Debugging Distributed Systems with Why-Across-Time Provenance

Michael Whittaker, Cristina Teodoropol, Peter Alvaro, Joseph M. Hellerstein
Reasoning about the causes of events in a distributed system is hard
Causality

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport, Massachusetts Institute of Technology

The concept of order is fundamental in our way of thinking. It is derived from the natural concept of the order in which events occur. We say that something happened first if it occurred before something else. The concept of the temporal ordering of events provides our thinking about systems. For example, in an office automation system, we specify that a request for a reservation should be granted if it is made before the flight is filled. However, we will see that this concept must be carefully reexamined when considering ordering in a distributed system.

A distributed system consists of a collection of spatially separated processes which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPANET, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit and the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will consider systems with processes of spatially separated computers. However, many of our arguments will apply more generally. In particular, a multiprocessor system on a single computer is another example of a distributed system because the unpredictable order in which events can occur in a distributed system is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that this problem often arises because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm also provides a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, asynchronous behavior can occur if the ordering obtained by this algorithm differs from that predicted by the user. This can be fixed by choosing real physical clocks. We describe a simple method for synchronizing real clocks, and derive an upper bound on how far our synchronizer can drift.

The Partial Ordering

Most people would probably say that an event happened before another event if it happened at an earlier time than the other. They might predict the assertions in terms of physical moments of time. However, if a system is to maintain a specification correctly, then that specification must be given in terms of events observable within the system. If a specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that each clock is not perfectly accurate and does not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

We begin by defining a system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of actions. Depending upon the application, the simulation of a subset of a computer could be one event, or the execution of a single instruction could be one event.
Causality is general-purpose but too coarse-grained
Why-Provenance
## Why-Provenance

### Database Instance $I$

<table>
<thead>
<tr>
<th></th>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>10</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>$r_2$</td>
<td>10</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>$r_3$</td>
<td>600</td>
<td>700</td>
<td></td>
</tr>
<tr>
<td>$s_1$</td>
<td>20</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td>40</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td>700</td>
<td>800</td>
<td></td>
</tr>
</tbody>
</table>
Why-Provenance

**Database Instance I**

**SELECT x**
FROM R, S
WHERE R.y = S.y

**SQL Query Q**
Why-Provenance

```
SELECT x
FROM R, S
WHERE R.y = S.y
```

**SQL Query Q**

**Output Q(1)**

**Database Instance I**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>10</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>10</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td>600</td>
<td>700</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>20</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>40</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>700</td>
<td>800</td>
<td></td>
</tr>
</tbody>
</table>
Why-Provenance

Database Instance $I$

### Table R

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>r2</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>r3</td>
<td>600</td>
<td>700</td>
</tr>
</tbody>
</table>

### Table S

<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>s2</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>s3</td>
<td>700</td>
<td>800</td>
</tr>
</tbody>
</table>

**SQL Query $Q$**

```
SELECT x 
FROM R, S 
WHERE R.y = S.y
```

**Output Tuple $t$**

```
10
600
```
A subinstance \( l' \) of \( l \) is a witness of \( t \) if \( t \) is in \( Q(l') \)
Why-Provenance

**Database Subinstance I’**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>r2</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>r3</td>
<td>600</td>
<td>700</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>s2</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>s3</td>
<td>700</td>
<td>800</td>
</tr>
</tbody>
</table>

**SQL Query Q**

```sql
SELECT x
FROM R, S
WHERE R.y = S.y
```
**Why-Provenance**

**Database Subinstance \( I' \)**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>r2</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>r3</td>
<td>600</td>
<td>700</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>s2</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>s3</td>
<td>700</td>
<td>800</td>
</tr>
</tbody>
</table>

**SQL Query \( Q \)**

```sql
SELECT x
FROM R, S
WHERE R.y = S.y
```

**Output Tuple \( t \)**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

**Output \( Q(I') \)**
A witness \( l' \) of \( t \) is a \textbf{minimal witness} of \( t \) if no proper subinstance of \( l' \) is also a witness of \( t \)
**Why-Provenance**

