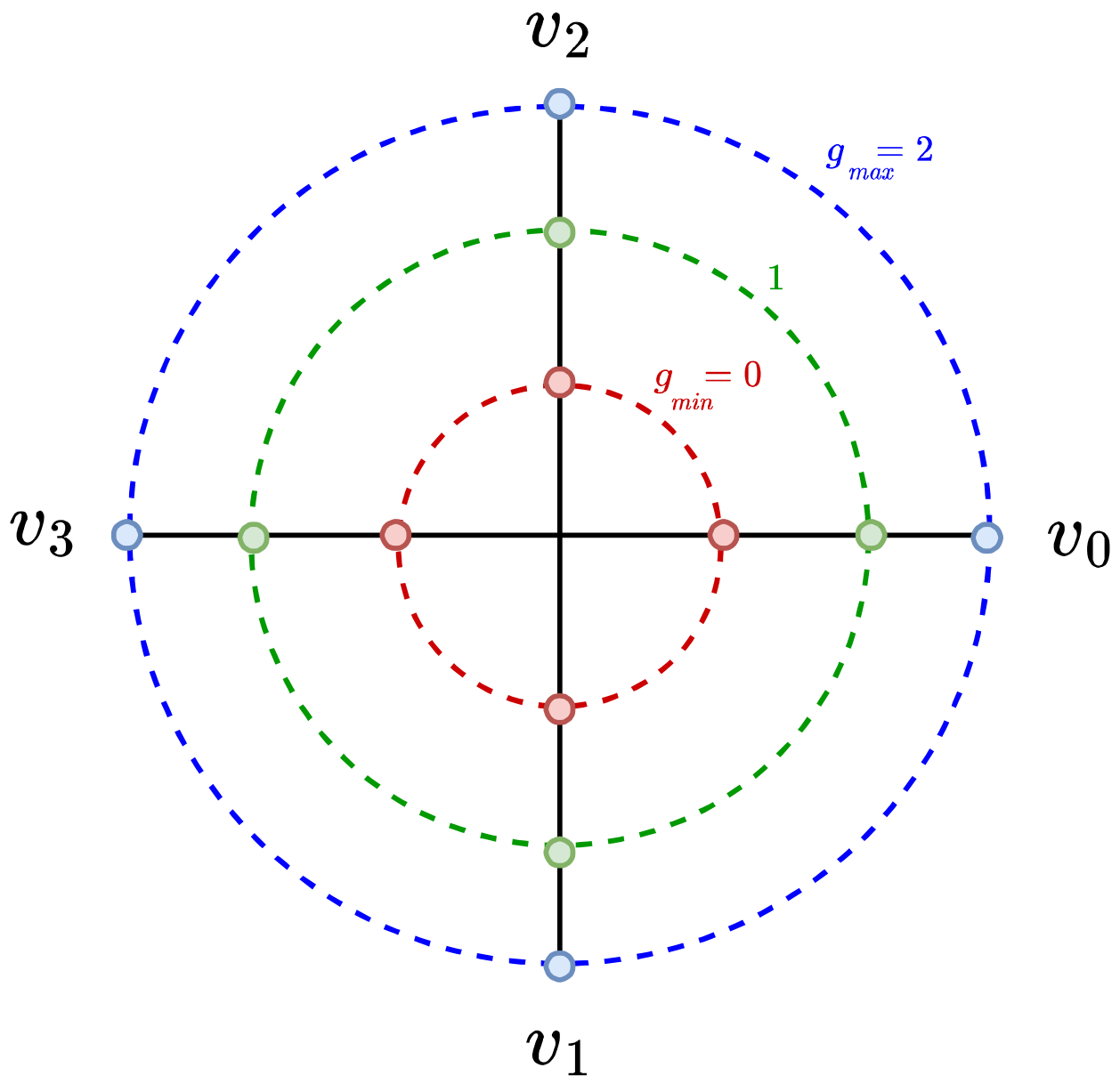
We prove that asynchronous byzantine agreement extension can be solved with perfect security and optimal resilience in $O(nL + n^2 \log n)$ total communication (in bits) in addition to a single call to a binary asynchronous Byzantine agreement protocol. For $L = O(n \log n)$, this gives an asymptotically optimal protocol, resolving a question that remained open for nearly two decades.

List decoding is a fundamental concept in theoretical computer science and cryptography, enabling error correction beyond the unique decoding radius and playing a critical role in constructing robust codes, hardness amplification, and secure cryptographic protocols. A key novelty of our perfectly secure and optimally resilient asynchronous byzantine agreement extension protocol is that it uses list decoding - making a striking new connection between list decoding and asynchronous Byzantine agreement.



One more refinement

$GC^{R=3}$



Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1$, $\mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1$, $\mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade

LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter $R = 3$.*

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1$, $\mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade

LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter* $R = 3$.

PROOF.

- Termination follows directly from the termination of $\mathcal{GC}_1$ and $\mathcal{GC}_2$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

Uses:
  Binary Graded Consensus, **instances** $\mathcal{GC}_1$, $\mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
Local Variables:
  Integer $g_i \leftarrow 0$ ▷ Grade
upon propose$(v_i \in \text{Value})$:
  invoke $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
  upon $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
    $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
    invoke $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
  upon $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
    $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
    trigger decide(v_i^2, g_i) ▷ Decide the final value and grade

LEMMA 1.2. *Algorithm 4 implements graded consensus with the refinement parameter $R = 3$.*

PROOF.

- Termination follows directly from the termination of $\mathcal{GC}_1$ and $\mathcal{GC}_2$.
- Unanimity property follows directly from the unanimity property of $\mathcal{GC}_1$ and $\mathcal{GC}_2$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

Uses:
  Binary Graded Consensus, **instances** $\mathcal{GC}_1$, $\mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
Local Variables:
  Integer $g_i \leftarrow 0$ ▷ Grade

upon propose$(v_i \in \text{Value})$:
  invoke $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
  upon $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
    $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
    invoke $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
  upon $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
    $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
    trigger decide(v_i^2, g_i) ▷ Decide the final value and grade

LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter* $R = 3$.

PROOF.

- Termination follows directly from the termination of $\mathcal{GC}_1$ and $\mathcal{GC}_2$.
- Unanimity property follows directly from the unanimity property of $\mathcal{GC}_1$ and $\mathcal{GC}_2$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade

LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter* $R = 3$.

PROOF.

- Termination follows directly from the termination of $\mathcal{GC}_1$ and $\mathcal{GC}_2$.
- Unanimity property follows directly from the unanimity property of $\mathcal{GC}_1$ and $\mathcal{GC}_2$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade

LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter* $R = 3$.

PROOF.

- Termination follows directly from the termination of $\mathcal{GC}_1$ and $\mathcal{GC}_2$.
- Unanimity property follows directly from the unanimity property of $\mathcal{GC}_1$ and $\mathcal{GC}_2$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

Uses:
  Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
Local Variables:
  Integer $g_i \leftarrow 0$ ▷ Grade
upon propose$(v_i \in \text{Value})$:
  invoke $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
  upon $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
    $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
    invoke $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
  upon $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
    $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
    trigger decide(v_i^2, g_i) ▷ Decide the final value and grade

LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter* $R = 3$.

PROOF.

- Termination follows directly from the termination of $\mathcal{GC}_1$ and $\mathcal{GC}_2$.
- Unanimity property follows directly from the unanimity property of $\mathcal{GC}_1$ and $\mathcal{GC}_2$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

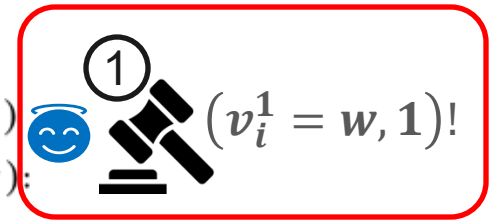
1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade

LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter $R = 3$.*

- To prove consistency, let j be the first instance of graded consensus where some process outputs $(\cdot, 1)$ from $\mathcal{GC}_j$. If no such j exists, the result is immediate.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade



LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter $R = 3$.*

- To prove consistency, let j be the first instance of graded consensus where some process outputs $(\cdot, 1)$ from $\mathcal{GC}_j$. If no such j exists, the result is immediate.

Case 1: $j = 1$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i)  $(v_i^1 = w, 1)!$  $(v_j^1 = w, *)!$ ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade

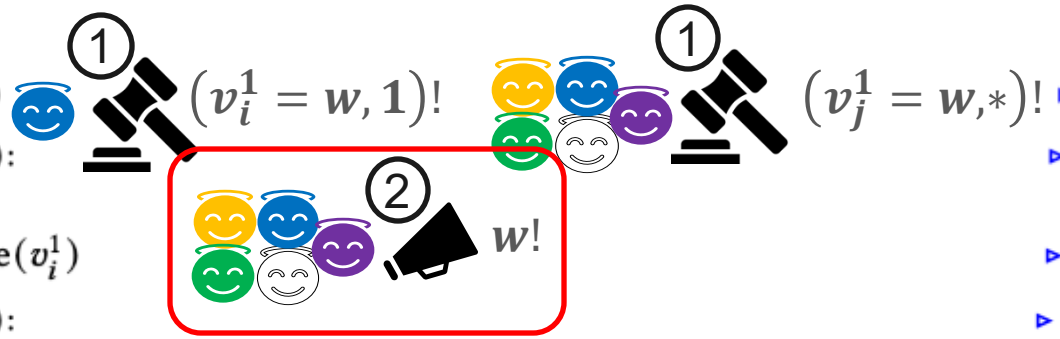
LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter $R = 3$.*

- To prove consistency, let j be the first instance of graded consensus where some process outputs $(\cdot, 1)$ from $\mathcal{GC}_j$. If no such j exists, the result is immediate.

Case 1: $j = 1$. By consistency, each correct process outputs $(w, \cdot)$ from $\mathcal{GC}_1$ for some value w, and thus proposes w to $\mathcal{GC}_2$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade



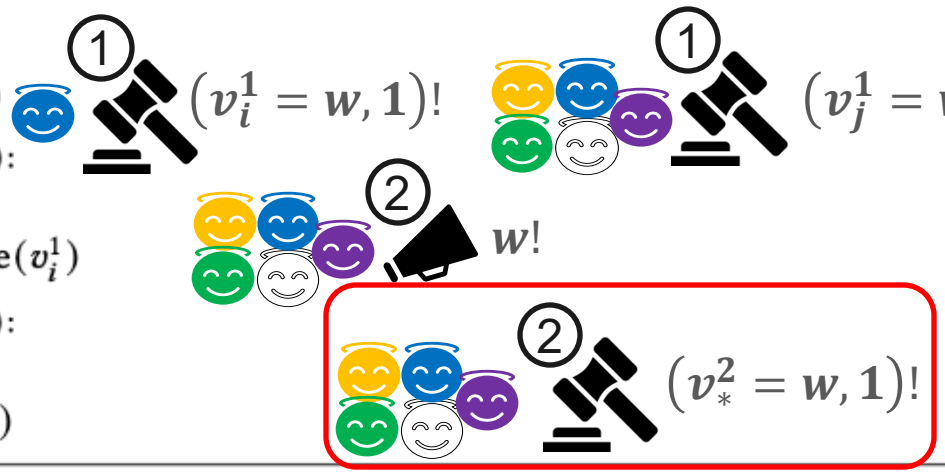
LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter $R = 3$.*

- To prove consistency, let j be the first instance of graded consensus where some process outputs $(\cdot, 1)$ from $\mathcal{GC}_j$. If no such j exists, the result is immediate.

Case 1: $j = 1$. By consistency, each correct process outputs $(w, \cdot)$ from $\mathcal{GC}_1$ for some value w, and thus proposes w to $\mathcal{GC}_2$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade



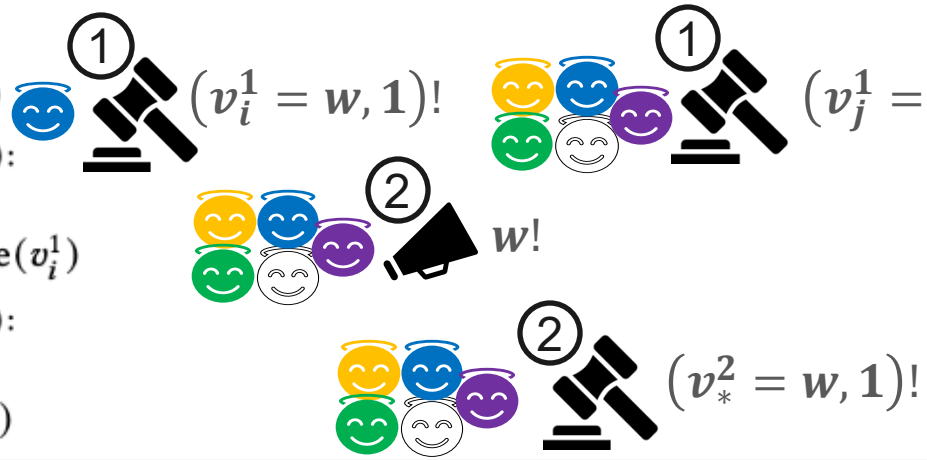
LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter $R = 3$.*

- To prove consistency, let j be the first instance of graded consensus where some process outputs $(\cdot, 1)$ from $\mathcal{GC}_j$. If no such j exists, the result is immediate.

Case 1: $j = 1$. By consistency, each correct process outputs $(w, \cdot)$ from $\mathcal{GC}_1$ for some value w, and thus proposes w to $\mathcal{GC}_2$. Therefore, due to the unanimity property of graded consensus $\mathcal{GC}_2$, every correct process returns $(w, 1)$ from $\mathcal{GC}_2$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade



LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter $R = 3$.*

- To prove consistency, let j be the first instance of graded consensus where some process outputs $(\cdot, 1)$ from $\mathcal{GC}_j$. If no such j exists, the result is immediate.

Case 1: $j = 1$. By consistency, each correct process outputs $(w, \cdot)$ from $\mathcal{GC}_1$ for some value w, and thus proposes w to $\mathcal{GC}_2$. Therefore, due to the unanimity property of graded consensus $\mathcal{GC}_2$, every correct process returns $(w, 1)$ from $\mathcal{GC}_2$. Hence, consistency follows directly from the consistency of $\mathcal{GC}_1$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

Uses:
  Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
Local Variables:
  Integer $g_i \leftarrow 0$ ▷ Grade
upon propose$(v_i \in \text{Value})$:
  invoke $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
  upon $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
    $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
    invoke $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
  upon $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
    $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
    trigger decide(v_i^2, g_i) ▷ Decide the final value and grade

LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter $R = 3$.*

- To prove consistency, let j be the first instance of graded consensus where some process outputs $(\cdot, 1)$ from $\mathcal{GC}_j$. If no such j exists, the result is immediate.

Case 2: $j = 2$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade

LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter* $R = 3$.

- To prove consistency, let j be the first instance of graded consensus where some process outputs $(\cdot, 1)$ from $\mathcal{GC}_j$. If no such j exists, the result is immediate.

Case 2: $j = 2$. By construction, each correct process outputs $(\cdot, 0)$ from $\mathcal{GC}_1$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade

LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter* $R = 3$.

- To prove consistency, let j be the first instance of graded consensus where some process outputs $(\cdot, 1)$ from $\mathcal{GC}_j$. If no such j exists, the result is immediate.

Case 2: $j = 2$. By construction, each correct process outputs $(\cdot, 0)$ from $\mathcal{GC}_1$.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade

LEMMA 1.2. *Algorithm 2 implements graded consensus with the refinement parameter $R = 3$.*

- To prove consistency, let j be the first instance of graded consensus where some process outputs $(\cdot, 1)$ from $\mathcal{GC}_j$. If no such j exists, the result is immediate.

Case 2: $j = 2$. By construction, each correct process outputs $(\cdot, 0)$ from $\mathcal{GC}_1$. Therefore, due to the consistency property of graded consensus $\mathcal{GC}_2$, if two correct processes p_i and p_j decide on (v_i, g_i) and (v_j, g_j), respectively, then $|g_i - g_j| \leq 1$. Moreover, if $g_i \neq 0$, $v_i = v_j$. This implies consistency.

Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

1: **Uses:**
2: Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
3: **Local Variables:**
4: Integer $g_i \leftarrow 0$ ▷ Grade
5: **upon** propose$(v_i \in \text{Value})$:
6: **invoke** $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
7: **upon** $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
8: $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
9: **invoke** $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
10: **upon** $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
11: $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
12: **trigger** decide(v_i^2, g_i) ▷ Decide the final value and grade

$$Alg_2 = GC^{R=2} \rhd GC^{R=2} \sqsubseteq GC^{R=3}$$

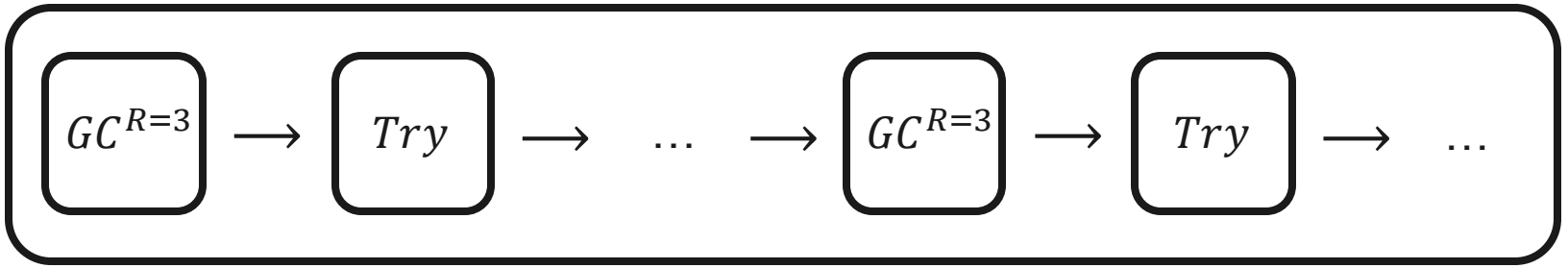
Algorithm 2 Binary Graded Consensus (BGC) with refinement $R = 3$ on top of BGC with refinement $R = 2$

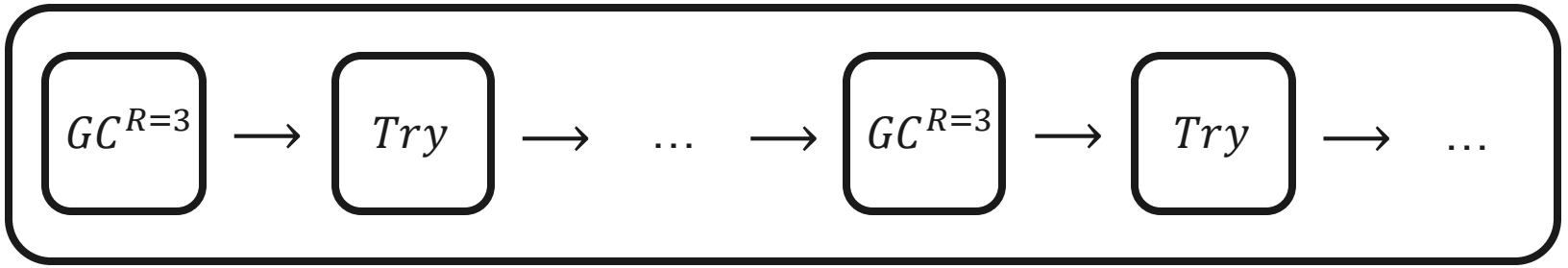
Uses:
  Binary Graded Consensus, **instances** $\mathcal{GC}_1, \mathcal{GC}_2$ ▷ 2 instances of the Binary Graded Consensus protocol with Refinement $R' = 2$
Local Variables:
  Integer $g_i \leftarrow 0$ ▷ Grade
upon propose$(v_i \in \text{Value})$:
  invoke $\mathcal{GC}_1$.propose(v_i) ▷ Propose input value to the 1st graded consensus instance
  upon $\mathcal{GC}_1$.decide(v_i^1, g_i^1): ▷ Received decision from the 1st graded consensus instance
    $g_i \leftarrow g_i + g_i^1$ ▷ Update the grade (confidence)
    invoke $\mathcal{GC}_2$.propose(v_i^1) ▷ Propose input value to the 2nd graded consensus instance
  upon $\mathcal{GC}_2$.decide(v_i^2, g_i^2): ▷ Received decision from the 2nd graded consensus instance
    $g_i \leftarrow g_i + g_i^2$ ▷ Update the grade (confidence)
    trigger decide(v_i^2, g_i) ▷ Decide the final value and grade

$$Alg_2 = GC^{R=2} \rhd GC^{R=2} \sqsubseteq GC^{R=3}$$

$$Alg_2 \sqsubseteq GC^{R=3}$$

Consensus

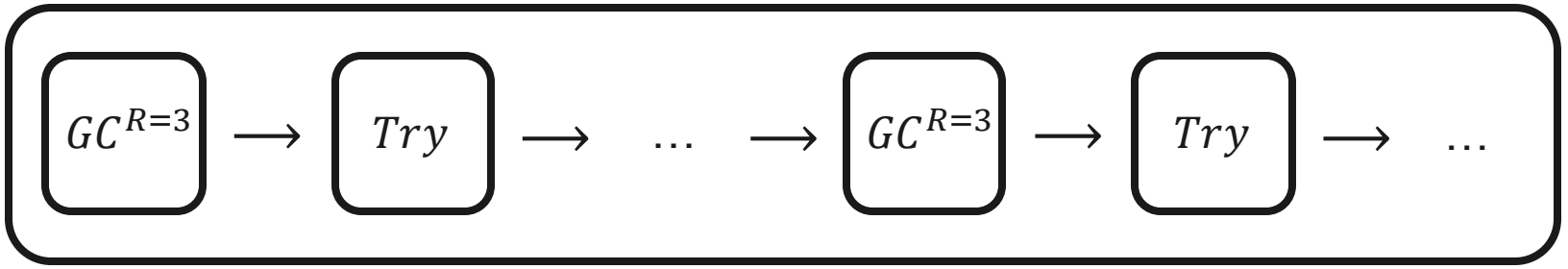

$GC^{R=3}$
Try
…
$GC^{R=3}$
Try
…



(v^k, g^k) v^k_{try} (v^{k+1}, g^{k+1}) v^{k+1}_{try}

if $g^k = g_{max}$:
decide$(v^k) \wedge$
halt after $\mathcal{I}^{k+1}$

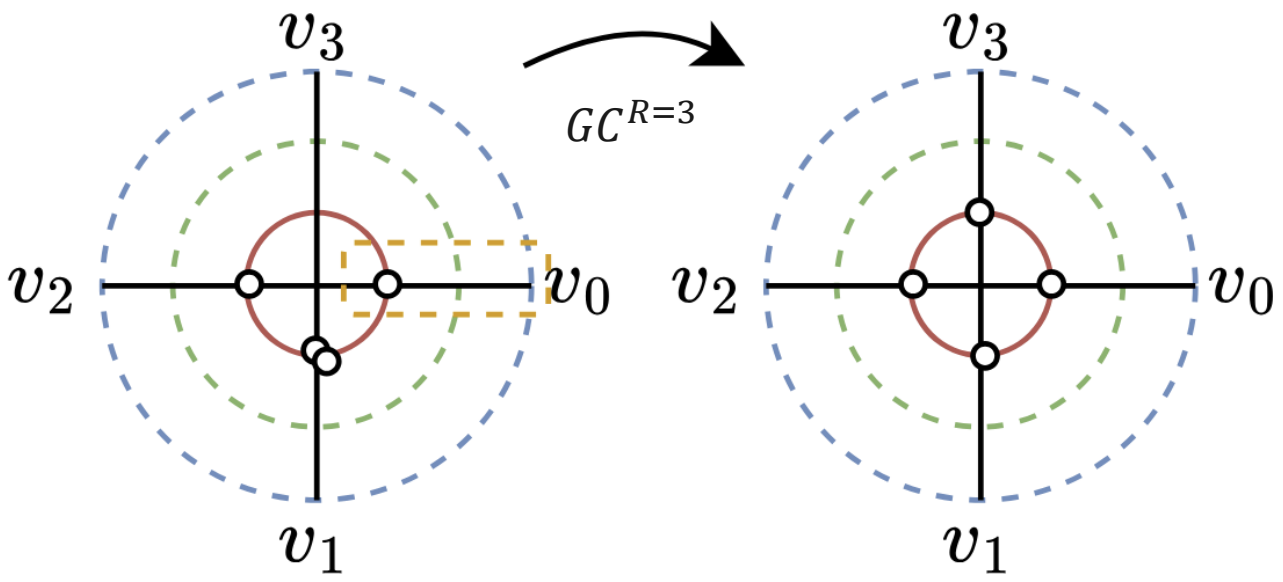
if $g^k > g_{min}$:
$v^k_{try} \leftarrow v^k$



(v^k, g^k) v^k_{try} (v^{k+1}, g^{k+1}) v^{k+1}_{try}

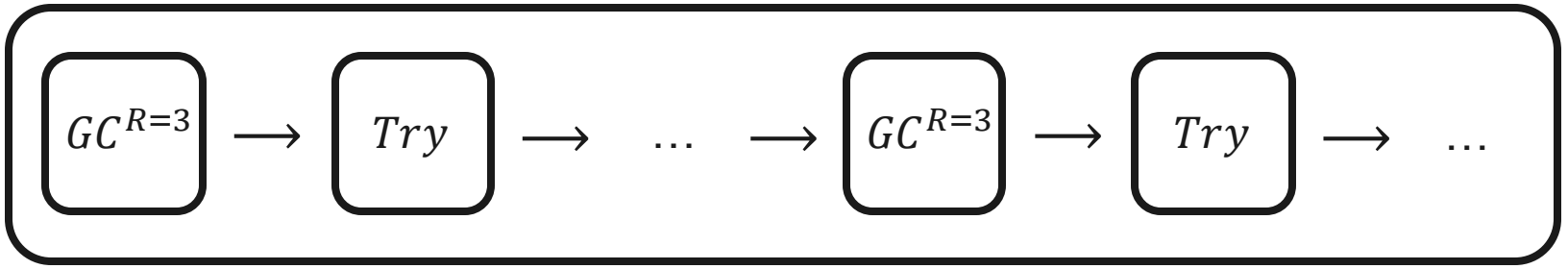
if $g^k = g_{max}$:
decide$(v^k)\wedge$
halt after $\mathcal{I}^{k+1}$

if $g^k > g_{min}$:
$v^k_{try} \leftarrow v^k$



Initially divergent (view start)

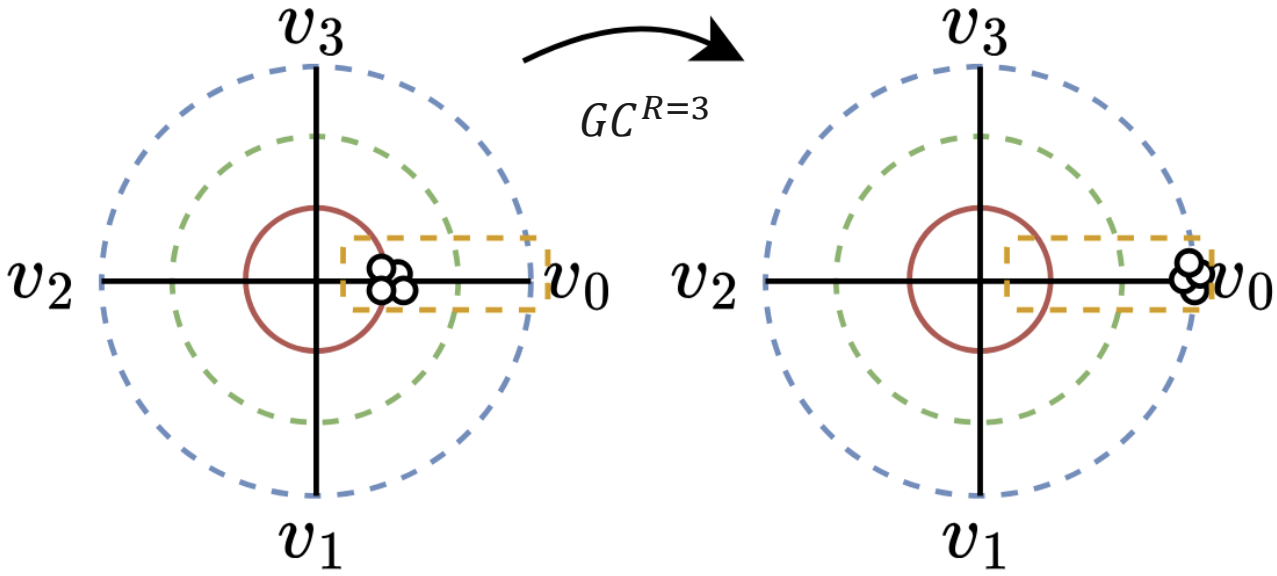
Still divergent (worst-case)



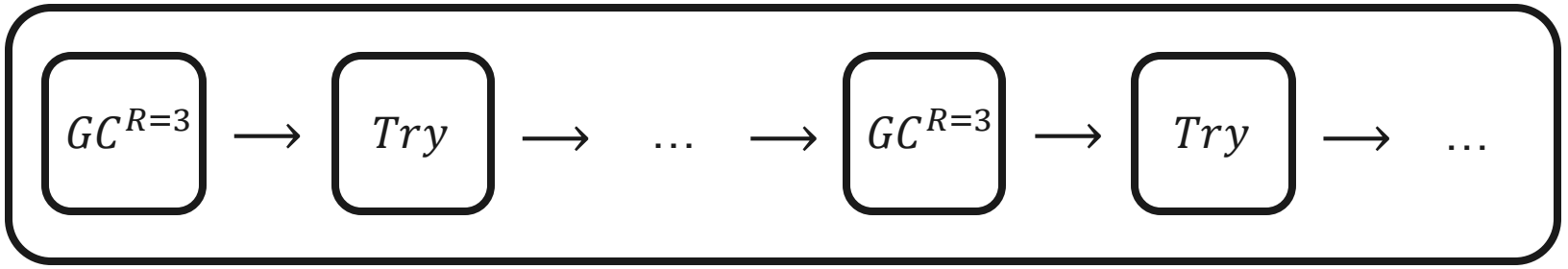
(v^k, g^k) v^k_{try} (v^{k+1}, g^{k+1}) v^{k+1}_{try}

if $g^k = g_{max}$:
decide$(v^k)\wedge$
halt after $\mathcal{I}^{k+1}$

if $g^k > g_{min}$:
$v^k_{try} \leftarrow v^k$



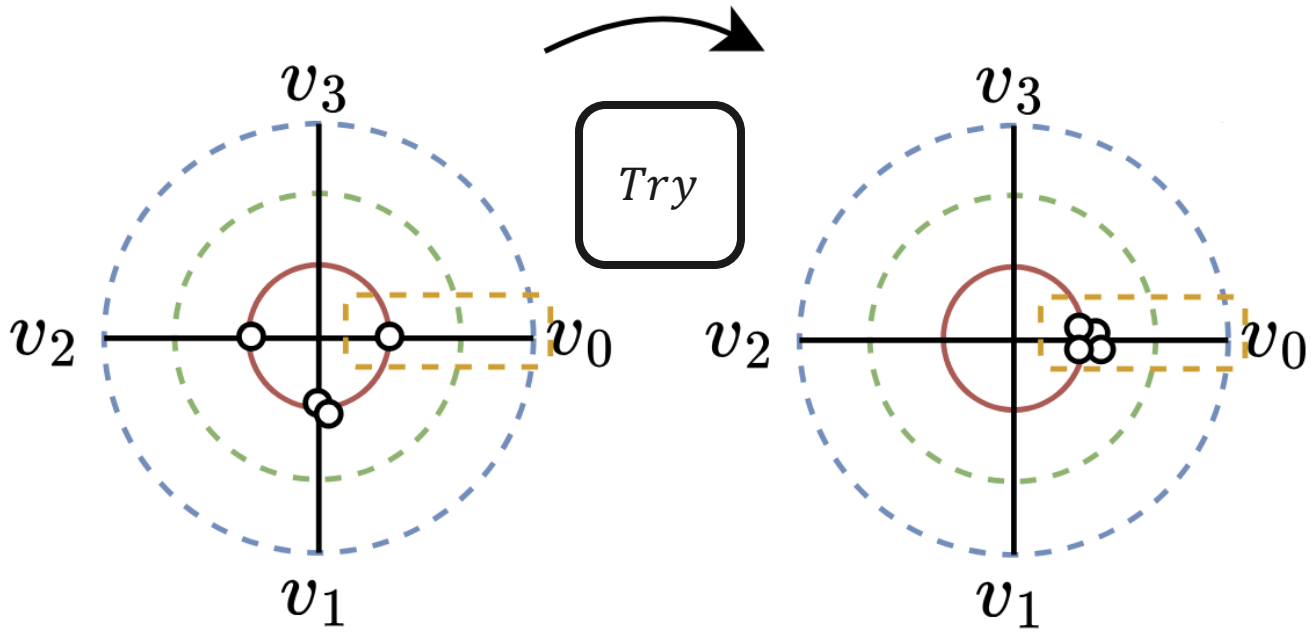
Achieves convergence All processes decide



