Specifying and Verifying RDMA Synchronisation

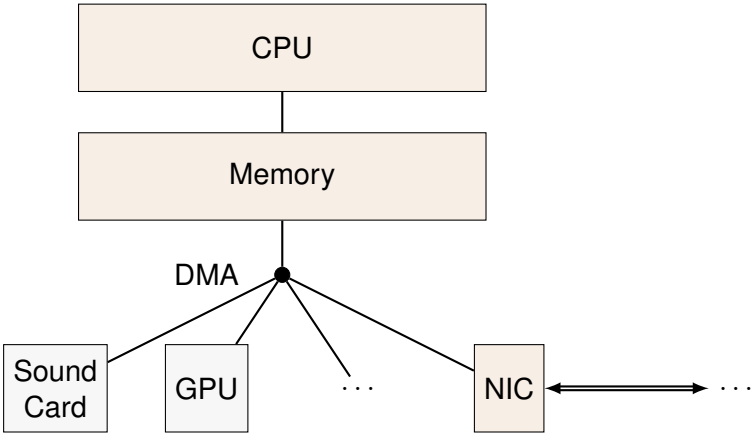
ESOP 2026

Guillaume Ambal[1], Max Stupple[1], Brijesh Dongol[2], Azalea Raad[1]

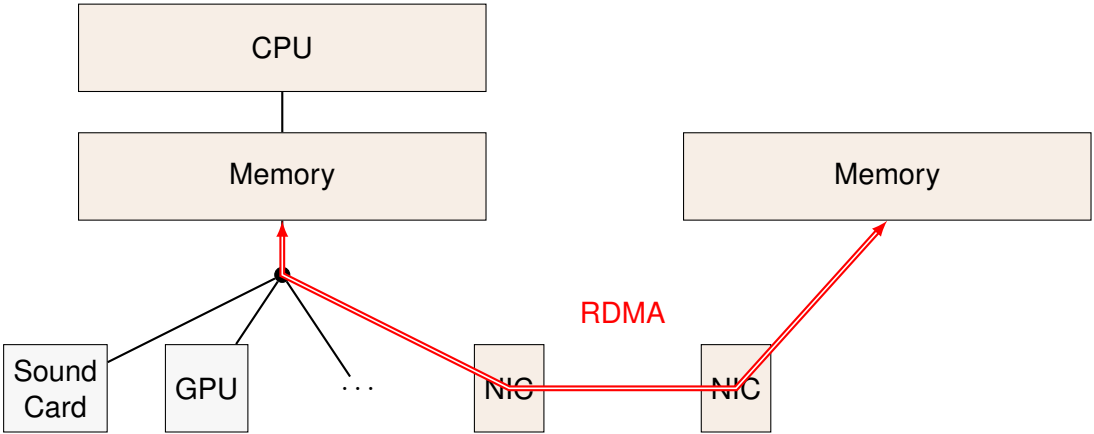
[1]Imperial College London, [2]University of Surrey

April 14, 2026

Direct Memory Access (DMA)



Remote Direct Memory Access (RDMA)



RDMA: Data-Transfer Protocol for High-Performance Computing
Low Latency: $\sim \mu$s

RDMA Example

Semantics of basic operations in OOPSLA '24

Two nodes, one thread

$x = 0$	$z = 0$
$\widetilde{z} := x$ $x := 1$	

$z = 0$ ✓ $z = 1$**?**

Can the output be $z = 1$?

RDMA Example

Semantics of basic operations in OOPSLA '24

Two nodes, one thread

$x = 0$	$z = 0$
$\widetilde{z} := x$ $x := 1$	

$z = 0$ ✓ $z = 1$ ✓

Can the output be $z = 1$?

Yes! like this:

- CPU offloads "$\widetilde{z} := x$" to the NIC
- CPU executes "$x := 1$"
- NIC picks up "$\widetilde{z} := x$" and reads 1
- ...

RDMA Example

Semantics of basic operations in OOPSLA '24

Two nodes, one thread

$x = 0$	$z = 0$
$\widetilde{z} :=^{d} x$ $\texttt{wait}(d)$ $x := 1$	

$z = 0$ ✓ $z = 1$ ✗

Can the output be $z = 1$?

To prevent it: Wait*

(*not an RDMA primitive, but easy to define)

Previous and New Results

Previous works on RDMA **basic** operations (RDMA Write and RDMA Read):

- Basic **semantics**
- RDMA **libraries** for shared memory (LOCO)
- Framework for **specifying** and **proving** implementations (Mowgli, POPL '26)

Previous and New Results

Previous works on RDMA **basic** operations (RDMA Write and RDMA Read):

- Basic **semantics**
- RDMA **libraries** for shared memory (LOCO)
- Framework for **specifying** and **proving** implementations (Mowgli, POPL '26)

New:

- Formalisation of **RDMA Atomics***
- Different formalisations of **RDMA lock** libraries
- **Verified implementations** of different RDMA locks

*official poorly-chosen name

(Usual) CPU Locks Semantics

Locks, aka mutex (**mut**ual **ex**clusion): synchronisation tool for concurrent programs.
Each lock can only be held by a single thread at a time.