Database Subinstance $I'$

SELECT $x$
FROM $R$, $S$
WHERE $R.y = S.y$

**SQL Query $Q$**

Output Tuple $t$

Output $Q(I')$
Why-Provenance

Database Subinstance $I'$

\[
\begin{array}{c|c|c}
\text{R} & \text{x} & \text{y} \\
\hline
r1 & 10 & 20 \\
r2 & 10 & 40 \\
r3 & 600 & 700 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{S} & \text{y} & \text{z} \\
\hline
s1 & 20 & 30 \\
s2 & 40 & 50 \\
s3 & 700 & 800 \\
\end{array}
\]

SQL Query $Q$

\[
\text{SELECT x}
\text{FROM R, S}
\text{WHERE R.y = S.y}
\]

Output Tuple $t$

Output $Q(I')$
Why-Provenance

Database Subinstance $I'$

**SELECT** $x$
**FROM** $R, S$
**WHERE** $R.y = S.y$

Output Tuple $t$

Output $Q(I')$
The **why-provenance** of \( t \) is the set of minimal witnesses of \( t \)
Why-provenance is fine-grained but not generally applicable
Causality is general-purpose but too coarse-grained.

Why-provenance is fine-grained but not generally applicable.
Wat-provenance is general-purpose and fine-grained
Wat-provenance generalizes why-provenance from static relational databases to arbitrary state machines
Wat-provenance is to state machines what why-provenance is to relational databases
Wat-Provenance
Wat-Provenance

<table>
<thead>
<tr>
<th></th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>700</td>
<td>800</td>
</tr>
</tbody>
</table>

Database Instance I
SELECT x
FROM R, S
WHERE R.y = S.y

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>700</td>
<td>800</td>
</tr>
</tbody>
</table>
Wat-Provenance

Database Instance I

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>700</td>
<td>800</td>
</tr>
</tbody>
</table>

Query Q

```
SELECT x
FROM R, S
WHERE R.y = S.y
```

Output Tuple t

```
<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>700</td>
</tr>
</tbody>
</table>
```
Wat-Provenance

**Database Instance I**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>700</td>
<td>800</td>
</tr>
</tbody>
</table>

**Query Q**

```sql
SELECT x
FROM R, S
WHERE R.y = S.y
```

**Output Tuple t**

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>700</td>
</tr>
</tbody>
</table>

**State Machine M**
Wat-Provenance

Database Instance I

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>700</td>
<td>800</td>
</tr>
</tbody>
</table>

Query Q

```
SELECT x
FROM R, S
WHERE R.y = S.y
```

Output Tuple t

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>700</td>
</tr>
</tbody>
</table>

State Machine M

Input Trace T

```
[abc]d
```
Wat-Provenance

Database Instance I

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>700</td>
<td>800</td>
</tr>
</tbody>
</table>

Query Q

```
SELECT x
FROM R, S
WHERE R.y = S.y
```

Output Tuple t

![](image)

State Machine M

Input Trace T

```
get(x); 1
```

Input i, Output o
Example 1: Key-Value Store
Example 1: Key-Value Store

Trace T

| set(x, 1) | set(y, 2) |
Example 1: Key-Value Store

Trace T

\[
\begin{array}{|c|c|}
\hline
\text{set(x,1)} & \text{set(y,2)} \\
\hline
\end{array}
\]

Input i

get(x)
Example 1: Key-Value Store

Trace T

set(x,1) set(y,2)

Input i get(x)

Output o 1
Example 1: Key-Value Store

Trace $T$: \text{set}(x, 1) \quad \text{set}(y, 2)

Input $i$: \text{get}(x)

Output $o$: 1
Lessons

1. The “cause” of an output $o$ should be a subtrace of the input that suffices to generate $o$. We call such a subtrace a witness of $o$. 
Example 1: Key-Value Store

Trace T

set(x,1) set(y,2)

Input i get(x)

Output o 1
Example 1: Key-Value Store

Trace T: set(x,1) set(y,2)

Input i: get(x)

Output o: 1
1. The “cause” of an output $o$ is a subtrace of the input that suffices to generate $o$. We call such a subtrace a witness of $o$.

