Byzantine Generals problem

Amirreza Taghizadeh

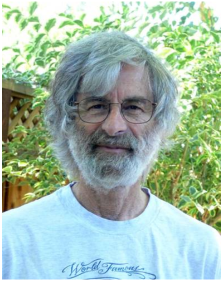
April 27, 2023

In this presentation, we aim to discuss a problem in distributed systems known as "**Byzantine generals problem"** proposed by **Leslie Lamport.**

Outline

Outline

Brief history of Lamport's works



- **Inventing LaTeX:** a group of macros, which can make the life of TeXusers a lot easier!!
- **One way authentication** in Whitfield Diffie's "*New directions in cryptography*" (1976)

brief description of **One-way Function**

- A one way function $\boldsymbol{f}$ is a function that is **easy to compute** but whose **inverse is difficult to compute**:

[1] Lamport, "*Constructing Digital Signatures from a One Way Function*"

[2] i.e. to say: $\forall$ data object $d' \neq d : \phi(d') \neq \phi(d)$

brief description of **One-way Function**

- A one way function $\boldsymbol{f}$ is a function that is **easy to compute** but whose **inverse is difficult to compute**:
- or to say f is one-way function iff (adapted from[1]):

[1] Lamport, "*Constructing Digital Signatures from a One Way Function*"

[2] i.e. to say: $\forall$ data object $d' \neq d : \phi(d') \neq \phi(d)$

brief description of **One-way Function**

- A one way function $\boldsymbol{f}$ is a function that is **easy to compute** but whose **inverse is difficult to compute**:
- or to say f is one-way function iff (adapted from[1]):
  1. for all value v, finding data object d s.t. $\phi(d) = v$ is **computationally infeasible.**

[1] Lamport, "*Constructing Digital Signatures from a One Way Function*"
[2] i.e. to say: $\forall$ data object $d' \neq d : \phi(d') \neq \phi(d)$

brief description of **One-way Function**

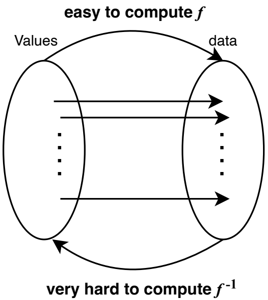
- A one way function $\boldsymbol{f}$ is a function that is **easy to compute** but whose **inverse is difficult to compute**:
- or to say f is one-way function iff (adapted from[1]):
    1. for all value v, finding data object d s.t. $\phi(d) = v$ is **computationally infeasible.**
    2. f is not **one-to-one** or proving it otherwise is **computationally infeasible.**[2]

[1] Lamport, "*Constructing Digital Signatures from a One Way Function*"

[2] i.e. to say: $\forall$ data object $d' \neq d : \phi(d') \neq \phi(d)$

brief description of **One-way Function**

- A one way function $\boldsymbol{f}$ is a function that is **easy to compute** but whose **inverse is difficult to compute**:
- or to say f is one-way function iff (adapted from[1]):
    1. for all value v, finding data object d s.t. $\phi(d) = v$ is **computationally infeasible.**
    2. f is not **one-to-one** or proving it otherwise is **computationally infeasible.**[2]



[1] Lamport, "*Constructing Digital Signatures from a One Way Function*"

[2] i.e. to say: $\forall$ data object $d' \neq d : \phi(d') \neq \phi(d)$

brief description of **One-way Authentication**

Definition (Digital signature)

A digital signature created by **sender P** for **document m** is a data item $\sigma_p(m)$ that is when received together with **m**, one can determine (e.g. in a court of law) that **P** generated document **m**.

[1] Lamport, L. (1979) "*Constructing Digital Signatures from a One Way Function*"

brief description of **One-way Authentication**

Definition (Digital signature)

A digital signature created by **sender P** for **document m** is a data item $\sigma_p(m)$ that is when received together with **m**, one can determine (e.g. in a court of law) that **P** generated document **m**.
Hence A tool for determining **validity** of something sent.[1]

[1] Lamport, L. (1979) "*Constructing Digital Signatures from a One Way Function*"

brief description of **One-way Authentication**

Definition (One way authentication)

It must be **easy for anyone** to recognize the signature as **authentic** but **impossible** for anyone other than the signer to produce it![1]

[1]Diffie, W. (1976)" *New Directions in Cryptography*"

A practical Example of a **One-way function** in **one-way authentication**
Login Problem