(v^k, g^k) v^k_{try} (v^{k+1}, g^{k+1}) v^{k+1}_{try}

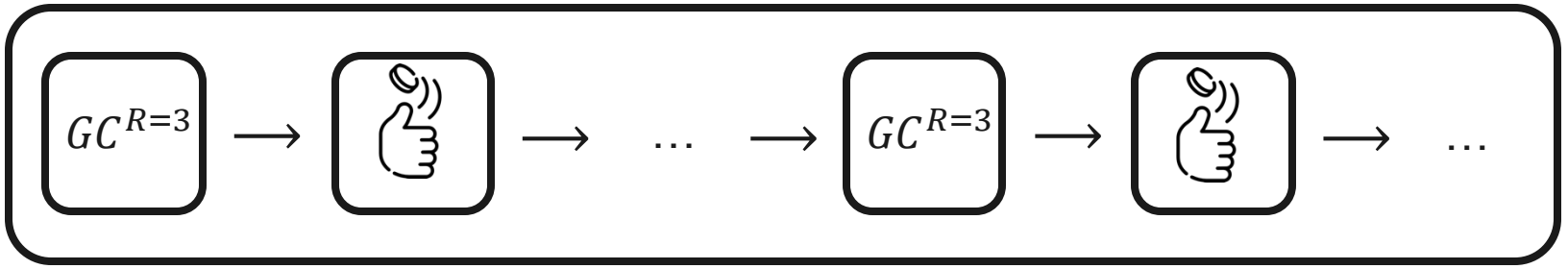
if $g^k = g_{max}$:
decide$(v^k)\wedge$
halt after $\mathcal{I}^{k+1}$

if $g^k > g_{min}$:
$v^k_{try} \leftarrow v^k$



Initially divergent
(view start)

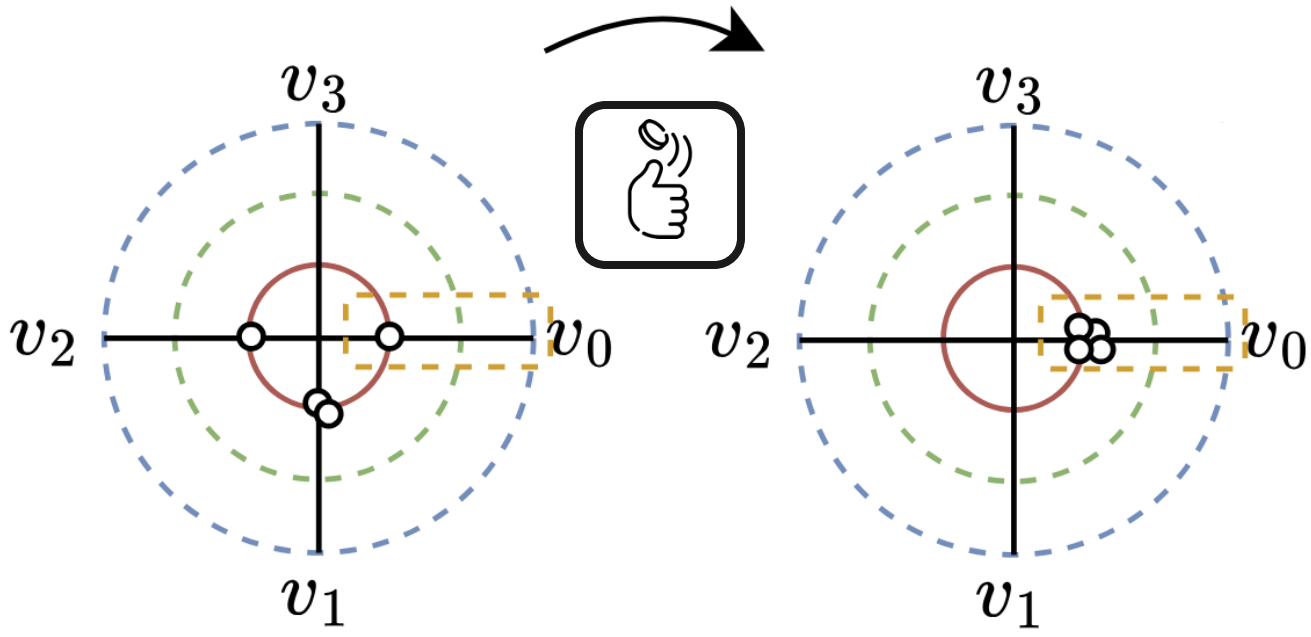
Achieves convergence



(v^k, g^k) v^k_{try} (v^{k+1}, g^{k+1}) v^{k+1}_{try}

if $g^k = g_{max}$:
decide$(v^k)\wedge$
halt after $\mathcal{I}^{k+1}$

if $g^k > g_{min}$:
$v^k_{try} \leftarrow v^k$



Initially divergent
(view start)

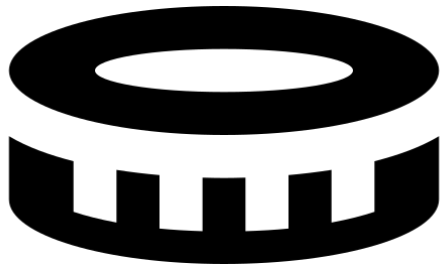
Achieves convergence

Common Coin

Interface of the common coin

Flip
Operation

Yield
Callback

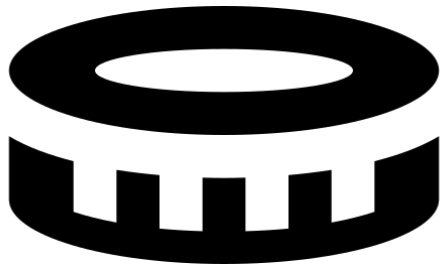


Interface of the common coin

Flip
Operation

∅

Yield
Callback

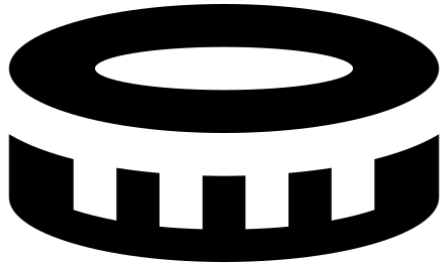


Interface of the common coin

Flip
Operation

Ø

Yield
Callback



v!

Properties of the Common Coin

- **Termination:** All correct processes eventually yield a binary value.
- **Agreement:** All correct processes agree on 0 (or 1) with probability >0.
- **Unpredictability:** As long as no correct process has triggered ‘flip’, the adversary cannot predict the output with probability greater than 1/2.

A naive implementation of the Common Coin

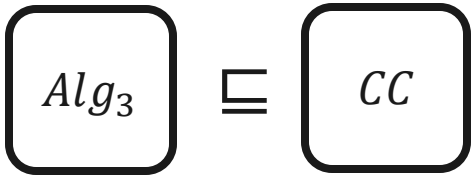
Algorithm 3 Common Coin

1: **upon** flip():
2: $b \xleftarrow{\$} \{0, 1\}$ ▷ Choose either 0 or 1 with probability 1/2
3: **trigger** yield(b)

A naive implementation of the Common Coin

Algorithm 3 Common Coin

1: **upon** flip():
2: $b \xleftarrow{\$} \{0, 1\}$ ▷ Choose either 0 or 1 with probability 1/2
3: **trigger** yield(b)



Consensus = Stay safe + Try (and try again)

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

Uses:
  Binary "Extended" Graded Consensus, **instances** $\mathcal{EGC}_1, \mathcal{EGC}_2, \ldots$ ▷ ∞ instances of the Binary Graded Consensus with Refinement $R = 3$

Constants:
  Integer $g_{min} \leftarrow 0$
  Integer $g_{max} \leftarrow 2$

Local Variables:
  Binary_Value $est_i \leftarrow 0$ ▷ Estimate Value
  Integer $g_i \leftarrow g_{min}$ ▷ Grade (Confidence) in $\{0, 1, 2\}$
  Integer $attempt \leftarrow 0$
  Integer $halt \leftarrow \infty$
  Boolean $decided \leftarrow false$

upon propose$(v_i \in \text{Value})$:
  $est_i \leftarrow v_i$:
  while $halt \geq attempt$:
    //safety guard
    $(est_i, g_i) \leftarrow \mathcal{EGC}_{attempt}$.propose$(est_i)$ ▷ Execute instance of extended graded consensus
    if $g_i == g_{max} \wedge decided = false$:
      trigger decide(est_i) ▷ Decide
      $decided \leftarrow true$
      $halt \leftarrow attempt + 1$ ▷ Halt after the next attempt after having helped the remaining processes to decide
    //try to converge
    $b_i \leftarrow CC_{attempt}$.flip() ▷ Execute instance of common coin
    if $g_i == g_{min}$:
      $est_i \leftarrow b_i$
    $attempt \leftarrow attempt + 1$

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

LEMMA 1.3. *If all correct processes begin attempt k with the same estimate value v, they will all decide on v by attempt k and halt by attempt $k + 1$.*

PROOF.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

LEMMA 1.3. *If all correct processes begin attempt k with the same estimate value v, they will all decide on v by attempt k and halt by attempt k + 1.*

PROOF. By the unanimity property of $\mathcal{EGC}_k$,

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

LEMMA 1.3. *If all correct processes begin attempt k with the same estimate value v, they will all decide on v by attempt k and halt by attempt k + 1.*

PROOF. By the unanimity property of $\mathcal{EGC}_k$, all correct processes return (v, g_{max}) from $\mathcal{EGC}_k$.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)
19:             decided ← true
20:             halt ← attempt + 1
21:         //try to converge
22:         b_i ← CC_attempt.flip()
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

▷ Execute instance of extended graded consensus

▷ Decide

▷ Halt after the next attempt after having helped the remaining processes to decide

▷ Execute instance of common coin

LEMMA 1.3. *If all correct processes begin attempt k with the same estimate value v, they will all decide on v by attempt k and halt by attempt k + 1.*

PROOF. By the unanimity property of $\mathcal{EGC}_k$, all correct processes return (v, g_{max}) from $\mathcal{EGC}_k$.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

LEMMA 1.3. *If all correct processes begin attempt k with the same estimate value v, they will all decide on v by attempt k and halt by attempt $k + 1$.*

PROOF. By the unanimity property of $\mathcal{EGC}_k$, all correct processes return (v, g_{max}) from $\mathcal{EGC}_k$. The rest follows directly from the protocol. □

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

v!

LEMMA 1.4. *If a correct process decides on v in attempt k, then all correct processes will decide on v by attempt $k+1$.*

PROOF. Let p_i be the first correct process to decide, and assume it decides on v at attempt k.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

LEMMA 1.4. *If a correct process decides on v in attempt k, then all correct processes will decide on v by attempt $k+1$.*

PROOF. Let p_i be the first correct process to decide, and assume it decides on v at attempt k. This implies that p_i returned (v, g_{max}) from $\mathcal{EGC}_k$.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

LEMMA 1.4. *If a correct process decides on v in attempt k, then all correct processes will decide on v by attempt $k + 1$.*

PROOF. Let p_i be the first correct process to decide, and assume it decides on v at attempt k. This implies that p_i returned (v, g_{max}) from $\mathcal{EGC}_k$. By the consistency property of $\mathcal{EGC}_k$, every correct process p_j returns $(v, g_j \in \{1, 2\})$ from $\mathcal{EGC}_k$

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

LEMMA 1.4. *If a correct process decides on v in attempt k, then all correct processes will decide on v by attempt $k + 1$.*

PROOF. Let p_i be the first correct process to decide, and assume it decides on v at attempt k. This implies that p_i returned (v, g_{max}) from $\mathcal{EGC}_k$. By the consistency property of $\mathcal{EGC}_k$, every correct process p_j returns $(v, g_j \in \{1, 2\})$ from $\mathcal{EGC}_k$, updating its estimate est_j to v.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

LEMMA 1.4. *If a correct process decides on v in attempt k, then all correct processes will decide on v by attempt $k+1$.*

PROOF. Let p_i be the first correct process to decide, and assume it decides on v at attempt k. This implies that p_i returned (v, g_{max}) from $\mathcal{EGC}_k$. By the consistency property of $\mathcal{EGC}_k$, every correct process p_j returns $(v, g_j \in \{1, 2\})$ from $\mathcal{EGC}_k$, updating its estimate est_j to v. Thus, $g_j > g_{min}$,

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

LEMMA 1.4. *If a correct process decides on v in attempt k, then all correct processes will decide on v by attempt $k+1$.*

PROOF. Let p_i be the first correct process to decide, and assume it decides on v at attempt k. This implies that p_i returned (v, g_{max}) from $\mathcal{EGC}_k$. By the consistency property of $\mathcal{EGC}_k$, every correct process p_j returns $(v, g_j \in \{1, 2\})$ from $\mathcal{EGC}_k$, updating its estimate est_j to v. Thus, $g_j > g_{min}$, so all correct processes ignore the output of CC_k and retain $est_j = v$.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

LEMMA 1.4. *If a correct process decides on v in attempt k, then all correct processes will decide on v by attempt $k+1$.*

PROOF. Let p_i be the first correct process to decide, and assume it decides on v at attempt k. This implies that p_i returned (v, g_{max}) from $\mathcal{EGC}_k$. By the consistency property of $\mathcal{EGC}_k$, every correct process p_j returns $(v, g_j \in \{1,2\})$ from $\mathcal{EGC}_k$, updating its estimate est_j to v. Thus, $g_j > g_{min}$, so all correct processes ignore the output of CC_k and retain $est_j = v$. Consequently, all correct processes begin attempt $k+1$ with estimate value v. □

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

LEMMA 1.4. *If a correct process decides on v in attempt k, then all correct processes will decide on v by attempt $k+1$.*

PROOF. Let p_i be the first correct process to decide, and assume it decides on v at attempt k. This implies that p_i returned (v, g_{max}) from $\mathcal{EGC}_k$. By the consistency property of $\mathcal{EGC}_k$, every correct process p_j returns $(v, g_j \in \{1, 2\})$ from $\mathcal{EGC}_k$, updating its estimate est_j to v. Thus, $g_j > g_{min}$, so all correct processes ignore the output of CC_k and retain $est_j = v$. Consequently, all correct processes begin attempt $k+1$ with estimate value v. Lemma 1.3 then completes the proof. □

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

PROOF.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

PROOF. **Validity** follows from

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

PROOF. **Validity** follows from Lemma 1.3,

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

PROOF. **Validity** follows from Lemma 1.3, and **Agreement** follows from

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

PROOF. **Validity** follows from Lemma 1.3, and **Agreement** follows from Lemma 1.4.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

PROOF. **Validity** follows from Lemma 1.3, and **Agreement** follows from Lemma 1.4. We now prove **Termination**.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

PROOF. **Validity** follows from Lemma 1.3, and **Agreement** follows from Lemma 1.4. We now prove **Termination**. Let p_i be the first correct process calling CC_k for some attempt k.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

PROOF. **Validity** follows from Lemma 1.3, and **Agreement** follows from Lemma 1.4. We now prove **Termination**. Let p_i be the first correct process calling CC_k for some attempt k. Let (b, g_i) the pair returned by p_i from $\mathcal{EGC}_k$.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

