Leaderless State Machine Replication: Specification, Properties, Limits

Tuanir França Rezende [1]   Pierre Sutra [1]

[1] Telecom SudParis

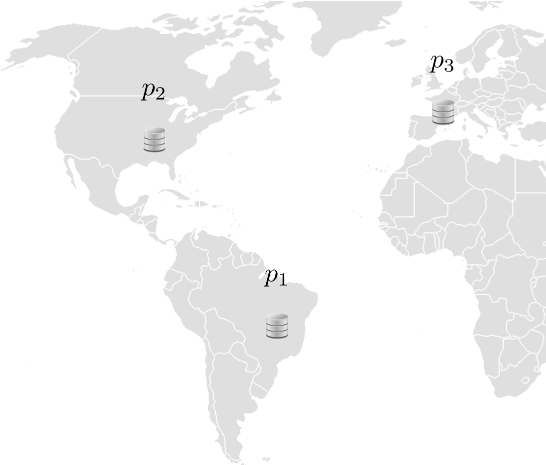
14 Oct. 2020, DISC'20

Context - Principles

- Modern internet services often maintain more than one **copy** of an object.

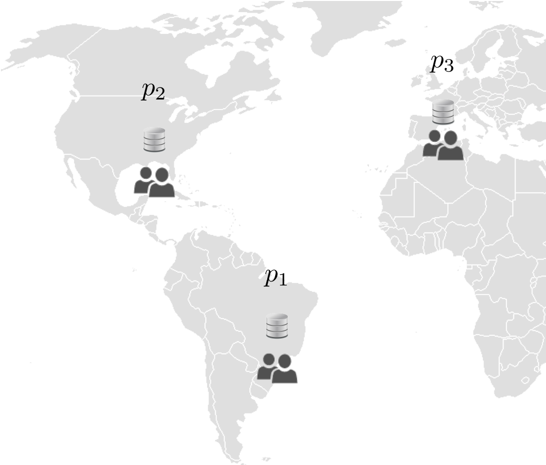
Context - Principles

- Each copy is located on a separate server, or **replica**.



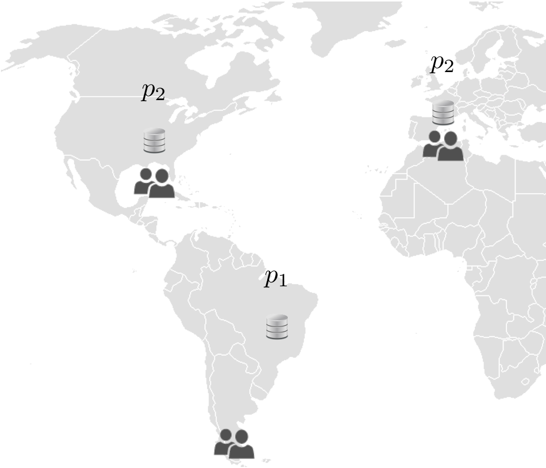
Context - Principles

- Clients interact with these replicas by sending **commands** and **share logically** the **objects**.



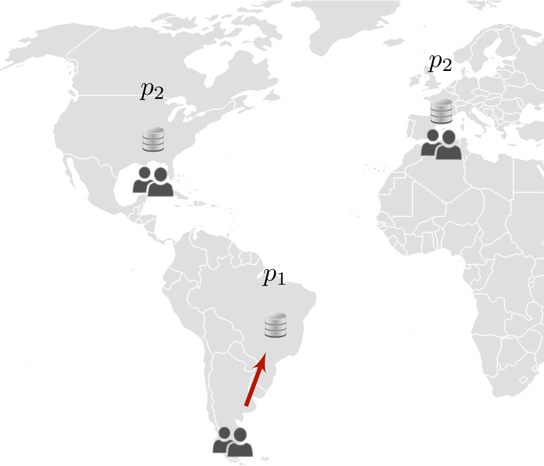
Context - Why to replicate data?

- **Performance:** Improve latency and/or throughput.



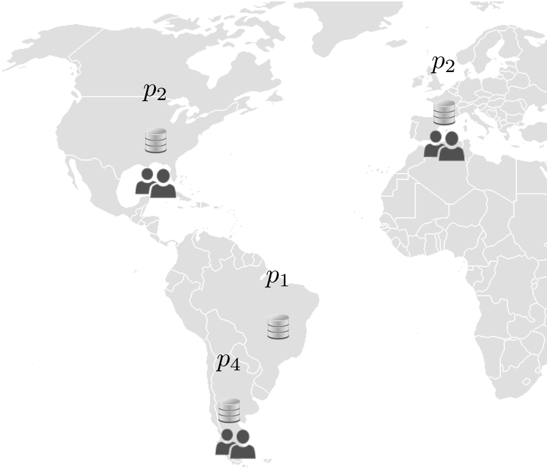
Context - Why to replicate data?

- **Performance:** Improve latency and/or throughput.



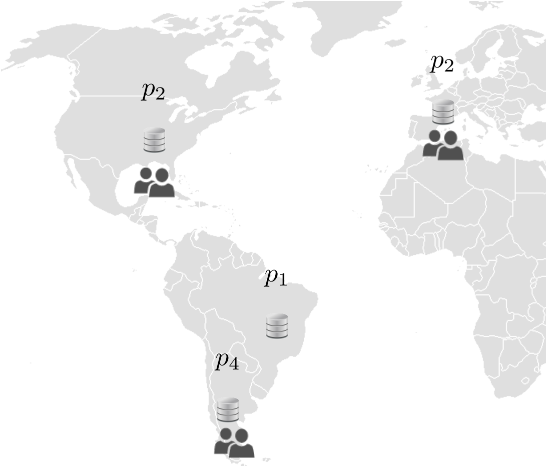
Context - Why to replicate data?

- **Performance:** Improve latency and/or throughput.



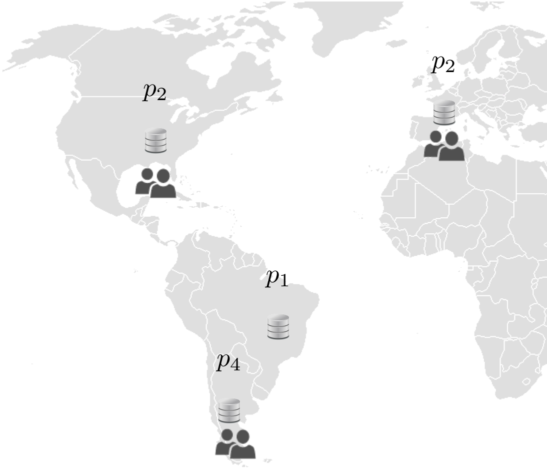
Context - Why to replicate data?

- **Performance:** Improve latency and/or throughput.



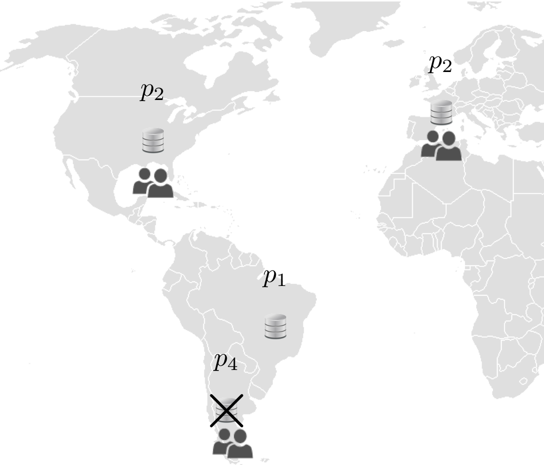
Context - Why to replicate data?

- **Reliability:** Mask replica and network failures.