Allows easy implementation of **critical sections** or **transactions**.
Can protect **concurrent objects**.

$x, y = 0, 0$	
`Acquire`(l)	`Acquire`(l)
$x := 1$	$a := x$
$y := 1$	$b := y$
`Release`(l)	`Release`(l)

$a \neq b$ ✗

(Usual) CPU RMW

Read-Modify-Write (RMW) operations, aka atomics. E.g.:

- compare-and-swap (CAS)
- fetch-and-add (FAA)

```
def FAA(x,v) =
{
  a := x
  x := a+v
  return a
}
```

(Usual) CPU RMW

Read-Modify-Write (RMW) operations, aka atomics. E.g.:

- compare-and-swap (CAS)
- fetch-and-add (FAA)

```
def FAA(x,v) =
{
  a := x
  x := a+v
  return a
}
```

Atomic!

$x = 0$	
$a := \texttt{FAA}(x, 2)$	$x := 1$

$x = 1$ ✓ $x = 2$ ✗ $x = 3$ ✓

(Usual) CPU Ticket Lock

Ticket lock example. Each lock `l` has:

- a ticket dispenser l_t
- and a serving display l_s

Each thread can hold a local ticket.

```
def acquire(l) = {
  ticket := FAA(l_t, 1)
  while true {
    if (l_s == ticket) {break}
  }}

def release(l) = {
  l_s := ticket+1
}
```

RDMA Atomics

RDMA offers compare-and-swap (RCAS) and fetch-and-add (RFAA) operations!

Semantics not yet properly formalised, and... it's not atomic...

	$x = 0$
$\texttt{RFAA}(\widetilde{x}, 2)$	$x := 1$

$x = 2$ ✓

		$x = 0$
$\texttt{RFAA}(\widetilde{x}, 2)$	$\widetilde{x} := 1$	

$x = 2$ ✓

		$x = 0$
$\texttt{RFAA}(\widetilde{x}, 2)$	$\texttt{RFAA}(\widetilde{x}, 1)$	

$x = 2$ ✗

RDMA Atomics are only atomic **with respect to each other**.

RDMA Atomics

RDMA offers compare-and-swap (RCAS) and fetch-and-add (RFAA) operations!

Semantics not yet properly formalised, and... it's not atomic...

	$x = 0$
$\texttt{RFAA}(\widetilde{x}, 2)$	$x := 1$

$x = 2$ ✓

		$x = 0$
$\texttt{RFAA}(\widetilde{x}, 2)$	$\widetilde{x} := 1$	

$x = 2$ ✓

		$x = 0$
$\texttt{RFAA}(\widetilde{x}, 2)$	$\texttt{RFAA}(\widetilde{x}, 1)$	

$x = 2$ ✗

RDMA Atomics are only atomic **with respect to each other**.

In collaboration with NVidia, we extended the previous semantics:

- **operational** semantics (abstract machine state + reduction rules);
- **declarative** semantics (graph of events + dependencies to be respected);
- proved **equivalence** between the two.

RDMA Ticket Lock

```
def acquire(l) = {
  ticket :=^d RFAA(l_t, 1)  # RDMA Atomic
  while true {
    temp :=^d l_s     # RDMA Read
    wait(d)
    if (temp == ticket) {break}
  }}

def release(l) = {
  temp := ticket+1
  l_s := temp     # RDMA Write
}
```

Let's adapt previous code to RDMA!

- Lock accessed via RDMA
- We wait when needed

Seems reasonable...
But does it work?

RDMA Lock Example

	$x, y = 0, 0$	
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := \widetilde{x}$ $b := \widetilde{y}$ `Release`(l)

$a \neq b$?

ticket : ?

temp : ?

l_t : 0

l_s : 0

x,y : 0,0

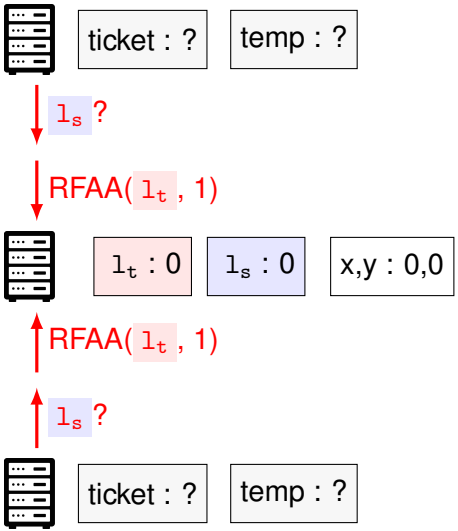
ticket : ?

temp : ?

