Impossibility of Distributed Consensus with One Faulty Process

The Weakest Failure Detector for Solving Consensus

October 22, 2015

The Consensus Problem

The Consensus Problem

- Set of n processes. Each process starts with a value

The Consensus Problem

- Set of n processes. Each process starts with a value
- Every correct process at the end outputs a value

The Consensus Problem

- Set of n processes. Each process starts with a value
- Every correct process at the end outputs a value

The solution must satisfy

- Termination : Every correct process must decide some value

The Consensus Problem

- Set of n processes. Each process starts with a value
- Every correct process at the end outputs a value

The solution must satisfy

- Termination : Every correct process must decide some value
- Validity : If all processes start with the same input value v, then the correct processes decide v

The Consensus Problem

- Set of n processes. Each process starts with a value
- Every correct process at the end outputs a value

The solution must satisfy

- Termination : Every correct process must decide some value
- Validity : If all processes start with the same input value v, then the correct processes decide v
- Agreement : Every correct process decides the same value

System Model

System Model

- Asynchronous processing : A process can take arbitrarily long to execute its next step

System Model

- Asynchronous processing : A process can take arbitrarily long to execute its next step
- Crash failures : A process cannot detect the failure of another process

System Model

- Asynchronous processing : A process can take arbitrarily long to execute its next step
- Crash failures : A process cannot detect the failure of another process
- Every message is eventually delivered, but can take arbitrarily long to reach or delivered out of order

System Model

System Model

- Message Buffer : Consists of messages sent, but not yet delivered. Supports send and receive operations

System Model

- Message Buffer : Consists of messages sent, but not yet delivered. Supports send and receive operations
- Configuration : Consists of the internal state of each process along with the state of the message buffer

System Model

- Message Buffer : Consists of messages sent, but not yet delivered. Supports send and receive operations
- Configuration : Consists of the internal state of each process along with the state of the message buffer
- Event : (p, m). Denotes the receipt of message m (possibly Φ) by p.

- Step : Consists of a step by a single process p, which is the change in its internal state based on event (p, m)

- Step : Consists of a step by a single process p, which is the change in its internal state based on event (p, m)
- Let C be a configuration. $e(C)$ denotes the resulting configuration on event e, if e can be applied.

- Step : Consists of a step by a single process p, which is the change in its internal state based on event (p, m)
- Let C be a configuration. $e(C)$ denotes the resulting configuration on event e, if e can be applied.
- Run : A sequence of steps (or events) σ

- Step : Consists of a step by a single process p, which is the change in its internal state based on event (p, m)
- Let C be a configuration. $e(C)$ denotes the resulting configuration on event e, if e can be applied.
- Run : A sequence of steps (or events) σ
- A configuration C' is reachable from C, if there exists a from C that ends in C'

- Step : Consists of a step by a single process p, which is the change in its internal state based on event (p, m)
- Let C be a configuration. $e(C)$ denotes the resulting configuration on event e, if e can be applied.
- Run : A sequence of steps (or events) σ
- A configuration C' is reachable from C, if there exists a from C that ends in C'
- Deciding Run : A run is a deciding run if some process reaches a decision in that run

- Bivalent Configuration : A configuration from which runs deciding both 0 and 1 are possible

- Bivalent Configuration : A configuration from which runs deciding both 0 and 1 are possible
- Univalent Configuration : A configuration from which runs deciding either 0 or 1 are possible

- Bivalent Configuration : A configuration from which runs deciding both 0 and 1 are possible
- Univalent Configuration : A configuration from which runs deciding either 0 or 1 are possible
- 0(1)-valent configuration : A configuration from which runs deciding only 0(1) exist

Theorem

There is no consensus protocol that can tolerate the failure of one process

Theorem

There is no consensus protocol that can tolerate the failure of one process

What does impossibility mean? Any consensus protocol that respects validity and agreement conditions, must have a possible run, in which no correct process terminates.