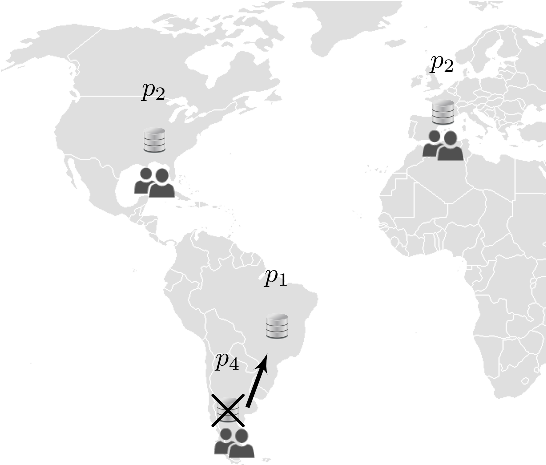
Context - Why to replicate data?

- **Reliability:** Mask replica and network failures.



Context - Why to replicate data?

- **Reliability:** Mask replica and network failures.



Context - How to replicate data?

Through **State Machine Replication (SMR)**:

- Classic SMR (Paxos). Execute commands in the same order.

$$p_1 \quad b \longrightarrow a \longrightarrow c$$

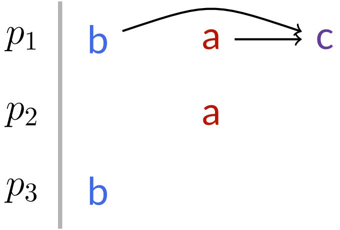
$$p_2 \quad b \longrightarrow a$$

$$p_3 \quad b$$

Context - How to replicate data?

Through **State Machine Replication (SMR)**:

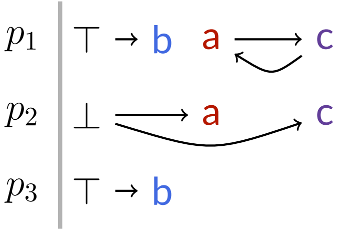
- Classic SMR (Paxos). Execute commands in the same order.
- Generic SMR. Execute *non-commuting* commands in the same order.



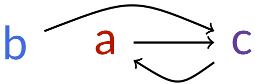
Context - How to replicate data?

Through **State Machine Replication (SMR)**:

- Classic SMR (Paxos). Execute commands in the same order.
- Generic SMR. Execute *non-commuting* commands in the same order.
- Leaderless SMR?

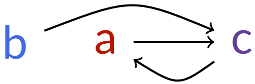


Leaderless SMR - Dependency Graph



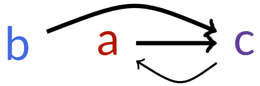
Relies on the notion of **dependency graph** instead of a partially ordered log as found in Generic SMR.

Leaderless SMR - Dependency Graph



A **dependency graph** is a directed graph that records the constraints defining how commands are executed.

Leaderless SMR - Dependency Graph



For some command c, the **incoming neighbors** of c in the dependency graph are its **dependencies**.

Leaderless SMR - Understanding the abstraction

In Leaderless SMR, a process holds two mapping: $deps$ and $phase$.

Leaderless SMR - Understanding the abstraction

The mapping $deps$ is a dependency graph storing a relation from $\mathcal{C}$ to $2^{\mathcal{C}} \cup \{\perp, \top\}$. [1]

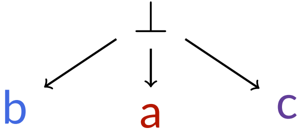
[1] $\mathcal{C}$ is the set of commands.

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd} & \perp \arrow[dl] \arrow[d] \arrow[dr] & \\ b & a & c \end{tikzcd}$$

Initially, for every command c, $deps(\mathrm{c})$ is set to $\perp$. This corresponds to the **pending** phase.

Leaderless SMR - Understanding the abstraction



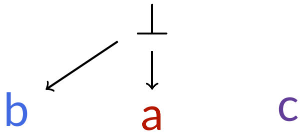
When a process decides a command c, it changes the mapping $deps(\mathsf{c})$ to a non-$\perp$ value. Operation $commit(\mathsf{c}, D)$ assigns D taken in $2^{\mathcal{C}}$ to $deps(\mathsf{c})$.

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd} & \perp \arrow[dl] \arrow[d] \arrow[dr] & \\ \mathsf{b} & \mathsf{a} & \mathsf{c} \end{tikzcd}$$

$$commit(\mathsf{c}, \{\mathsf{a}, \mathsf{b}\})$$

Leaderless SMR - Understanding the abstraction



$$commit(\mathsf{c}, \{\mathsf{a}, \mathsf{b}\})$$

Leaderless SMR - Understanding the abstraction

$$
\begin{tikzcd}
 & \perp \arrow[dl] \arrow[d] & \\
b & a \arrow[r] & c
\end{tikzcd}
$$

$$commit(c, \{a, b\})$$

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd} & & \perp \arrow[dll] \arrow[d] & \\ b \arrow[rrr, bend right] & & a \arrow[r] & c \end{tikzcd}$$

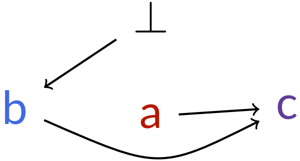
$$commit(\mathsf{c}, \{\mathsf{a}, \mathsf{b}\})$$

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd} & \perp \arrow[dl] \arrow[d] & \\ b \arrow[rr, bend right] & a \arrow[r] & c \end{tikzcd}$$

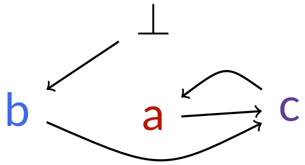
$$commit(\mathsf{a}, \{\mathsf{c}\})$$

Leaderless SMR - Understanding the abstraction



$$commit(\mathsf{a}, \{\mathsf{c}\})$$

Leaderless SMR - Understanding the abstraction



$$commit(a, \{c\})$$

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd}
\mathsf{a} \arrow[rr, bend right] & & \mathsf{b} \arrow[ll, bend right] & \mathsf{c} \arrow[l] \\
\perp \arrow[r] & \mathsf{d} \arrow[ur] & &
\end{tikzcd}$$

Let $deps^*(\mathsf{c})$ be the transitive closure of the $deps$ relation starting from a command $\{\mathsf{c}\}$.

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd} a \arrow[rr, bend right] & & b \arrow[ll, bend right] & c \arrow[l] \\ \perp \arrow[r] & d \arrow[ur] & & \end{tikzcd}$$

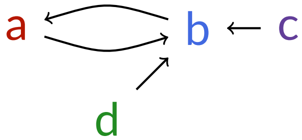
A command c is stable once it is committed and no command in $deps^*(\text{c})$ is pending.

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd}
a \arrow[rr, bend right] & & b \arrow[ll, bend right] & c \arrow[l] \\
\perp \arrow[r] & d \arrow[ur] & &
\end{tikzcd}$$

Here, only c is stable. All others are still pending, since their $deps^*$ include d (which is pending since it has $\perp$ as a dependency).

Leaderless SMR - Understanding the abstraction



In this example, all commands are stable.

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd} \perp \arrow[r] & b \arrow[rr, bend left] & a \arrow[r] \arrow[r, leftarrow, bend right] & c \end{tikzcd}$$

A command c gets aborted when $deps(\mathsf{c})$ is set to $\top$.

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd} \top \arrow[r] & b \arrow[rr, bend left] & a \arrow[r] & c \arrow[l, bend left] \end{tikzcd}$$

A command c gets aborted when $deps$(c) is set to $\top$.