RDMA Lock Example

	$x, y = 0, 0$	
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := \widetilde{x}$ $b := \widetilde{y}$ `Release`(l)

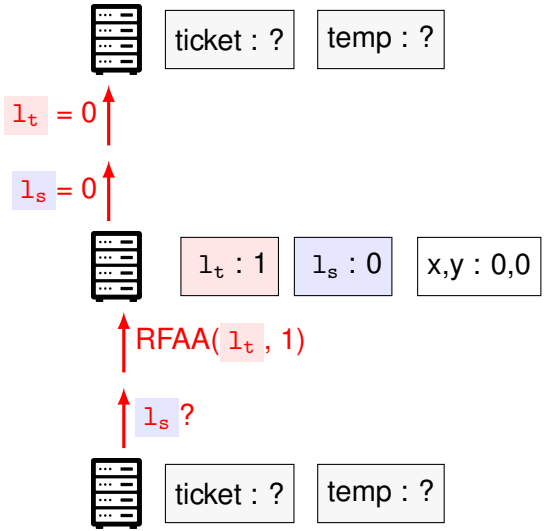
$a \neq b$?



RDMA Lock Example

	$x, y = 0, 0$	
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := \widetilde{x}$ $b := \widetilde{y}$ `Release`(l)

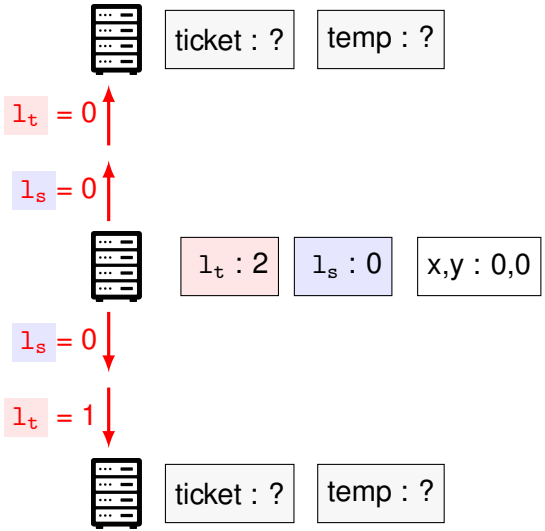
$a \neq b$?



RDMA Lock Example

	$x, y = 0, 0$	
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := \widetilde{x}$ $b := \widetilde{y}$ `Release`(l)

$a \neq b$?



RDMA Lock Example

	$x, y = 0, 0$	
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := \widetilde{x}$ $b := \widetilde{y}$ `Release`(l)

$a \neq b$?

ticket : 0

temp : 0

l_t : 2

l_s : 0

x,y : 0,0

ticket : 1

temp : 0

RDMA Lock Example

	$x, y = 0, 0$	
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := \widetilde{x}$ $b := \widetilde{y}$ `Release`(l)

$a \neq b$?

ticket : 0

temp : 0

x,y := 1,1

l_t : 2

l_s : 0

x,y : 0,0

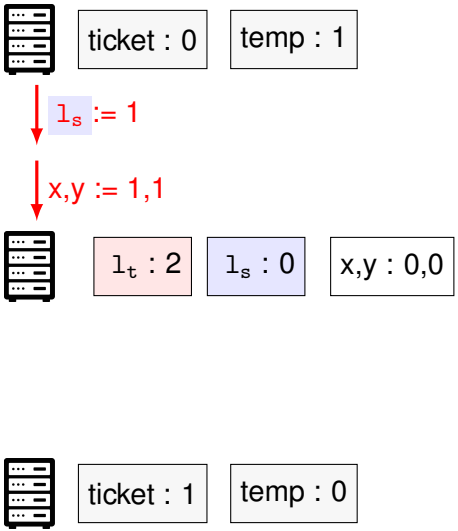
ticket : 1

temp : 0

RDMA Lock Example

	$x, y = 0, 0$	
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := \widetilde{x}$ $b := \widetilde{y}$ `Release`(l)

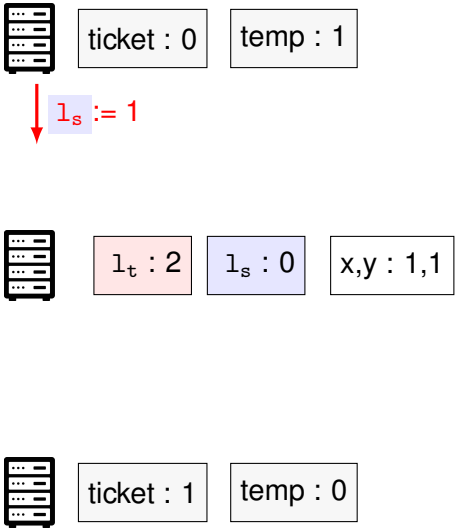
$a \neq b$?



RDMA Lock Example

	$x, y = 0, 0$	
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := \widetilde{x}$ $b := \widetilde{y}$ `Release`(l)

$a \neq b$?



