Definition (Synchronous distributed system)

A distributed system comprising of processes is synchronous iff it has the following properties[1]

1. There is a real-time physical clock which provides synchronization among multiple processes
2. A known upper bound ϵ on processing delays
3. A known upper bound δ on message transmission delays.

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

Definition (Synchronous distributed system)

A distributed system comprising of processes is synchronous iff it has the following properties[1]

1. There is a real-time physical clock which provides synchronization among multiple processes
2. A known upper bound ϵ on processing delays
3. A known upper bound δ on message transmission delays.

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

Asynchronous systems → Lamport's Time clocks revisted

Definition (Asynchronous system[2])

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

Asynchronous systems → Lamport's Time clocks revisted

Definition (Asynchronous system[2])

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

Asynchronous systems → Lamport's Time clocks revisted

Definition (Asynchronous system[2])

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

Asynchronous systems → Lamport's Time clocks revisted

Definition (Asynchronous system[2])

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

A brief description of Lamport's Algorithm for Time clocks

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

A brief description of Lamport's Algorithm for Time clocks

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

A brief description of 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.

A brief description of 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.

A brief description of 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 $t(e)$.

A brief description of 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 $t(e)$.

A brief description of 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: $l_p := \max\{l_p, t_m\} + 1$

A brief description of 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: $l_p := \max\{l_p, t_m\} + 1$

A brief description of Lamport's Algorithm for Time clocks *cont.*

- now that we have a mechanism for time in asynchronous world, let us define the *happened before* relation

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

A brief description of Lamport's Algorithm for Time clocks *cont.*

- now that we have a mechanism for time in asynchronous world, let us define the *happened before* relation

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

A brief description of Lamport's Algorithm for Time clocks *cont.*

- now that we have a mechanism for time in asynchronous world, let us define the *happened before* relation

Definition (Happened before relation $e_1 \to e_2$)

For event e_1 and e_2 we say "e_1 *has happened before* e_2" ($e_1 \to e_2$) if e_1 and e_2 have the following conditions:

1. e_1 and e_2 occurred at the **same process p** and e_1 occurred before e_2 (i.e. $t(e_2) = t(e_1) + 1$)
2. e_1 corresponds to the transmission of a message $\boldsymbol{m}$ at process p and e_2 to the reception of m at process q.
3. $\to$ be transitive.

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

A brief description of Lamport's Algorithm for Time clocks *cont.*

- now that we have a mechanism for time in asynchronous world, let us define the *happened before* relation

Definition (Happened before relation $e_1 \to e_2$)

For event e_1 and e_2 we say "e_1 *has happened before* e_2" ($e_1 \to e_2$) if e_1 and e_2 have the following conditions:

1. e_1 and e_2 occurred at the **same process p** and e_1 occurred before e_2 (i.e. $t(e_2) = t(e_1) + 1$)
2. e_1 corresponds to the transmission of a message $\boldsymbol{m}$ at process p and e_2 to the reception of m at process q.
3. $\to$ be transitive.

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

---

- event is something such as sending or receiving message.
- there exists some event e' s.t. $e_1 \to e'$ and $e' \to e_2$

A brief description of Lamport's Algorithm for Time clocks *cont.*

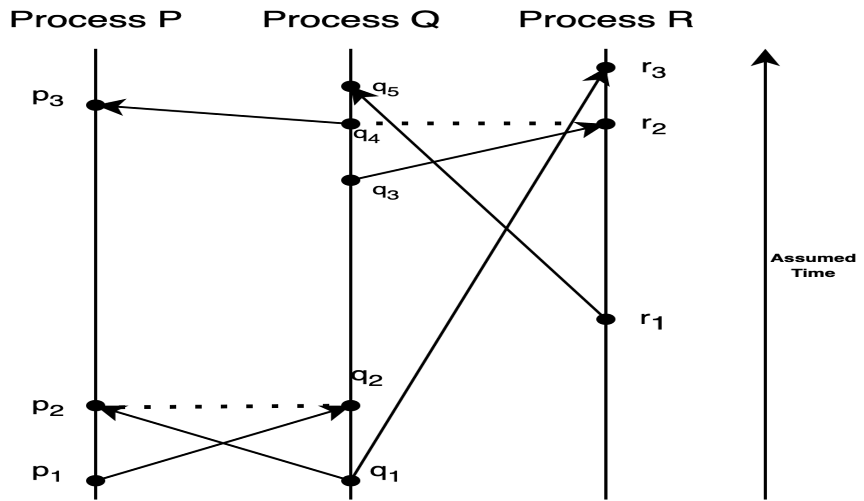
The above definition implies that if $e_1 \to e_2 \implies t(e_1) < t(e_2)$