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd} \top \arrow[r] & b \arrow[rr, bend left] & a \arrow[r] & c \arrow[l, bend left] \end{tikzcd}$$

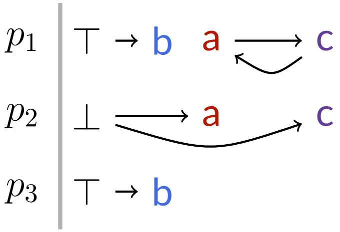
In that case, the command is removed from any $deps(\mathrm{d})$ and it will not appear later on.

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd} \top \arrow[r] & b & a \arrow[r] & c \arrow[l, bend left] \end{tikzcd}$$

In that case, the command is removed from any $deps(\mathrm{d})$ and it will not appear later on.

Leaderless SMR - Understanding the abstraction

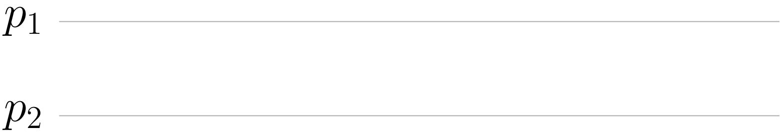


Guarantees:

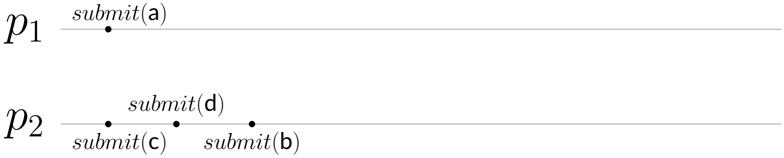
Stability: For each command c, there exists D such that if c is stable then $deps(\mathsf{c}) = D$.

Consistency: If a and b are both committed and conflicting, then a $\in deps(\mathsf{b})$ or b $\in deps(\mathsf{a})$.

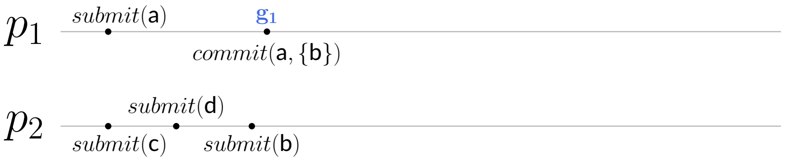
Leaderless SMR - Understanding the abstraction

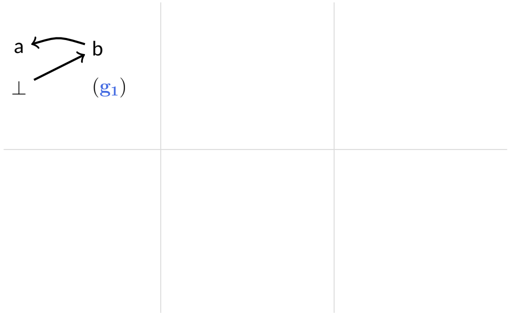


Leaderless SMR - Understanding the abstraction

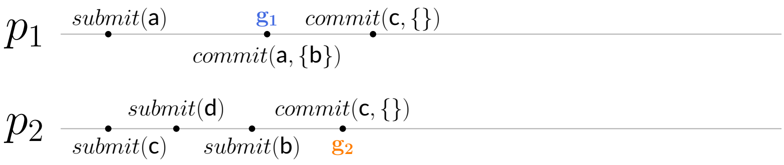


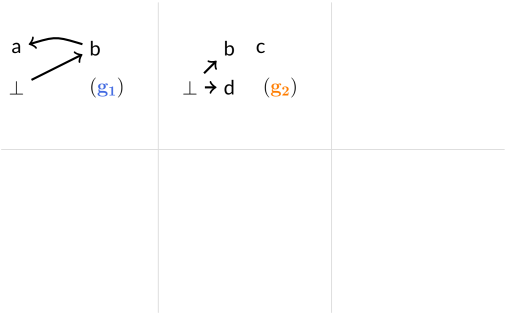
Leaderless SMR - Understanding the abstraction



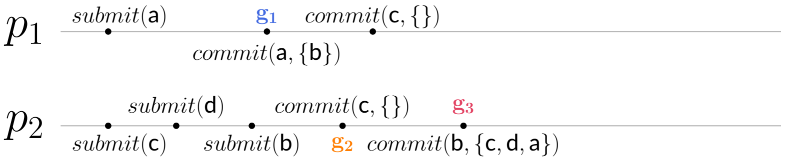


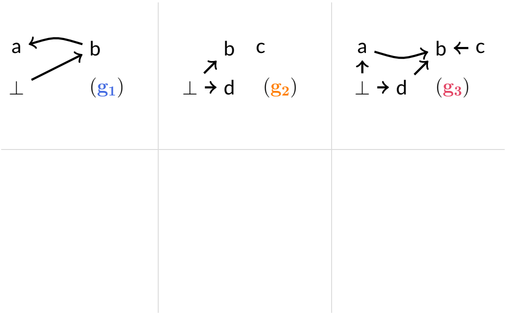
Leaderless SMR - Understanding the abstraction



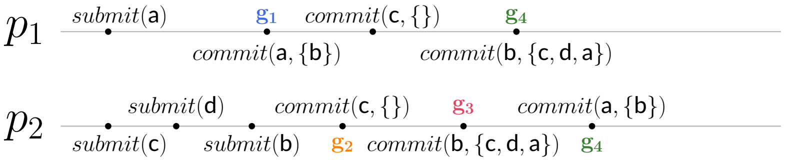


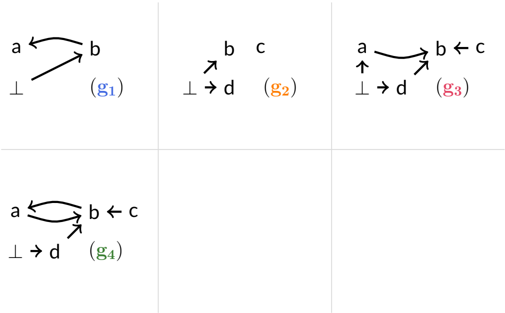
Leaderless SMR - Understanding the abstraction



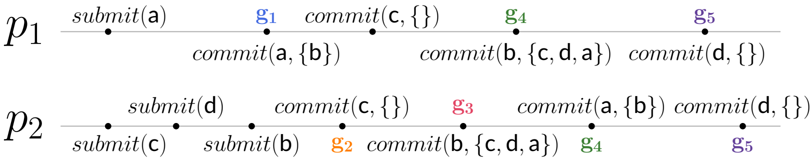


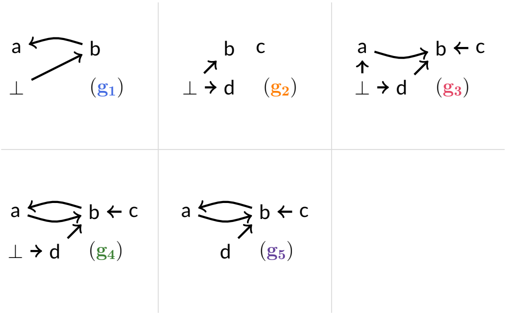
Leaderless SMR - Understanding the abstraction





Leaderless SMR - Understanding the abstraction





Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd} a \arrow[rr, bend right] & & b \arrow[ll, bend right] & c \arrow[l] \\ & d \arrow[ur] & (g_5) & \end{tikzcd}$$

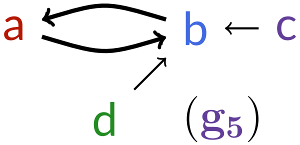
A command gets executed once it is stable.

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd}
\color{red}{a} \arrow[rr, bend right] & & \color{blue}{b} \arrow[ll, bend right] & \color{violet}{c} \arrow[l] \\
& \color{green}{d} \arrow[ur] & (\mathbf{g_5}) &
\end{tikzcd}$$

All commands are stable in $\mathbf{g_5}$, therefore they can be executed.