- User **A** enters Password PW and computer store it as $f(PW)$

A practical Example of a **One-way function** in **one-way authentication**
Login Problem

- User **A** enters Password PW and computer store it as $f(PW)$
- where $f(PW)$ is a one-way function of <u>10 million instructions</u>

A practical Example of a **One-way function** in **one-way authentication**
Login Problem

- User **A** enters Password PW and computer store it as $f(PW)$
- where $f(PW)$ is a one-way function of 10 million instructions
- and its inverse has 10^{30} more instructions (or computations), which practically makes it **noninvertible**

A practical Example of a **One-way function** in **one-way authentication**
Login Problem

- User **A** enters Password PW and computer store it as $f(PW)$
- where $f(PW)$ is a one-way function of <u>10 million instructions</u>
- and its inverse has 10^{30} more instructions (or computations), which practically makes it **noninvertible**
- for example, finding square root of x_0 given in $f(x) = x^2$ is much harder than computing x^2 at x_0.

brief description of **One-way Authentication** *Cont'd*

- However, determining exactly what the one-way function should be is originally solved by **Lamport**
  which further lead to the publication of the paper: " *Constructing Digital Signatures from a One Way Function*"

brief description of **One-way Authentication** *Cont'd*

- But how this solution relates to the ecosystem of **public keys** is out of the scope of the presentation and discussed in the paper: *"New Directions in Cryptography"* by Whitfield Diffie (1976)

Brief history of Lamport's works

- **Inventing LaTeX:** a group of macros, which can make the life of TeXusers a lot easier!!
- **One way authentication** in Whitfield Diffie's "*New directions in cryptography*" (1976)
- **bakery algorithm** an algorithm to ensure mutual exclusion in concurrent processes

[1] Silberschatz, "Database System Concepts", Ch. 19, P. 965

Brief history of Lamport's works

- **Inventing LaTeX:** a group of macros, which can make the life of TeXusers a lot easier!!
- **One way authentication** in Whitfield Diffie's "*New directions in cryptography*" (1976)
- **bakery algorithm** an algorithm to ensure mutual exclusion in concurrent processes
- that is to ensure a data structure is modified by at most one process at a time
  and no process is reading a data structure while it is being written by other processes.

[1] Silberschatz, "Database System Concepts", Ch. 19, P. 965

Brief history of Lamport's works

- **Inventing LaTeX:** a group of macros, which can make the life of TeXusers a lot easier!!
- **One way authentication** in Whitfield Diffie's "*New directions in cryptography*" (1976)
- **bakery algorithm** an algorithm to ensure mutual exclusion in concurrent processes
- that is to ensure a data structure is modified by at most one process at a time
  and no process is reading a data structure while it is being written by other processes.
- **Paxos algorithm**: an algorithm used in distributed systems for reaching consensus, used in distributed storage systems

[1] Silberschatz, "Database System Concepts", Ch. 19, P. 965

Brief history of Lamport's works *cont'd*

- Time, clocks and ordering of events in a distributed system:

Brief history of Lamport's works *cont'd*

- Time, clocks and ordering of events in a distributed system:
  - in a distributed system, sometimes it is **impossible** to say an event has happened before something else.

Brief history of Lamport's works *cont'd*

- Time, clocks and ordering of events in a distributed system:
  - in a distributed system, sometimes it is **impossible** to say an event has happened before something else.
  - hence, relation "*happened before* is a partial ordering relation.

Brief history of Lamport's works *cont'd*

- Time, clocks and ordering of events in a distributed system:
  - in a distributed system, sometimes it is **impossible** to say an event has happened before something else.
  - hence, relation "*happened before* is a partial ordering relation.
  - **Partial ordering? sounds familiar...**

Brief history of Lamport's works *cont'd*

- Time, clocks and ordering of events in a distributed system:
  - in a distributed system, sometimes it is **impossible** to say an event has happened before something else.
  - hence, relation "*happened before* is a partial ordering relation.
  - **Partial ordering? sounds familiar...**

Definition (Partial ordering)

Partial ordering relation is an ordering relation in which not all members of the set need to be comparable!

Brief history of Lamport's works *cont'd*

- Time, clocks and ordering of events in a distributed system:
  - in a distributed system, sometimes it is **impossible** to say an event has happened before something else.
  - hence, relation "*happened before* is a partial ordering relation.
  - He proposed in that paper a partial ordering of "happened before" and gave a **distributed algorithm** for **extending it to a total ordering of events**

Brief history of Lamport's works *cont'd*

- Time, clocks and ordering of events in a distributed system:
  - in a distributed system, sometimes it is **impossible** to say an event has happened before something else.
  - hence, relation "*happened before* is a partial ordering relation.

