Lecture 21: Distributed Algorithms

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

July 8, 2024

Overview

- Distributed Computing: The Models
- Consensus with Byzantine Failures
- Conclusion
- Acknowledgements

What are Distributed Algorithms?

- Algorithms which run on a network, or multiprocessors within a computer which share memory

What are Distributed Algorithms?

- Algorithms which run on a network, or multiprocessors within a computer which share memory
- Problems they solve:
    - Resource Management
    - Data Management and Transmission
    - Synchronization
    - Consensus
    - many more

What are Distributed Algorithms?

- Algorithms which run on a network, or multiprocessors within a computer which share memory
- Problems they solve:
  - Resource Management
  - Data Management and Transmission
  - Synchronization
  - Consensus
  - many more
- Challenges in this setting:
  - Concurrent Activity
  - Uncertainty of order of events
  - Failure and recovery of processors or channels

What are Distributed Algorithms?

- Algorithms which run on a network, or multiprocessors within a computer which share memory
- Problems they solve:
  - Resource Management
  - Data Management and Transmission
  - Synchronization
  - Consensus
  - many more
- Challenges in this setting:
  - Concurrent Activity
  - Uncertainty of order of events
  - Failure and recovery of processors or channels
- Many models
  - *Memory & Communication*: shared memory, message-passing
  - *Timing*: synchronous (rounds), asynchronous, partially synchronous (bounds on message delay, processor speeds, clock rates)
  - *Failures*: processor (stop, Byzantine), communication (message loss/altered), system state corruption

Synchronous Model

- processors are vertices of directed graph
  - *Memory*: each processor has its own memory
  - *Communication*: each processor can send messages to its *outgoing* neighbours
  - *Timing*: processors communicate in synchronous rounds
  - *Failures*: may or may not have failures (different settings today)

Synchronous Model

- processors are vertices of directed graph
  - *Memory*: each processor has its own memory
  - *Communication*: each processor can send messages to its *outgoing* neighbours
  - *Timing*: processors communicate in synchronous rounds
  - *Failures*: may or may not have failures (different settings today)
- Σ is the message alphabet, plus special symbol $\perp$

Synchronous Model

- processors are vertices of directed graph
  - *Memory*: each processor has its own memory
  - *Communication*: each processor can send messages to its *outgoing* neighbours
  - *Timing*: processors communicate in synchronous rounds
  - *Failures*: may or may not have failures (different settings today)
- Σ is the message alphabet, plus special symbol $\perp$
- For each vertex $i \in [n]$, a processor consists of:
  - S_i = non-empty set of states
  - σ_i = a start state
  - $\mu_i : S_i \times out_i \to \Sigma \cup \{\perp\}$ — Message function
  - $\tau_i : S_i \times (\Sigma \cup \{\perp\})^{in_i} \to S_i$ — Transition function

Synchronous Model

- processors are vertices of directed graph
  - *Memory*: each processor has its own memory
  - *Communication*: each processor can send messages to its *outgoing* neighbours
  - *Timing*: processors communicate in synchronous rounds
  - *Failures*: may or may not have failures (different settings today)
- Σ is the message alphabet, plus special symbol $\perp$
- For each vertex $i \in [n]$, a processor consists of:
  - S_i = non-empty set of states
  - σ_i = a start state
  - $\mu_i : S_i \times out_i \to \Sigma \cup \{\perp\}$ Message function
  - $\tau_i : S_i \times (\Sigma \cup \{\perp\})^{in_i} \to S_i$ Transition function
- Complexity Measure: *number of rounds* (*total data communicated*) needed to solve problem
  - processors have *unlimited internal resources* (i.e., can compute anything)
  - For today, will assume each processor deterministic

Example: Leader Election (i.e. breaking symmetry)

- Input: network of processors
- Output: want to distinguish exactly one process, as the *leader*

Example: Leader Election (i.e. breaking symmetry)

- Input: network of processors
- Output: want to distinguish exactly one process, as the *leader*
- Motivation: leader can take charge of
  - communication
  - coordination
  - allocating resources
  - other tasks

Example: Leader Election (i.e. breaking symmetry)

- Input: network of processors
- Output: want to distinguish exactly one process, as the *leader*
- Motivation: leader can take charge of
    - communication
    - coordination
    - allocating resources
    - other tasks
- Simple case: ring network, bi-directional communication
- processors numbered clockwise (but they don't know their numbers)

Example: Leader Election (i.e. breaking symmetry)

- Input: network of processors
- Output: want to distinguish exactly one process, as the *leader*
- Motivation: leader can take charge of
  - communication
  - coordination
  - allocating resources
  - other tasks
- Simple case: ring network, bi-directional communication
- processors numbered clockwise (but they don't know their numbers)
- **Fact:** all processors identical (same set of states and transition functions) and deterministic then it is *impossible* to elect a leader!