RDMA Lock Example

	$x, y = 0, 0$	
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := \widetilde{x}$ $b := \widetilde{y}$ `Release`(l)

$a \neq b$?

ticket : 0

temp : 1

$\mathtt{l_t}$: 2

$\mathtt{l_s}$: 1

x,y : 1,1

ticket : 1

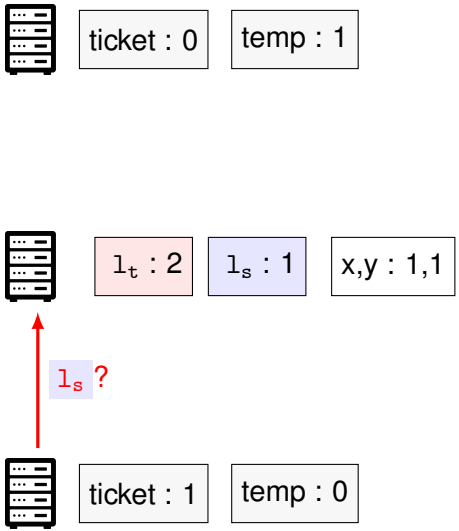
temp : 0

RDMA Lock Example

	$x, y = 0, 0$	
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := \widetilde{x}$ $b := \widetilde{y}$ `Release`(l)

$a \neq b$ ✗

Seems to work!

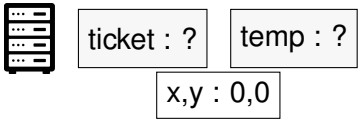


RDMA Lock Example v2

Let's double check with another case.

		$x, y = 0, 0$
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := x$ $b := y$ `Release`(l)

$a \neq b$?

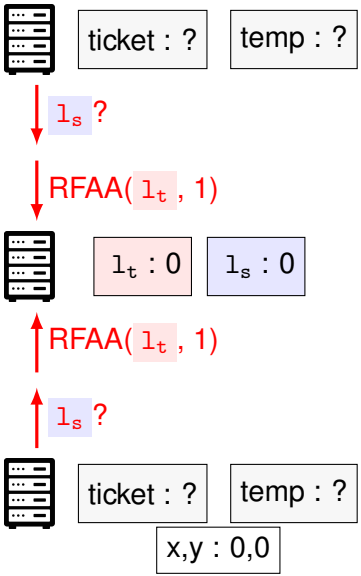


RDMA Lock Example v2

Let's double check with another case.

		$x, y = 0, 0$
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := x$ $b := y$ `Release`(l)

$a \neq b$?

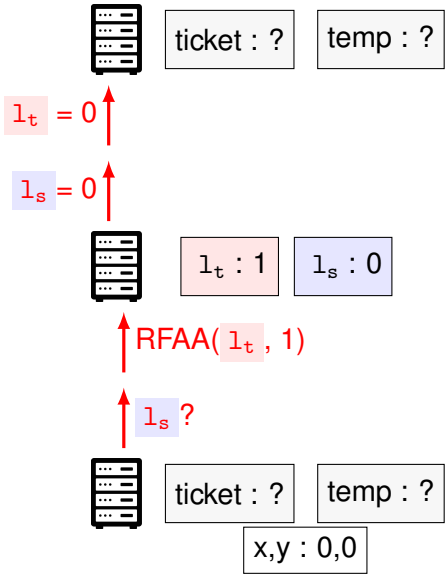


RDMA Lock Example v2

Let's double check with another case.

		$x, y = 0, 0$
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := x$ $b := y$ `Release`(l)

$a \neq b$?

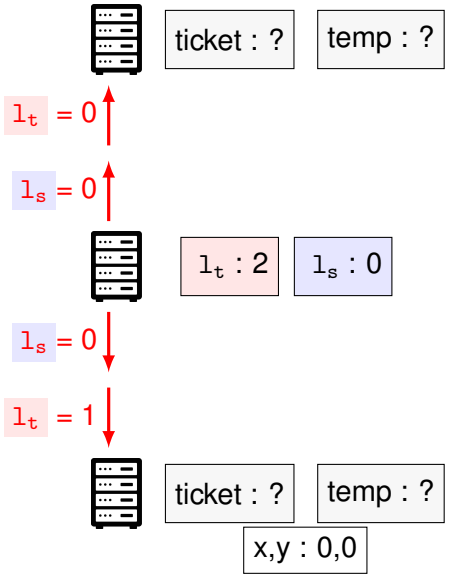


RDMA Lock Example v2

Let's double check with another case.

		$x, y = 0, 0$
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := x$ $b := y$ `Release`(l)

$a \neq b$?



RDMA Lock Example v2

Let's double check with another case.

		$x, y = 0, 0$
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := x$ $b := y$ `Release`(l)

$a \neq b$?

ticket : 0

temp : 0

$l_t : 2$

$l_s : 0$

ticket : 1

temp : 0

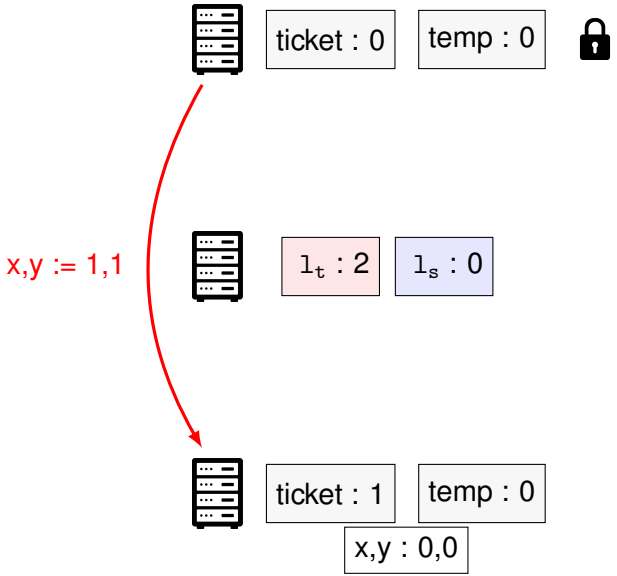
x,y : 0,0

RDMA Lock Example v2

Let's double check with another case.

