Leaderless State Machine Replication: Specification, Properties, Limits

Tuanir França Rezende \textsuperscript{1}  
Pierre Sutra \textsuperscript{1}

\textsuperscript{1}Telecom SudParis

14 Oct. 2020, DISC’20
Modern internet services often maintain more than one **copy** of an object.
Context - Principles

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

```
p1 | b → a → c

p2 | b → a

p3 | 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?

\[
\begin{array}{c|c}
  p_1 & \top \rightarrow b \quad a \quad \rightarrow c \\
  p_2 & \bot \quad \rightarrow a \quad \rightarrow c \\
  p_3 & \top \rightarrow b \\
\end{array}
\]
Leaderless SMR - Dependency Graph

Relies on the notion of **dependency graph** instead of a partially ordered log as found in Generic SMR.
A dependency graph is a directed graph that records the constraints defining how commands are executed.
For some command c, the incoming neighbors of c in the dependency graph are its dependencies.
In Leaderless SMR, a process holds two mapping: $deps$ and $phase$. 
The mapping $deps$ is a dependency graph storing a relation from $\mathcal{C}$ to $2^{\mathcal{C}} \cup \{\bot, \top\}$. \(^1\)

---

\(^1\mathcal{C}\) is the set of commands.
Initially, for every command $c$, $\text{deps}(c)$ is set to $\bot$. This corresponds to the pending phase.
When a process decides a command $c$, it changes the mapping $\text{deps}(c)$ to a non-$\bot$ value. Operation $\text{commit}(c, D)$ assigns $D$ taken in $2^c$ to $\text{deps}(c)$. 
Leaderless SMR - Understanding the abstraction

commit(c, \{a, b\})
Leaderless SMR - Understanding the abstraction

\[
\text{commit}(c, \{a, b\})
\]
Leaderless SMR - Understanding the abstraction

\[
\text{commit}(c, \{a, b\})
\]
Leaderless SMR - Understanding the abstraction

\[
\text{commit}(c, \{a, b\})
\]
Leaderless SMR - Understanding the abstraction

\[ \text{commit}(a, \{c\}) \]
Leaderless SMR - Understanding the abstraction

\[ \text{commit}(a, \{c\}) \]

\[ \begin{aligned} & b \quad a \quad c \\ & \downarrow \quad \downarrow \quad \downarrow \\ & b \quad a \quad c \\ & \end{aligned} \]
Leaderless SMR - Understanding the abstraction

\[ \text{commit}(a, \{c\}) \]
Let $\text{deps}^*(c)$ be the transitive closure of the $\text{deps}$ relation starting from a command \{c\}.
A command $c$ is stable once it is committed and no command in $\text{deps}^*(c)$ is pending.
Here, only $c$ is stable. All others are still pending, since their $deps^*$ include $d$ (which is pending since it has $\bot$ as a dependency).
In this example, all commands are stable.
A command $c$ gets aborted when $\text{deps}(c)$ is set to $\top$. 
Leaderless SMR - Understanding the abstraction

A command $c$ gets aborted when $\text{deps}(c)$ is set to $\top$. 
In that case, the command is removed from any $\text{deps}(d)$ and it will not appear later on.
In that case, the command is removed from any $\text{deps}(d)$ and it will not appear later on.
Leaderless SMR - Understanding the abstraction

\[
\begin{align*}
\text{p}_1 & : T \rightarrow b \quad a \xrightarrow{\sim} c \\
\text{p}_2 & : T \rightarrow a \xrightarrow{\sim} c \\
\text{p}_3 & : T \rightarrow b
\end{align*}
\]

Guarantees:

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

**Consistency:** If \( a \) and \( b \) are both committed and conflicting, then \( a \in \text{deps}(b) \) or \( b \in \text{deps}(a) \).