Example: Leader Election (i.e. breaking symmetry)

- Input: network of processors
- Output: want to distinguish exactly one process, as the *leader*
- Motivation: leader can take charge of
  - communication
  - coordination
  - allocating resources
  - other tasks
- Simple case: ring network, bi-directional communication
- processors numbered clockwise (but they don't know their numbers)
- **Fact:** all processors identical (same set of states and transition functions) and deterministic then it is *impossible* to elect a leader!
- To show this, simply look at execution and check that all processors will always be at identical states.

Leader Election: Algorithm

- Let's assume that each processor also has a unique ID (UID)
- But they don't know size of the network (i.e. n)

Leader Election: Algorithm

- Let's assume that each processor also has a unique ID (UID)
- But they don't know size of the network (i.e. n)
  - Idea: each processor sends its UID in a message, to be relayed step-by-step around the ring.

Leader Election: Algorithm

- Let's assume that each processor also has a unique ID (UID)
- But they don't know size of the network (i.e. n)
  - Idea: each processor sends its UID in a message, to be relayed step-by-step around the ring.
  - When processor receives UID, compares it with its own
    - if it is bigger, pass it on
    - if smaller, discard
    - equal $\Rightarrow$ processor declares itself leader
    - leader then notifies everyone else (by message relaying in network)

Leader Election: Algorithm

- Let's assume that each processor also has a unique ID (UID)
- But they don't know size of the network (i.e. n)
  - Idea: each processor sends its UID in a message, to be relayed step-by-step around the ring.
  - When processor receives UID, compares it with its own
    - if it is bigger, pass it on
    - if smaller, discard
    - equal $\Rightarrow$ processor declares itself leader
    - leader then notifies everyone else (by message relaying in network)
- Algorithm terminates, and elects leader with largest UID

Leader Election: Algorithm

- Let's assume that each processor also has a unique ID (UID)
- But they don't know size of the network (i.e. n)
  - Idea: each processor sends its UID in a message, to be relayed step-by-step around the ring.
  - When processor receives UID, compares it with its own
    - if it is bigger, pass it on
    - if smaller, discard
    - equal $\Rightarrow$ processor declares itself leader
    - leader then notifies everyone else (by message relaying in network)
- Algorithm terminates, and elects leader with largest UID
- After n rounds, element with maximum UID will declare itself the leader (and no other processor will)

Leader Election: Algorithm

- Let's assume that each processor also has a unique ID (UID)
- But they don't know size of the network (i.e. n)
  - Idea: each processor sends its UID in a message, to be relayed step-by-step around the ring.
  - When processor receives UID, compares it with its own
    - if it is bigger, pass it on
    - if smaller, discard
    - equal $\Rightarrow$ processor declares itself leader
    - leader then notifies everyone else (by message relaying in network)
- Algorithm terminates, and elects leader with largest UID
- After n rounds, element with maximum UID will declare itself the leader (and no other processor will)
- Complexity:
  - Number of rounds: $O(n)$
  - Communication: $O(n^2)$

Leader Election: Algorithm

- Let's assume that each processor also has a unique ID (UID)
- But they don't know size of the network (i.e. n)
  - Idea: each processor sends its UID in a message, to be relayed step-by-step around the ring.
  - When processor receives UID, compares it with its own
    - if it is bigger, pass it on
    - if smaller, discard
    - equal $\Rightarrow$ processor declares itself leader
    - leader then notifies everyone else (by message relaying in network)
- Algorithm terminates, and elects leader with largest UID
- After n rounds, element with maximum UID will declare itself the leader (and no other processor will)
- Complexity:
  - Number of rounds: $O(n)$
  - Communication: $O(n^2)$
- Can reduce communication to $O(n \log n)$ by successively doubling (see reference)

- Distributed Computing: The Models
- Consensus with Byzantine Failures
- Conclusion
- Acknowledgements

Consensus Problem - Setup

- Several generals and their armies surround an enemy city

Consensus Problem - Setup

- Several generals and their armies surround an enemy city
- Generals want to plan a coordinated attack to an enemy

Consensus Problem - Setup

- Several generals and their armies surround an enemy city
- Generals want to plan a coordinated attack to an enemy
- Some generals may not have their armies ready...