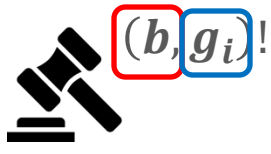
```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

PROOF. **Validity** follows from Lemma 1.3, and **Agreement** follows from Lemma 1.4. We now prove **Termination**. Let p_i be the first correct process calling CC_k for some attempt k. Let (b, g_i) the pair returned by p_i from $\mathcal{EGC}_k$. With non-zero probability ρ, all correct processes return b from CC_k.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

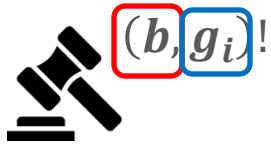


THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

PROOF. **Validity** follows from Lemma 1.3, and **Agreement** follows from Lemma 1.4. We now prove **Termination**. Let p_i be the first correct process calling CC_k for some attempt k. Let (b, g_i) the pair returned by p_i from $\mathcal{EGC}_k$. With non-zero probability ρ, all correct processes return b from CC_k. We consider two cases depending on the grades obtained by correct processes from $\mathcal{EGC}_k$. In both cases, all correct processes start the next attempt with the same estimate value, allowing us to apply Lemma 1.3.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```



THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

We consider two cases depending on the grades obtained by correct processes from $\mathcal{EGC}_k$. In both cases, all correct processes start the next attempt with the same estimate value, allowing us to apply Lemma 1.3.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

We consider two cases depending on the grades obtained by correct processes from $\mathcal{EGC}_k$. In both cases, all correct processes start the next attempt with the same estimate value, allowing us to apply Lemma 1.3.

- **Case** $\exists p_j \in \text{Correct}, g_j > g_{min}$:

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

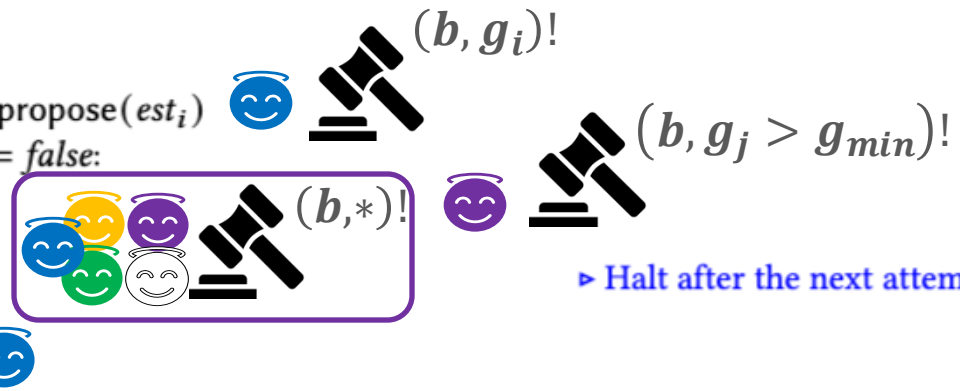
THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

We consider two cases depending on the grades obtained by correct processes from $\mathcal{EGC}_k$. In both cases, all correct processes start the next attempt with the same estimate value, allowing us to apply Lemma 1.3.

- **Case** $\exists p_j \in \text{Correct}, g_j > g_{min}$: By $\mathcal{EGC}_k$'s consistency,

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```



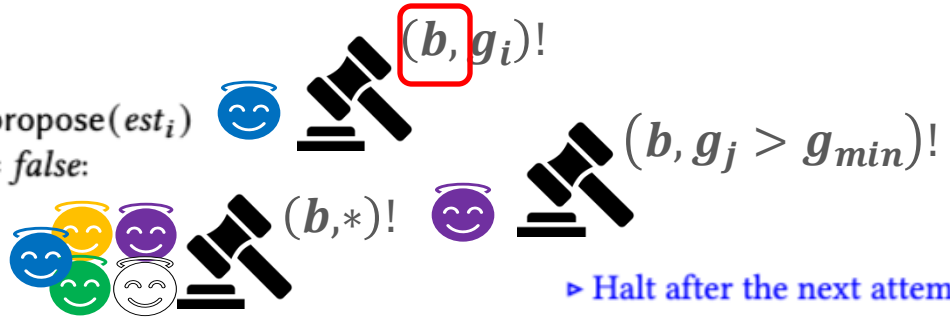
THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

We consider two cases depending on the grades obtained by correct processes from $\mathcal{EGC}_k$. In both cases, all correct processes start the next attempt with the same estimate value, allowing us to apply Lemma 1.3.

- **Case** $\exists p_j \in \text{Correct}, g_j > g_{min}$: By $\mathcal{EGC}_k$'s consistency, (a) all processes return $(b, \cdot)$ from $\mathcal{EGC}_k$,

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```



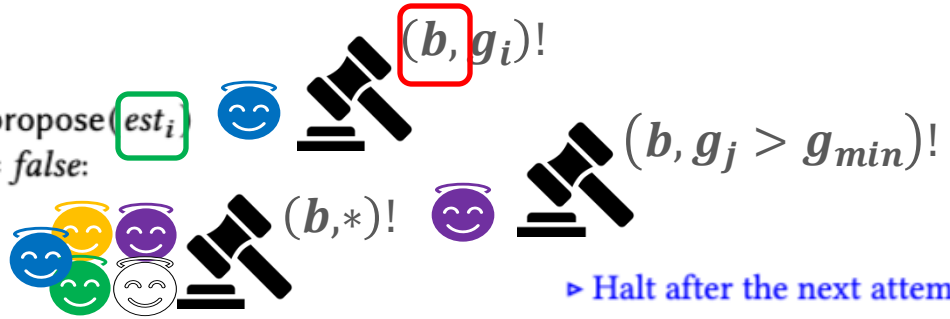
THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

We consider two cases depending on the grades obtained by correct processes from $\mathcal{EGC}_k$. In both cases, all correct processes start the next attempt with the same estimate value, allowing us to apply Lemma 1.3.

- **Case** $\exists p_j \in \text{Correct}, g_j > g_{min}$**:** By $\mathcal{EGC}_k$'s consistency, (a) all processes return $(b, \cdot)$ from $\mathcal{EGC}_k$, so they will all have the same estimate value at attempt $k+1$, either by adopting CC_k's output or by retaining the value from $\mathcal{EGC}_k$.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```



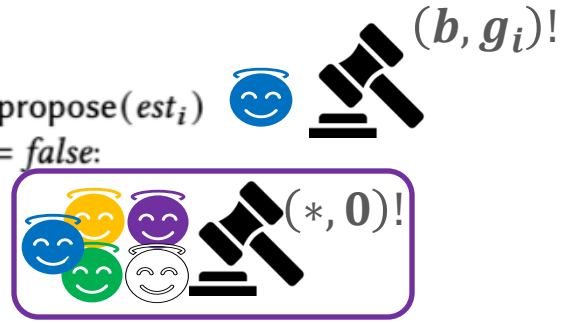
THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

We consider two cases depending on the grades obtained by correct processes from $\mathcal{EGC}_k$. In both cases, all correct processes start the next attempt with the same estimate value, allowing us to apply Lemma 1.3.

- **Case** $\exists p_j \in \text{Correct}, g_j > g_{min}$**:** By $\mathcal{EGC}_k$'s consistency, (a) all processes return $(b, \cdot)$ from $\mathcal{EGC}_k$, so they will all have the same estimate value at attempt $k+1$, either by adopting CC_k's output or by retaining the value from $\mathcal{EGC}_k$.

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```



THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

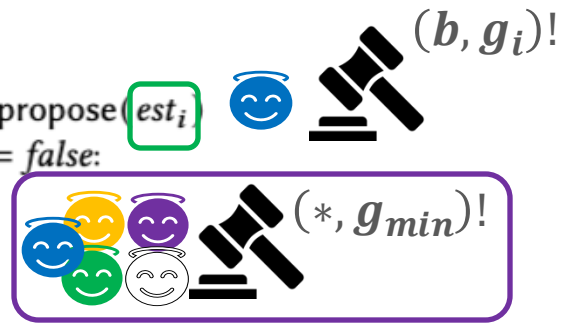
We consider two cases depending on the grades obtained by correct processes from $\mathcal{EGC}_k$. In both cases, all correct processes start the next attempt with the same estimate value, allowing us to apply Lemma 1.3.

- **Case** $\exists p_j \in \text{Correct}, g_j > g_{min}$**:** By $\mathcal{EGC}_k$'s consistency, (a) all processes return $(b, \cdot)$ from $\mathcal{EGC}_k$, so they will all have the same estimate value at attempt $k+1$, either by adopting CC_k's output or by retaining the value from $\mathcal{EGC}_k$.

- **Case** $\forall p_j \in \text{Correct}, g_j = g_{min}$**:**

Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt ≥ attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```



THEOREM 1.5. *Algorithm 4 implements binary Byzantine consensus with probability* 1 *and has the same resiliency as the underlying graded consensus object.*

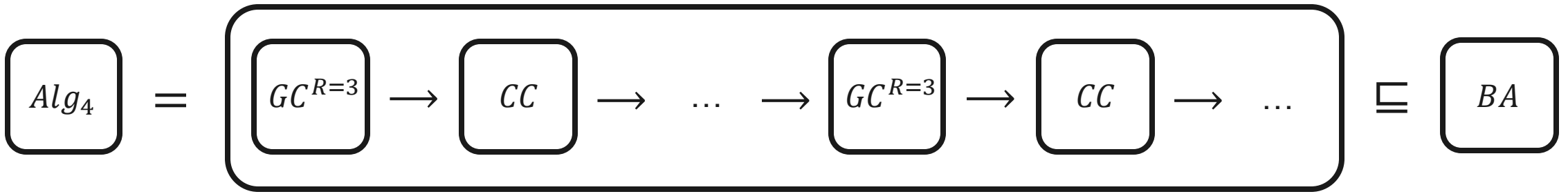
We consider two cases depending on the grades obtained by correct processes from $\mathcal{EGC}_k$. In both cases, all correct processes start the next attempt with the same estimate value, allowing us to apply Lemma 1.3.

- **Case** $\exists p_j \in \text{Correct}, g_j > g_{min}$**:** By $\mathcal{EGC}_k$'s consistency, (a) all processes return $(b, \cdot)$ from $\mathcal{EGC}_k$, so they will all have the same estimate value at attempt $k+1$, either by adopting CC_k's output or by retaining the value from $\mathcal{EGC}_k$.

- **Case** $\forall p_j \in \text{Correct}, g_j = g_{min}$**:** All processes adopt the value provided by the common coin, which is identical across processes. □

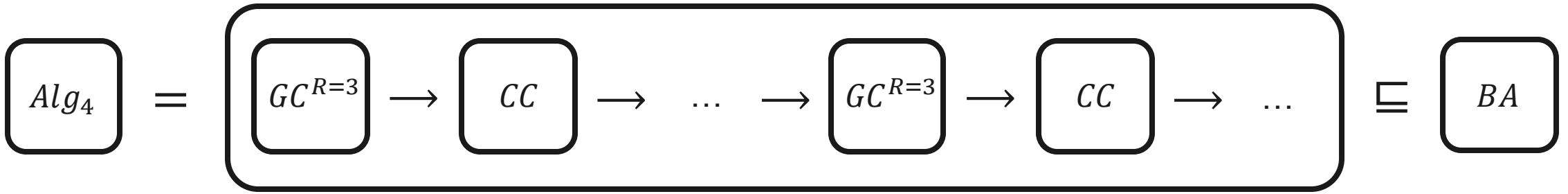
Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

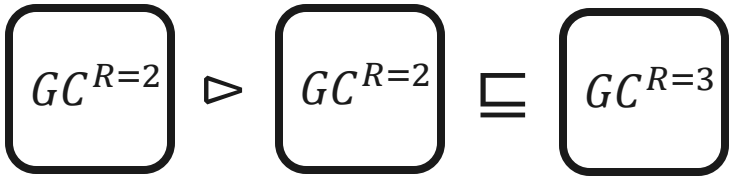
```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt > attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```



Algorithm 4 Byzantine Agreement Protocol with Extended Graded Consensus

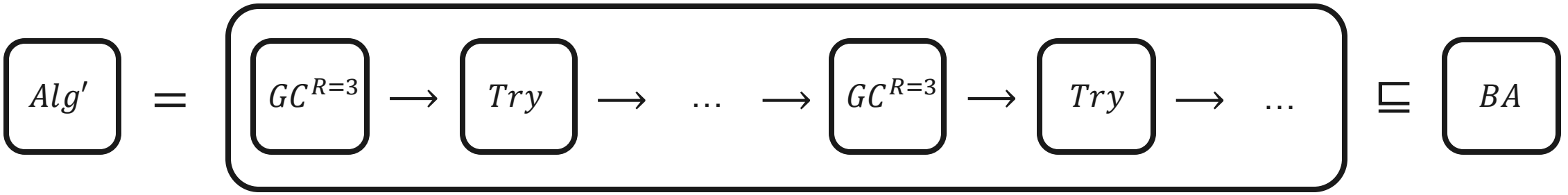
```
12: upon propose(v_i ∈ Value):
13:     est_i ← v_i:
14:     while halt > attempt:
15:         //safety guard
16:         (est_i, g_i) ← EGC_attempt.propose(est_i)        ▷ Execute instance of extended graded consensus
17:         if g_i == g_max ∧ decided = false:
18:             trigger decide(est_i)                        ▷ Decide
19:             decided ← true
20:             halt ← attempt + 1                           ▷ Halt after the next attempt after having helped the remaining processes to decide
21:         //try to converge
22:         b_i ← CC_attempt.flip()                          ▷ Execute instance of common coin
23:         if g_i == g_min:
24:             est_i ← b_i
25:         attempt ← attempt + 1
```