Intuition : 2 process case

Intuition : 2 process case

Scenario 1:

Intuition : 2 process case

Scenario 1:

- p_1 starts with input 0

Intuition : 2 process case

Scenario 1:

- p_1 starts with input 0
- p_2 fails without executing any step

Intuition : 2 process case

Scenario 1:

- p_1 starts with input 0
- p_2 fails without executing any step
- p_1 decides 0 and terminates

Intuition : 2 process case

Scenario 1:

- p_1 starts with input 0
- p_2 fails without executing any step
- p_1 decides 0 and terminates

Scenario 2:

Intuition : 2 process case

Scenario 1:

- p_1 starts with input 0
- p_2 fails without executing any step
- p_1 decides 0 and terminates

Scenario 2:

- p_1 fails without executing any step

Intuition : 2 process case

Scenario 1:

- p_1 starts with input 0
- p_2 fails without executing any step
- p_1 decides 0 and terminates

Scenario 2:

- p_1 fails without executing any step
- p_2 starts with input 1

Intuition : 2 process case

Scenario 1:

- p_1 starts with input 0
- p_2 fails without executing any step
- p_1 decides 0 and terminates

Scenario 2:

- p_1 fails without executing any step
- p_2 starts with input 1
- p_2 decides 1 and terminates

Intuition : 2 process case

Scenario 1:

- p_1 starts with input 0
- p_2 fails without executing any step
- p_1 decides 0 and terminates

Scenario 2:

- p_1 fails without executing any step
- p_2 starts with input 1
- p_2 decides 1 and terminates

Scenario 3:

Intuition : 2 process case

Scenario 1:

- p_1 starts with input 0
- p_2 fails without executing any step
- p_1 decides 0 and terminates

Scenario 2:

- p_1 fails without executing any step
- p_2 starts with input 1
- p_2 decides 1 and terminates

Scenario 3:

- p_1 starts with 0 and p_2 stars with 1

Intuition : 2 process case

Scenario 1:

- p_1 starts with input 0
- p_2 fails without executing any step
- p_1 decides 0 and terminates

Scenario 2:

- p_1 fails without executing any step
- p_2 starts with input 1
- p_2 decides 1 and terminates

Scenario 3:

- p_1 starts with 0 and p_2 stars with 1
- Messages take a long time to reach, so p_1's and p_2's view of the system is same as Scenario 1 and 2, resp.

Intuition : 2 process case

Scenario 1:

- p_1 starts with input 0
- p_2 fails without executing any step
- p_1 decides 0 and terminates

Scenario 2:

- p_1 fails without executing any step
- p_2 starts with input 1
- p_2 decides 1 and terminates

Scenario 3:

- p_1 starts with 0 and p_2 stars with 1
- Messages take a long time to reach, so p_1's and p_2's view of the system is same as Scenario 1 and 2, resp.
- p_1 decides 0 and p_2 decides 1

Proof of the Impossibility Result

Proof of the Impossibility Result

- The proof proceeds by contradiction. Suppose an algorithm P exists that solves consensus despite one failure

Proof of the Impossibility Result

- The proof proceeds by contradiction. Suppose an algorithm P exists that solves consensus despite one failure
- We show that P has a bivalent initial configuration

Proof of the Impossibility Result

- The proof proceeds by contradiction. Suppose an algorithm P exists that solves consensus despite one failure
- We show that P has a bivalent initial configuration
- Then we show that from every bivalent configuration, a possible sequence of events can again result in a bivalent configuration

Lemma

There exists a bivalent initial configuration of P

Lemma

There exists a bivalent initial configuration of P

Suppose not.

Lemma

There exists a bivalent initial configuration of P

Suppose not.Initial configuration $(0, 0, ..., 0$ is $0-$valent while $(1, 1, ..., 1$ is $1-$valent.

Lemma

There exists a bivalent initial configuration of P

Suppose not.Initial configuration $(0, 0, ..., 0$ is $0-$valent while $(1, 1, ..., 1$ is $1-$valent.
Take a path
$(0, 0, 0, ..., 0), (1, 0, 0, ..., 0), (1, 1, 0, ..., 0), ..., (1, 1, 1, ..., 1)$

Lemma

There exists a bivalent initial configuration of P

Suppose not.Initial configuration $(0, 0, ..., 0$ is $0-$valent while $(1, 1, ..., 1$ is $1-$valent.
Take a path
$(0, 0, 0, ..., 0), (1, 0, 0, ..., 0), (1, 1, 0, ..., 0), ..., (1, 1, 1, ..., 1)$
There exists two adjacent configurations in the path that are of different valency. And they differ in the input value of only one process *i*