  - He proposed in that paper a partial ordering of "happened before" and gave a **distributed algorithm** for **extending it to a total ordering of events**

But where is the "Byzantine generals" problem in the list?

Outline

Brief history of the origination of the problem

- **Naming:** "As the Edsger W. Dijkstra's "Dining philosophers (a classic problem discussed in Operating Systems classes) involves a story, I decided to **include a story with the problem** related to Digital signatures with a recursive solution algorithm" -Lamport

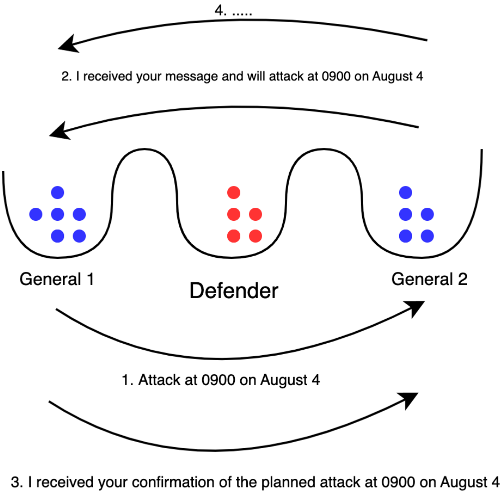
Brief history of the origination of the problem

- **Naming:** "As the Edsger W. Dijkstra's "Dining philosophers *(*a classic problem discussed in Operating Systems classes) involves a story, I decided to **include a story with the problem** related to Digital signatures with a recursive solution algorithm" -Lamport
- **Motivation:** Two General's problem.

Brief history of the origination of the problem

- **Naming:** "As the Edsger W. Dijkstra's "Dining philosophers *(*a classic problem discussed in Operating Systems classes) involves a story, I decided to **include a story with the problem** related to Digital signatures with a recursive solution algorithm" -Lamport
- **Motivation:** Two General's problem. in fact, byzantine generals are a more general form of this problem.

A brief Overview of Two general's problem
Stating the problem through visualization!



Outline

A review of the Byzantine problem

- There is an **enemy city** and a group of **General** i, each deciding to reach **an agreed upon plant** (which is the exact definition of **consensus**)

A review of the Byzantine problem

- There is an **enemy city** and a group of **General** i, each deciding to reach **an agreed upon plant** (which is the exact definition of **consensus**)
- and each general i is equipped with a messaging method for sending value $v(i)$

A review of the Byzantine problem

- There is an **enemy city** and a group of **General** i, each deciding to reach **an agreed upon plant** (which is the exact definition of **consensus**)
- and each general i is equipped with a messaging method for sending value $v(i)$
- AND there are a bunch of **traitorous generals** sending **conflicting** messages, aim to prevent **loyal generals to reach a plan**

A review of the Byzantine problem

- There is an **enemy city** and a group of **General** i, each deciding to reach **an agreed upon plant** (which is the exact definition of **consensus**)
- and each general i is equipped with a messaging method for sending value $v(i)$
- AND there are a bunch of **traitorous generals** sending **conflicting** messages, aim to prevent **loyal generals to reach a plan**
- In order for them to a reach consensus, two conditions must be satisfied:

A review of the Byzantine problem

- There is an **enemy city** and a group of **General** i, each deciding to reach **an agreed upon plant** (which is the exact definition of **consensus**)
- and each general i is equipped with a messaging method for sending value $v(i)$
- AND there are a bunch of **traitorous generals** sending **conflicting** messages, aim to prevent **loyal generals to reach a plan**
- In order for them to a reach consensus, two conditions must be satisfied:
  1. Every loyal general must obtain the same information $v(1), v(2) \ldots, v(n)$
  2. The value sent by a loyal general should be used by all loyal generals

A review of the Byzantine problem *cont*

How should the generals send their messages?