A brief description of Lamport's Algorithm for Time clocks *cont.*

The above definition implies that if $e_1 \to e_2 \implies t(e_1) < t(e_2)$

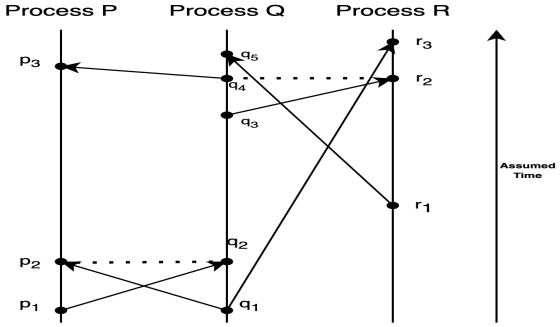
A brief description of Lamport's Algorithm for Time clocks *cont.*

The above definition implies that if $e_1 \rightarrow e_2 \implies t(e_1) < t(e_2)$



A brief description of Lamport's Algorithm for Time clocks *cont.*

The above definition implies that if $e_1 \rightarrow e_2 \implies t(e_1) < t(e_2)$



Practical Byzantine Fault Tolerance (PBFT)

This Algorithm describes the implementation of a
Byzantine-fault-tolerant disributed file system.
Assumptions:

1. We have a **asynchronous distributed system** $\implies$ network may have: **Delays, failure to deliver messages; duplicate them, delivery out of order**

Practical Byzantine Fault Tolerance (PBFT)

This Algorithm describes the implementation of a
Byzantine-fault-tolerant disributed file system.
Assumptions:

1. We have a **asynchronous distributed system** $\implies$ network may have: **Delays, failure to deliver messages; duplicate them, delivery out of order**

Practical Byzantine Fault Tolerance (PBFT)

This Algorithm describes the implementation of a **Byzantine-fault-tolerant disributed file system.**
Assumptions:

1. We have a **asynchronous distributed system** $\implies$ network may have: **Delays, failure to deliver messages; duplicate them, delivery out of order**
2. For Byzantine assumption (nodes behaving arbitrarily): Independent node failures $\implies$ each node runs service code **independently**, have different root password and a different admin.

Practical Byzantine Fault Tolerance (PBFT)

This Algorithm describes the implementation of a **Byzantine-fault-tolerant disributed file system.**
Assumptions:

1. We have a **asynchronous distributed system** $\implies$ network may have: **Delays, failure to deliver messages; duplicate them, delivery out of order**
2. For Byzantine assumption (nodes behaving arbitrarily): Independent node failures $\implies$ each node runs service code **independently**, have different root password and a different admin.

Practical Byzantine Fault Tolerance (PBFT)

This Algorithm describes the implementation of a
Byzantine-fault-tolerant disributed file system.

Assumptions:

1. We have a **asynchronous distributed system** $\implies$ network may have: **Delays, failure to deliver messages; duplicate them, delivery out of order**
2. For Byzantine assumption (nodes behaving arbitrarily): Independent node failures $\implies$ each node runs service code **independently**, have different root password and a different admin.
3. Use cryptographic techniques to prevent spoofing and to detect corrupted messages $\rightarrow$ messages have *public key signatures*, message authentication codes.

Practical Byzantine Fault Tolerance (PBFT)

This Algorithm describes the implementation of a
Byzantine-fault-tolerant disributed file system.

Assumptions:

1. We have a **asynchronous distributed system** $\implies$ network may have: **Delays, failure to deliver messages; duplicate them, delivery out of order**
2. For Byzantine assumption (nodes behaving arbitrarily): Independent node failures $\implies$ each node runs service code **independently**, have different root password and a different admin.
3. Use cryptographic techniques to prevent spoofing and to detect corrupted messages $\rightarrow$ messages have *public key signatures*, message authentication codes.

Practical Byzantine Fault Tolerance (PBFT)

This Algorithm describes the implementation of a
Byzantine-fault-tolerant disributed file system.
Assumptions:

1. We have a **asynchronous distributed system** $\implies$ network may have: **Delays, failure to deliver messages; duplicate them, delivery out of order**
2. For Byzantine assumption (nodes behaving arbitrarily): Independent node failures $\implies$ each node runs service code **independently**, have different root password and a different admin.
3. Use cryptographic techniques to prevent spoofing and to detect corrupted messages $\rightarrow$ messages have *public key signatures*, message authentication codes.
4. A strong adversary coordinates faulty nodes, delay communication and delay correct nodes
5. The adversary cannot violate cryptographic techniques used by nodes.