		$x, y = 0, 0$
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := x$ $b := y$ `Release`(l)

$a \neq b$?

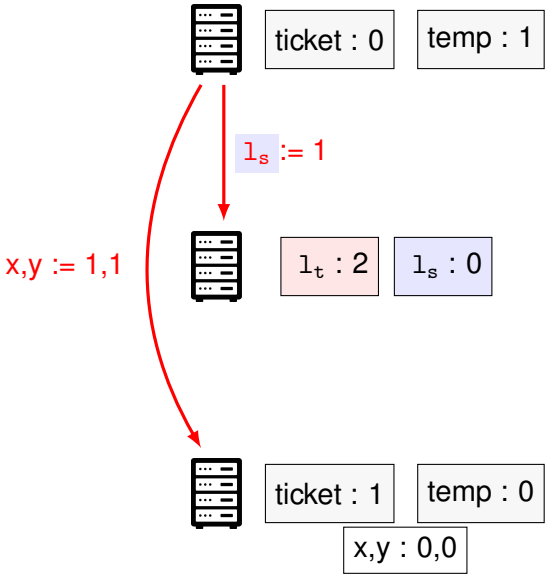


RDMA Lock Example v2

Let's double check with another case.

		$x, y = 0, 0$
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := x$ $b := y$ `Release`(l)

$a \neq b$?

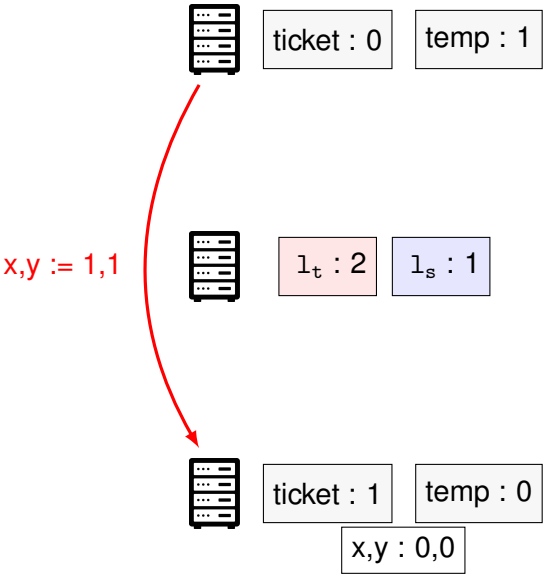


RDMA Lock Example v2

Let's double check with another case.

		$x, y = 0, 0$
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := x$ $b := y$ `Release`(l)

$a \neq b$?

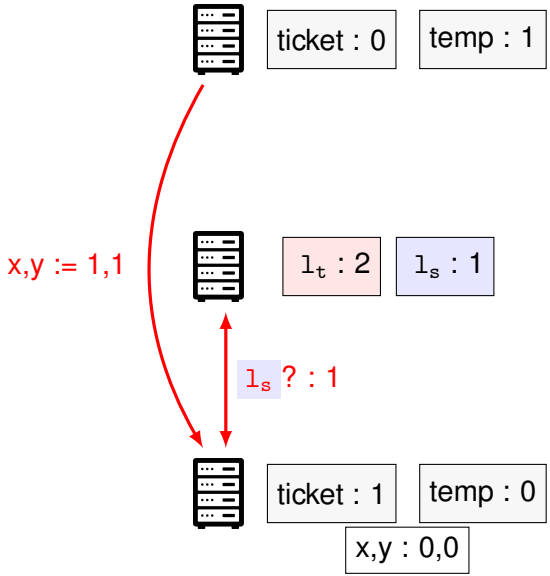


RDMA Lock Example v2

Let's double check with another case.

		$x, y = 0, 0$
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := x$ $b := y$ `Release`(l)

$a \neq b$?