A review of the Byzantine problem *cont*

How should the generals send their messages?
Let's examine how **a single general** i **should send the message** $v(i)$ that is formally defined by:

A review of the Byzantine problem *cont*

How should the generals send their messages?
Let's examine how **a single general i should send the message $v(i)$** that is formally defined by: (which is made by grouping generals into two groups, namely **commander and lieutenant generals**)

Definition (Byzantine Generals Problem)

A **commanding general** must send an order to his $n-1$ **lieutenant generals** s.t.:
IC1. All loyal lieutenants obey the same order.
IC2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

A review of the Byzantine problem *cont*

How should the generals send their messages?
Let's examine how **a single general i should send the message $v(i)$** that is formally defined by: (which is made by grouping generals into two groups, namely **commander and lieutenant generals**)

Definition (Byzantine Generals Problem)

A **commanding general** must send an order to his $n-1$ **lieutenant generals** s.t.:
IC1. All loyal lieutenants obey the same order.
IC2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

Note that IC2 $\implies$ IC1

A review of the Byzantine problem *cont*

Lamport gave a recursive algorithm based on **majority function** for the mentioned problem in case of having **oral messages** whose content is solely managed by the sender.

A review of the Byzantine problem *cont*

Lamport gave a recursive algorithm based on **majority function** for the mentioned problem in case of having **oral messages** whose content is solely managed by the sender.

Unfortunately, the algorithm works only, in case of having m traitors, for $n \geq 3m + 1$ generals

A review of the Byzantine problem *cont*

Lamport gave a recursive algorithm based on **majority function** for the mentioned problem in case of having **oral messages** whose content is solely managed by the sender.

Unfortunately, the algorithm works only, in case of having m traitors, for $n \geq 3m + 1$ generals

This is where he proposed an algorithm based on **unforgeable messages.**

Outline

Signed messages

What was wrong with the oral messages??

Signed messages

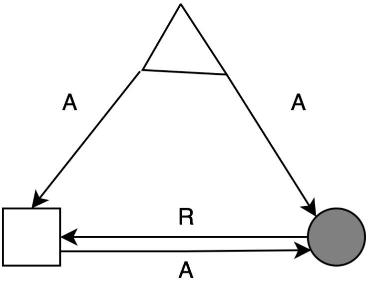
What was wrong with the oral messages??

Answer: because **traitors lie**, they **alter the contents** of the messages they receive and send

Signed messages

What was wrong with the oral messages??

Answer: because **traitors lie**, they **alter the contents** of the messages they receive and send



Signed messages

What was wrong with the oral messages??

Answer: because **traitors lie**, they **alter the contents** of the messages they receive and send

Let's make messages that can not be lied (forged) or any alterations could be detected

Signed messages algorithm

Note that we know **the number of m** traitors when running the Alg.

Signed messages algorithm

1. Commander sends a message having a value v and a signature (which is a sequence of IDs) $\rightarrow \boldsymbol{v : 0}$

Signed messages algorithm

1. Commander sends a message having a value v and a signature (which is a sequence of IDs) $\rightarrow$ $\boldsymbol{v:0}$
2. each lieutenant i receives the message of length k,
   adds the v to a V_i set, **adds his ID** to the message and **sends it to those child lieutenants not having received this message before**

Signed messages algorithm

1. Commander sends a message having a value v and a signature (which is a sequence of IDs) $\rightarrow \boldsymbol{v:0}$
2. each lieutenant i receives the message of length k, **adds the v to a V_i set**, **adds his ID** to the message and **sends it to those child lieutenants not having received this message before**
3. when lieutenant i receives no more messages, the lieutenant i applies the **a Choice function** to V_i in order to retrieve an order.

Signed messages algorithm

1. Commander sends a message having a value v and a signature (which is a sequence of IDs) $\rightarrow \boldsymbol{v : 0}$
2. each lieutenant i receives the message of length k, **adds the v to a V_i set**, **adds his ID** to the message and **sends it to those child lieutenants not having received this message before**
3. when lieutenant i receives no more messages, the lieutenant i applies the **a Choice function** to V_i in order to retrieve an order.

what is that **Choice function?**

Choice function

The Choice function could be any **aggregate function** (such as median, average, etc) BUT, it needs to have two essential properties:

1. if Set V_i consists of single value v Then, $Choice(V_i) = v$

Choice function

The Choice function could be any **aggregate function** (such as median, average, etc) BUT, it needs to have two essential properties:

1. if Set V_i consists of single value v Then, $Choice(V_i) = v$
2. $Choice(\emptyset) = RETREAT$

Formal statement of Signed messages

Initially $V_i = \emptyset$

(1) The commander signs and sends message $v : 0$ to all lieutenants

(2) For each i:

(A) If Lieutenant i receives a message of the form $v : 0$ from the commander and he hasn't receieved any order, then:

(i) he lets $V_i = v$

(ii) he sends the message $v : 0 : i$ to every other lieutenant.