Practical Byzantine Fault Tolerance (PBFT)

This Algorithm describes the implementation of a
Byzantine-fault-tolerant disributed file system.
Assumptions:

1. We have a **asynchronous distributed system** $\implies$ network may have: **Delays, failure to deliver messages; duplicate them, delivery out of order**
2. For Byzantine assumption (nodes behaving arbitrarily): Independent node failures $\implies$ each node runs service code **independently**, have different root password and a different admin.
3. Use cryptographic techniques to prevent spoofing and to detect corrupted messages $\rightarrow$ messages have *public key signatures*, message authentication codes.
4. A strong adversary coordinates faulty nodes, delay communication and delay correct nodes
5. The adversary cannot violate cryptographic techniques used by nodes.

For example, the adversary cannot produce a valid signature of a non-faulty node, compute the information summarized by a digest from the digest, or find two messages with the same digest. The cryptographic techniques we use are thought to have these properties

Practical Byzantine Fault Tolerance (PBFT)

Properties:

1. The algorithm used to implement any **replicated service** with *state* and *some operations*, including but not limited to *read and writes on replicas*

Practical Byzantine Fault Tolerance (PBFT)

Properties:

1. The algorithm used to implement any **replicated service** with *state* and *some operations*, including but not limited to *read and writes on replicas*

Practical Byzantine Fault Tolerance (PBFT)

Properties:

1. The algorithm used to implement any **replicated service** with *state* and *some operations*, including but not limited to *read and writes on replicas*
2. There are some **faulty** and **non-faulty clients(requesting to invoke operations)** and **replicas** (we have f faulty replicas and $n = 3f + 1$ number of all replicas))

Practical Byzantine Fault Tolerance (PBFT)

Properties:

1. The algorithm used to implement any **replicated service** with *state* and *some operations*, including but not limited to *read and writes on replicas*
2. There are some **faulty** and **non-faulty clients(requesting to invoke operations)** and **replicas** (we have f faulty replicas and $n = 3f + 1$ number of all replicas))

Practical Byzantine Fault Tolerance (PBFT)

Properties:

1. The algorithm used to implement any **replicated service** with *state* and *some operations*, including but not limited to *read and writes on replicas*
2. There are some **faulty** and **non-faulty clients(requesting to invoke operations)** and **replicas** (we have f faulty replicas and $n = 3f + 1$ number of all replicas))

Practical Byzantine Fault Tolerance (PBFT)

Properties:

1. The algorithm used to implement any **replicated service** with *state* and *some operations*, including but not limited to *read and writes on replicas*
2. There are some **faulty** and **non-faulty clients(requesting to invoke operations)** and **replicas** (we have f faulty replicas and $n = 3f + 1$ number of all replicas))

Faulty clients are dealt with by **strong authentication**

Practical Byzantine Fault Tolerance (PBFT)

Properties:

1. The algorithm used to implement any **replicated service** with *state* and *some operations*, including but not limited to *read and writes on replicas*
2. There are some **faulty** and **non-faulty clients(requesting to invoke operations)** and **replicas** (we have f faulty replicas and $n = 3f + 1$ number of all replicas))
3. The Algorithm both provides **safety** and **liveness**

Practical Byzantine Fault Tolerance (PBFT)

Properties:

1. The algorithm used to implement any **replicated service** with *state* and *some operations*, including but not limited to *read and writes on replicas*
2. There are some **faulty** and **non-faulty clients(requesting to invoke operations)** and **replicas** (we have f faulty replicas and $n = 3f + 1$ number of all replicas))
3. The Algorithm both provides **safety** and **liveness**

Faulty clients are dealt with by **strong authentication**

Practical Byzantine Fault Tolerance (PBFT)

Properties:

1. The algorithm used to implement any **replicated service** with *state* and *some operations*, including but not limited to *read and writes on replicas*
2. There are some **faulty** and **non-faulty clients(requesting to invoke operations)** and **replicas** (we have f faulty replicas and $n = 3f + 1$ number of all replicas))
3. The Algorithm both provides **safety** and **liveness**
   1. **Safety (Accuracy):** replicated service satisfies linearizability.

Practical Byzantine Fault Tolerance (PBFT)

Properties:

1. The algorithm used to implement any **replicated service** with *state* and *some operations*, including but not limited to *read and writes on replicas*
2. There are some **faulty** and **non-faulty clients(requesting to invoke operations)** and **replicas** (we have f faulty replicas and $n = 3f + 1$ number of all replicas))
3. The Algorithm both provides **safety** and **liveness**
   1. **Safety (Accuracy):** replicated service satisfies linearizability.