Leaderless SMR - Understanding the abstraction

\[ p_1 \]

\[ p_2 \]
Leaderless SMR - Understanding the abstraction

\[ p_1 \quad \text{submit}(a) \]

\[ p_2 \quad \begin{align*}
& \text{submit}(d) \\
& \text{commit}(c) \quad \text{commit}(b)
\end{align*} \]
Leaderless SMR - Understanding the abstraction

\[ p_1 \quad \begin{array}{c}
\text{submit}(a) \\
\text{commit}(a, \{b\})
\end{array} \]

\[ p_2 \quad \begin{array}{c}
\text{submit}(c) \\
\text{submit}(b)
\end{array} \]

a ← b

\( \perp \quad (g_1) \)
Leaderless SMR - Understanding the abstraction

\[ p_1 \quad \text{submit}(a) \rightarrow^* g_1 \quad \text{commit}(c, \{\}) \]
\[ \quad \text{commit}(a, \{b\}) \]

\[ p_2 \quad \text{commit}(c, \{\}) \]
\[ \quad \text{submit}(d) \rightarrow^* g_2 \quad \text{commit}(c, \{\}) \]
\[ \quad \text{submit}(c) \quad \text{submit}(b) \]

\[ a \xleftrightarrow{\perp} b \quad (g_1) \]
\[ b \xrightarrow{\perp} (g_2) \]
Leaderless SMR - Understanding the abstraction

\[ p_1 \quad \frac{\text{submit}(a)}{\text{commit}(a, \{b\})} \]

\[ p_2 \quad \frac{\text{submit}(d) \quad \text{commit}(c, \{\}) \quad \text{commit}(b, \{c, d, a\})}{\text{commit}(b, \{c, d, a\})} \]
Leaderless SMR - Understanding the abstraction

\[ p_1 \begin{array}{l}
\text{submit}(a) \\
\text{commit}(c, \{\}) \\
\text{commit}(a, \{b\}) \\
\text{commit}(b, \{c, d, a\}) \\
\end{array} \]

\[ p_2 \begin{array}{l}
\text{submit}(d) \\
\text{commit}(c, \{\}) \\
\text{commit}(b, \{c, d, a\}) \\
\end{array} \]

\[ a \overset{\perp}{\leftarrow} b \text{ (g_1)} \]
\[ b \overset{\perp}{\rightarrow} d \text{ (g_2)} \]
\[ a \overset{\perp}{\leftarrow} b \overset{\perp}{\rightarrow} c \text{ (g_3)} \]
\[ a \overset{\perp}{\rightarrow} b \text{ (g_4)} \]
Leaderless SMR - Understanding the abstraction

\[ p_1 \begin{array}{lllll}
\text{submit}(a) & g_1 & \text{commit}(c, \{\}) & g_4 & g_5 \\
\text{commit}(a, \{b\}) & commit(b, \{c, d, a\}) & commit(d, \{\})
\end{array} \]

\[ p_2 \begin{array}{lllll}
\text{submit}(d) & \text{commit}(c, \{\}) & g_3 & \text{commit}(a, \{b\}) & \text{commit}(d, \{\}) \\
\text{commit}(b, \{c, d, a\}) & g_4 & g_5
\end{array} \]
A command gets executed once it is stable.
Leaderless SMR - Understanding the abstraction

All commands are stable in $g_5$, therefore they can be executed.
Leaderless SMR - Understanding the abstraction

Cycles are broken deterministically.
Leaderless SMR - Understanding the abstraction

commit(c, {}); commit(d, {}); commit(a, {b}); commit(b, {a});
Leaderless SMR - Understanding the abstraction

\[
\begin{align*}
&\text{execute}(c) ; \text{execute}(d) ; \text{execute}(a) ; \text{execute}(b) \text{ if } a < b
\end{align*}
\]
Leaderless SMR - Understanding the abstraction

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

1: $\text{submit}(c) :=$
2: $\textbf{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in \mathcal{D}$
3: $\textbf{eff: } (D, b) \leftarrow \text{DDS.announce}(c)$
4: \hspace{1em} $\text{if } b = \text{false then } D \leftarrow \text{CONS}_c.\text{propose}(D)$
5: \hspace{1em} $\text{deps}(c) \leftarrow D$
6: \hspace{1em} $\text{send}(c, \text{deps}(c))$ to $\Pi \setminus \{p\}$
7: 
8: $\textbf{when } \text{recv}(c, D)$
9: $\textbf{eff: } \text{deps}(c) \leftarrow D$
10:

- 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 \( p \)

1: \( \text{submit}(c) := \)
2: \( \text{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in \mathcal{D} \)
3: \( \text{eff: } (D, b) \leftarrow \text{DDS.announce}(c) \)
4: \( \text{if } b = \text{false then } D \leftarrow \text{CONS}_c.\text{propose}(D) \)