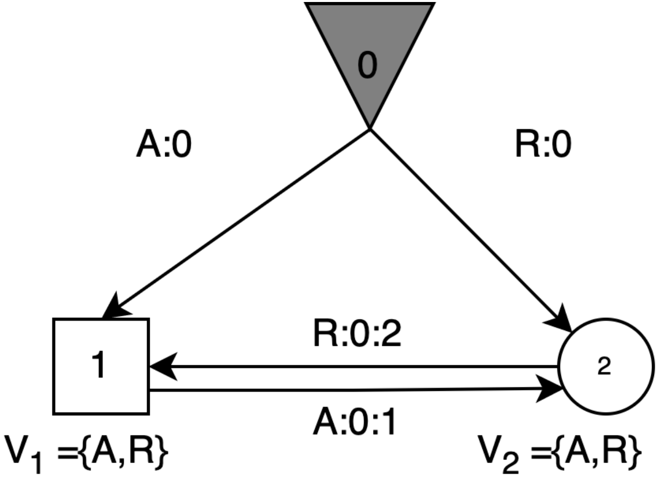
(B) If Lieutenant i receives a message of the form $v : 0 : j_1 \cdots : j_k$ **and** $v \notin V_i$ then:

(i) he adds v to V_i;

(ii) if $k < m$, then he sends the message $v : 0 : j_1 \cdots : j_k : i$ to every lieutenant other than $j_1, \ldots, j_k$

(3) For each i: When Lieutenant i will receive no more messages, he obeys the order $Choice(V_i)$

The impossible case revisited



Outline

Through the end of the presentation, we will be talking about **Consensus in asynchronous** systems
and an important Article in this regard, by Fischer-Lynch-Paterson (FLP)
Which proves the impossibility of **reaching consensus** in **asynchronous system** in the presence of any failure.
However, in the paper

Asynchronous systems → Lamport's Time clocks revisted

Definition (Asynchronous system[1])

In such systems, there is no time assumptions about processes. That is, we do not have **physical clock**. Instead **Time** is defined with respect to communication and measured by a *Logical clock*.

[1] "*Reliable and Secure Distributed Programming*" A book by Rodrigues L. & Cachin C.

Asynchronous systems → Lamport's Time clocks revisted

Definition (Asynchronous system[1])

In such systems, there is no time assumptions about processes. That is, we do not have **physical clock**. Instead **Time** is defined with respect to communication and measured by a *Logical clock*.

Here is Lamport's Algorithm for finding something has "happened before" something else.

[1] "*Reliable and Secure Distributed Programming*" A book by Rodrigues L. & Cachin C.

Lamport's Algorithm for Time clocks

1. Each process p keeps an ineger called *logical clock* l_p initially 0.

[1] "*Reliable and Secure Distributed Programming*" A book by Rodrigues L. & Cachin C.

Lamport's Algorithm for Time clocks

1. Each process p keeps an ineger called *logical clock* l_p initially 0.
2. Whenver an event occurs at Node p, the logical clock l_p is incremented by one unit.

[1] "*Reliable and Secure Distributed Programming*" A book by Rodrigues L. & Cachin C.

Lamport's Algorithm for Time clocks

1. Each process p keeps an ineger called *logical clock* l_p initially 0.
2. Whenver an event occurs at Node p, the logical clock l_p is incremented by one unit.
3. When a process **sends a message,** it adds a timestamp to the message with the value of its logical clock at the moment the message is sent. The timestamp of an event e is denoted by $t(e)$.

[1] "*Reliable and Secure Distributed Programming*" A book by Rodrigues L. & Cachin C.

Lamport's Algorithm for Time clocks

1. Each process p keeps an ineger called *logical clock* l_p initially 0.
2. Whenver an event occurs at Node p, the logical clock l_p is incremented by one unit.
3. When a process **sends a message,** it adds a timestamp to the message with the value of its logical clock at the moment the message is sent. The timestamp of an event e is denoted by $t(e)$.
4. When a Node p receives a message m with timestamp t_m, Node p increments its logical clock in the following way: $l_p := \max\{l_p, t_m\} + 1$

[1] "*Reliable and Secure Distributed Programming*" A book by Rodrigues L. & Cachin C.