Faulty clients are dealt with by **strong authentication-** It behaves like a centralized implementation that executes operations **atomically** one at a time.

Practical Byzantine Fault Tolerance (PBFT)

Properties:

1. The algorithm used to implement any **replicated service** with *state* and *some operations*, including but not limited to *read and writes on replicas*
2. There are some **faulty** and **non-faulty clients(requesting to invoke operations)** and **replicas** (we have f faulty replicas and $n = 3f + 1$ number of all replicas))
3. The Algorithm both provides **safety** and **liveness**
   1. **Safety (Accuracy):** replicated service satisfies linearizability.
   2. **Liveness (Completeness):** Clients eventually receive replies to their requests, provided at most $\lfloor \frac{n-1}{3} \rfloor$ are faulty and $\underline{delay(t)}$ does not grow faster than t indefinitely.

Practical Byzantine Fault Tolerance (PBFT)

Properties:

1. The algorithm used to implement any **replicated service** with *state* and *some operations*, including but not limited to *read and writes on replicas*
2. There are some **faulty** and **non-faulty clients(requesting to invoke operations)** and **replicas** (we have f faulty replicas and $n = 3f + 1$ number of all replicas))
3. The Algorithm both provides **safety** and **liveness**
   1. **Safety (Accuracy):** replicated service satisfies linearizability.
   2. **Liveness (Completeness):** Clients eventually receive replies to their requests, provided at most $\lfloor \frac{n-1}{3} \rfloor$ are faulty and $\underline{delay(t)}$ does not grow faster than t indefinitely.

Faulty clients are dealt with by **strong authentication-** It behaves like a centralized implementation that executes operations **atomically** one at a time.

-$delay(t)$ is the time between the moment when a message is sent for the first time and the moment when it is received by its destination (assuming the sender keeps retransmitting the message until it is received)

4. Algorithm does not rely on **synchrony** to provide **safety**

4. Algorithm does not rely on **synchrony** to provide **safety**

4. Algorithm does not rely on **synchrony** to provide **safety**
5. However, **It must** rely on synchrony to provide **liveness** $\rightarrow$ **FLP's impossibility**

4. Algorithm does not rely on **synchrony** to provide **safety**
5. However, **It must** rely on synchrony to provide **liveness** $\rightarrow$ **FLP's impossibility**

4. Algorithm does not rely on **synchrony** to provide **safety**
5. However, **It must** rely on synchrony to provide **liveness** $\rightarrow$ **FLP's impossibility**
6. The algorithm uses a 3-phase commit protocol: Pre-prepare, prepare, commit

4. Algorithm does not rely on **synchrony** to provide **safety**
5. However, **It must** rely on synchrony to provide **liveness** $\rightarrow$ **FLP's impossibility**
6. The algorithm uses a 3-phase commit protocol: Pre-prepare, prepare, commit

PBFT Algorithm

- There is a primary replica in each request of a client c.

PBFT Algorithm

- There is a primary replica in each request of a client c.

PBFT Algorithm

- There is a primary replica in each request of a client c.
- Replicas move through a seccession of configurations called **views**.

PBFT Algorithm

- There is a primary replica in each request of a client c.
- Replicas move through a seccession of configurations called **views**.

PBFT Algorithm

- There is a primary replica in each request of a client c.
- Replicas move through a seccession of configurations called **views**.
- in a view, one replica is the primary and others are backups.

PBFT Algorithm

- There is a primary replica in each request of a client c.
- Replicas move through a seccession of configurations called **views**.
- in a view, one replica is the primary and others are backups.

PBFT Alg. Cont.

A very brief description of the Algorithm:

PBFT Alg. Cont.

A very brief description of the Algorithm:

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$

---

c is client, o is the operation needed, timestamp t is used to ensure **exactly once** semantics for the execution of client requests.

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$
- The primary multicasts the request to the backups

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$
- The primary multicasts the request to the backups

c is client, o is the operation needed, timestamp t is used to ensure **exactly once** semantics for the execution of client requests.

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$
- The primary multicasts the request to the backups
- Replicas execute the request and send a reply to the client

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$
- The primary multicasts the request to the backups
- Replicas execute the request and send a reply to the client

---

c is client, o is the operation needed, timestamp t is used to ensure **exactly once** semantics for the execution of client requests.

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$
- The primary multicasts the request to the backups
- Replicas execute the request and send a reply to the client
- The client waits for $f + 1$ replies from different replicas with the same result; this the result of the operation.

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$
- The primary multicasts the request to the backups
- Replicas execute the request and send a reply to the client
- The client waits for $f + 1$ replies from different replicas with the same result; this the result of the operation.