Leaderless SMR - Understanding the abstraction



Cycles are broken deterministically.

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd} \mathsf{a} \arrow[r, bend right] & \mathsf{b} \arrow[l, bend right] & \mathsf{c} \arrow[l] \\ \mathsf{d} \arrow[ur] & (g_5) \end{tikzcd}$$

$$commit(\mathsf{c}, \{\});\ commit(\mathsf{d}, \{\});\ commit(\mathsf{a}, \{\mathsf{b}\});\ commit(\mathsf{b}, \{\mathsf{a}\});$$

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd} a \arrow[rr, bend right] & & b \arrow[ll, bend right] & c \arrow[l] \\ & d \arrow[ur] & (g_5) & \end{tikzcd}$$

$$execute(c); execute(d); execute(a); execute(b) \text{ if } a < b$$

Leaderless SMR - Understanding the abstraction

$$\begin{tikzcd} a \arrow[rr, bend right] & & b \arrow[ll, bend right] & c \arrow[l] \\ & d \arrow[ur] & & (g_5) \end{tikzcd}$$

$$execute(c); execute(d); execute(b); execute(a) \text{ if } b < a$$

A General Framework for Leaderless SMR

- What is the essence of Leaderless SMR?
    - Leaderless SMR **does not** compute an ordering over conflicting commands.
    - Conflicting commands must simply observe one another (**Consistency** property).
- How to capture this feature?
    - Decomposition into two services: Dependency Discovery Service and Consensus Service.

Abstract protocol to decide a command

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$

  pre: $p = coord(\mathsf{c}) \lor coord(\mathsf{c}) \in \mathcal{D}$

  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$

    if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$

    $deps(\mathsf{c}) \leftarrow D$

    $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$

  eff: $deps(\mathsf{c}) \leftarrow D$

- Algorithm 2 depicts an abstract protocol to decide a command.

Abstract protocol to decide a command

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c})$:=
  pre: $\boxed{p = coord(\mathsf{c})} \lor coord(\mathsf{c}) \in \mathcal{D}$
  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$
    if $b = false$ **then** $D \leftarrow \mathsf{CONS_c}.propose(D)$
    $deps(\mathsf{c}) \leftarrow D$
    $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$
  eff: $deps(\mathsf{c}) \leftarrow D$

- $coord(\mathsf{c})$ is the initial process in charge of submitting the command to the replicated state machine.

Abstract protocol to decide a command

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$
    pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$
    eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$
        if $b = \mathit{false}$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$
        $\boxed{deps(\mathsf{c}) \leftarrow D}$
        $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$
    eff: $\boxed{deps(\mathsf{c}) \leftarrow D}$

- The algorithm computes a set of dependencies of a command abiding to properties **Stability** and **Consistency**

Abstract protocol to decide a command

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$

  pre: $p = coord(\mathsf{c}) \lor coord(\mathsf{c}) \in \mathcal{D}$

  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$

    if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$

    $deps(\mathsf{c}) \leftarrow D$

    $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$

  eff: $deps(\mathsf{c}) \leftarrow D$

- It uses a dependency discovery service (DDS).

Abstract protocol to decide a command

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$

  pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$

  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$

    if $b = false$ **then** $D \leftarrow \boxed{\mathsf{CONS}_\mathsf{c}.}propose(D)$

    $deps(\mathsf{c}) \leftarrow D$

    $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$

  eff: $deps(\mathsf{c}) \leftarrow D$

- A family of consensus objects ($(\mathsf{CONS}_\mathsf{c})_{\mathsf{c} \in \mathcal{C}}$).

Abstract protocol to decide a command

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$

  pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$

  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$

    if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$

    $deps(\mathsf{c}) \leftarrow D$

    $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$

  eff: $deps(\mathsf{c}) \leftarrow D$

- A failure detector ($\mathcal{D}$) that returns a set of suspected processes.

Abstract protocol to decide a command

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c})$:=
    pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$
    eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$
        if $b = false$ **then** $D \leftarrow \mathsf{CONS_c}.propose(D)$
        $deps(\mathsf{c}) \leftarrow D$
        $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$
    eff: $deps(\mathsf{c}) \leftarrow D$

- To create a valid proposal for $(\mathsf{CONS_c})_{\mathsf{c} \in \mathcal{C}}$, a process p relies on the DDS.

Abstract protocol to decide a command

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$
    pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$
    eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$
        if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$
        $deps(\mathsf{c}) \leftarrow D$
        $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$
    eff: $deps(\mathsf{c}) \leftarrow D$

- This shared object offers a single operation $announce(\mathsf{c})$ that returns a pair (D, b), where $D \in 2^{\mathcal{C}} \cup \{\top\}$ and $b \in \{0, 1\}$ is a flag.

Abstract protocol to decide a command

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$

  pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$

  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$

    if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$

    $deps(\mathsf{c}) \leftarrow D$

    $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$

  eff: $deps(\mathsf{c}) \leftarrow D$

- When the return value is in $2^{\mathcal{C}}$, the service suggests to **commit** the command. Otherwise, the command should be **aborted**.

Abstract protocol to decide a command

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$

  pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$

  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$

   if $b = false$ **then** $D \leftarrow \mathsf{CONS_c}.propose(D)$

   $deps(\mathsf{c}) \leftarrow D$

   $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$

  eff: $deps(\mathsf{c}) \leftarrow D$

- When the flag is set, the service indicates that a **spontaneous agreement occurs.**

Abstract protocol to decide a command

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$

  pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$

  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$

    if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$

    $deps(\mathsf{c}) \leftarrow D$

    $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$

  eff: $deps(\mathsf{c}) \leftarrow D$

- In such a case, p can directly commit c with the return value of the DDS service and bypass CONS_c; this is called a fast path.

Abstract protocol to decide a command

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c})$:=
    pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$
    eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$
        if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$
        $deps(\mathsf{c}) \leftarrow D$
        $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$
    eff: $deps(\mathsf{c}) \leftarrow D$

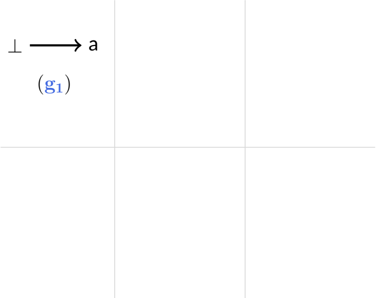
Abstract protocol to decide a command

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$
  pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$
  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$
    if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$
    $deps(\mathsf{c}) \leftarrow D$
    $send(\mathsf{c}, deps(\mathsf{c}))$ to $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$
  eff: $deps(\mathsf{c}) \leftarrow D$

Abstract protocol to decide a command

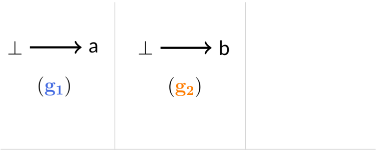


Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c})$:=
    pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$
    eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$
        if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$
        $deps(\mathsf{c}) \leftarrow D$
        $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$
    eff: $deps(\mathsf{c}) \leftarrow D$

Abstract protocol to decide a command

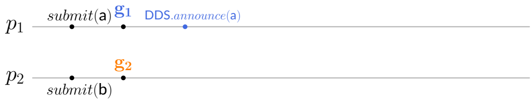


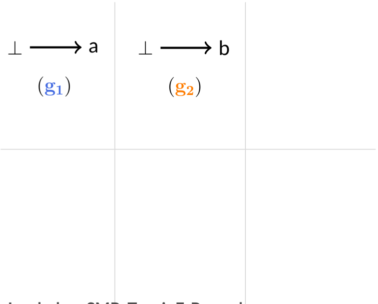
Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c})$:=
    pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$
    eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$
        if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$
        $deps(\mathsf{c}) \leftarrow D$
        $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$
    eff: $deps(\mathsf{c}) \leftarrow D$