Consensus Problem - Setup

- Several generals and their armies surround an enemy city
- Generals want to plan a coordinated attack to an enemy
- Some generals may not have their armies ready...
- Generals can communicate by sending messengers to others' bases
    - Unreliable, as messenger can get lost or captured
    - Routes between bases are undirected graph, known to all generals
    - know bound on time it takes for message to be delivered successfully

Consensus Problem - Setup

- Several generals and their armies surround an enemy city
- Generals want to plan a coordinated attack to an enemy
- Some generals may not have their armies ready...
- Generals can communicate by sending messengers to others' bases
    - Unreliable, as messenger can get lost or captured
    - Routes between bases are undirected graph, known to all generals
    - know bound on time it takes for message to be delivered successfully
- For them to attack, *all generals* must *agree to attack*

Consensus Problem - Setup

- Several generals and their armies surround an enemy city
- Generals want to plan a coordinated attack to an enemy
- Some generals may not have their armies ready...
- Generals can communicate by sending messengers to others' bases
  - Unreliable, as messenger can get lost or captured
  - Routes between bases are undirected graph, known to all generals
  - know bound on time it takes for message to be delivered successfully
- For them to attack, *all generals* must *agree to attack*
- Model: synchronous model, arbitrary number of message failures.

Consensus Problem - Setup

- Several generals and their armies surround an enemy city
- Generals want to plan a coordinated attack to an enemy
- Some generals may not have their armies ready...
- Generals can communicate by sending messengers to others' bases
  - Unreliable, as messenger can get lost or captured
  - Routes between bases are undirected graph, known to all generals
  - know bound on time it takes for message to be delivered successfully
- For them to attack, *all generals* must *agree to attack*
- Model: synchronous model, arbitrary number of message failures.
- **Input**: Each processor has one bit. 1 (attack) or 0 (don't attack)
- **Output**: *same decision bit* b satisfying *strong validity*.
  - if all processors start with bit b, then b is only allowed decision [1]
  - if all start with 1 and *all messages successfully delivered*, then 1 is the only allowed decision.

[1] Weak validity: the agreed upon output should be the initial value of some non-faulty processor

Consensus Problem - Byzantine Failures

- Unbounded message failures $\Rightarrow$ impossible, even for 2 generals
- In the end $\rightarrow$ have to make a decision without communicating

Consensus Problem - Byzantine Failures

- Unbounded message failures $\Rightarrow$ impossible, even for 2 generals
- In the end $\rightarrow$ have to make a decision without communicating
- Not very illuminating.

What if we allow only a finite number of failures?

Consensus Problem - Byzantine Failures

- Unbounded message failures $\Rightarrow$ impossible, even for 2 generals
- In the end $\rightarrow$ have to make a decision without communicating
- Not very illuminating.

What if we allow only a finite number of failures?

- Two types of failures:
  - Stopping Failures: all generals honest, but some may not be able to communicate at all (node crash in network)
  - *Byzantine Failures*: some generals *dishonest*. Similar to malicious attacker in a network.

Consensus Problem - Byzantine Failures

- Unbounded message failures $\Rightarrow$ impossible, even for 2 generals
- In the end $\rightarrow$ have to make a decision without communicating
- Not very illuminating.

What if we allow only a finite number of failures?

- Two types of failures:
  - Stopping Failures: all generals honest, but some may not be able to communicate at all (node crash in network)
  - *Byzantine Failures*: some generals *dishonest*. Similar to malicious attacker in a network.
- **Input**: Each processor has one bit of input. 1 (attack) or 0 (don't attack). Faulty processors can behave arbitrarily.

Consensus Problem - Byzantine Failures

- Unbounded message failures $\Rightarrow$ impossible, even for 2 generals
- In the end $\rightarrow$ have to make a decision without communicating
- Not very illuminating.

What if we allow only a finite number of failures?

- Two types of failures:
  - Stopping Failures: all generals honest, but some may not be able to communicate at all (node crash in network)
  - *Byzantine Failures*: some generals *dishonest*. Similar to malicious attacker in a network.
- **Input**: Each processor has one bit of input. 1 (attack) or 0 (don't attack). Faulty processors can behave arbitrarily.
- **Output**: all *non-faulty processors* should *terminate* and have
  1. *Agreement*: same decision bit b
  2. *Strong Validity*: if all *non-faulty processors* start with bit a, then b must be equal to a.