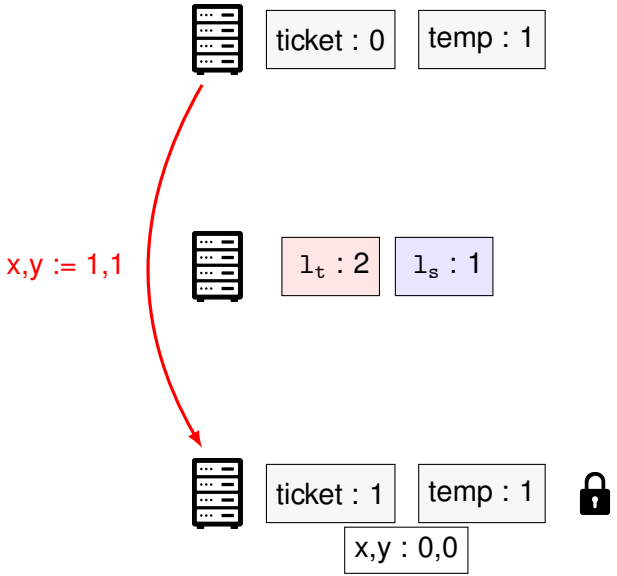
RDMA Lock Example v2

Let's double check with another case.

		$x, y = 0, 0$
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := x$ $b := y$ `Release`(l)

$a \neq b$ ✓

Oops...



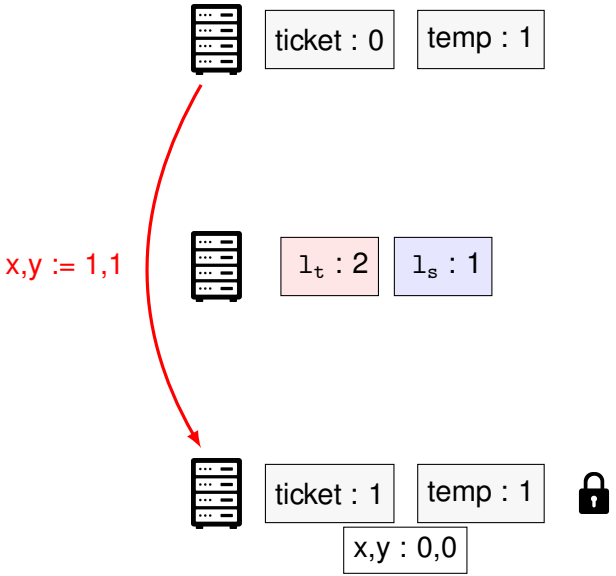
RDMA Lock Example v2

Let's double check with another case.

		$x, y = 0, 0$
	`lock` l	
`Acquire`(l) $\widetilde{x} := 1$ $\widetilde{y} := 1$ `Release`(l)		`Acquire`(l) $a := x$ $b := y$ `Release`(l)

$a \neq b$ ✓

Oops... Is it a **bug** or a **feature**?



Three Semantics

Need to decide how a lock is **supposed** to behave.

Trade-off between guarantees and efficiency.

We define:

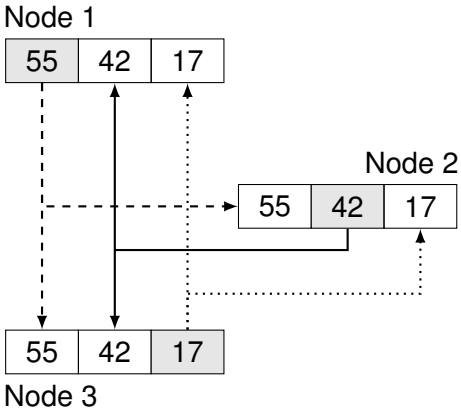
- **Strong Lock**: protects **everything** (similar to CPU locks)
- **Node Lock**: protects objects on **one computer** (e.g. previous ticket lock)
- **Weak Lock**: only **mutual exclusion** (programmers add restrictions if necessary)

Weak lock semantics allows optimisations!

Optimised (Weak) Ticket Lock (1/2)

Shared State Table (SST)[1]:

Data structure where each node has a value and a copy of other node's values.

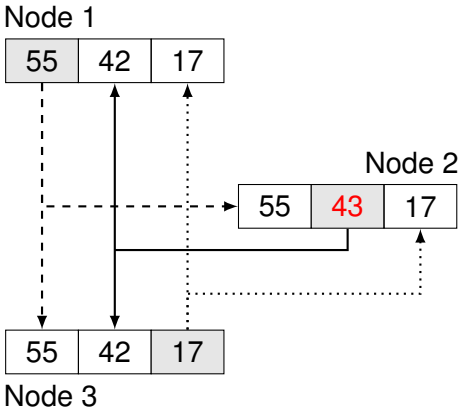


[1] Jha et al. 2018, "Derecho: Fast State Machine Replication for Cloud Services".

Optimised (Weak) Ticket Lock (1/2)

Shared State Table (SST)[1]:

Data structure where each node has a value and a copy of other node's values.

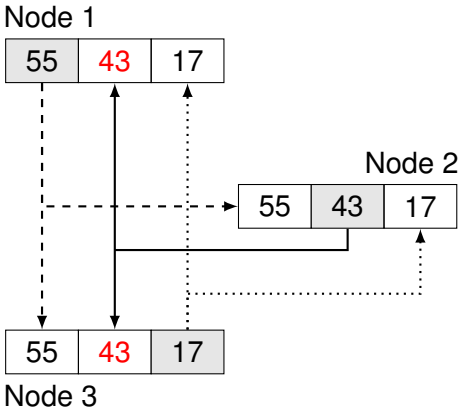


[1] Jha et al. 2018, "Derecho: Fast State Machine Replication for Cloud Services".