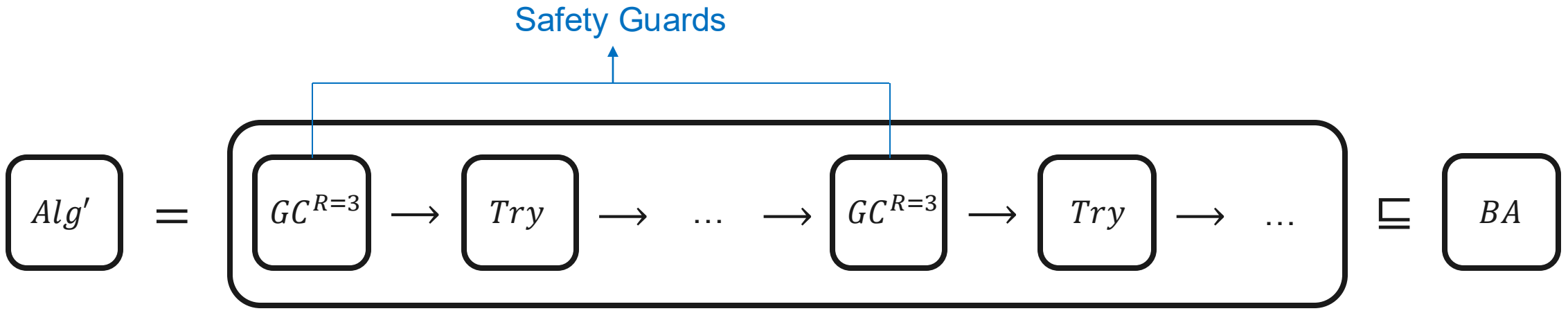


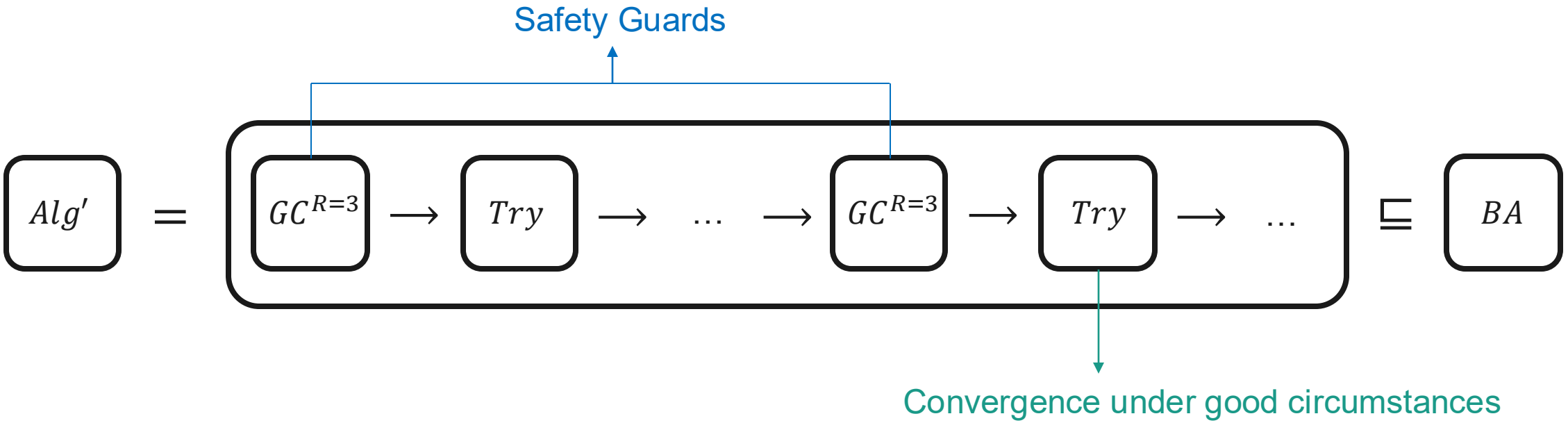


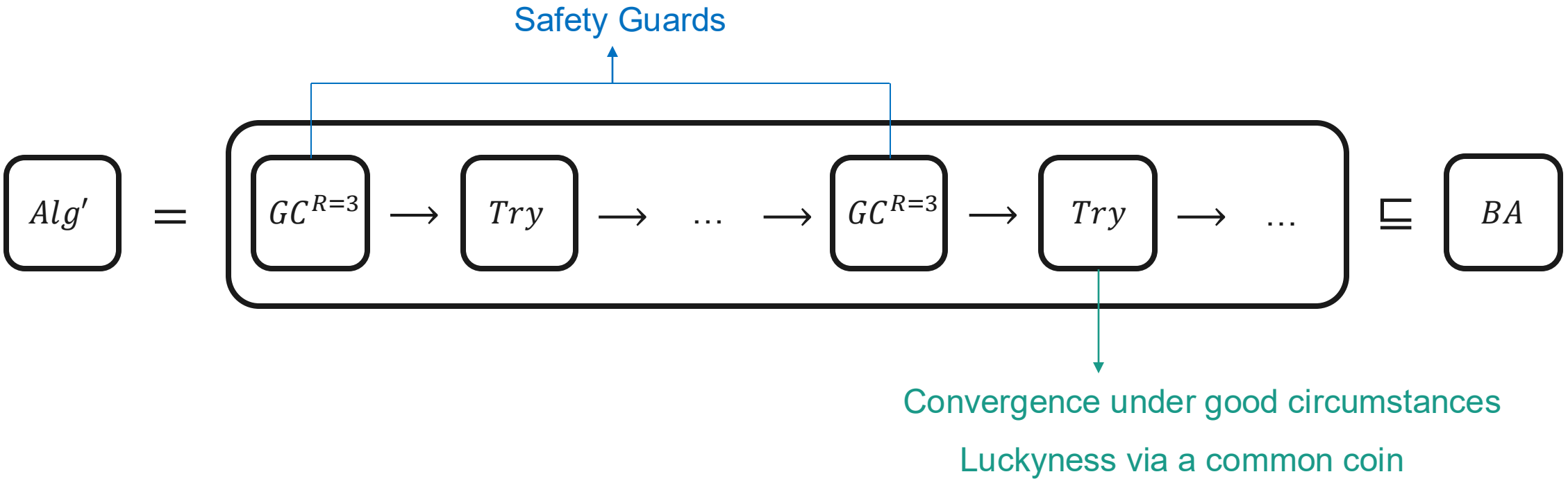
Positive Result 1: *There exists* a **randomized asynchronous** protocol that solves consensus, while tolerating arbitrary (**Byzantine**) failures

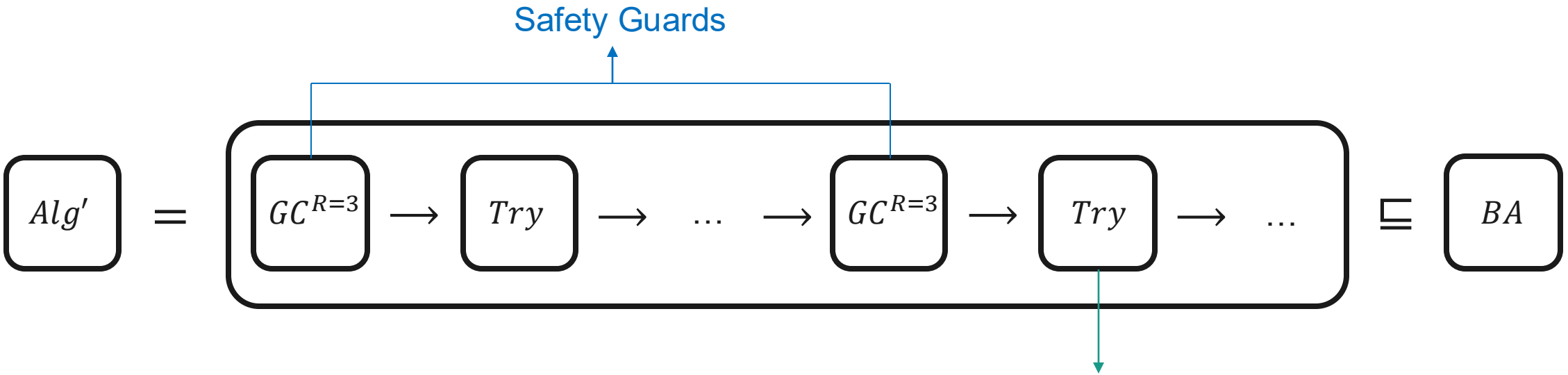
A general perspective

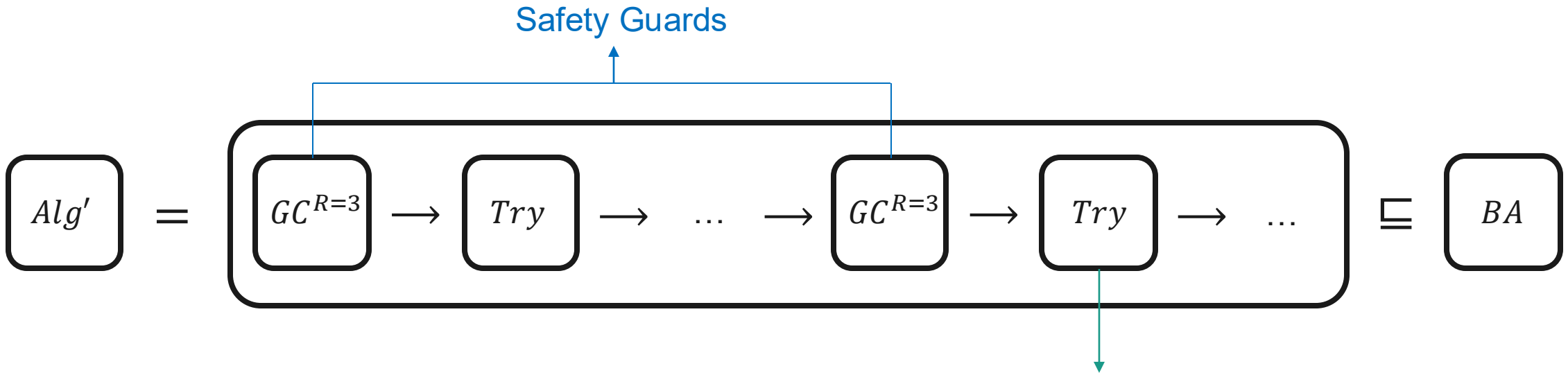

Alg'
=
$GC^{R=3}$
Try
…
$GC^{R=3}$
Try
…
⊑
BA

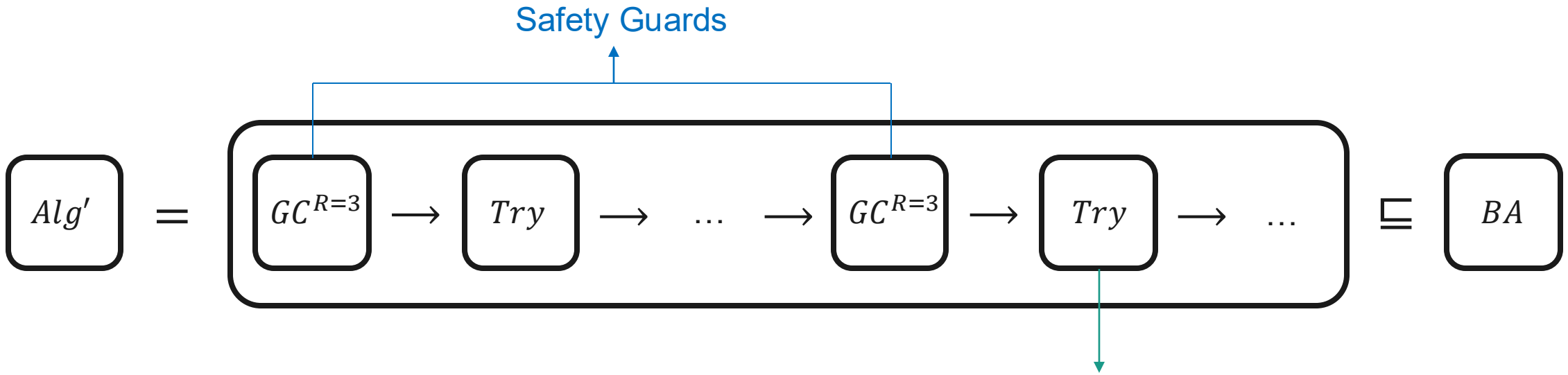

Safety Guards
Alg'
=
$GC^{R=3}$
Try
…
$GC^{R=3}$
Try
…
⊑
BA


Safety Guards
Alg'
=
$GC^{R=3}$
Try
…
$GC^{R=3}$
Try
…
⊑
BA
Convergence under good circumstances


Safety Guards
Alg'
=
$GC^{R=3}$
Try
…
$GC^{R=3}$
Try
…
⊑
BA
Convergence under good circumstances
Luckyness via a common coin

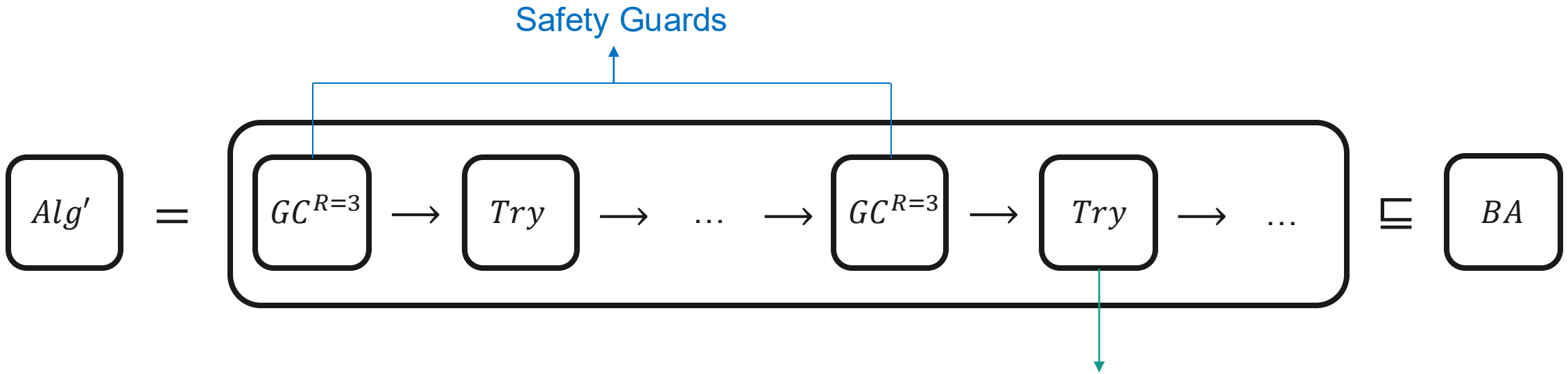

Safety Guards
Alg'
=
$GC^{R=3}$
Try
…
$GC^{R=3}$
Try
…
⊑
BA
Convergence under good circumstances
Luckyness via a common coin
Eventual Synchrony + Synchronization


Safety Guards
Alg'
=
$GC^{R=3}$
Try
…
$GC^{R=3}$
Try
…
⊑
BA
Convergence under good circumstances
Luckyness via a common coin
Eventual Synchrony + Synchronization
Unreliable Failure Detectors



Convergence under good circumstances

- Luckyness via a common coin
- Eventual Synchrony + Synchronization
- Unreliable Failure Detectors
- Fair scheduling / Noisy Environement



Convergence under good circumstances

- Luckyness via a common coin
- Eventual Synchrony + Synchronization
- Unreliable Failure Detectors
- Fair scheduling / Noisy Environment
- Synchrony + round-robin rotating leader

Consensus

Solvability

Asynchronous Model

No notion of time

1982

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

1983

Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols

(Extended Abstract)

Michael Ben-Or †

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

1. Introduction

Recently, Fischer, Lynch and Paterson [3] proved that no completely asynchronous consensus protocol can tolerate even a single unannounced process death. We exhibit here a probabilistic solution for this problem, which guarantees that as long as a majority of the processes continues to operate, a decision will be made (Theorem 1). Our solution is completely asynchronous and is rather strong: As in [4], it is guaranteed to work with probability 1 even against an adversary scheduler who knows all about the system.

process.

problem.

tic protocol. Previous examples required the processes to be symmetric.

The protocols presented here are not necessarily efficient. However, if the number of faulty processes, t, is $O(\sqrt{N})$, then when running the processes synchronously, the expected time to reach agreement is constant (Theorem 3). This result shows another advantage of probabilistic protocols, since any deterministic solution to the "Byzantine Generals" problem cannot reach agreement in less than $t+1$ rounds, (see [1,2]).

2. The Consensus Problem

1989

Solvability in Asynchronous Environments*
(extended abstract)

Benny Chor Lior Moscovici

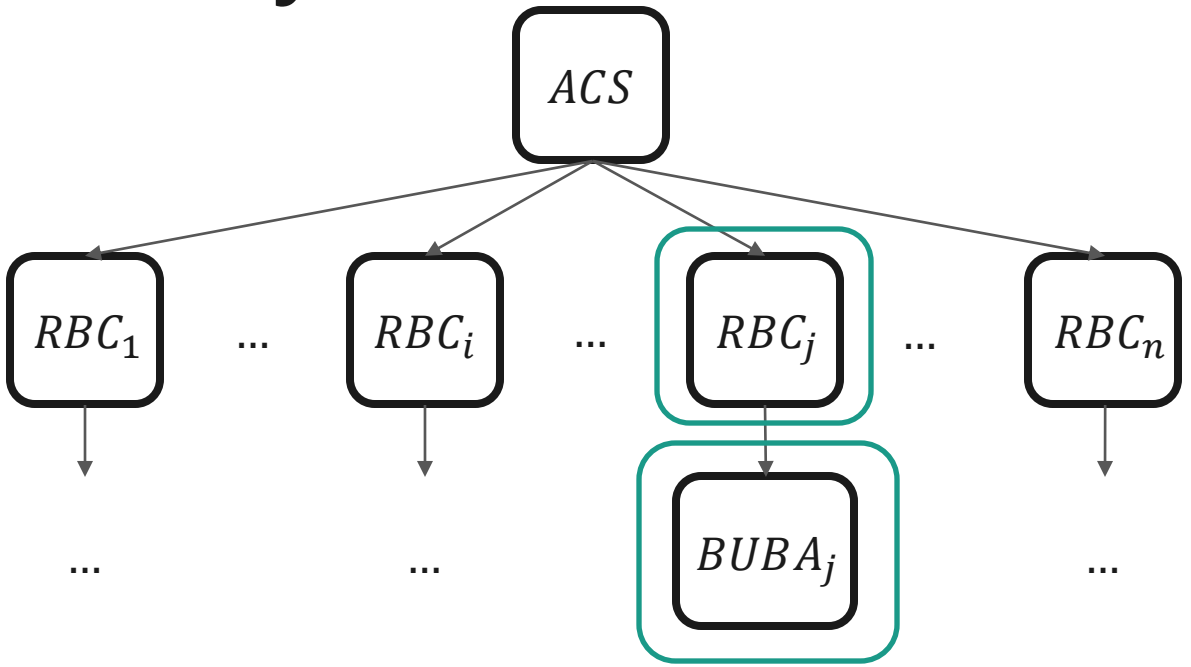
Department of Computer Science
Technion, Haifa 32000, Israel

Vector Consensus (a.k.a. Agreement on a Core Set, a.k.a. Asynchronous Common Subset)

ACS

- ***Vector Validity:*** A decision is a vector which contains at least $n - t$ process-proposal pairs, such that, if (p, v) belongs to the vector, and p is correct, the p proposed v.