Abstract protocol to decide a command





Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$

  pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$

  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$

    if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$

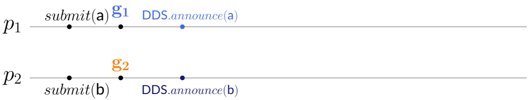
    $deps(\mathsf{c}) \leftarrow D$

    $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$

  eff: $deps(\mathsf{c}) \leftarrow D$

Abstract protocol to decide a command



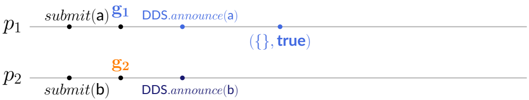
$\perp \longrightarrow \mathsf{a}$	$\perp \longrightarrow \mathsf{b}$
(g_1)	(g_2)

Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$
  pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$
  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$
    if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$
    $deps(\mathsf{c}) \leftarrow D$
    $send(\mathsf{c}, deps(\mathsf{c}))$ to $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$
  eff: $deps(\mathsf{c}) \leftarrow D$

Abstract protocol to decide a command



$\perp \longrightarrow \mathsf{a}$ (g_1)

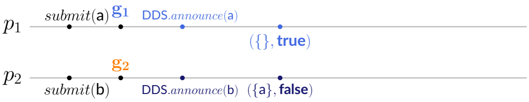
$\perp \longrightarrow \mathsf{b}$ (g_2)

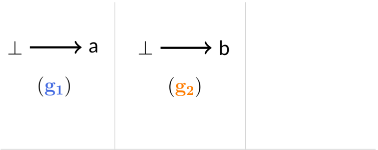
Algorithm 2 Deciding a command c – code at process p

```
submit(c) :=
    pre: p = coord(c) ∨ coord(c) ∈ D
    eff: (D, b) ← DDS.announce(c)
        if b = false then D ← CONS_c.propose(D)
        deps(c) ← D
        send(c, deps(c)) to Π \ {p}

when recv(c, D)
    eff: deps(c) ← D
```

Abstract protocol to decide a command





Algorithm 2 Deciding a command c – code at process p

$\mathit{submit}(\mathsf{c}) :=$

  pre: $p = \mathit{coord}(\mathsf{c}) \lor \mathit{coord}(\mathsf{c}) \in \mathcal{D}$

  eff: $(D, b) \leftarrow \mathsf{DDS}.\mathit{announce}(\mathsf{c})$

    if $b = \mathit{false}$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.\mathit{propose}(D)$

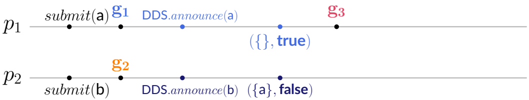
    $\mathit{deps}(\mathsf{c}) \leftarrow D$

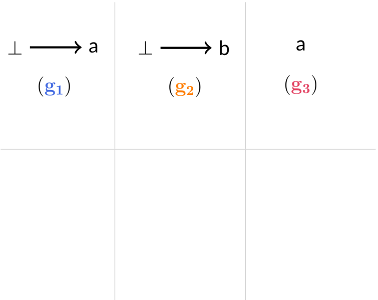
    $\mathit{send}(\mathsf{c}, \mathit{deps}(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $\mathit{recv}(\mathsf{c}, D)$

  eff: $\mathit{deps}(\mathsf{c}) \leftarrow D$

Abstract protocol to decide a command





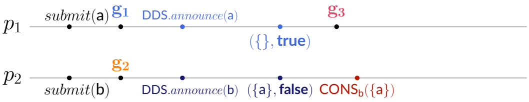
Algorithm 2 Deciding a command c – code at process p

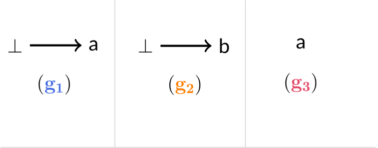
```
submit(c) :=
    pre:  p = coord(c) ∨ coord(c) ∈ 𝒟
    eff:  (D, b) ← DDS.announce(c)
          if b = false then D ← CONS_c.propose(D)
          deps(c) ← D
          send(c, deps(c)) to Π \ {p}

when recv(c, D)
    eff:  deps(c) ← D

```

Abstract protocol to decide a command





Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$

  pre: $p = coord(\mathsf{c}) \lor coord(\mathsf{c}) \in \mathcal{D}$

  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$

    if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$

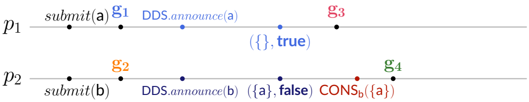
    $deps(\mathsf{c}) \leftarrow D$

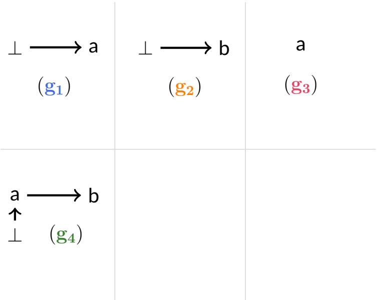
    $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$

  eff: $deps(\mathsf{c}) \leftarrow D$

Abstract protocol to decide a command



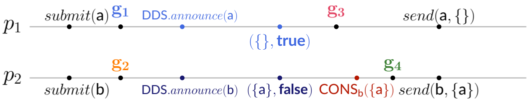


Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$
  pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$
  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$
    if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$
    $deps(\mathsf{c}) \leftarrow D$
    $send(\mathsf{c}, deps(\mathsf{c}))$ to $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$
  eff: $deps(\mathsf{c}) \leftarrow D$

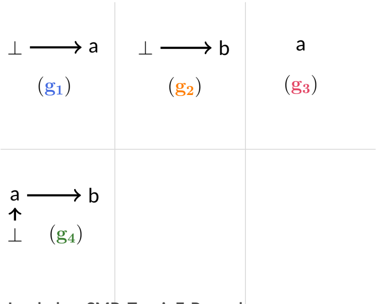
Abstract protocol to decide a command



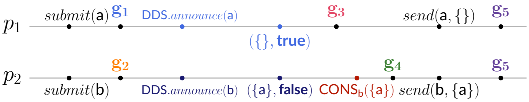
Algorithm 2 Deciding a command c – code at process p

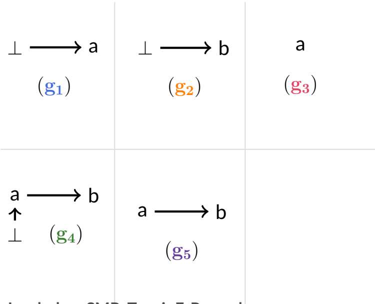
$submit(\mathsf{c}) :=$
  pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$
  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$
    if $b = false$ **then** $D \leftarrow \mathsf{CONS_c}.propose(D)$
    $deps(\mathsf{c}) \leftarrow D$
    $\boxed{send(\mathsf{c}, deps(\mathsf{c}))\ to\ \Pi \setminus \{p\}}$

when $recv(\mathsf{c}, D)$
  eff: $deps(\mathsf{c}) \leftarrow D$



Abstract protocol to decide a command





Algorithm 2 Deciding a command c – code at process p

$submit(\mathsf{c}) :=$
  pre: $p = coord(\mathsf{c}) \vee coord(\mathsf{c}) \in \mathcal{D}$
  eff: $(D, b) \leftarrow \mathsf{DDS}.announce(\mathsf{c})$
    if $b = false$ **then** $D \leftarrow \mathsf{CONS}_{\mathsf{c}}.propose(D)$
    $deps(\mathsf{c}) \leftarrow D$
    $send(\mathsf{c}, deps(\mathsf{c}))$ *to* $\Pi \setminus \{p\}$

when $recv(\mathsf{c}, D)$
  eff: $deps(\mathsf{c}) \leftarrow D$

Core Properties

Multiple implementations of Dependency Discovery Service and Consensus Service are possible.