Lemma

There exists a bivalent initial configuration of P

Suppose not.Initial configuration $(0, 0, ..., 0$ is $0-$valent while $(1, 1, ..., 1$ is $1-$valent.
Take a path
$(0, 0, 0, ..., 0), (1, 0, 0, ..., 0), (1, 1, 0, ..., 0), ..., (1, 1, 1, ..., 1)$
There exists two adjacent configurations in the path that are of different valency. And they differ in the input value of only one process i
Now construct a run where i crashes without taking any steps. Then, processes $< i$ decide on 0 and process $> i$ decide on 1.

Lemma

Let C be a bivalent configuration and $e = (p, m)$ be an event applicable to C. Then, there exists a bivalent configuration reachable from C in which e has been applied.

What to do now?

What to do now?

- Even if there is no “perfect” protocol, cases when processes do not terminate may be rare

What to do now?

- Even if there is no “perfect” protocol, cases when processes do not terminate may be rare
- Look for relaxation in the model or make extra assumptions

What to do now?

- Even if there is no "perfect" protocol, cases when processes do not terminate may be rare
- Look for relaxation in the model or make extra assumptions

One approach : Every process has access to a local failure detector module

What to do now?

- Even if there is no “perfect” protocol, cases when processes do not terminate may be rare
- Look for relaxation in the model or make extra assumptions

One approach : Every process has access to a local failure detector module

- The module need not be perfect. It can suspect a correct process to have failed or not suspect a failed process

We reason about failure detectors abstractly on the basis of properties they satisfy.

We reason about failure detectors abstractly on the basis of properties they satisfy.

- Strong Completeness : There is a time after which every process that crashes is suspected by **all** correct processes

We reason about failure detectors abstractly on the basis of properties they satisfy.

- Strong Completeness : There is a time after which every process that crashes is suspected by **all** correct processes
- Weak Completeness : There is a time after which every process that crashes is suspected by **some** correct processes

We reason about failure detectors abstractly on the basis of properties they satisfy.

- Strong Completeness : There is a time after which every process that crashes is suspected by **all** correct processes
- Weak Completeness : There is a time after which every process that crashes is suspected by **some** correct processes
- Perpetual Strong Accuracy : **Any** correct process is never suspected by any process

We reason about failure detectors abstractly on the basis of properties they satisfy.

- Strong Completeness : There is a time after which every process that crashes is suspected by **all** correct processes
- Weak Completeness : There is a time after which every process that crashes is suspected by **some** correct processes
- Perpetual Strong Accuracy : **Any** correct process is never suspected by any process
- Perpetual Weak Accuracy : **Some** correct process is never suspected by any process

We reason about failure detectors abstractly on the basis of properties they satisfy.

- Strong Completeness : There is a time after which every process that crashes is suspected by **all** correct processes
- Weak Completeness : There is a time after which every process that crashes is suspected by **some** correct processes
- Perpetual Strong Accuracy : **Any** correct process is never suspected by any process
- Perpetual Weak Accuracy : **Some** correct process is never suspected by any process
- Eventual Strong Accuracy : **There is a time** after which **any** correct process is never suspected by any correct process

We reason about failure detectors abstractly on the basis of properties they satisfy.

- Strong Completeness : There is a time after which every process that crashes is suspected by **all** correct processes
- Weak Completeness : There is a time after which every process that crashes is suspected by **some** correct processes
- Perpetual Strong Accuracy : **Any** correct process is never suspected by any process
- Perpetual Weak Accuracy : **Some** correct process is never suspected by any process
- Eventual Strong Accuracy : **There is a time** after which **any** correct process is never suspected by any correct process
- Eventual Weak Accuracy : **There is a time** after which **some** correct process is never suspected by any correct process

Why do we need both Completeness and Accuracy properties?

Why do we need both Completeness and Accuracy properties?

- A failure detector that suspects all the processes is complete

Why do we need both Completeness and Accuracy properties?

- A failure detector that suspects all the processes is complete
- A failure detector that never suspects any process is accurate

Why do we need both Completeness and Accuracy properties?