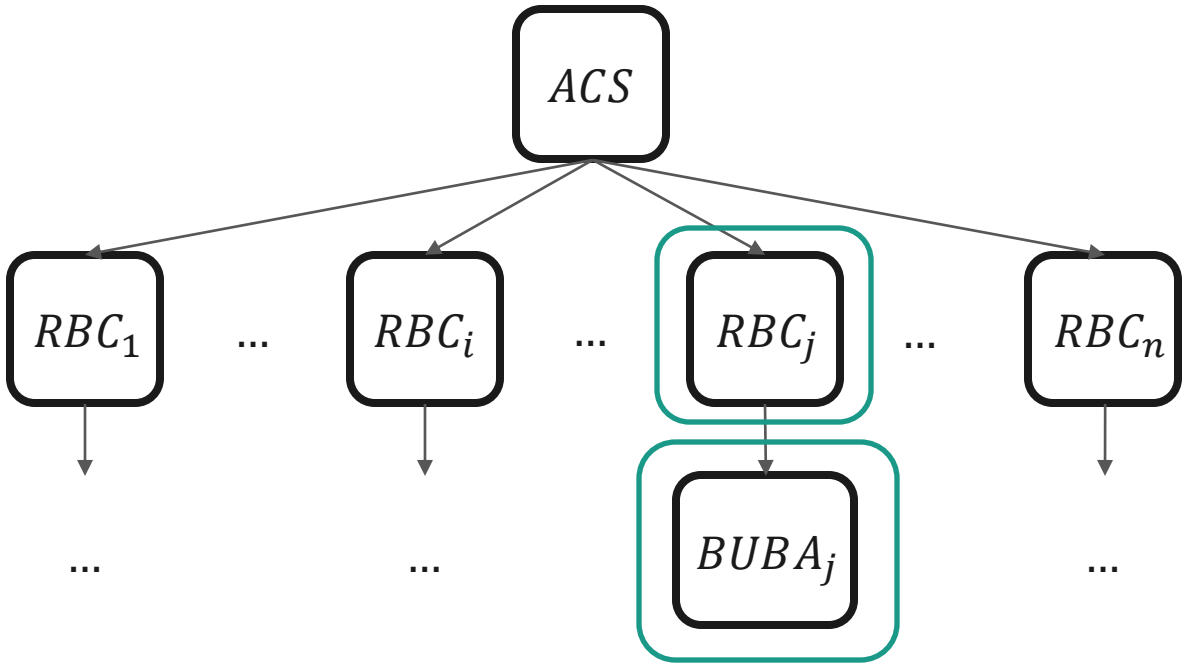
Vector Consensus (a.k.a. Agreement on a Core Set, a.k.a. Asynchronous Common Subset)



RBC: *Binary Reliable Broadcast*

$BUBA$: *Binary Byzantine Agreement with Strong Unanimity Validity property*

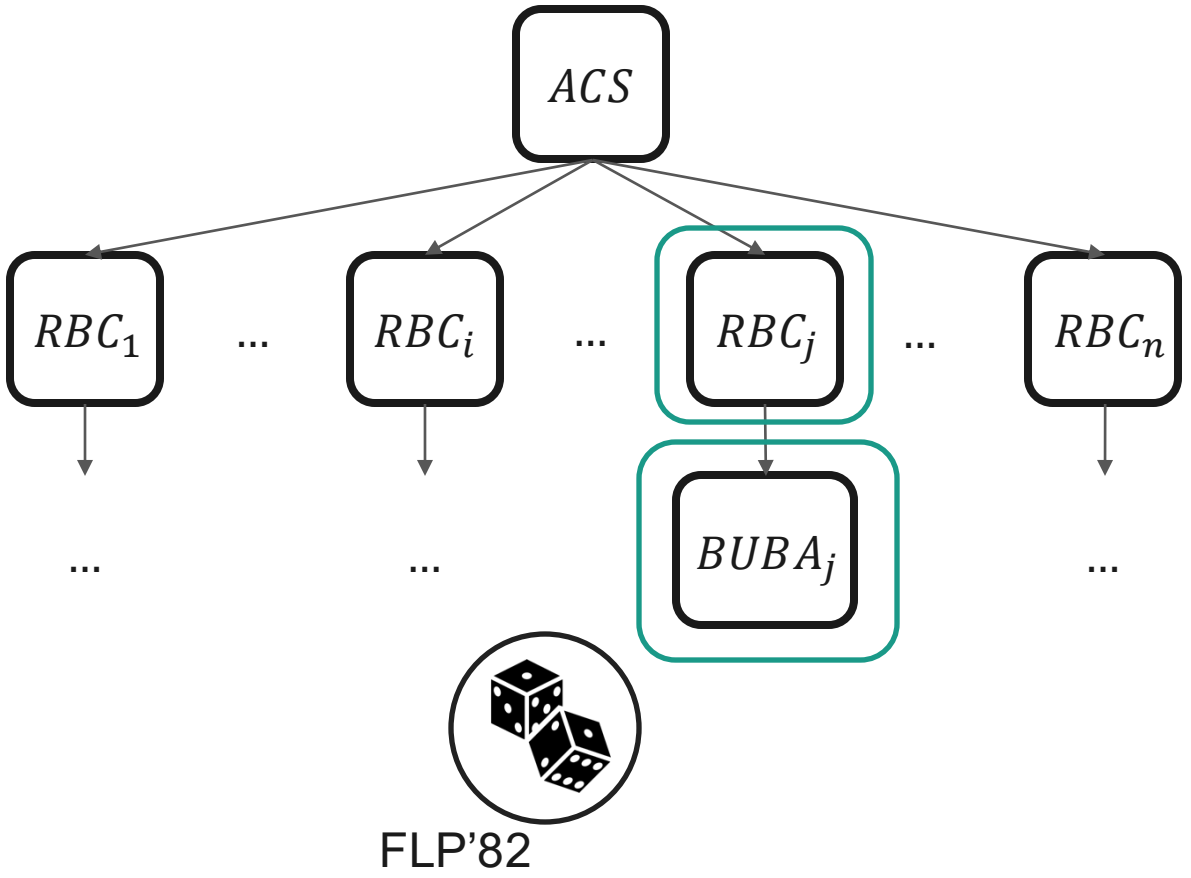
Vector Consensus



$n > 3t$

partition (Bracha-Toueg'83)

Vector Consensus



$n > 3t$

Reliable Broadcast

Reliable Broadcast

Specification

Interface of reliable broadcast

Rb.broadcast

Operation

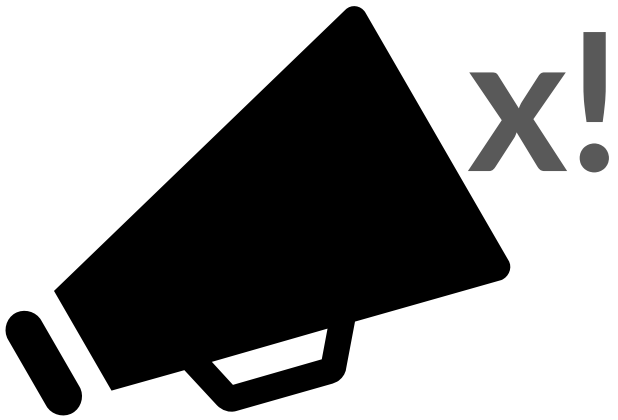
Rb.deliver

Callback

Interface of reliable broadcast

Rb.broadcast

Operation



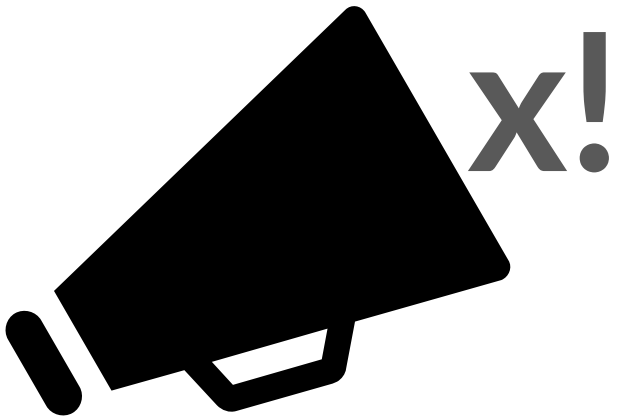
Rb.deliver

Callback

Interface of reliable broadcast

Rb.broadcast

Operation



Rb.deliver

Callback

Properties of Reliable Broadcast

Let s be the source (possibly faulty). Let (p, q) be any pair of correct processes.

- **Justification:** If p delivers m and s is correct, then broadcasted m
- **Agreement:** If p delivers m and q delivers, then m = m'.
- **Obligation:** If s is correct and broadcast m, then p eventually delivers m
- **Totality:** If p delivers m, then q eventually delivers m'.

Reliable Broadcast

Asynchronous implementation with t<n/5 Byzantine failures

Implementation of Reliable Broadcast

Algorithm 5 Reliable Broadcast (RB): Pseudocode (for process p_i)

1: **Local Variables:**
2: Boolean $init_i \leftarrow false$ ▷ init message received yet
3: Boolean $del_i \leftarrow false$ ▷ delivered yet
4: **upon** RB_broadcast(v_i):
5: **broadcast** $\langle \text{INIT}, i, v_i \rangle$
6: **upon** $\langle \text{INIT}, j, v \rangle$ is received from $p_j \wedge init_i = false$:
7: $init_i \leftarrow true$
8: **broadcast** $\langle \text{WITNESS}, j, v \rangle$
9: **upon** $\langle \text{WITNESS}, j, v \rangle$ is received from $n - 2t$ processes:
10: **broadcast** $\langle \text{WITNESS}, j, v \rangle$ ▷ Echo once a $n - 2t$ quorum is observed
11: **upon** $\langle \text{WITNESS}, j, v \rangle$ is received from $n - t$ processes $\wedge\ del_i = false$:
12: $del_i \leftarrow true$
13: **trigger** RB_deliver(v) ▷ Deliver on $n - t$ witnesses

Algorithm 5 Reliable Broadcast (RB): Pseudocode (for process p_i)

```
1:  Local Variables:
2:      Boolean init_i ← false                          ▷ init message received yet
3:      Boolean del_i ← false                           ▷ delivered yet
4:  upon RB_broadcast(v_i):
5:      broadcast ⟨INIT, i, v_i⟩
6:  upon ⟨INIT, j, v⟩ is received from p_j ∧ init_i = false:
7:      init_i ← true
8:      broadcast ⟨WITNESS, j, v⟩
9:  upon ⟨WITNESS, j, v⟩ is received from n − 2t processes:
10:     broadcast ⟨WITNESS, j, v⟩                       ▷ Echo once a n − 2t quorum is observed
11: upon ⟨WITNESS, j, v⟩ is received from n − t processes ∧ del_i = false:
12:     del_i ← true
13:     trigger RB_deliver(v)                           ▷ Deliver on n − t witnesses
```

THEOREM 6.2. *Algorithm 5 implements the reliable broadcast abstraction in an asynchronous n-process message-passing system with up to $t < n/5$ Byzantine faults.*

Algorithm 5 Reliable Broadcast (RB): Pseudocode (for process p_i)

```
1:  Local Variables:
2:      Boolean init_i ← false                          ▷ init message received yet
3:      Boolean del_i ← false                           ▷ delivered yet
4:  upon RB_broadcast(v_i):
5:      broadcast ⟨INIT, i, v_i⟩
6:  upon ⟨INIT, j, v⟩ is received from p_j ∧ init_i = false:
7:      init_i ← true
8:      broadcast ⟨WITNESS, j, v⟩
9:  upon ⟨WITNESS, j, v⟩ is received from n − 2t processes:
10:     broadcast ⟨WITNESS, j, v⟩                       ▷ Echo once a n − 2t quorum is observed
11: upon ⟨WITNESS, j, v⟩ is received from n − t processes ∧ del_i = false:
12:     del_i ← true
13:     trigger RB_deliver(v)                           ▷ Deliver on n − t witnesses
```

LEMMA 6.1. *Assume the source s is correct and never send a message of the form $\langle \text{INIT}, s, v \rangle$. No correct process ever sends a message $\langle \text{WITNESS}, s, v \rangle$.*

Algorithm 5 Reliable Broadcast (RB): Pseudocode (for process p_i)

```
1:  Local Variables:
2:      Boolean init_i ← false                          ▷ init message received yet
3:      Boolean del_i ← false                           ▷ delivered yet
4:  upon RB_broadcast(v_i):
5:      broadcast ⟨INIT, i, v_i⟩
6:  upon ⟨INIT, j, v⟩ is received from p_j ∧ init_i = false:
7:      init_i ← true
8:      broadcast ⟨WITNESS, j, v⟩
9:  upon ⟨WITNESS, j, v⟩ is received from n − 2t processes:
10:     broadcast ⟨WITNESS, j, v⟩                       ▷ Echo once a n − 2t quorum is observed
11: upon ⟨WITNESS, j, v⟩ is received from n − t processes ∧ del_i = false:
12:     del_i ← true
13:     trigger RB_deliver(v)                           ▷ Deliver on n − t witnesses
```

LEMMA 6.1. *Assume the source s is correct and never send a message of the form $\langle \text{INIT}, s, v\rangle$. No correct process ever sends a message $\langle \text{WITNESS}, s, v\rangle$.*

PROOF. By contradiction, assume there exists a set Q of processes who sent messages $\langle \text{WITNESS}, s, v\rangle$. Since no correct process receives $\langle \text{INIT}, s, v\rangle$, no correct process sends a WITNESS message line 8 for a value different from v. Hence, if a correct process sends a WITNESS message, it is necessarily line 10. Let q be the first correct of them. The precondition line 9 must hold, meaning q must have received $n - 2t > t$ WITNESS messages, which is impossible. We have reached a contradiction. Thus, the post-condition of the lemma. □

Algorithm 5 Reliable Broadcast (RB): Pseudocode (for process p_i)

```
1:  Local Variables:
2:      Boolean init_i ← false                          ▷ init message received yet
3:      Boolean del_i ← false                           ▷ delivered yet
4:  upon RB_broadcast(v_i):
5:      broadcast ⟨INIT, i, v_i⟩
6:  upon ⟨INIT, j, v⟩ is received from p_j ∧ init_i = false:
7:      init_i ← true
8:      broadcast ⟨WITNESS, j, v⟩
9:  upon ⟨WITNESS, j, v⟩ is received from n − 2t processes:
10:     broadcast ⟨WITNESS, j, v⟩                       ▷ Echo once a n − 2t quorum is observed
11: upon ⟨WITNESS, j, v⟩ is received from n − t processes ∧ del_i = false:
12:     del_i ← true
13:     trigger RB_deliver(v)                           ▷ Deliver on n − t witnesses
```

THEOREM 6.2. *Algorithm 5 implements the reliable broadcast abstraction in an asynchronous n-process message-passing system with up to $t < n/5$ Byzantine faults.*

PROOF. We show that the algorithm satisfies the four defining properties: *Justification*, *Agreement*, *Obligation*, and *Totality*.