Consensus Problem - Byzantine Failures

- Unbounded message failures $\Rightarrow$ impossible, even for 2 generals
- In the end $\rightarrow$ have to make a decision without communicating
- Not very illuminating.

What if we allow only a finite number of failures?

- Two types of failures:
  - Stopping Failures: all generals honest, but some may not be able to communicate at all (node crash in network)
  - *Byzantine Failures*: some generals *dishonest*. Similar to malicious attacker in a network.
- **Input**: Each processor has one bit of input. 1 (attack) or 0 (don't attack). Faulty processors can behave arbitrarily.
- **Output**: all *non-faulty processors* should *terminate* and have
  1. *Agreement*: same decision bit b
  2. *Strong Validity*: if all *non-faulty processors* start with bit a, then b must be equal to a.
- Complexity measures: *number of rounds* & *communication* (# messages exchanged in bit-size).

Byzantine Consensus - Complete Graph

- Assume all vertices can talk to any other vertex ("broadcast" setting)

Byzantine Consensus - Complete Graph

- Assume all vertices can talk to any other vertex ("broadcast" setting)
- First attempt: simply send our value to other nodes (if non-faulty), then take majority.

Byzantine Consensus - Complete Graph

- Assume all vertices can talk to any other vertex ("broadcast" setting)
- First attempt: simply send our value to other nodes (if non-faulty), then take majority.
- Well, that didn't work - violated the *agreement* property!

Byzantine Consensus - Complete Graph

- Assume all vertices can talk to any other vertex ("broadcast" setting)
- First attempt: simply send our value to other nodes (if non-faulty), then take majority.
- Well, that didn't work - violated the *agreement* property!
- New Idea: make all nodes *gossip*!

  Each node now will keep track of what each node has told another and so on...
- At each round, each vertex broadcasts its knowledge
- After a number of rounds, everyone must make a decision

Byzantine Consensus - Complete Graph

- Assume all vertices can talk to any other vertex ("broadcast" setting)
- First attempt: simply send our value to other nodes (if non-faulty), then take majority.
- Well, that didn't work - violated the *agreement* property!
- New Idea: make all nodes *gossip*!

  Each node now will keep track of what each node has told another and so on...
- At each round, each vertex broadcasts its knowledge
- After a number of rounds, everyone must make a decision
- Does this work?
- How many rounds do we need?
- How many Byzantine failures can it tolerate?

Byzantine Consensus - Bad Example

- 3 vertices $\{v_1, v_2, v_3\}$, 1 faulty vertex
- Scenario 1: v_1, v_2 good with value 1, v_3 faulty with value 0
  1. Round 1: all vertices truthful
  2. Round 2: v_3 lies to v_1, saying that v_2 said 0, all other communications truthful
  3. Validity $\Rightarrow v_1, v_2$ must decide 1

Byzantine Consensus - Bad Example

- 3 vertices $\{v_1, v_2, v_3\}$, 1 faulty vertex
- Scenario 2: v_2, v_3 good with value 0, v_1 faulty with value 1
  1. Round 1: all vertices truthful
  2. Round 2: v_1 lies to v_3, saying that v_2 said 1, all other communications truthful
  3. Validity $\Rightarrow$ v_2, v_3 must decide 0

Byzantine Consensus - Bad Example

- 3 vertices $\{v_1, v_2, v_3\}$, 1 faulty vertex
- Scenario 3: v_1, v_3 good with values $1, 0$ (resp.), v_2 faulty with value 0
  1. Round 1: v_2 tells v_1 its value is 1, tells v_3 its value is 0
  2. Round 2: all truthful

Byzantine Consensus - Bad Example

- 3 vertices $\{v_1, v_2, v_3\}$, 1 faulty vertex
- Scenario 1: v_1, v_2 good with value 1, v_3 faulty with value 0
  1. Round 1: all vertices truthful
  2. Round 2: v_3 lies to v_1, saying that v_2 said 0, all other communications truthful
  3. Validity $\Rightarrow v_1, v_2$ must decide 1
- Scenario 2: v_2, v_3 good with value 0, v_1 faulty with value 1
  1. Round 1: all vertices truthful
  2. Round 2: v_1 lies to v_3, saying that v_2 said 1, all other communications truthful
  3. Validity $\Rightarrow v_2, v_3$ must decide 0
- Scenario 3: v_1, v_3 good with values $1, 0$ (resp.), v_2 faulty with value 0
  1. Round 1: v_2 tells v_1 its value is 1, tells v_3 its value is 0
  2. Round 2: all truthful
- Scenarios 1 and 3 identical to v_1, so it must return 1 (validity)
- Scenarios 2 and 3 identical to v_3, so it must return 0 (validity)
- Contradicts *agreement* in Scenario 3!