- A failure detector that suspects all the processes is complete
- A failure detector that never suspects any process is accurate

And both of these are useless!

Eventually weak failure detector, $\diamond W$

Eventually weak failure detector, $\diamond W$

- [**Weak Completeness**] : After some time, every process that crashes is suspected by some correct process

Eventually weak failure detector, $\diamond W$

- [**Weak Completeness**] : After some time, every process that crashes is suspected by some correct process
- [**Eventual Weak Accuracy**] : After some time, some correct process is never suspected by any correct process

Eventually weak failure detector, $\diamond W$

- [**Weak Completeness**] : After some time, every process that crashes is suspected by some correct process
- [**Eventual Weak Accuracy**] : After some time, some correct process is never suspected by any correct process

These are examples of eventually forever properties : Properties that forever hold true after some finite amount of time

Theorem

It is possible to solve consensus using $\diamond W$ *if* $n > 2f$

Theorem

It is possible to solve consensus using $\diamond W$ *if* $n > 2f$

Theorem

W *is the weakest failure detector that solves consensus*

Theorem

It is possible to solve consensus using $\diamond W$ *if* $n > 2f$

Theorem

W is the weakest failure detector that solves consensus

What do we mean by the "weakest" failure detector?

Theorem

It is possible to solve consensus using $\diamond W$ *if* $n > 2f$

Theorem

W is the weakest failure detector that solves consensus

What do we mean by the "weakest" failure detector?
Any failure detector that solves consensus with $n > 2f$ can emulate $\diamond W$

A practical implementation of $\diamond W$

A practical implementation of $\diamond W$

- Every process sends "I am alive" messages periodically

A practical implementation of $\diamond W$

- Every process sends "I am alive" messages periodically
- If a process p does not hear from another process q for some time, it adds q to the list of processes suspected to have failed

A practical implementation of $\diamond W$

- Every process sends "I am alive" messages periodically
- If a process p does not hear from another process q for some time, it adds q to the list of processes suspected to have failed
- If p later receives the "I am alive" message from q, it removes q from its list and increases length of timeout for q

A practical implementation of $\diamond W$

- Every process sends "I am alive" messages periodically
- If a process p does not hear from another process q for some time, it adds q to the list of processes suspected to have failed
- If p later receives the "I am alive" message from q, it removes q from its list and increases length of timeout for q

Works well in practice, but does not guarantee $\diamond W$

Solving consensus using $\diamond W$

Outline of the Algorithm

Solving consensus using $\diamond W$

Outline of the Algorithm

- Proceeds in rounds. Each round has a coordinator that rotates among the set of processes

Solving consensus using $\diamond W$

Outline of the Algorithm

- Proceeds in rounds. Each round has a coordinator that rotates among the set of processes
- In each round all messages are sent to or from the coordinator

Solving consensus using $\diamond W$

Outline of the Algorithm

- Proceeds in rounds. Each round has a coordinator that rotates among the set of processes
- In each round all messages are sent to or from the coordinator
- In each round, the coordinator tries to determine a consistent value

Solving consensus using $\diamond W$

Outline of the Algorithm

- Proceeds in rounds. Each round has a coordinator that rotates among the set of processes
- In each round all messages are sent to or from the coordinator
- In each round, the coordinator tries to determine a consistent value
- If in a round, the coordinator is not suspected by any correct process, then it succeeds

Solving consensus using $\diamond W$

Outline of the Algorithm

- Proceeds in rounds. Each round has a coordinator that rotates among the set of processes
- In each round all messages are sent to or from the coordinator
- In each round, the coordinator tries to determine a consistent value
- If in a round, the coordinator is not suspected by any correct process, then it succeeds
- Otherwise, the algorithm enters the next round

Weakest failure detector

Instead of emulating $\diamond W$, we show that any failure detector can emulate Ω (defined below) which can in turn emulate $\diamond W$

Weakest failure detector

Instead of emulating $\diamond W$, we show that any failure detector can emulate Ω (defined below) which can in turn emulate $\diamond W$

Definition

A failure detector Ω satisfies the following properties :

- Its output at a process p is a single process q that p trusts to be correct at that time

Weakest failure detector

Instead of emulating $\diamond W$, we show that any failure detector can emulate Ω (defined below) which can in turn emulate $\diamond W$