5: \( \text{deps}(c) \leftarrow D \)
6: \( \text{send}(c, \text{deps}(c)) \) to \( \Pi \setminus \{p\} \)
7: 
8: \( \text{when } \text{recv}(c, D) \)
9: \( \text{eff: } \text{deps}(c) \leftarrow D \)
10: 

- \( \text{coord}(c) \) 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 $p$

1: $\text{submit}(c) :=$
2: $\text{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in \mathcal{D}$
3: $\text{eff: } (D, b) \leftarrow \text{DDS.announce}(c)$
4: $\text{if } b = \text{false then } D \leftarrow \text{CONS}_c\text{.propose}(D)$
5: $\qquad \textbf{deps}(c) \leftarrow D$
6: $\text{send}(c, \text{deps}(c)) \text{ to } \Pi \setminus \{p\}$
7: $\text{when } \text{recv}(c, D)$
8: $\text{eff: } \textbf{deps}(c) \leftarrow D$
9: $\text{10: }$

- 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 $p$

1: $\textit{submit}(c) :=$
2: \hspace{1em} \textbf{pre: } p = \textit{coord}(c) \lor \textit{coord}(c) \in \mathcal{D}$
3: \hspace{1em} \textbf{eff: } $(D, b) \leftarrow \texttt{DDS.announce}(c)$
4: \hspace{2em} \textbf{if } b = \textit{false } \textbf{then } D \leftarrow \textit{CONS}_c.\textit{propose}(D)$
5: \hspace{1em} \hspace{1em} $\textit{deps}(c) \leftarrow D$
6: \hspace{1em} \hspace{1em} $\textit{send}(c, \textit{deps}(c))$ to $\Pi \setminus \{p\}$
7: 
8: \hspace{1em} \textbf{when } \textit{recv}(c, D)$
9: \hspace{1em} \hspace{1em} \textbf{eff: } $\textit{deps}(c) \leftarrow D$
10: 

- It uses a dependency discovery service (DDS).
Abstract protocol to decide a command

Algorithm 2 Deciding a command $c$ – code at process $p$

1: $\text{submit}(c) :=$
2: \hspace{1em} \textbf{pre:} $\ p = \text{coord}(c) \lor \text{coord}(c) \in \mathcal{D}$
3: \hspace{1em} \textbf{eff:} $(D, b) \leftarrow \text{DDS.announce}(c)$
4: \hspace{2em} \text{if } b = \text{false} \text{ then } D \leftarrow \text{CONS}_c.\text{propose}(D)$
5: \hspace{1em} $\ \text{deps}(c) \leftarrow D$
6: \hspace{1em} $\text{send}(c, \text{deps}(c))$ to $\Pi \setminus \{p\}$
7: \hspace{1em} $\ \text{when } \text{recv}(c, D)$
8: \hspace{2em} $\ \text{eff: } \text{deps}(c) \leftarrow D$
9: \hspace{1em} $\text{deps}(c) \leftarrow D$
10:

• A family of consensus objects $((\text{CONS}_c)_{c \in \mathcal{C}})$. 
Abstract protocol to decide a command

Algorithm 2 Deciding a command \( c \) – code at process \( p \)

1: \( \text{submit}(c) \) :=
2: \( \text{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in \mathcal{D} \)
3: \( \text{eff: } (D, b) \leftarrow \text{DDS.announce}(c) \)
4: \( \text{if } b = \text{false then } D \leftarrow \text{CONS}_c.\text{propose}(D) \)
5: \( \text{deps}(c) \leftarrow D \)
6: \( \text{send}(c, \text{deps}(c)) \) to \( \Pi \setminus \{p\} \)
7:  
8: \( \text{when } \text{recv}(c, D) \)
9: \( \text{eff: } \text{deps}(c) \leftarrow D \)
10: 

- A failure detector \( \mathcal{D} \) that returns a set of suspected processes.
Abstract protocol to decide a command

To create a valid proposal for \((\text{CONS}_c)_{c \in C}\), a process \(p\) relies on the DDS.

**Algorithm 2** Deciding a command \(c\) – code at process \(p\)

1: \(\text{submit}(c) :=\)
2: \textbf{pre:} \(p = \text{coord}(c) \lor \text{coord}(c) \in \mathcal{D}\)
3: \textbf{eff:} \((D, b) \leftarrow \text{DDS.announce}(c)\)
4: \quad \text{if} \ b = \text{false} \text{ then } D \leftarrow \text{CONS}_c.\text{propose}(D)\
5: \quad \text{deps}(c) \leftarrow D
6: \quad \text{send}(c, \text{deps}(c)) \text{ to } \Pi \setminus \{p\}
7: \quad \textbf{when} \ \text{recv}(c, D)
8: \quad \textbf{eff:} \ \text{deps}(c) \leftarrow D
9: 
10: 

DISC’20. Leaderless SMR. Tuanir F. Rezende.