2. The cause of an output $o$ should be a “minimal” witness of $o$. 
Example 1: Key-Value Store

Trace $T$

Input $i$

get($x$)

Output $o$

1
Example 1: Key-Value Store

Trace $T$

Input $i$

Output $o$

set($x, 1$) set($y, 2$)

get($x$)

1
Example 2: Boolean Formulas
Example 2: Boolean Formulas

Trace $T$:

| set(a) | set(b) | set(c) | set(d) |
Example 2: Boolean Formulas

Trace $T$:

| set(a) | set(b) | set(c) | set(d) |

Input $i$:

```
eval((a and d) or (b and c))
```
### Example 2: Boolean Formulas

<table>
<thead>
<tr>
<th>Trace T</th>
</tr>
</thead>
<tbody>
<tr>
<td>set(a)</td>
</tr>
<tr>
<td>set(b)</td>
</tr>
<tr>
<td>set(c)</td>
</tr>
<tr>
<td>set(d)</td>
</tr>
</tbody>
</table>

**Input i**

\[
eval((a \text{ and } d) \text{ or } (b \text{ and } c))
\]

**Output o**

true
Example 2: Boolean Formulas

Trace $T$:

- set(a)
- set(b)
- set(c)
- set(d)

Input $i$:

$$\text{eval}((a \text{ and } d) \text{ or } (b \text{ and } c))$$

Output $o$:

true
Example 2: Boolean Formulas

Trace T

Input i

eval((a and d) or (b and c))

Output o

true
Lessons

1. The “cause” of an output $o$ is a subtrace of the input that suffices to generate $o$. We call such a subtrace a witness of $o$.

2. The cause of an output $o$ should be a “minimal” witness of $o$.

3. An output $o$ could have multiple “minimal” witnesses.
Example 3: Negation

Trace T

| set(a) | set(b) | set(c) |

Input i
eval((a and not b) or c)
Example 3: Negation

Trace T

| set(a) | set(b) | set(c) |

Input i

eval((a and not b) or c)
Example 3: Negation

<table>
<thead>
<tr>
<th>Trace T</th>
<th>set(a)</th>
<th>set(b)</th>
<th>set(c)</th>
</tr>
</thead>
</table>

Input i  eval((a and not b) or c)
Example 3: Negation

Trace T

| set(a) | set(b) | set(c) |

Input i

\[ \text{eval}((a \text{ and not } b) \text{ or } c) \]
Example 3: Negation

Trace T

\[
\begin{array}{c|c|c}
\text{set}(a) & \text{set}(b) & \text{set}(c) \\
\end{array}
\]

Input \( i \) \quad \text{eval}( (a \text{ and } \text{not } b) \text{ or } c )
Example 3: Negation

Trace $T$:  

| set(a) | set(b) | set(c) |

Input $i$:  

eval($(a \text{ and} \neg b) \text{ or } c$)

Output $o$:  

true
Example 3: Negation

Trace T

set(a)  set(b)  set(c)

Input i

eval((a and not b) or c)

Output o
	true
Example 3: Negation

Trace $T$ \hspace{1cm} \text{set(a)} \hspace{1cm} \text{set(b)} \hspace{1cm} \text{set(c)}

Input $i$ \hspace{1cm} \text{eval}((a \text{ and not } b) \text{ or } c)

Output $o$ \hspace{1cm} \text{true}
Example 3: Negation

Trace $T$  

Input $i$  
$\text{eval((}a\text{ and not }b\text{) or }c)$

Output $o$  
true
Example 3: Negation

Trace T

set(a)  set(b)  set(c)