Definition

A failure detector Ω satisfies the following properties :

- Its output at a process p is a single process q that p trusts to be correct at that time
- There is a time after which all correct processes trust the same correct process

Weakest failure detector

Instead of emulating $\diamond W$, we show that any failure detector can emulate Ω (defined below) which can in turn emulate $\diamond W$

Definition

A failure detector Ω satisfies the following properties :

- Its output at a process p is a single process q that p trusts to be correct at that time
- There is a time after which all correct processes trust the same correct process

- Easy to see that Ω is at least as strong as $\diamond W$

Weakest failure detector

Instead of emulating $\diamond W$, we show that any failure detector can emulate Ω (defined below) which can in turn emulate $\diamond W$

Definition

A failure detector Ω satisfies the following properties :

- Its output at a process p is a single process q that p trusts to be correct at that time
- There is a time after which all correct processes trust the same correct process

- Easy to see that Ω is at least as strong as $\diamond W$
- An emulator for $\diamond W$ using Ω outputs the set of processes that are not trusted in Ω

Construction Outline

Construction Outline

- Every process maintains a DAG which models a causal relation between queries to the failure detector

Construction Outline

- Every process maintains a DAG which models a causal relation between queries to the failure detector
- A processes p queries its failure detector D for the k^{th} time and gets response d

Construction Outline

- Every process maintains a DAG which models a causal relation between queries to the failure detector
- A processes p queries its failure detector D for the k^{th} time and gets response d
- It sends (p, d, k) to other processes which add this node to their DAGs

Construction Outline

- Every process maintains a DAG which models a causal relation between queries to the failure detector
- A processes p queries its failure detector D for the k^{th} time and gets response d
- It sends (p, d, k) to other processes which add this node to their DAGs
- After process q adds a node (p, d, k), all nodes corresponding to future queries of q to its failure detector take an edge from (p, d, k)

Construction Outline

- Every process maintains a DAG which models a causal relation between queries to the failure detector
- A processes p queries its failure detector D for the k^{th} time and gets response d
- It sends (p, d, k) to other processes which add this node to their DAGs
- After process q adds a node (p, d, k), all nodes corresponding to future queries of q to its failure detector take an edge from (p, d, k)
- Processes exchange and update their graphs

Construction Outline

- Every process maintains a DAG which models a causal relation between queries to the failure detector
- A processes p queries its failure detector D for the k^{th} time and gets response d
- It sends (p, d, k) to other processes which add this node to their DAGs
- After process q adds a node (p, d, k), all nodes corresponding to future queries of q to its failure detector take an edge from (p, d, k)
- Processes exchange and update their graphs
- A finite subgraph of this graph contains the node that every process should trust

Conclusion

Conclusion

- A consensus algorithm satisfying all the three properties in an asynchronous environment tolerating a single node failure is impossible

Conclusion

- A consensus algorithm satisfying all the three properties in an asynchronous environment tolerating a single node failure is impossible
- Since a purely asynchronous system does not exist, it tells us any practical algorithm can get into infinite executions, however rare they are

Conclusion

- A consensus algorithm satisfying all the three properties in an asynchronous environment tolerating a single node failure is impossible
- Since a purely asynchronous system does not exist, it tells us any practical algorithm can get into infinite executions, however rare they are
- We need to relax constraints that make extra assumptions about the system to solve consensus

Conclusion

- A consensus algorithm satisfying all the three properties in an asynchronous environment tolerating a single node failure is impossible
- Since a purely asynchronous system does not exist, it tells us any practical algorithm can get into infinite executions, however rare they are
- We need to relax constraints that make extra assumptions about the system to solve consensus
- $\diamond W$ solves consensus algorithm by assuming weak properties about the failure detection module

Conclusion

- A consensus algorithm satisfying all the three properties in an asynchronous environment tolerating a single node failure is impossible
- Since a purely asynchronous system does not exist, it tells us any practical algorithm can get into infinite executions, however rare they are
- We need to relax constraints that make extra assumptions about the system to solve consensus
- $\diamond W$ solves consensus algorithm by assuming weak properties about the failure detection module
- It is the weakest failure detection module using which we can solve consensus