Algorithm 5 Reliable Broadcast (RB): Pseudocode (for process p_i)

```
1:  Local Variables:
2:      Boolean init_i ← false                          ▷ init message received yet
3:      Boolean del_i ← false                           ▷ delivered yet
4:  upon RB_broadcast(v_i):
5:      broadcast ⟨INIT, i, v_i⟩
6:  upon ⟨INIT, j, v⟩ is received from p_j ∧ init_i = false:
7:      init_i ← true
8:      broadcast ⟨WITNESS, j, v⟩
9:  upon ⟨WITNESS, j, v⟩ is received from n − 2t processes:
10:     broadcast ⟨WITNESS, j, v⟩                       ▷ Echo once a n − 2t quorum is observed
11: upon ⟨WITNESS, j, v⟩ is received from n − t processes ∧ del_i = false:
12:     del_i ← true
13:     trigger RB_deliver(v)                           ▷ Deliver on n − t witnesses
```

Justification. Assume the source is correct and triggered RB_broadcast(v). This means the source never sent a message $\langle \text{INIT}, s, v' \neq v \rangle$. Thus, by Lemma 6.1. No correct process ever sends a $\langle \text{WITNESS}, s, v' \neq v \rangle$ message. Hence, no correct process ever receive more than t $\langle \text{WITNESS}, s, v' \neq v \rangle$, which means that if a correct process triggers RB_deliver(v'), $v' = v$.

Algorithm 5 Reliable Broadcast (RB): Pseudocode (for process p_i)

```
1:  Local Variables:
2:      Boolean init_i ← false                          ▷ init message received yet
3:      Boolean del_i ← false                           ▷ delivered yet
4:  upon RB_broadcast(v_i):
5:      broadcast ⟨INIT, i, v_i⟩
6:  upon ⟨INIT, j, v⟩ is received from p_j ∧ init_i = false:
7:      init_i ← true
8:      broadcast ⟨WITNESS, j, v⟩
9:  upon ⟨WITNESS, j, v⟩ is received from n − 2t processes:
10:     broadcast ⟨WITNESS, j, v⟩                       ▷ Echo once a n − 2t quorum is observed
11: upon ⟨WITNESS, j, v⟩ is received from n − t processes ∧ del_i = false:
12:     del_i ← true
13:     trigger RB_deliver(v)                           ▷ Deliver on n − t witnesses
```

Agreement. Let p_i and p_j be two correct processes that trigger RB_deliver(v_i) and RB_deliver(v_j) respectively (line 13). Let Q_i (resp. Q_j) be the set of processes from which p_i (resp. p_j) received $\langle \text{WITNESS}, s, v_i \rangle$ (resp. $\langle \text{WITNESS}, s, v_j \rangle$). By definition, $|Q_i| = |Q_j| = n - t$. Assume, for contradiction, that $v_i \neq v_j$.

Let $C_k^{init} \subseteq Q_k$ denote the subset of correct processes that sent their WITNESS for value v_k at line 8, and $C_k^{echo} \subseteq Q_k$ denote those that sent it at line 10. Since only the source can send $\langle \text{INIT}, s, \cdot \rangle$ messages, each correct process can belong to at most one such set C_x^{init}. If $v_i \neq v_j$, then $C_i^{init} \cap C_j^{init} = \emptyset$. Let B be the set of Byzantine processes.

We have $|C_i^{init}| + |C_j^{init}| + |B| \leq n$. Without loss of generality, assume $|C_i^{init}| \leq |C_j^{init}|$. Necessarily, $|C_i^{init}| + |B| \leq 3t < n - 2t$. Thus, for process p_i to deliver v_i, some correct process must have sent a WITNESS at line 10, i.e., $C_i^{echo} \neq \emptyset$. Let q be the first such correct process. To send its WITNESS, q must have received $n - 2t > |C_i^{init} \cup B|$ WITNESS messages (line 9) from processes in $C_i^{init} \cup B$. We reach a contradiction. Thus, $v_i = v_j$ and agreement holds.

Algorithm 5 Reliable Broadcast (RB): Pseudocode (for process p_i)

```
1:  Local Variables:
2:      Boolean init_i ← false                          ▷ init message received yet
3:      Boolean del_i ← false                           ▷ delivered yet
4:  upon RB_broadcast(v_i):
5:      broadcast ⟨INIT, i, v_i⟩
6:  upon ⟨INIT, j, v⟩ is received from p_j ∧ init_i = false:
7:      init_i ← true
8:      broadcast ⟨WITNESS, j, v⟩
9:  upon ⟨WITNESS, j, v⟩ is received from n − 2t processes:
10:     broadcast ⟨WITNESS, j, v⟩                       ▷ Echo once a n − 2t quorum is observed
11: upon ⟨WITNESS, j, v⟩ is received from n − t processes ∧ del_i = false:
12:     del_i ← true
13:     trigger RB_deliver(v)                           ▷ Deliver on n − t witnesses
```

Totality. Assume RB_deliver(v) is triggered by a correct process p. It must have received $\langle \text{WITNESS}, i, v \rangle$ from at least $n - t$ processes, including at least $n - 2t$ correct ones. Those correct processes will eventually cause all others to satisfy the forwarding condition (line 8) and broadcast $\langle \text{WITNESS}, i, v \rangle$. Consequently, every correct process eventually receives the message from $n - t$ distinct sources and delivers RB_deliver(v).

Algorithm 5 Reliable Broadcast (RB): Pseudocode (for process p_i)

```
1:  Local Variables:
2:      Boolean init_i ← false                          ▷ init message received yet
3:      Boolean del_i ← false                           ▷ delivered yet
4:  upon RB_broadcast(v_i):
5:      broadcast ⟨INIT, i, v_i⟩
6:  upon ⟨INIT, j, v⟩ is received from p_j ∧ init_i = false:
7:      init_i ← true
8:      broadcast ⟨WITNESS, j, v⟩
9:  upon ⟨WITNESS, j, v⟩ is received from n − 2t processes:
10:     broadcast ⟨WITNESS, j, v⟩                       ▷ Echo once a n − 2t quorum is observed
11: upon ⟨WITNESS, j, v⟩ is received from n − t processes ∧ del_i = false:
12:     del_i ← true
13:     trigger RB_deliver(v)                           ▷ Deliver on n − t witnesses
```

Obligation. Assume the source s is correct and invokes RB_broadcast(v) (line 4). Then s broadcasts $\langle \text{INIT}, s, v \rangle$ (line 5). Every correct process eventually receives this message and, by line 8, broadcasts $\langle \text{WITNESS}, s, v \rangle$. By Justification, no correct process delivers a value $v' \neq v$. Hence, all correct processes eventually receive $\langle \text{WITNESS}, s, v \rangle$ from at least $n - t$ distinct sources and trigger RB_deliver(v) at line 13.

Vector Consensus

Vector Consensus

Reduction to Strong Binary Consensus and Reliable Broadcast

Implementation of Vector Consensus

Algorithm 6 Vector Consensus: Pseudocode (for process p_i)

1: **Local variables:**
2: Array$_{[n]}$(Value $\cup \{\perp\}$) $delivered_i \leftarrow \{\perp, \ldots, \perp\}$
3: Array$_{[n]}$(Boolean) $proposed_i \leftarrow \{false, \ldots, false\}$
4: Set(Process) $selected_i \leftarrow \emptyset$
5: Set(Process) $ignored_i \leftarrow \emptyset$
6: Array$_{[n]}$(Value $\cup \{\perp\}$) $output_i \leftarrow \{\perp, \ldots, \perp\}$
7: **Uses:**
8: Reliable Broadcast, **instance** $\mathcal{RBC}_j$ with source p_j, $\forall j \in [n]$
9: Strong Binary Byzantine Agreement, **instance** $\mathcal{BA}_j$, $\forall j \in [n]$
10: **upon** propose($v_i \in$ Value):
11: **invoke** $\mathcal{RBC}_i$.rb_broadcast(v_i)
12: **upon first** $\mathcal{RBC}_j$.deliver($v_j \in$ Value):
13: $delivered_i[p_j] \leftarrow v_j$
14: **invoke** $\mathcal{BA}_j$.propose(1)
15: $proposed_i[p_j] \leftarrow true$
16: **upon** $\mathcal{BA}_j$.decide($b_j \in \{0, 1\}$):
17: **if** $b_j = 1$:
18: $selected_i \leftarrow selected_i \cup \{p_j\}$
19: **else**:
20: $ignored_i \leftarrow ignored_i \cup \{p_j\}$
21: **upon** $|selected_i| \geq n - t$:
22: Let $K = \{k \in [n] \mid proposed_i[p_k] = false\}$
23: **for each** $k \in K$:
24: **invoke** $\mathcal{BA}_k$.propose(0)
25: $proposed_i[p_k] \leftarrow true$
26: **upon** $|selected_i \cup ignored_i| = n$:
27: **wait until** $\forall p_j \in selected_i$, $delivered_i[p_j] \neq \perp$
28: **for each** $p_j \in selected_i$:
29: $output_i[j] \leftarrow delivered_i[p_j]$
30: **trigger** decide($output_i$)

Algorithm 6 Vector Consensus: Pseudocode (for process p_i)

```
1:  Local variables:
2:      Array_[n](Value ∪ {⊥}) delivered_i ← {⊥, ..., ⊥}
3:      Array_[n](Boolean) proposed_i ← {false, ..., false}
4:      Set(Process) selected_i ← ∅
5:      Set(Process) ignored_i ← ∅
6:      Array_[n](Value ∪ {⊥}) output_i ← {⊥, ..., ⊥}
7:  Uses:
8:      Reliable Broadcast, instance RBC_j with source p_j, ∀j ∈ [n]
9:      Strong Binary Byzantine Agreement, instance BA_j, ∀j ∈ [n]
10: upon propose(v_i ∈ Value):
11:     invoke RBC_i.rb_broadcast(v_i)
12: upon first RBC_j.deliver(v_j ∈ Value):
13:     delivered_i[p_j] ← v_j
14:     invoke BA_j.propose(1)
15:     proposed_i[p_j] ← true
16: upon BA_j.decide(b_j ∈ {0, 1}):
17:     if b_j = 1:
18:         selected_i ← selected_i ∪ {p_j}
19:     else:
20:         ignored_i ← ignored_i ∪ {p_j}
21: upon |selected_i| ≥ n − t:
22:     Let K = {k ∈ [n] | proposed_i[p_k] = false}
23:     for each k ∈ K:
24:         invoke BA_k.propose(0)
25:         proposed_i[p_k] ← true
26: upon |selected_i ∪ ignored_i| = n:
27:     wait until ∀p_j ∈ selected_i, delivered_i[p_j] ≠ ⊥
28:     for each p_j ∈ selected_i:
29:         output_i[j] ← delivered_i[p_j]
30:     trigger decide(output_i)
```

Agreement

Agreement follows directly from the Agreement properties of both reliable broadcast (line 8) and binary Byzantine agreement (line 9). For each index $j \in [n]$, all correct processes decide the same bit b_j in $\mathcal{BA}_j$ (line 16). Hence, all correct processes have identical sets $selected_i$ and $ignored_i$ (lines 18–20). Furthermore, if a value v' is delivered by one correct process through $\mathcal{RBC}_j$ (line 13), it is delivered by all correct processes with the same content, by reliable broadcast Agreement. Thus, when processes fill their output vectors (lines 28–29), they assign the same value to every index j, and therefore decide identical vectors at line 30.

Algorithm 6 Vector Consensus: Pseudocode (for process p_i)

```
1:  Local variables:
2:      Array_[n](Value ∪ {⊥}) delivered_i ← {⊥, ..., ⊥}
3:      Array_[n](Boolean) proposed_i ← {false, ..., false}
4:      Set(Process) selected_i ← ∅
5:      Set(Process) ignored_i ← ∅
6:      Array_[n](Value ∪ {⊥}) output_i ← {⊥, ..., ⊥}
7:  Uses:
8:      Reliable Broadcast, instance RBC_j with source p_j, ∀j ∈ [n]
9:      Strong Binary Byzantine Agreement, instance BA_j, ∀j ∈ [n]
10: upon propose(v_i ∈ Value):
11:     invoke RBC_i.rb_broadcast(v_i)
12: upon first RBC_j.deliver(v_j ∈ Value):
13:     delivered_i[p_j] ← v_j
14:     invoke BA_j.propose(1)
15:     proposed_i[p_j] ← true
16: upon BA_j.decide(b_j ∈ {0, 1}):
17:     if b_j = 1:
18:         selected_i ← selected_i ∪ {p_j}
19:     else:
20:         ignored_i ← ignored_i ∪ {p_j}
21: upon |selected_i| ≥ n − t:
22:     Let K = {k ∈ [n] | proposed_i[p_k] = false}
23:     for each k ∈ K:
24:         invoke BA_k.propose(0)
25:         proposed_i[p_k] ← true
26: upon |selected_i ∪ ignored_i| = n:
27:     wait until ∀p_j ∈ selected_i, delivered_i[p_j] ≠ ⊥
28:     for each p_j ∈ selected_i:
29:         output_i[j] ← delivered_i[p_j]
30:     trigger decide(output_i)
```

Vector Validity

We prove two properties: (a) the decided vector contains at least $n - t$ non-$\perp$ values, and (b) every non-$\perp$ entry originates from a value proposed by the corresponding process if that process is correct.

Algorithm 6 Vector Consensus: Pseudocode (for process p_i)

```
1:  Local variables:
2:      Array_[n](Value ∪ {⊥}) delivered_i ← {⊥, ..., ⊥}
3:      Array_[n](Boolean) proposed_i ← {false, ..., false}
4:      Set(Process) selected_i ← ∅
5:      Set(Process) ignored_i ← ∅
6:      Array_[n](Value ∪ {⊥}) output_i ← {⊥, ..., ⊥}
7:  Uses:
8:      Reliable Broadcast, instance RBC_j with source p_j, ∀j ∈ [n]
9:      Strong Binary Byzantine Agreement, instance BA_j, ∀j ∈ [n]
10: upon propose(v_i ∈ Value):
11:     invoke RBC_i.rb_broadcast(v_i)
12: upon first RBC_j.deliver(v_j ∈ Value):
13:     delivered_i[p_j] ← v_j
14:     invoke BA_j.propose(1)
15:     proposed_i[p_j] ← true
16: upon BA_j.decide(b_j ∈ {0, 1}):
17:     if b_j = 1:
18:         selected_i ← selected_i ∪ {p_j}
19:     else:
20:         ignored_i ← ignored_i ∪ {p_j}
21: upon |selected_i| ≥ n − t:
22:     Let K = {k ∈ [n] | proposed_i[p_k] = false}
23:     for each k ∈ K:
24:         invoke BA_k.propose(0)
25:         proposed_i[p_k] ← true
26: upon |selected_i ∪ ignored_i| = n:
27:     wait until ∀p_j ∈ selected_i, delivered_i[p_j] ≠ ⊥
28:     for each p_j ∈ selected_i:
29:         output_i[j] ← delivered_i[p_j]
30:     trigger decide(output_i)
```

PROOF. **(a) At least $n - t$ non-$\perp$ values.** Two cases arise:

(1) No correct process ever proposes 0 in any $\mathcal{BA}_j$ instance. By strong unanimity validity of binary agreement, no instance can decide 0. Therefore, once a correct process p_i decides (line 30), it holds that $|selected_i| = n$.

(2) Some correct process eventually proposes 0 to a binary instance (line 24). Let p_j be the first such process. By construction, this only occurs once $|selected_j| \geq n - t$ (line 21). By Agreement of binary consensus, all correct processes decide with the same $selected_i = selected_j$, and thus $|selected_i| \geq n - t$.