---

c is client, o is the operation needed, timestamp t is used to ensure **exactly once** semantics for the execution of client requests.

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$
- The primary multicasts the request to the backups
- Replicas execute the request and send a reply to the client
- The client waits for $f + 1$ replies from different replicas with the same result; this the result of the operation.
- The replica's reply is of the form $< REQUEST, \nu, t, c, i, r >_{\sigma_i}$

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$
- The primary multicasts the request to the backups
- Replicas execute the request and send a reply to the client
- The client waits for $f + 1$ replies from different replicas with the same result; this the result of the operation.
- The replica's reply is of the form $< REQUEST, \nu, t, c, i, r >_{\sigma_i}$

---

c is client, o is the operation needed, timestamp t is used to ensure **exactly once** semantics for the execution of client requests.

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$
- The primary multicasts the request to the backups
- Replicas execute the request and send a reply to the client
- The client waits for $f+1$ replies from different replicas with the same result; this the result of the operation.
- The replica's reply is of the form $< REQUEST, \nu, t, c, i, r >_{\sigma_i}$

PBFT Alg. Cont.

A very brief description of the Algorithm:

- A client sends a request to invoke a service operation to the primary replica
- The client's request message is of the form $< REQUEST, o, t, c >_{\sigma_c}$
- The primary multicasts the request to the backups
- Replicas execute the request and send a reply to the client
- The client waits for $f+1$ replies from different replicas with the same result; this the result of the operation.
- The replica's reply is of the form $< REQUEST, \nu, t, c, i, r >_{\sigma_i}$

---

c is client, o is the operation needed, timestamp t is used to ensure **exactly once** semantics for the execution of client requests.v is the current view number, t is the timestamp of the corresponding request, i is the replica number, and r is the result of executing the requested operation

PBFT Alg. Cont.

- The client waits for $f+1$ replies with valid signatures from different replicas, and with the same t and r, before accepting the result r

PBFT Alg. Cont.

- The client waits for $f+1$ replies with valid signatures from different replicas, and with the same t and r, before accepting the result r

PBFT Alg. Cont.

- The client waits for $f+1$ replies with valid signatures from different replicas, and with the same t and r, before accepting the result r
- If client doesn't receive replies soon, it broadcasts the request to all replicas.

PBFT Alg. Cont.

- The client waits for $f+1$ replies with valid signatures from different replicas, and with the same t and r, before accepting the result r
- If client doesn't receive replies soon, it broadcasts the request to all replicas.

PBFT Alg. Cont.

- The client waits for $f+1$ replies with valid signatures from different replicas, and with the same t and r, before accepting the result r
- If client doesn't receive replies soon, it broadcasts the request to all replicas.
- If the request has already been processed, the replicas simply re-send the reply;

PBFT Alg. Cont.

- The client waits for $f+1$ replies with valid signatures from different replicas, and with the same t and r, before accepting the result r
- If client doesn't receive replies soon, it broadcasts the request to all replicas.
- If the request has already been processed, the replicas simply re-send the reply;

PBFT Alg. Cont.

- The client waits for $f+1$ replies with valid signatures from different replicas, and with the same t and r, before accepting the result r
- If client doesn't receive replies soon, it broadcasts the request to all replicas.
- If the request has already been processed, the replicas simply re-send the reply;
- otherwise, if the replica is not the primary, it relays the request to the primary.

PBFT Alg. Cont.

- The client waits for $f+1$ replies with valid signatures from different replicas, and with the same t and r, before accepting the result r
- If client doesn't receive replies soon, it broadcasts the request to all replicas.
- If the request has already been processed, the replicas simply re-send the reply;
- otherwise, if the replica is not the primary, it relays the request to the primary.

PBFT Alg. Cont.

- The client waits for $f + 1$ replies with valid signatures from different replicas, and with the same t and r, before accepting the result r
- If client doesn't receive replies soon, it broadcasts the request to all replicas.
- If the request has already been processed, the replicas simply re-send the reply;
- otherwise, if the replica is not the primary, it relays the request to the primary.
- If the primary doesn't multicast the request to the group, it will eventually be suspected to be faulty by enough replicas to cause a view change.

PBFT Alg. Cont.

- The client waits for $f + 1$ replies with valid signatures from different replicas, and with the same t and r, before accepting the result r
- If client doesn't receive replies soon, it broadcasts the request to all replicas.
- If the request has already been processed, the replicas simply re-send the reply;
- otherwise, if the replica is not the primary, it relays the request to the primary.
- If the primary doesn't multicast the request to the group, it will eventually be suspected to be faulty by enough replicas to cause a view change.

Thank you for your attention!

Thank you for your attention!