- We have cast different leaderless SMR protocols into our framework.
- How to further **characterize** the protocols?

Core Properties

Multiple implementations of Dependency Discovery Service and Consensus Service are possible.

- We have cast different leaderless SMR protocols into our framework.
- How to further **characterize** the protocols?

Core properties:

(Reliability)

(Optimal Latency)

(Load Balancing)

Core Properties

(Reliability) In every run, if there are at most f failures, every submitted command gets eventually decided at every correct process.

Core Properties

(Reliability) In every run, if there are at most f failures, every submitted command gets eventually decided at every correct process.

- Rationale:
  - SMR helps to mask failures and asynchrony in a distributed system.
  - Parameter f captures the largest number of failures tolerated by a protocol.

Core Properties

(Optimal Latency) During a nice run, every call to $announce(\mathsf{c})$ returns after two message delays in the absence of conflicting commands

Core Properties

(Optimal Latency) During a nice run, every call to $announce(\mathsf{c})$ returns after two message delays in the absence of conflicting commands

- Rationale:
  - Leaderless SMR protocols exploit the absence of contention to boost performance.
  - Some protocols are able to execute a command after a single round-trip (optimal).

Core Properties

(Load Balancing) During a nice run, any $n - F$ replicas can be used to announce a command c.

Core Properties

(Load Balancing) During a nice run, any $n - F$ replicas can be used to announce a command c.

- Rationale:
  - The replicas that take place in the fast path vary from one protocol to another.
  - For example, Mencius use all the processes.
  - Captures scalability, as any quorum of $n - F$ processes may be used to order a command.

Complexity result - ROLL theorem

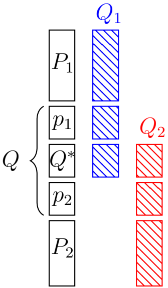
Theorem

Consider an SMR protocol that satisfies the ROLL properties. Then, it is true that $2F + f - 1 \leq n$.

- The inequation presented in the theorem expresses the trade-off between scalability and fault-tolerance.
- Differently than Lamport's lower bound (*Fast Learning Theorem*), ROLL captures more accurately this trade-off in leaderless protocols.

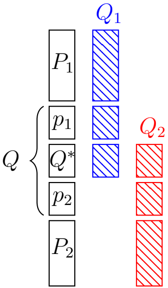
ROLL theorem - Proof sketch

By contradiction, using a round-based reasoning. Let us assume a protocol $\mathcal{P}$ that satisfies all the ROLL properties with $2F + f - 1 > n$.