Byzantine Consensus - Algorithm

- Assumption:[2] $n > 3f$ (number of bad vertices $<$ third total vertices)

[2] It turns out that $n \leq 3f \Rightarrow$ *no algorithm* can reach consensus!

Byzantine Consensus - Algorithm

- Assumption:[2] $n > 3f$ (number of bad vertices $<$ third total vertices)
- How to perfectly gossip?

[2] It turns out that $n \leq 3f \Rightarrow$ *no algorithm* can reach consensus!

Byzantine Consensus - Algorithm

- Assumption:[2] $n > 3f$ (number of bad vertices $<$ third total vertices)
- How to perfectly gossip?
- Data structure: *Exponential Information Gathering* (EIG) tree $T_{n,f}$
  - Depth: $f + 1$ (so $f + 2$ node levels)
  - Each tree node at level $k + 1$ labeled by string $i_1 i_2 \cdots i_k$ ($i_a \neq i_b$)

[2] It turns out that $n \leq 3f \Rightarrow$ *no algorithm* can reach consensus!

Byzantine Consensus - Algorithm

- Assumption:[2] $n > 3f$ (number of bad vertices $<$ third total vertices)
- How to perfectly gossip?
- Data structure: *Exponential Information Gathering* (EIG) tree $T_{n,f}$
  - Depth: $f + 1$ (so $f + 2$ node levels)
  - Each tree node at level $k + 1$ labeled by string $i_1 i_2 \cdots i_k$ ($i_a \neq i_b$)
  - Node $i_1 i_2 \cdots i_k$ will store value v if the following happens: i_k told you that i_{k-1} told i_k that i_{k-2} told i_{k-1} ... that i_1 told i_2 that its initial value was v

[2] It turns out that $n \leq 3f \Rightarrow$ *no algorithm* can reach consensus!

Byzantine Consensus - EIG Algorithm

1. Each vertex has own EIG tree $T_{n,f}$, with root labeled by its own value

Byzantine Consensus - EIG Algorithm

1. Each vertex has own EIG tree $T_{n,f}$, with root labeled by its own value
2. Relay messages for $f + 1$ rounds
   - At round r, each vertex sends the values of level r of its EIG tree
   - Each vertex decorates values of its $(r+1)^{th}$ level with values from messages

Byzantine Consensus - EIG Algorithm

1. Each vertex has own EIG tree $T_{n,f}$, with root labeled by its own value
2. Relay messages for $f + 1$ rounds
   - At round r, each vertex sends the values of level r of its EIG tree
   - Each vertex decorates values of its $(r+1)^{th}$ level with values from messages
3. After $f + 1$ rounds, redecorate tree bottom-up, taking strict majority of children (otherwise set value of tree node to $\perp$)

EIG Algorithm - Example

- $n = 4$, $f = 1$
- p_3 is faulty, initial values are $p_1 = p_2 = 1$, $p_3 = p_4 = 0$
- round 1: p_3 lies to p_2 and p_4
- round 2: p_3 lies to p_2 about p_1 and lies to p_1 about p_2

EIG Algorithm - Analysis

Lemma (Consistency of Non-Faulty Messages)

If i, j, k are non-faulty, then $T_i(x) = T_j(x)$ whenever label x ends with k.

(This is value of the tree before relabeling)

EIG Algorithm - Analysis

Lemma (Consistency of Upwards Relabeling)

If label x ends with non-faulty process, then for any two non-faulty processors i, j the new values of $T_i(x)$ and $T_j(x)$ are the same.

EIG Algorithm - Analysis

Lemma (Consistency of Upwards Relabeling)

If label x ends with non-faulty process, then for any two non-faulty processors i, j the new values of $T_i(x)$ and $T_j(x)$ are the same.

- Base case: if x is the label of leaf, previous lemma handles it.

EIG Algorithm - Analysis

Lemma (Consistency of Upwards Relabeling)

If label x ends with non-faulty process, then for any two non-faulty processors i, j the new values of $T_i(x)$ and $T_j(x)$ are the same.

- Base case: if x is the label of leaf, previous lemma handles it.
- Inductive step: $|x| = t < f$ (x not a leaf)
  - By induction, if ℓ is a non-faulty element the new value of $T_i(x \circ \ell)$ is the same for any non-faulty $i \in [n]$.