Abstract protocol to decide a command

Algorithm 2 Deciding a command \( c \) – code at process \( p \)

1: \( \text{submit}(c) := \) 
2: \( \text{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in \mathcal{D} \) 
3: \( \text{eff: } (D, b) \leftarrow \text{DDS.announce}(c) \) 
4: \( \text{if } b = \text{false} \text{ then } D \leftarrow \text{CONSc.propose}(D) \) 
5: \( \text{deps}(c) \leftarrow D \) 
6: \( \text{send}(c, \text{deps}(c)) \) to \( \Pi \setminus \{p\} \) 
7: 
8: \( \text{when } \text{recv}(c, D) \) 
9: \( \text{eff: } \text{deps}(c) \leftarrow D \) 
10: 

- This shared object offers a single operation \( \text{announce}(c) \) 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 $p$

1: $submit(c) :=$
2: **pre:** $p = coord(c) \lor coord(c) \in \mathcal{D}$
3: **eff:** $(D, b) \leftarrow DDS.announce(c)$
4: \hspace{1em} if $b = false$ then $D \leftarrow CONS_c.propose(D)$
5: \hspace{1em} $deps(c) \leftarrow D$
6: \hspace{1em} $send(c, deps(c)) \to \Pi \setminus \{p\}$
7: 
8: **when** $recv(c, D)$
9: \hspace{1em} **eff:** $deps(c) \leftarrow D$
10: 

- 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 \( p \)

1: \( \text{submit}(c) := \)
2: \[ \text{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in \mathcal{D} \]
3: \[ \text{eff: } (D, b) \leftarrow \text{DDS.announce}(c) \]
4: \[ \text{if } b = \text{false} \text{ then } D \leftarrow \text{CONS}_c.\text{propose}(D) \]
5: \[ \text{deps}(c) \leftarrow D \]
6: \[ \text{send}(c, \text{deps}(c)) \text{ to } \Pi \setminus \{p\} \]
7: 
8: \[ \text{when } \text{recv}(c, D) \]
9: \[ \text{eff: } \text{deps}(c) \leftarrow D \]
10: 

- 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 $p$

1: \( \text{submit}(c) := \)
2: \quad \text{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in \mathcal{D} 
3: \quad \text{eff: } (D, p) \leftarrow \text{DDS.announce}(c) 
4: \quad \text{if } b = \text{false then } D \leftarrow \text{CONS}_c.\text{propose}(D) 
5: \quad \text{deps}(c) \leftarrow D 
6: \quad \text{send}(c, \text{deps}(c)) \text{ to } \Pi \setminus \{p\} 
7: 
8: \text{when } \text{recv}(c, D) 
9: \quad \text{eff: } \text{deps}(c) \leftarrow D 
10: 

- 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

\[ p_1 \]

\[ p_2 \]

\[ g_1 \]

\[ g_2 \]

\[ g_3 \]

\[ g_4 \]

\[ g_5 \]

\[ \bot \]

Algorithm 2 Deciding a command \( c \) – code at process \( p \)

1: \( \text{submit}(c) := \)
2: \( \text{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in D \)
3: \( \text{eff: } (D, b) \leftarrow \text{DDS.announce}(c) \)
4: \( \text{if } b = \text{false then } D \leftarrow \text{CONS}_{c,p}(D) \)
5: \( \text{deps}(c) \leftarrow D \)
6: \( \text{send}(c, \text{deps}(c)) \text{ to } II \setminus \{p\} \)
7: \( \) 
8: \( \text{when } \text{recv}(c, D) \)
9: \( \text{eff: } \text{deps}(c) \leftarrow D \)
10: \( \)
Abstract protocol to decide a command

\[ p_1 \quad \text{submit}(a) \]

\[ p_2 \quad \text{submit}(b) \]

---

**Algorithm 2** Deciding a command $c$ – code at process $p$

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>$\text{submit}(c) :=$</td>
</tr>
<tr>
<td>2:</td>
<td>$\textbf{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in D$</td>
</tr>
<tr>
<td>3:</td>
<td>$\textbf{eff: } (D, b) \leftarrow \text{DDS.announce}(c)$</td>
</tr>
<tr>
<td>4:</td>