Algorithm 6 Vector Consensus: Pseudocode (for process p_i)

```
1:  Local variables:
2:      Array_[n](Value ∪ {⊥}) delivered_i ← {⊥, ..., ⊥}
3:      Array_[n](Boolean) proposed_i ← {false, ..., false}
4:      Set(Process) selected_i ← ∅
5:      Set(Process) ignored_i ← ∅
6:      Array_[n](Value ∪ {⊥}) output_i ← {⊥, ..., ⊥}
7:  Uses:
8:      Reliable Broadcast, instance RBC_j with source p_j, ∀j ∈ [n]
9:      Strong Binary Byzantine Agreement, instance BA_j, ∀j ∈ [n]
10: upon propose(v_i ∈ Value):
11:     invoke RBC_i.rb_broadcast(v_i)
12: upon first RBC_j.deliver(v_j ∈ Value):
13:     delivered_i[p_j] ← v_j
14:     invoke BA_j.propose(1)
15:     proposed_i[p_j] ← true
16: upon BA_j.decide(b_j ∈ {0, 1}):
17:     if b_j = 1:
18:         selected_i ← selected_i ∪ {p_j}
19:     else:
20:         ignored_i ← ignored_i ∪ {p_j}
21: upon |selected_i| ≥ n − t:
22:     Let K = {k ∈ [n] | proposed_i[p_k] = false}
23:     for each k ∈ K:
24:         invoke BA_k.propose(0)
25:         proposed_i[p_k] ← true
26: upon |selected_i ∪ ignored_i| = n:
27:     wait until ∀p_j ∈ selected_i, delivered_i[p_j] ≠ ⊥
28:     for each p_j ∈ selected_i:
29:         output_i[j] ← delivered_i[p_j]
30:     trigger decide(output_i)
```

PROOF. **(a) At least $n - t$ non-$\perp$ values.** Two cases arise:

In both cases, every correct process decides with at least $n - t$ indices in $selected_i$, and since it assigns non-$\perp$ values to these indices (lines 28–29), the decided vector contains at least $n - t$ non-$\perp$ entries.

Algorithm 6 Vector Consensus: Pseudocode (for process p_i)

```
1:  Local variables:
2:      Array_[n](Value ∪ {⊥}) delivered_i ← {⊥, ..., ⊥}
3:      Array_[n](Boolean) proposed_i ← {false, ..., false}
4:      Set(Process) selected_i ← ∅
5:      Set(Process) ignored_i ← ∅
6:      Array_[n](Value ∪ {⊥}) output_i ← {⊥, ..., ⊥}
7:  Uses:
8:      Reliable Broadcast, instance RBC_j with source p_j, ∀j ∈ [n]
9:      Strong Binary Byzantine Agreement, instance BA_j, ∀j ∈ [n]
10: upon propose(v_i ∈ Value):
11:     invoke RBC_i.rb_broadcast(v_i)
12: upon first RBC_j.deliver(v_j ∈ Value):
13:     delivered_i[p_j] ← v_j
14:     invoke BA_j.propose(1)
15:     proposed_i[p_j] ← true
16: upon BA_j.decide(b_j ∈ {0, 1}):
17:     if b_j = 1:
18:         selected_i ← selected_i ∪ {p_j}
19:     else:
20:         ignored_i ← ignored_i ∪ {p_j}
21: upon |selected_i| ≥ n − t:
22:     Let K = {k ∈ [n] | proposed_i[p_k] = false}
23:     for each k ∈ K:
24:         invoke BA_k.propose(0)
25:         proposed_i[p_k] ← true
26: upon |selected_i ∪ ignored_i| = n:
27:     wait until ∀p_j ∈ selected_i, delivered_i[p_j] ≠ ⊥
28:     for each p_j ∈ selected_i:
29:         output_i[j] ← delivered_i[p_j]
30:     trigger decide(output_i)
```

(b) Origin of non-⊥ entries. If $output_i[j] \neq \perp$ at decision time (line 30), then $p_j \in selected_i$ and $delivered_i[p_j] \neq \perp$ (line 27). By reliable broadcast Justification, a correct process can deliver a value from $\mathcal{RBC}_j$ only if p_j broadcast that value (line 11). Therefore, every non-⊥ entry in $output_i$ corresponds to the value proposed by its sender. □

Algorithm 6 Vector Consensus: Pseudocode (for process p_i)

```
1:  Local variables:
2:      Array_[n](Value ∪ {⊥}) delivered_i ← {⊥, ..., ⊥}
3:      Array_[n](Boolean) proposed_i ← {false, ..., false}
4:      Set(Process) selected_i ← ∅
5:      Set(Process) ignored_i ← ∅
6:      Array_[n](Value ∪ {⊥}) output_i ← {⊥, ..., ⊥}
7:  Uses:
8:      Reliable Broadcast, instance RBC_j with source p_j, ∀j ∈ [n]
9:      Strong Binary Byzantine Agreement, instance BA_j, ∀j ∈ [n]
10: upon propose(v_i ∈ Value):
11:     invoke RBC_i.rb_broadcast(v_i)
12: upon first RBC_j.deliver(v_j ∈ Value):
13:     delivered_i[p_j] ← v_j
14:     invoke BA_j.propose(1)
15:     proposed_i[p_j] ← true
16: upon BA_j.decide(b_j ∈ {0, 1}):
17:     if b_j = 1:
18:         selected_i ← selected_i ∪ {p_j}
19:     else:
20:         ignored_i ← ignored_i ∪ {p_j}
21: upon |selected_i| ≥ n − t:
22:     Let K = {k ∈ [n] | proposed_i[p_k] = false}
23:     for each k ∈ K:
24:         invoke BA_k.propose(0)
25:         proposed_i[p_k] ← true
26: upon |selected_i ∪ ignored_i| = n:
27:     wait until ∀p_j ∈ selected_i, delivered_i[p_j] ≠ ⊥
28:     for each p_j ∈ selected_i:
29:         output_i[j] ← delivered_i[p_j]
30:     trigger decide(output_i)
```

Termination

Proof. We first show that every correct process eventually proposes a value to each binary agreement. Assume, for contradiction, that no correct process ever reaches the condition $|selected_i| \geq n - t$ (line 21). Then, no correct process ever executes $\mathcal{BA}_k$.propose(0) (line 24). Hence, no value 0 is ever proposed in any binary agreement instance.

Algorithm 6 Vector Consensus: Pseudocode (for process p_i)

```
1:  Local variables:
2:      Array_[n](Value ∪ {⊥}) delivered_i ← {⊥, ..., ⊥}
3:      Array_[n](Boolean) proposed_i ← {false, ..., false}
4:      Set(Process) selected_i ← ∅
5:      Set(Process) ignored_i ← ∅
6:      Array_[n](Value ∪ {⊥}) output_i ← {⊥, ..., ⊥}
7:  Uses:
8:      Reliable Broadcast, instance RBC_j with source p_j, ∀j ∈ [n]
9:      Strong Binary Byzantine Agreement, instance BA_j, ∀j ∈ [n]
10: upon propose(v_i ∈ Value):
11:     invoke RBC_i.rb_broadcast(v_i)
12: upon first RBC_j.deliver(v_j ∈ Value):
13:     delivered_i[p_j] ← v_j
14:     invoke BA_j.propose(1)
15:     proposed_i[p_j] ← true
16: upon BA_j.decide(b_j ∈ {0, 1}):
17:     if b_j = 1:
18:         selected_i ← selected_i ∪ {p_j}
19:     else:
20:         ignored_i ← ignored_i ∪ {p_j}
21: upon |selected_i| ≥ n − t:
22:     Let K = {k ∈ [n] | proposed_i[p_k] = false}
23:     for each k ∈ K:
24:         invoke BA_k.propose(0)
25:         proposed_i[p_k] ← true
26: upon |selected_i ∪ ignored_i| = n:
27:     wait until ∀p_j ∈ selected_i, delivered_i[p_j] ≠ ⊥
28:     for each p_j ∈ selected_i:
29:         output_i[j] ← delivered_i[p_j]
30:     trigger decide(output_i)
```

By the *Obligation* property of reliable broadcast, every correct process eventually delivers the proposal of every correct sender p_j (line 13) and thus proposes 1 to the corresponding binary agreement instance (line 14). Since every correct process proposes 1 to all $\mathcal{BA}_j$ instances for correct p_j, the *Termination* and *Validity* properties of binary agreement imply that all correct processes eventually decide 1 for each of these instances. Consequently, each correct process eventually inserts at least $n - t$ processes into $selected_i$ (line 18), reaching $|selected_i| \geq n - t$, contradicting our assumption.

Algorithm 6 Vector Consensus: Pseudocode (for process p_i)

```
1:  Local variables:
2:      Array_[n](Value ∪ {⊥}) delivered_i ← {⊥, ..., ⊥}
3:      Array_[n](Boolean) proposed_i ← {false, ..., false}
4:      Set(Process) selected_i ← ∅
5:      Set(Process) ignored_i ← ∅
6:      Array_[n](Value ∪ {⊥}) output_i ← {⊥, ..., ⊥}
7:  Uses:
8:      Reliable Broadcast, instance RBC_j with source p_j, ∀j ∈ [n]
9:      Strong Binary Byzantine Agreement, instance BA_j, ∀j ∈ [n]
10: upon propose(v_i ∈ Value):
11:     invoke RBC_i.rb_broadcast(v_i)
12: upon first RBC_j.deliver(v_j ∈ Value):
13:     delivered_i[p_j] ← v_j
14:     invoke BA_j.propose(1)
15:     proposed_i[p_j] ← true
16: upon BA_j.decide(b_j ∈ {0, 1}):
17:     if b_j = 1:
18:         selected_i ← selected_i ∪ {p_j}
19:     else:
20:         ignored_i ← ignored_i ∪ {p_j}
21: upon |selected_i| ≥ n − t:
22:     Let K = {k ∈ [n] | proposed_i[p_k] = false}
23:     for each k ∈ K:
24:         invoke BA_k.propose(0)
25:         proposed_i[p_k] ← true
26: upon |selected_i ∪ ignored_i| = n:
27:     wait until ∀p_j ∈ selected_i, delivered_i[p_j] ≠ ⊥
28:     for each p_j ∈ selected_i:
29:         output_i[j] ← delivered_i[p_j]
30:     trigger decide(output_i)
```

Therefore, there exists at least one correct process that reaches the condition $|selected_i| \geq n - t$ (line 21). Let p_i be the first such process.

Algorithm 6 Vector Consensus: Pseudocode (for process p_i)

```
1:  Local variables:
2:      Array_[n](Value ∪ {⊥}) delivered_i ← {⊥, ..., ⊥}
3:      Array_[n](Boolean) proposed_i ← {false, ..., false}
4:      Set(Process) selected_i ← ∅
5:      Set(Process) ignored_i ← ∅
6:      Array_[n](Value ∪ {⊥}) output_i ← {⊥, ..., ⊥}
7:  Uses:
8:      Reliable Broadcast, instance RBC_j with source p_j, ∀j ∈ [n]
9:      Strong Binary Byzantine Agreement, instance BA_j, ∀j ∈ [n]
10: upon propose(v_i ∈ Value):
11:     invoke RBC_i.rb_broadcast(v_i)
12: upon first RBC_j.deliver(v_j ∈ Value):
13:     delivered_i[p_j] ← v_j
14:     invoke BA_j.propose(1)
15:     proposed_i[p_j] ← true
16: upon BA_j.decide(b_j ∈ {0, 1}):
17:     if b_j = 1:
18:         selected_i ← selected_i ∪ {p_j}
19:     else:
20:         ignored_i ← ignored_i ∪ {p_j}
21: upon |selected_i| ≥ n − t:
22:     Let K = {k ∈ [n] | proposed_i[p_k] = false}
23:     for each k ∈ K:
24:         invoke BA_k.propose(0)
25:         proposed_i[p_k] ← true
26: upon |selected_i ∪ ignored_i| = n:
27:     wait until ∀p_j ∈ selected_i, delivered_i[p_j] ≠ ⊥
28:     for each p_j ∈ selected_i:
29:         output_i[j] ← delivered_i[p_j]
30:     trigger decide(output_i)
```

By the *strong unanimity validity* property of binary agreement, for every $p_j \in selected_i$, there exists at least one correct process p_k that proposed 1 to $\mathcal{BA}_j$ (line 14). By the construction of the algorithm, p_k could only propose 1 after reliably delivering the corresponding value from $\mathcal{RBC}_j$ (lines 12–15). By the *Totality* property of reliable broadcast, if one correct process delivers the value from $\mathcal{RBC}_j$, then all correct processes eventually deliver the same value. Hence, every correct process eventually delivers the value broadcast by each $p_j \in selected_i$.

Algorithm 6 Vector Consensus: Pseudocode (for process p_i)

```
1:  Local variables:
2:      Array_[n](Value ∪ {⊥}) delivered_i ← {⊥, ..., ⊥}
3:      Array_[n](Boolean) proposed_i ← {false, ..., false}
4:      Set(Process) selected_i ← ∅
5:      Set(Process) ignored_i ← ∅
6:      Array_[n](Value ∪ {⊥}) output_i ← {⊥, ..., ⊥}
7:  Uses:
8:      Reliable Broadcast, instance RBC_j with source p_j, ∀j ∈ [n]
9:      Strong Binary Byzantine Agreement, instance BA_j, ∀j ∈ [n]
10: upon propose(v_i ∈ Value):
11:     invoke RBC_i.rb_broadcast(v_i)
12: upon first RBC_j.deliver(v_j ∈ Value):
13:     delivered_i[p_j] ← v_j
14:     invoke BA_j.propose(1)
15:     proposed_i[p_j] ← true
16: upon BA_j.decide(b_j ∈ {0, 1}):
17:     if b_j = 1:
18:         selected_i ← selected_i ∪ {p_j}
19:     else:
20:         ignored_i ← ignored_i ∪ {p_j}
21: upon |selected_i| ≥ n − t:
22:     Let K = {k ∈ [n] | proposed_i[p_k] = false}
23:     for each k ∈ K:
24:         invoke BA_k.propose(0)
25:         proposed_i[p_k] ← true
26: upon |selected_i ∪ ignored_i| = n:
27:     wait until ∀p_j ∈ selected_i, delivered_i[p_j] ≠ ⊥
28:     for each p_j ∈ selected_i:
29:         output_i[j] ← delivered_i[p_j]
30:     trigger decide(output_i)
```

Every correct process will eventually propose to the corresponding binary agreement instances (either 1 at line 14 or 0 at line 24), ensuring that every $\mathcal{BA}_j$ instance receives an input from every correct process. By binary agreement *Termination*, all these instances eventually decide. Moreover, by *Agreement*, they decide the same bit across all correct processes. For all indices $j \in selected_i$, this decision is 1, since at least one correct process proposed 1 in each corresponding instance. Thus, every correct process eventually satisfies $|selected_*| \geq n - t$ and proposes to every binary agreement instance.

Algorithm 6 Vector Consensus: Pseudocode (for process p_i)

```
1:  Local variables:
2:      Array_[n](Value ∪ {⊥}) delivered_i ← {⊥, ..., ⊥}
3:      Array_[n](Boolean) proposed_i ← {false, ..., false}
4:      Set(Process) selected_i ← ∅
5:      Set(Process) ignored_i ← ∅
6:      Array_[n](Value ∪ {⊥}) output_i ← {⊥, ..., ⊥}
7:  Uses:
8:      Reliable Broadcast, instance RBC_j with source p_j, ∀j ∈ [n]
9:      Strong Binary Byzantine Agreement, instance BA_j, ∀j ∈ [n]
10: upon propose(v_i ∈ Value):
11:     invoke RBC_i.rb_broadcast(v_i)
12: upon first RBC_j.deliver(v_j ∈ Value):
13:     delivered_i[p_j] ← v_j
14:     invoke BA_j.propose(1)
15:     proposed_i[p_j] ← true
16: upon BA_j.decide(b_j ∈ {0, 1}):
17:     if b_j = 1:
18:         selected_i ← selected_i ∪ {p_j}
19:     else:
20:         ignored_i ← ignored_i ∪ {p_j}
21: upon |selected_i| ≥ n − t:
22:     Let K = {k ∈ [n] | proposed_i[p_k] = false}
23:     for each k ∈ K:
24:         invoke BA_k.propose(0)
25:         proposed_i[p_k] ← true
26: upon |selected_i ∪ ignored_i| = n:
27:     wait until ∀p_j ∈ selected_i, delivered_i[p_j] ≠ ⊥
28:     for each p_j ∈ selected_i:
29:         output_i[j] ← delivered_i[p_j]
30:     trigger decide(output_i)
```

Finally, since all $\mathcal{BA}_j$ instances eventually decide and each decision is consistent across correct processes, every correct process eventually reaches the state $|selected_i \cup ignored_i| = n$ (line 26) and subsequently decides (line 30). By reliable broadcast *Totality*, the values associated with the decided set $selected_*$ are eventually delivered by all correct processes. □