Optimised (Weak) Ticket Lock (1/2)

Shared State Table (SST)[1]:

Data structure where each node has a value and a copy of other node's values.

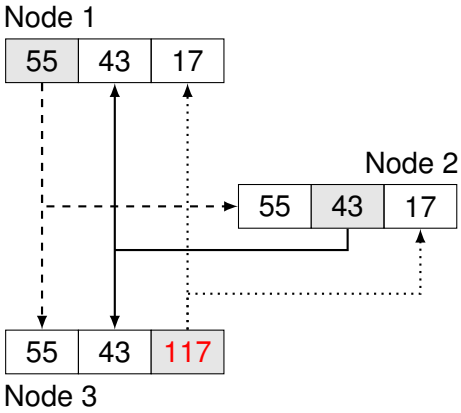


[1] Jha et al. 2018, "Derecho: Fast State Machine Replication for Cloud Services".

Optimised (Weak) Ticket Lock (1/2)

Shared State Table (SST)[1]:

Data structure where each node has a value and a copy of other node's values.

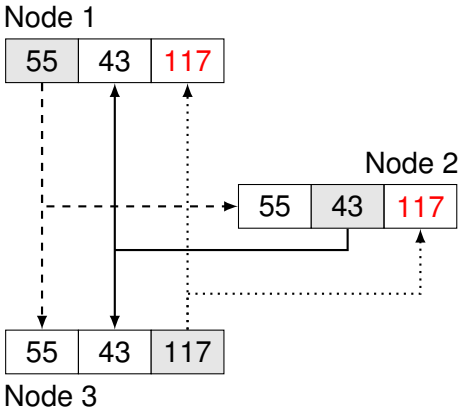


[1] Jha et al. 2018, "Derecho: Fast State Machine Replication for Cloud Services".

Optimised (Weak) Ticket Lock (1/2)

Shared State Table (SST)[1]:

Data structure where each node has a value and a copy of other node's values.



[1] Jha et al. 2018, "Derecho: Fast State Machine Replication for Cloud Services".

Optimised (Weak) Ticket Lock (2/2)

```
def acquire(l) = {
  ticket :=^d RFAA(l_t, 1)
  wait(d)
  while true {
    if (ticket in sst) {break}
  }}          CPU reads!

def release(l) = {
  sst.store_mine(ticket+1)
  sst.broadcast() # RDMA Writes
}
```

Let's use an SST for releasing the lock!
Served ticket is highest number in the SST.

$\Rightarrow$ Weak RDMA Lock (only mut. ex.)

Optimised (Weak) Ticket Lock (2/2)

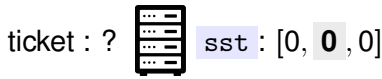
```
def acquire(l) = {
  ticket :=^d RFAA(l_t, 1)
  wait(d)
  while true {
    if (ticket in sst) {break}
  }}
```

CPU reads!

```
def release(l) = {
  sst.store_mine(ticket+1)
  sst.broadcast() # RDMA Writes
}
```

Let's use an SST for releasing the lock!
Served ticket is highest number in the SST.

$\Rightarrow$ Weak RDMA Lock (only mut. ex.)



Optimised (Weak) Ticket Lock (2/2)

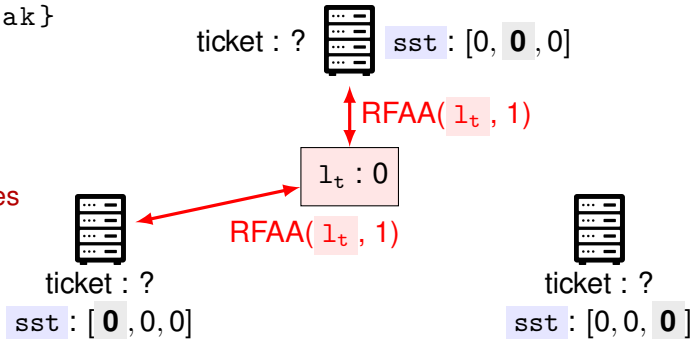
```
def acquire(l) = {
  ticket :=^d RFAA(l_t, 1)
  wait(d)
  while true {
    if (ticket in sst) {break}
  }}
```

CPU reads!

```
def release(l) = {
  sst.store_mine(ticket+1)
  sst.broadcast() # RDMA Writes
}
```

Let's use an SST for releasing the lock!
Served ticket is highest number in the SST.

$\Rightarrow$ Weak RDMA Lock (only mut. ex.)



Optimised (Weak) Ticket Lock (2/2)

```
def acquire(l) = {
  ticket :=^d RFAA(l_t, 1)
  wait(d)
  while true {
    if (ticket in sst) {break}
  }}
```

CPU reads!

```
def release(l) = {
  sst.store_mine(ticket+1)
  sst.broadcast() # RDMA Writes
}
```

Let's use an SST for releasing the lock!
Served ticket is highest number in the SST.