Input i

eval((a and not b) or c)

Output o

true
Example 3: Negation

Trace T: set(a) set(b) set(c)

Input i: eval((a and not b) or c)

Output o: true
Example 3: Negation

Trace $T$

Input $i$

Output $o$

eval\(((a \text{ and not } b) \text{ or } c)\)
Example 3: Negation

Trace T

| set(a) | set(b) | set(c) |

Input i
eval((a and not b) or c)

Output o
true
Lessons

1. The “cause” of an output $o$ is a subtrace of the input that suffices to generate $o$. We call such a subtrace a witness of $o$.

2. The cause of an output $o$ should be a “minimal” witness of $o$.

3. An output $o$ could have multiple “minimal” witnesses.

4. If a witness is a “cause” of an output $o$, then all supertraces of the witness should be too.
Example 3: Negation

Trace $T$ 

| set(a) | set(b) | set(c) |

Input $i$ 

$\text{eval}((a \text{ and not } b) \text{ or } c)$

Output $o$ 

true
Example 3: Negation

Trace T: set(a) set(b) set(c)

Input i: eval((a and not b) or c)

Output o: true
Wat-Provenance
Wat-Provenance

• Given state machine $M$, trace $T$, input $i$, and output $o$. 
Wat-Provenance

- Given state machine $M$, trace $T$, input $i$, and output $o$.
- A **witness** of $o$ is a subtrace of $T$ that suffices to produce $o$. 
Wat-Provenance

- Given state machine $M$, trace $T$, input $i$, and output $o$.
- A witness of $o$ is a subtrace of $T$ that suffices to produce $o$.
- A witness $T'$ of $o$ is closed under supertrace in $T$ if every supertrace of $T'$ in $T$ is also a witness of $o$. 
Wat-Provenance

- Given state machine $M$, trace $T$, input $i$, and output $o$.
- A witness of $o$ is a subtrace of $T$ that suffices to produce $o$.
- A witness $T'$ of $o$ is closed under supertrace in $T$ if every supertrace of $T'$ in $T$ is also a witness of $o$.
- The wat-provenance of $o$ is the set of minimal witnesses of $o$ that are closed under supertrace in $T$. 
Causality is **general-purpose** but too **coarse-grained**

Why-provenance is **fine-grained** but not generally applicable

Wat-provenance is **general-purpose** and **fine-grained**
Computing
wat-provenance
Causality is **general-purpose** but too **coarse-grained**

Why-provenance is **fine-grained** but not generally applicable

Wat-provenance is **general-purpose** and **fine-grained**
Wat-provenance is general-purpose but too coarse-grained and easy to compute

Causality is general-purpose but too coarse-grained and easy to compute

Why-provenance is fine-grained but not generally applicable and easy to compute

Wat-provenance is general-purpose and fine-grained but hard to compute
Computing Causal History
Computing Causal History

Use vector clocks

Time

A
A:0

B
B:0

C
C:0

<table>
<thead>
<tr>
<th>Cause</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>B:1</td>
<td>A:3</td>
</tr>
<tr>
<td>B:2</td>
<td>A:4</td>
</tr>
<tr>
<td>C:1</td>
<td>A:4</td>
</tr>
<tr>
<td>B:2</td>
<td>A:4</td>
</tr>
<tr>
<td>C:1</td>
<td>A:4</td>
</tr>
<tr>
<td>B:3</td>
<td>A:4</td>
</tr>
<tr>
<td>B:4</td>
<td>A:5</td>
</tr>
<tr>
<td>C:1</td>
<td>A:5</td>
</tr>
<tr>
<td>B:5</td>
<td>A:5</td>
</tr>
<tr>
<td>C:1</td>
<td>A:5</td>
</tr>
<tr>
<td>B:5</td>
<td>A:5</td>
</tr>
<tr>
<td>C:4</td>
<td>A:5</td>
</tr>
<tr>
<td>B:5</td>
<td>A:5</td>
</tr>
<tr>
<td>C:5</td>
<td>A:5</td>
</tr>
</tbody>
</table>
Computing Why-Provenance
Computing Why-Provenance