<td>$\text{if } b = \text{false }$ then $D \leftarrow \text{CONSc-propose}(D)$</td>
</tr>
<tr>
<td>5:</td>
<td>$\text{deps}(c) \leftarrow D$</td>
</tr>
<tr>
<td>6:</td>
<td>$\text{send}(c, \text{deps}(c))$ to $\Pi \setminus {p}$</td>
</tr>
<tr>
<td>7:</td>
<td>$\textbf{when } \text{recv}(c, D)$</td>
</tr>
<tr>
<td>8:</td>
<td>$\textbf{eff: } \text{deps}(c) \leftarrow D$</td>
</tr>
<tr>
<td>9:</td>
<td>$\textbf{eff: } D$</td>
</tr>
<tr>
<td>10:</td>
<td>$\bot$</td>
</tr>
</tbody>
</table>
Abstract protocol to decide a command

\begin{algorithm}
\caption{Deciding a command $c$ – code at process $p$}
\begin{algorithmic}[1]
\State $\text{submit}(c) :=$
\State \textbf{pre}: $p = \text{coord}(c) \lor \text{coord}(c) \in D$
\State \textbf{eff}: $(D, b) \leftarrow \text{DDS.announce}(c)$
\If{$b = \text{false}$} \State $D \leftarrow \text{CONS}_c\text{-propose}(D)$ \EndIf
\State $\text{deps}(c) \leftarrow D$
\State $\text{send}(c, \text{deps}(c))$ to $\Pi \setminus \{p\}$
\State \textbf{when} $\text{recv}(c, D)$
\State \textbf{eff}: $\text{deps}(c) \leftarrow D$
\end{algorithmic}
\end{algorithm}
Abstract protocol to decide a command

\[ p_1 \xrightarrow{\text{submit}(a)} g_1 \]

\[ p_2 \xrightarrow{\text{submit}(b)} g_2 \]

\[ \bot \xrightarrow{a} (g_1) \quad \bot \xrightarrow{b} (g_2) \]

---

**Algorithm 2** Deciding a command \( c \) – code at process \( p \)

1: \( \text{submit}(c) := \)
2: \( \text{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in D \)
3: \( \text{eff: } (D, b) \leftarrow \text{DDS.announce}(c) \)
4: \( \quad \text{if } b = \text{false} \text{ then } D \leftarrow \text{CONS}_c.\text{propose}(D) \)
5: \( \quad \text{deps}(c) \leftarrow D \)
6: \( \quad \text{send}(c, \text{deps}(c)) \text{ to } \Pi \setminus \{p\} \)
7: \( \)
8: \( \text{when } \text{recv}(c, D) \)
9: \( \text{eff: } \text{deps}(c) \leftarrow D \)
10: \( \)
Abstract protocol to decide a command

\[ p_1 \xrightarrow{\text{submit}(a)} g_1 \xrightarrow{\text{DDS}.\text{announce}(a)} g_2 \xrightarrow{\text{submit}(b)} \]

\[ \bot \xrightarrow{a} (g_1) \quad \bot \xrightarrow{b} (g_2) \]

\[ p_2 \]

\textbf{Algorithm 2} Deciding a command c - code at process p

1: \text{submit}(c) :=
2: \text{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in D
3: \text{eff: } (D, b) \leftarrow \text{DDS}.\text{announce}(c)
4: \quad \text{if } b = \text{false} \text{ then } D \leftarrow \text{CONS}_{c,\text{propose}}(D)
5: \quad \text{deps}(c) \leftarrow D
6: \quad \text{send}(c, \text{deps}(c)) \text{ to } \Pi \setminus \{p\}
7: \quad \text{when } \text{recv}(c, D)
8: \quad \text{eff: } \text{deps}(c) \leftarrow D
9: 10:
Abstract protocol to decide a command

\[
p_1 \xrightarrow{\text{submit}(a)} \text{DDS.announce}(a)
\]

\[
p_2 \xrightarrow{\text{submit}(b)} \text{DDS.announce}(b)
\]

\[
\perp \rightarrow a \quad \perp \rightarrow b
\]

**Algorithm 2** Deciding a command \(c\) – code at process \(p\)

1: \(\text{submit}(c) :=\)
2: \(\textbf{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in D\)
3: \(\textbf{eff: } (D, b) \leftarrow \text{DDS.announce}(c)\)
4: \[\text{if } b = \text{false} \text{ then } D \leftarrow \text{CONS}_{c,\text{propose}}(D)\]
5: \(\text{deps}(c) \leftarrow D\)
6: \(\text{send}(c, \text{deps}(c)) \text{ to } \Pi \setminus \{p\}\)
7: 
8: \(\textbf{when } \text{recv}(c, D)\)