EIG Algorithm - Analysis

Lemma (Consistency of Upwards Relabeling)

If label x ends with non-faulty process, then for any two non-faulty processors i, j the new values of $T_i(x)$ and $T_j(x)$ are the same.

- Base case: if x is the label of leaf, previous lemma handles it.
- Inductive step: $|x| = t < f$ (x not a leaf)
  - By induction, if ℓ is a non-faulty element the new value of $T_i(x \circ \ell)$ is the same for any non-faulty $i \in [n]$.
  - So label x has same labeled "honest" children across trees

EIG Algorithm - Analysis

Lemma (Consistency of Upwards Relabeling)

If label x ends with non-faulty process, then for any two non-faulty processors i, j the new values of $T_i(x)$ and $T_j(x)$ are the same.

- Base case: if x is the label of leaf, previous lemma handles it.
- Inductive step: $|x| = t < f$ (x not a leaf)
  - By induction, if ℓ is a non-faulty element the new value of $T_i(x \circ \ell)$ is the same for any non-faulty $i \in [n]$.
  - So label x has same labeled "honest" children across trees
  - Number of children of x:

$$= n - t > 3f - f = 2f$$

EIG Algorithm - Analysis

Lemma (Consistency of Upwards Relabeling)

If label x ends with non-faulty process, then for any two non-faulty processors i, j the new values of $T_i(x)$ and $T_j(x)$ are the same.

- Base case: if x is the label of leaf, previous lemma handles it.
- Inductive step: $|x| = t < f$ (x not a leaf)
  - By induction, if ℓ is a non-faulty element the new value of $T_i(x \circ \ell)$ is the same for any non-faulty $i \in [n]$.
  - So label x has same labeled "honest" children across trees
  - Number of children of x:

$$= n - t > 3f - f = 2f$$

  - At most f are faulty. By taking majority, we get that new values $T_i(x) = T_j(x)$

EIG Algorithm - Analysis

So far we have managed to prove:

1. *Termination*: after $f + 1$ rounds, all of them will decide.
    - every label x which has no faulty processor is able to update its value

EIG Algorithm - Analysis

So far we have managed to prove:

1. *Termination*: after $f + 1$ rounds, all of them will decide.
   - every label x which has no faulty processor is able to update its value
2. *Validity*: if all nodes start with b, then each label x with no faulty processor will be updated to b
   - proof analogous to the proof of previous lemma
   - just note that all values will be b, as it is value being propagated by non-faulty nodes

EIG Algorithm - Analysis

So far we have managed to prove:

1. *Termination*: after $f + 1$ rounds, all of them will decide.
   - every label x which has no faulty processor is able to update its value
2. *Validity*: if all nodes start with b, then each label x with no faulty processor will be updated to b
   - proof analogous to the proof of previous lemma
   - just note that all values will be b, as it is value being propagated by non-faulty nodes
3. *Agreement*: all nodes must agree on same value
   - By first lemma, all values in the leaves x are consistent across processors so long as x ends on a non-faulty process
   - By second lemma, majority will cause all values in nodes from level r ending in non-faulty nodes to be *the same* across processors
   - Induction and $n > 3f$ ensures that labels in level 1 will look the same on non-faulty nodes $\Rightarrow$ agreement

Conclusion

- Today we learned about distributed computation
- It is cool
- Widely used in practice
  - Cryptocurrencies - all of them need to solve Byzantine Agreement!

    Happening at UW: Sergey Gorbunov (Algorand & Axelar)
  - Other peer-to-peer systems
  - Multi-core programming

    Happening at UW: Trevor Brown
  - Biology (social insect colony algorithms)
  - many more...
- Learned an (inefficient) algorithm for Byzantine Agreement (check out the more efficient one in [Attiya and Welch 2004])

Acknowledgement

- Lecture based largely on:
  - Nancy Lynch's 6.852 Fall 2015 course - lectures 1 and 6
  - Lecture 1

    https://learning-modules.mit.edu/service/materials/groups/103042/files/271154f5-ea0f-41a0-9ed9-6f83a5222d8b/link?errorRedirect=%2Fmaterials%2Findex.html&download=true
  - Lecture 6

    https://learning-modules.mit.edu/service/materials/groups/103042/files/95f71f5e-7791-4a1a-aeb5-e3d97afb167f/link?errorRedirect=%2Fmaterials%2Findex.html&download=true

References I

Attiya, H. and Welch, J., 2004.
Distributed computing: fundamentals, simulations, and advanced topics (Vol. 19).
John Wiley & Sons.