$\Rightarrow$ Weak RDMA Lock (only mut. ex.)

Optimised (Weak) Ticket Lock (2/2)

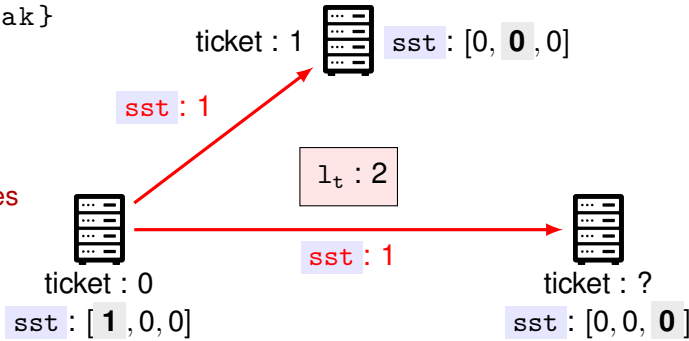
```
def acquire(l) = {
  ticket :=^d RFAA(l_t, 1)
  wait(d)
  while true {
    if (ticket in sst) {break}
  }}
```

CPU reads!

```
def release(l) = {
  sst.store_mine(ticket+1)
  sst.broadcast() # RDMA Writes
}
```

Let's use an SST for releasing the lock!
Served ticket is highest number in the SST.

$\Rightarrow$ Weak RDMA Lock (only mut. ex.)



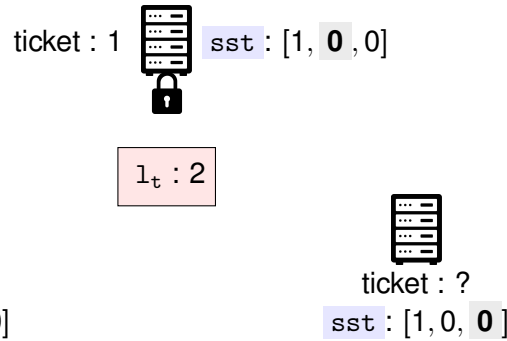
Optimised (Weak) Ticket Lock (2/2)

```
def acquire(l) = {
  ticket :=^d RFAA(l_t, 1)
  wait(d)
  while true {
    if (ticket in sst) {break}
  }}          CPU reads!

def release(l) = {
  sst.store_mine(ticket+1)
  sst.broadcast() # RDMA Writes
}
```

Let's use an SST for releasing the lock!
Served ticket is highest number in the SST.

$\Rightarrow$ Weak RDMA Lock (only mut. ex.)



Specifying and Verifying Implementations

Everything above (semantics, implementation, soundness) is formalised in Mowgli.

Mowgli: recent mathematical framework (POPL '26) for modular libraries in very weak memory models, using declarative semantics.

- Formal semantics for mutual exclusion (weak lock)
  (total order on acquire, dependency from release to next acquire)
- Additional guarantees for node/strong locks
- (Pen-and-paper) proof of an algorithm for each lock type
- Bonus: use-case proving node locks are enough to protect concurrent objects

Conclusion

Summary:

- First formalisation of **RDMA Atomics***
  (operational + declarative + equivalence)
- First formalisation of different **RDMA lock** semantics
  (weak, node, strong)
- First **verified implementations** of different RDMA lock libraries

*non-contractual name; terms and conditions may apply

Conclusion

Summary:

- First formalisation of **RDMA Atomics***
  (operational + declarative + equivalence)
- First formalisation of different **RDMA lock** semantics
  (weak, node, strong)
- First **verified implementations** of different RDMA lock libraries

*non-contractual name; terms and conditions may apply

Thanks for listening!

Poll vs Wait Semantics

Base RDMA (RDMA^{TSO}):

$x = 0$	$z = 0$
$\widetilde{z} := x$ $x := 1$	

$z = 1$ ✓

$x = 0$	$z = 0$
$\widetilde{z} := x$ $\texttt{poll}()$ $x := 1$	

$z = 1$ ✗

$x = 0$	$z = 0$
$\vdots$ $\widetilde{z} := x$ $\texttt{poll}()$ $x := 1$	

$z = 1$ ✓

Poll vs Wait Semantics

Base RDMA (RDMA^{TSO}):

$x = 0$	$z = 0$
$\widetilde{z} := x$ $x := 1$	

$z = 1$ ✓

$x = 0$	$z = 0$
$\widetilde{z} := x$ poll() $x := 1$	

$z = 1$ ✗

$x = 0$	$z = 0$
⋮ $\widetilde{z} := x$ poll() $x := 1$	

$z = 1$ ✓

LOCO ($\text{RDMA}^{\text{WAIT}}$):

$x = 0$	$z = 0$
$\widetilde{z} := x$ $x := 1$	

$z = 1$ ✓

$x = 0$	$z = 0$
$\widetilde{z} :=^{d} x$ wait(d) $x := 1$	

$z = 1$ ✗

$x = 0$	$z = 0$
⋮ $\widetilde{z} :=^{d} x$ wait(d) $x := 1$	

$z = 1$ ✗