9: \(\textbf{eff: } \text{deps}(c) \leftarrow D\)
10:
Abstract protocol to decide a command

\[
p_1 \xrightarrow{\text{submit}(a)} g_1 \xrightarrow{\text{DDS.announce}(a)} (\{\}, \text{true})
\]

\[
p_2 \xrightarrow{\text{submit}(b)} g_2 \xrightarrow{\text{DDS.announce}(b)}
\]

\[
\perp \longrightarrow a \quad \perp \longrightarrow b
\]

\[
(g_1) \quad (g_2)
\]

**Algorithm 2** Deciding a command \( c \) – code at process \( p \)

1: \( \text{submit}(c) := \)

2: \( \text{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in D \)

3: \( \text{eff: } (D, b) \leftarrow \text{DDS.announce}(c) \)

4: \( \quad \text{if } b \text{ = false then } D \leftarrow \text{CONS}_c\text{-propose}(D) \)

5: \( \quad \text{deps}(c) \leftarrow D \)

6: \( \quad \text{send}(c, \text{deps}(c)) \text{ to } \Pi \setminus \{p\} \)

7: \( \)

8: \( \text{when } \text{recv}(c, D) \)

9: \( \text{eff: } \text{deps}(c) \leftarrow D \)

10: \]
Abstract protocol to decide a command

\[ p_1 \xrightarrow{\text{submit}(a)} g_1 \xrightarrow{\text{DDS.announce}(a)} ((\{}), \text{true}) \]

\[ p_2 \xrightarrow{\text{submit}(b)} g_2 \xrightarrow{\text{DDS.announce}(b)} (\{}), \text{false} \]

\[ \bot \xrightarrow{a} (g_1) \]

\[ \bot \xrightarrow{b} (g_2) \]

---

**Algorithm 2** Deciding a command \( c \) – code at process \( p \)

1: \textit{submit}(c) :=

2: \textbf{pre:} \( p = \text{coord}(c) \lor \text{coord}(c) \in D \)

3: \textbf{eff:} \((D, b) \leftarrow \text{DDS.announce}(c)\)

4: \text{if} \( b = \text{false} \) \text{then} \( D \leftarrow \text{CONS}_c\text{-propose}(D) \)

5: \( \text{deps}(c) \leftarrow D \)

6: \( \text{send}(c, \text{deps}(c)) \) to \( \Pi \setminus \{p\} \)

7: 

8: \textbf{when} \text{recv}(c, D) \)

9: \textbf{eff:} \( \text{deps}(c) \leftarrow D \)

10:
Abstract protocol to decide a command

\begin{align*}
    p_1 \xrightarrow{\text{submit}(a)} g_1 \quad \text{DDS.announce}(a) \quad g_3 \\
    p_2 \quad g_2 \quad \text{DDS.announce}(b) \quad (\{a\}, \text{false})
\end{align*}

**Algorithm 2** Deciding a command $c$ \textendnote{c - code at process $p$}

\begin{enumerate}
    \item $\text{submit}(c) :=$
    \item \textbf{pre:} $p = \text{coord}(c) \lor \text{coord}(c) \in D$
    \item \textbf{eff:} $(D, b) \leftarrow \text{DDS.announce}(c)$
    \item \textbf{if} $b = \text{false}$ \textbf{then} $D \leftarrow \text{CONS}_c.\text{propose}(D)$
    \item $\text{deps}(c) \leftarrow D$
    \item $\text{send}(c, \text{deps}(c))$ to $\Pi \setminus \{p\}$
    \item $\text{when} \; \text{recv}(c, D)$
    \item \textbf{eff:} $\text{deps}(c) \leftarrow D$
\end{enumerate}
Abstract protocol to decide a command

\[ p_1 \xrightarrow{\text{submit}(a)} g_1 \xrightarrow{\text{DDS} \text{.announce}(a)} g_3 \]

\[ p_2 \xrightarrow{\text{submit}(b)} g_2 \xrightarrow{\text{DDS} \text{.announce}(b)} (\{a\}, \text{false}) \xrightarrow{\text{CONS}_b(\{a\})} g_5 \]

\[ \perp \xrightarrow{a} \quad \perp \xrightarrow{b} \quad a \]

\[ (g_1) \quad (g_2) \quad (g_3) \]

Algorithm 2 Deciding a command \( c \) – code at process \( p \)

1: \( \text{submit}(c) := \)
2: \( \textbf{pre:} \quad p = \text{coord}(c) \lor \text{coord}(c) \in D \)
3: \( \textbf{eff:} \quad (D, b) \leftarrow \text{DDS} \text{.announce}(c) \)
4: \( \quad \text{if } b = \text{false } \text{then } D \leftarrow \text{CONS}_c \text{.propose}(D) \)