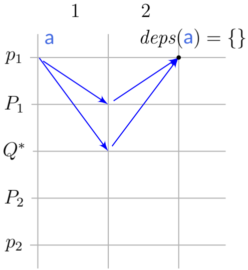


Quorums in use

ROLL theorem - Proof sketch

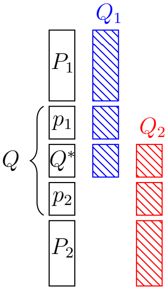


Quorums in use

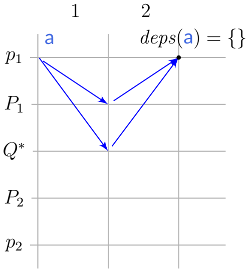


Run λ_1

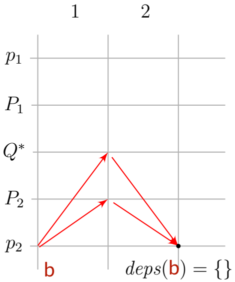
ROLL theorem - Proof sketch



Quorums in use

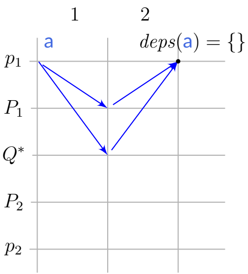


Run λ_1

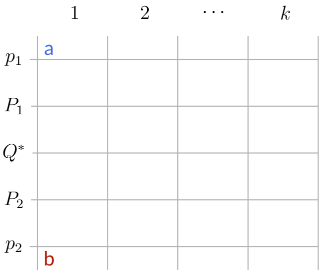


Run λ_2

ROLL theorem - Proof sketch

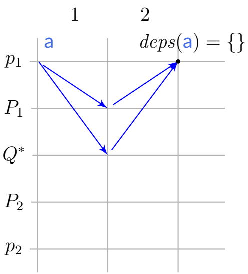


Run λ_1

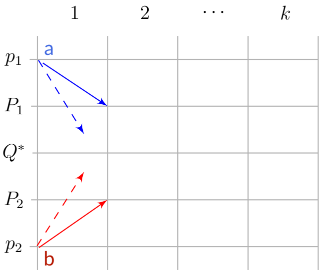


Run λ_3

ROLL theorem - Proof sketch

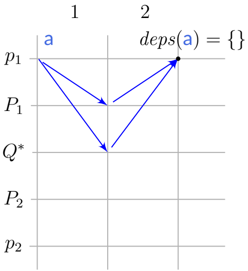


Run λ_1

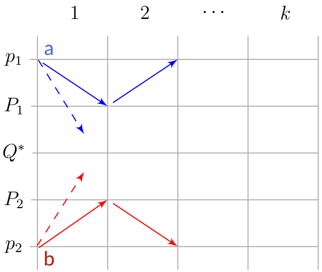


Run λ_3

ROLL theorem - Proof sketch

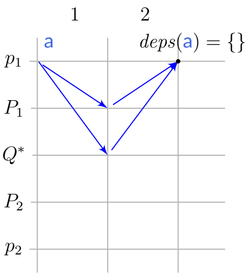


Run λ_1

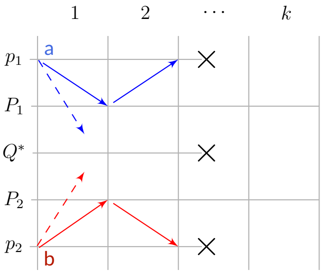


Run λ_3

ROLL theorem - Proof sketch

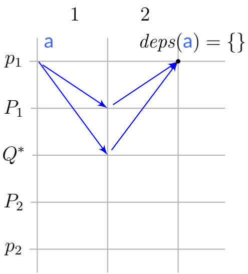


Run λ_1

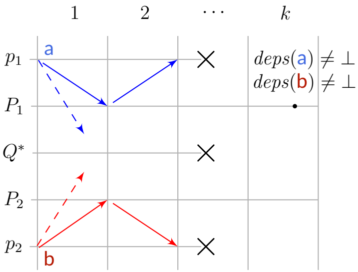


Run λ_3

ROLL theorem - Proof sketch

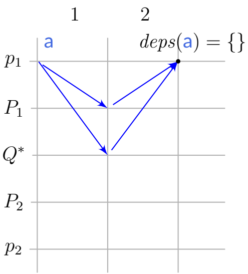


Run λ_1

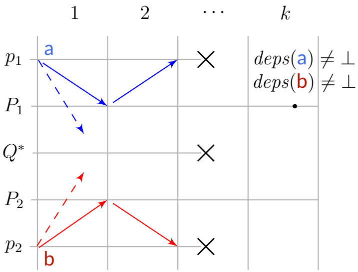


Run λ_3

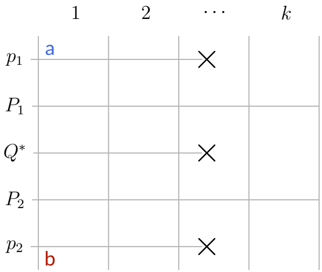
ROLL theorem - Proof sketch



Run λ_1

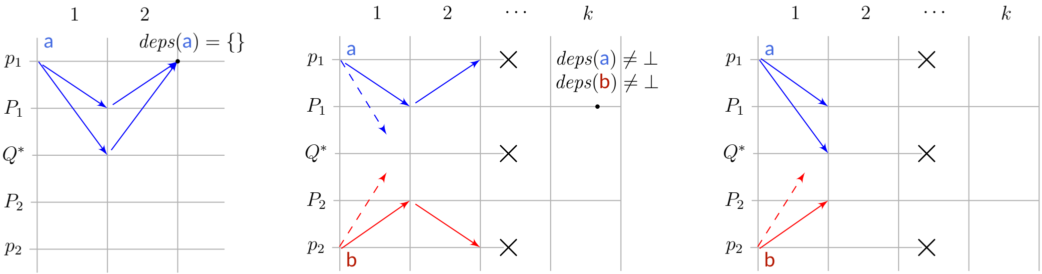


Run λ_3



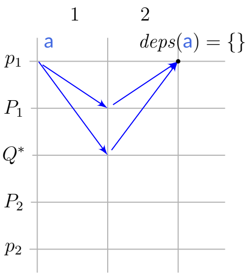
Run λ_4

ROLL theorem - Proof sketch

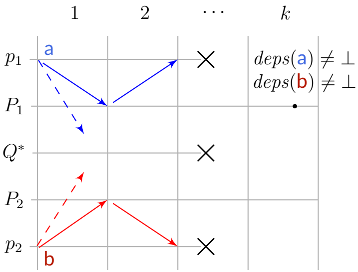


Run λ_1 Run λ_3 Run λ_4

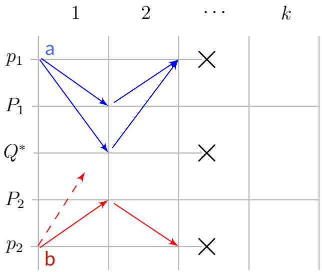
ROLL theorem - Proof sketch



Run λ_1

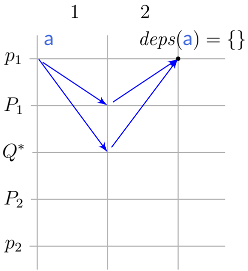


Run λ_3

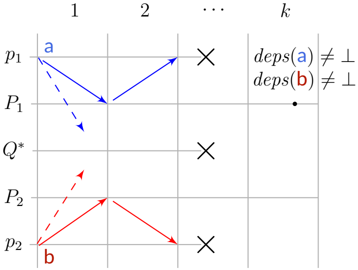


Run λ_4

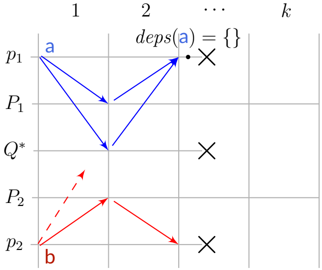
ROLL theorem - Proof sketch



Run λ_1

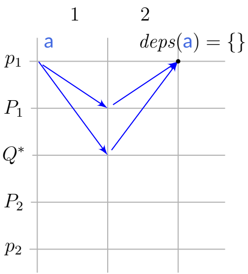


Run λ_3

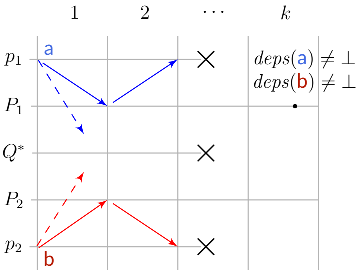


Run λ_4

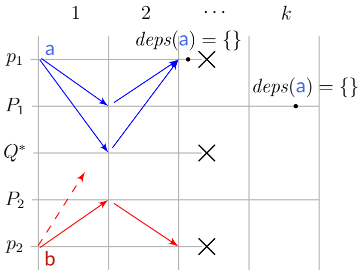
ROLL theorem - Proof sketch



Run λ_1

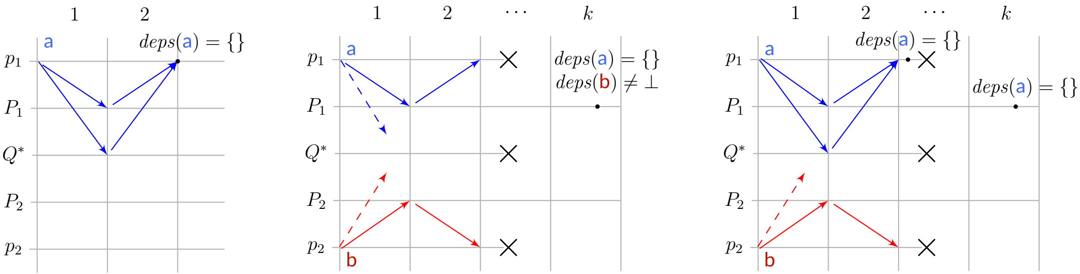


Run λ_3



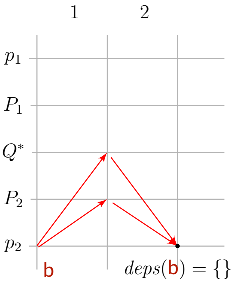
Run λ_4

ROLL theorem - Proof sketch

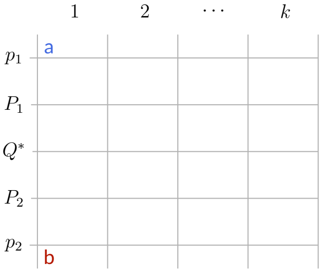


Run λ_1 Run λ_3 Run λ_4

ROLL theorem - Proof sketch

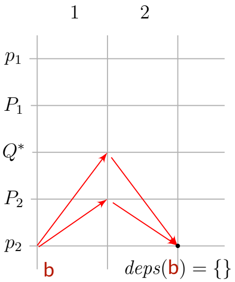


Run λ_2

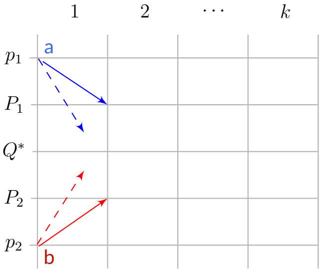


Run λ_3

ROLL theorem - Proof sketch

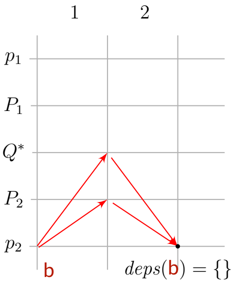


Run λ_2

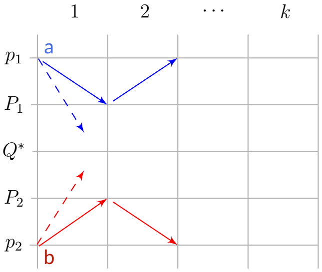


Run λ_3

ROLL theorem - Proof sketch

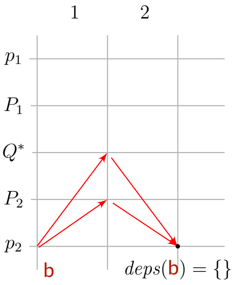


Run λ_2

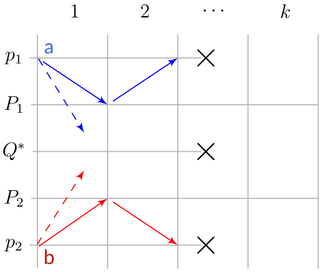


Run λ_3

ROLL theorem - Proof sketch

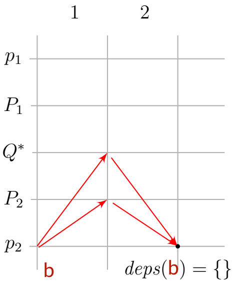


Run λ_2

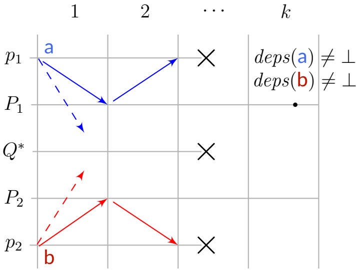


Run λ_3

ROLL theorem - Proof sketch

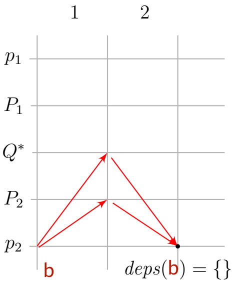


Run λ_2