Compute it explicitly

\[
\begin{align*}
\text{Why}([t], I, \{u\}) &= \begin{cases} 
\{\emptyset\}, & \text{if } (t = u), \\
\emptyset, & \text{otherwise}.
\end{cases} \\
\text{Why}(R, I, t) &= \begin{cases} 
\{\{(R, t)\}\}, & \text{if } (t \in R(I)), \\
\emptyset, & \text{otherwise}.
\end{cases} \\
\text{Why}(\sigma_{\theta}(Q), I, t) &= \begin{cases} 
\text{Why}(Q, I, t), & \text{if } \theta(t), \\
\emptyset, & \text{otherwise}.
\end{cases} \\
\text{Why}(\pi_{U}(Q), I, t) &= \bigcup\{\text{Why}(Q, I, u) \mid u \in Q(I), t = u[U]\} \\
\text{Why}(\rho_{A \rightarrow B}(Q), I, t) &= \text{Why}(Q, I, t[B \rightarrow A]) \\
\text{Why}(Q_1 \land Q_2, I, t) &= \text{Why}(Q_1, I, t[U_1]) \cup \text{Why}(Q_2, I, t[U_2]) \\
\text{Why}(Q_1 \lor Q_2, I, t) &= \text{Why}(Q_1, I, t) \cup \text{Why}(Q_2, I, t)
\end{align*}
\]
Computing Wat-Provenance

State Machine Implementation
Computing Wat-Provenance

State Machine Implementation → Infer → Wat-Provenance
Computing Wat-Provenance

State Machine Implementation → Infer Code Analysis → Wat-Provenance
Computing Wat-Provenance

State Machine Implementation → Infer Code Analysis → Wat-Provenance

State Machine Interface
Computing Wat-Provenance

State Machine Implementation $\xrightarrow{\text{Infer Code Analysis}}$ Wat-Provenance

State Machine Interface $\xrightarrow{\text{Specify}}$ Wat-Provenance
Computing Wat-Provenance

State Machine Implementation → Infer Code Analysis → Wat-Provenance

State Machine Interface → Specify Wat-Provenance Spec → Wat-Provenance
Example: Key-Value Store

Trace T

| 1 | set(a,3) | set(e,1) | set(f,4) |

Input $i = \text{get(b)}$

Output $o = 1$
Example: Key-Value Store

Trace T

Input $i = \text{get}(b)$

Output $o = 1$
Example: Key-Value Store

Trace T

<table>
<thead>
<tr>
<th></th>
<th>set(a,1)</th>
<th>set(c,2)</th>
<th>set(b,1)</th>
<th>set(a,3)</th>
<th>set(d,1)</th>
<th>set(e,2)</th>
</tr>
</thead>
</table>

Input $i = \text{get}(b)$

Output $o = 1$

English wat-provenance specification:
The wat-provenance of a get of key $k$ is the most recent set to $k$. 
Example: Key-Value Store

English wat-provenance specification:
The wat-provenance of a get of key $k$ is the most recent set to $k$.

Python wat-provenance specification:

```python
def get_prov(T: List[Request], i: GetRequest):
    for a in reversed(T):
        if (isinstance(a, SetRequest) and a.key == i.key):
            return {[a]}
    return {}"
Wat-Provenance Specifications

Simple wat-provenance specifications are not uncommon:

- Key-Value Stores
- Object Stores
- Distributed File Systems
- Coordination Services
- Load Balancers
- Stateless Services
Come to my poster!

✅ What is wat-provenance?
✅ How do you compute wat-provenance?
❌ How do you use wat-provenance?
❌ What are the limitations of wat-provenance?
Thank you!