5: \( \quad \text{deps}(c) \leftarrow D \)
6: \( \quad \text{send}(c, \text{deps}(c)) \) to \( \Pi \setminus \{p\} \)
7: \( \)
8: \( \textbf{when } \text{recv}(c, D) \)
9: \( \quad \textbf{eff: } \quad \text{deps}(c) \leftarrow D \)
10: \( \)
Abstract protocol to decide a command

\[ \text{Algorithm 2 Deciding a command } c \text{ – code at process } p \]

1: \textit{submit}(c) :=
2: \hspace{1em} \textbf{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in D
3: \hspace{1em} \textbf{eff: } (D, b) \leftarrow \text{DDS.announce}(c)
4: \hspace{1em} \hspace{1em} \textbf{if } b = \text{false} \text{ then } D \leftarrow \text{CONS}_c\text{-propose}(D)
5: \hspace{1em} \hspace{1em} \text{deps}(c) \leftarrow D
6: \hspace{1em} \hspace{1em} \text{send}(c, \text{deps}(c)) \text{ to } \Pi \setminus \{p\}
7: \hspace{1em} \hspace{1em} \\
8: \hspace{1em} \hspace{1em} \textbf{when } \text{recv}(c, D)
9: \hspace{1em} \hspace{1em} \textbf{eff: } \text{deps}(c) \leftarrow D
10: \hspace{1em} \\

\[ g_1 \xrightarrow{\text{submit}(a)} g_2 \xrightarrow{\text{DDSannounce}(a)} g_3 \]

\[ g_2 \xrightarrow{\text{submit}(b)} g_4 \]

\[ g_3 \]

\[ g_4 \]
Abstract protocol to decide a command

\[p_1 \xrightarrow{\text{submit}(a)} g_1 \xrightarrow{\text{DDS.announce}(a)} g_3 \xrightarrow{\text{send}(a, \{\})} \]

\[p_2 \xrightarrow{\text{submit}(b)} g_2 \xrightarrow{\text{DDS.announce}(b)} g_4 \]

**Algorithm 2** Deciding a command \(c\) – code at process \(p\)

1: \( \text{submit}(c) := \)
2: \quad \textbf{pre: } \(p = \text{coord}(c) \lor \text{coord}(c) \in D\)
3: \quad \textbf{eff: } (D, b) \leftarrow \text{DDS.announce}(c)
4: \quad \quad \text{if } b = \text{false} \text{ then } D \leftarrow \text{CONS}_c\text{-propose}(D)
5: \quad \quad \text{deps}(c) \leftarrow D
6: \quad \quad \text{send}(c, \text{deps}(c)) \text{ to } \Pi \setminus \{p\}
7: \quad \textbf{when } \text{recv}(c, D)
8: \quad \quad \textbf{eff: } \text{deps}(c) \leftarrow D
9: 
10:
Abstract protocol to decide a command

\[ p_1 \xrightarrow{\text{submit}(a)} g_1 \xrightarrow{\text{DDS.announce}(a)} g_3 \xrightarrow{\text{send}(a, \{\})} g_5 \]

\[ p_2 \xrightarrow{\text{submit}(b)} g_2 \xrightarrow{\text{DDS.announce}(b, \{a\}, \text{false}) \xrightarrow{\text{CONS}_p(\{a\}) \xrightarrow{\text{send}(b, \{a\})} g_5 \]

**Algorithm 2** Deciding a command \( c \) – code at process \( p \)

1: \( \text{submit}(c) := \)

2: \( \text{pre: } p = \text{coord}(c) \lor \text{coord}(c) \in D \)

3: \( \text{eff: } (D, b) \leftarrow \text{DDS.announce}(c) \)

4: \( \text{if } b = \text{false then } D \leftarrow \text{CONS}_c.\text{propose}(D) \)

5: \( \text{deps}(c) \leftarrow D \)

6: \( \text{send}(c, \text{deps}(c)) \text{ to } I \setminus \{p\} \)

7:

8: \( \text{when } \text{recv}(c, D) \)

9: \( \text{eff: } \text{deps}(c) \leftarrow D \)

10:
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

(Reiability) 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 $\text{announce}(c)$ returns after two message delays in the absence of conflicting commands.
Core Properties

**Optimal Latency** During a nice run, every call to \textit{announce(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 $c$. 

---

DISC'20. Leaderless SMR. Tuanir F. Rezende. 19
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.