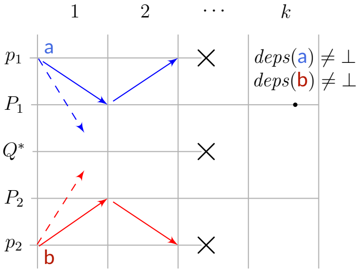


Run λ_3

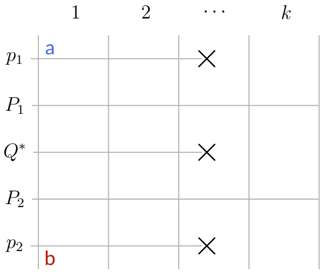
ROLL theorem - Proof sketch



Run λ_2

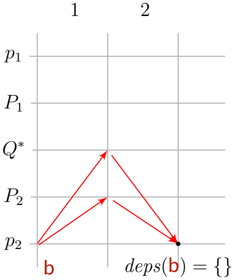


Run λ_3

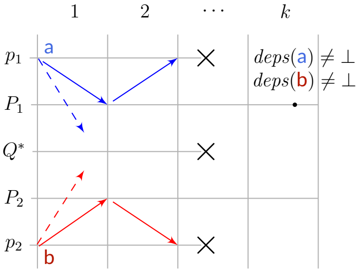


Run λ_5

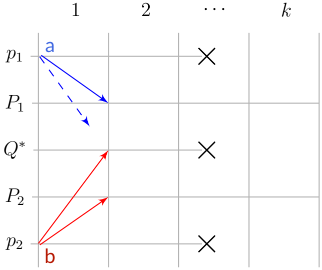
ROLL theorem - Proof sketch



Run λ_2

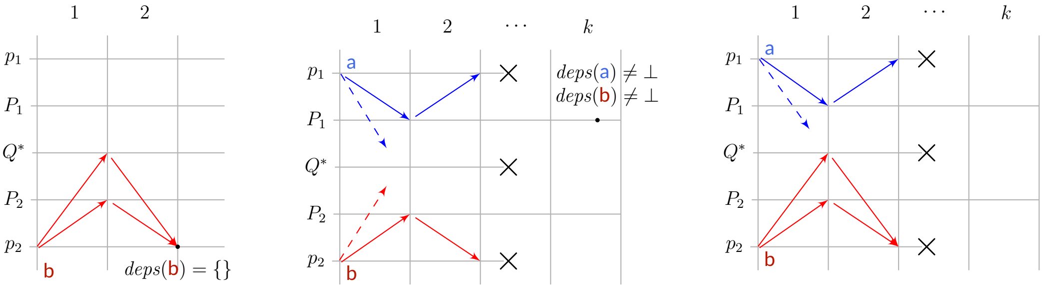


Run λ_3



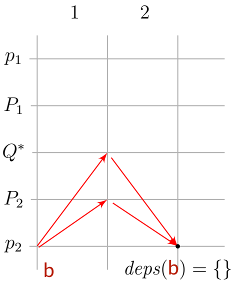
Run λ_5

ROLL theorem - Proof sketch

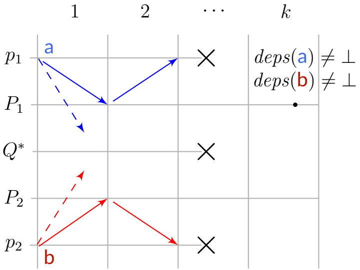


Run λ_2 Run λ_3 Run λ_5

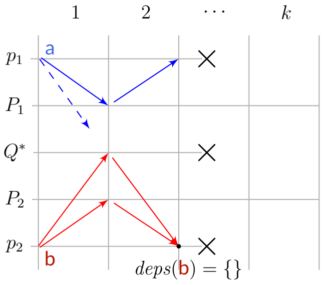
ROLL theorem - Proof sketch



Run λ_2

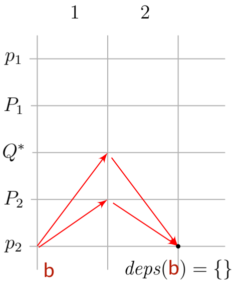


Run λ_3

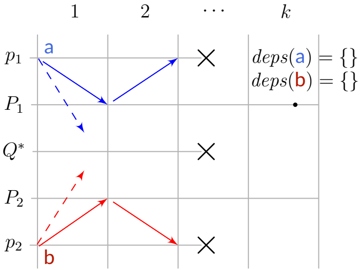


Run λ_5

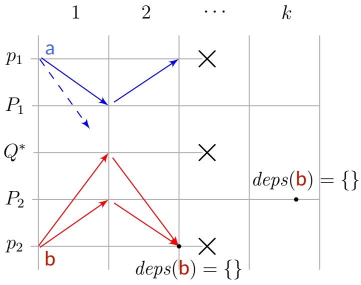
ROLL theorem - Proof sketch



Run λ_2



Run λ_3



Run λ_5

Complexity result - Optimality

Theorem

Consider an SMR protocol that satisfies the ROLL properties. Then, it is true that $2F + f - 1 \leq n$.

- A protocol is *ROLL-optimal* when the parameters F and f cannot be improved according to the theorem.

For example:

- With $n = 5$, there is a single such tuple: $(F, f) = (2, 2)$.

Complexity result - Optimality

Theorem

Consider an SMR protocol that satisfies the ROLL properties. Then, it is true that $2F + f - 1 \leq n$.

- A protocol is *ROLL-optimal* when the parameters F and f cannot be improved according to the theorem.

For example:

- With $n = 7$, there are two tuples possible: $(F, f) = (2, 3)$ and $(3, 2)$.

Complexity result - Optimality

Theorem

Consider an SMR protocol that satisfies the ROLL properties. Then, it is true that $2F + f - 1 \leq n$.

- A protocol is *ROLL-optimal* when the parameters F and f cannot be improved according to the theorem.

For example:

- Epaxos is optimal for $n = 5$.

Complexity result - Optimality

Theorem

Consider an SMR protocol that satisfies the ROLL properties. Then, it is true that $2F + f - 1 \leq n$.

- A protocol is *ROLL-optimal* when the parameters F and f cannot be improved according to the theorem.

For example:

- No known optimal protocol when $n = 7$.

Conclusion

- The decomposition into Dependency Discovery Service and Consensus Service.

Conclusion

- The decomposition into Dependency Discovery Service and Consensus Service.
- Leaderless SMR Properties.

Conclusion

- The decomposition into Dependency Discovery Service and Consensus Service.
- Leaderless SMR Properties.
- Complexity results:
    - ROLL theorem. Explains the trade-off between reliability and scalability.
    - Chaining effect. Explains the reason for the high latency found in real-world implementation of leaderless protocols.