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

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

Quorums in use

Run $\lambda_1$
ROLL theorem - Proof sketch

By contradiction, using a round-based reasoning. Let us assume a protocol $P$ that satisfies all the ROLL proper¬es with $2F + f - 1 > n$.

Quorums in use

Run $\lambda_1$

$\text{deps}(a) = \{\}$

Run $\lambda_2$

$\text{deps}(b) = \{\}$
ROLL theorem - Proof sketch

Run $\lambda_1$

$\text{deps}(a) = \{\}$

Run $\lambda_3$
ROLL theorem - Proof sketch

\[ \text{Run } \lambda_1 \]

\[ \text{Run } \lambda_3 \]

\[ \text{deps}(a) = \{ \} \]
ROLL theorem - Proof sketch

\[ \text{Run } \lambda_1 \]

\[ p_1 \rightarrow P_1 \rightarrow Q^* \rightarrow P_2 \rightarrow p_2 \]

\[ \text{deps}(a) = \{\} \]

\[ \text{Run } \lambda_3 \]

\[ p_1 \rightarrow P_1 \rightarrow Q^* \rightarrow P_2 \rightarrow p_2 \]

\[ b \]
ROLL theorem - Proof sketch

Run $\lambda_1$

Run $\lambda_3$
ROLL theorem - Proof sketch

Run $\lambda_1$

$$\text{deps}(a) = \{\}$$

Run $\lambda_3$

$$\text{deps}(a) \neq \bot$$

$$\text{deps}(b) \neq \bot$$
ROLL theorem - Proof sketch

Run $\lambda_1$

$\text{deps}(a) = \{\}$

Run $\lambda_3$

$\text{deps}(a) \neq \bot$

$\text{deps}(b) \neq \bot$

Run $\lambda_4$
ROLL theorem - Proof sketch

Run $\lambda_1$

Run $\lambda_3$

Run $\lambda_4$

$\text{deps}(a) = \{\}$

$\text{deps}(a) \neq \bot$

$\text{deps}(b) \neq \bot$
ROLL theorem - Proof sketch

Run $\lambda_1$

$\text{deps}(a) = \{\}$

Run $\lambda_3$

$\text{deps}(a) \neq \perp$

$\text{deps}(b) \neq \perp$

Run $\lambda_4$
ROLL theorem - Proof sketch

Run $\lambda_1$

Run $\lambda_3$

Run $\lambda_4$
ROLL theorem - Proof sketch

Run $\lambda_1$

$\text{deps}(a) = \{\}$

Run $\lambda_3$

$\text{deps}(a) \neq \bot$

$\text{deps}(b) \neq \bot$

Run $\lambda_4$

$\text{deps}(a) = \{\}$
ROLL theorem - Proof sketch

Run $\lambda_1$

$\text{deps}(a) = \{\}$

Run $\lambda_3$

$\text{deps}(a) = \{\}$

$\text{deps}(b) \neq \perp$

Run $\lambda_4$

$\text{deps}(a) = \{\}$

DISC'20. Leaderless SMR. Tuanir F. Rezende.
ROLL theorem - Proof sketch

Run $\lambda_2$

Run $\lambda_3$
ROLL theorem - Proof sketch

Run $\lambda_2$

Run $\lambda_3$
ROLL theorem - Proof sketch

Run $\lambda_2$

Run $\lambda_3$
ROLL theorem - Proof sketch

Run $\lambda_2$  Run $\lambda_3$
ROLL theorem - Proof sketch

Run $\lambda_2$

Run $\lambda_3$
ROLL theorem - Proof sketch

Run $\lambda_2$

Run $\lambda_3$

Run $\lambda_5$
ROLL theorem - Proof sketch

Run $\lambda_2$

Run $\lambda_3$

Run $\lambda_5$
ROLL theorem - Proof sketch

Run $\lambda_2$

$\text{deps}(b) = \emptyset$

Run $\lambda_3$

$\text{deps}(a) \neq \perp$
$\text{deps}(b) \neq \perp$

Run $\lambda_5$

DISC'20. Leaderless SMR. Tuanir F. Rezende.
ROLL theorem - Proof sketch

Run $\lambda_2$

Run $\lambda_3$

Run $\lambda_5$
ROLL theorem - Proof sketch

Run $\lambda_2$

Run $\lambda_3$

Run $\lambda_5$
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: $(F, f) = (2, 2)$.
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.