

T. HOEFLER, M. PUESCHEL

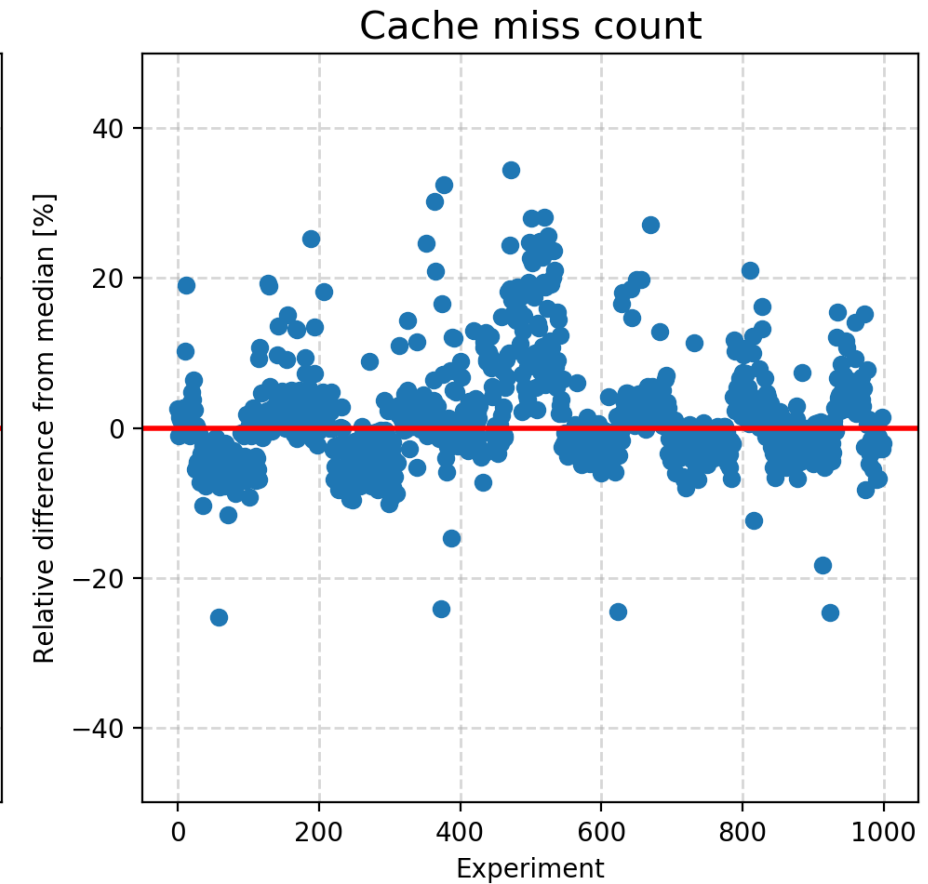
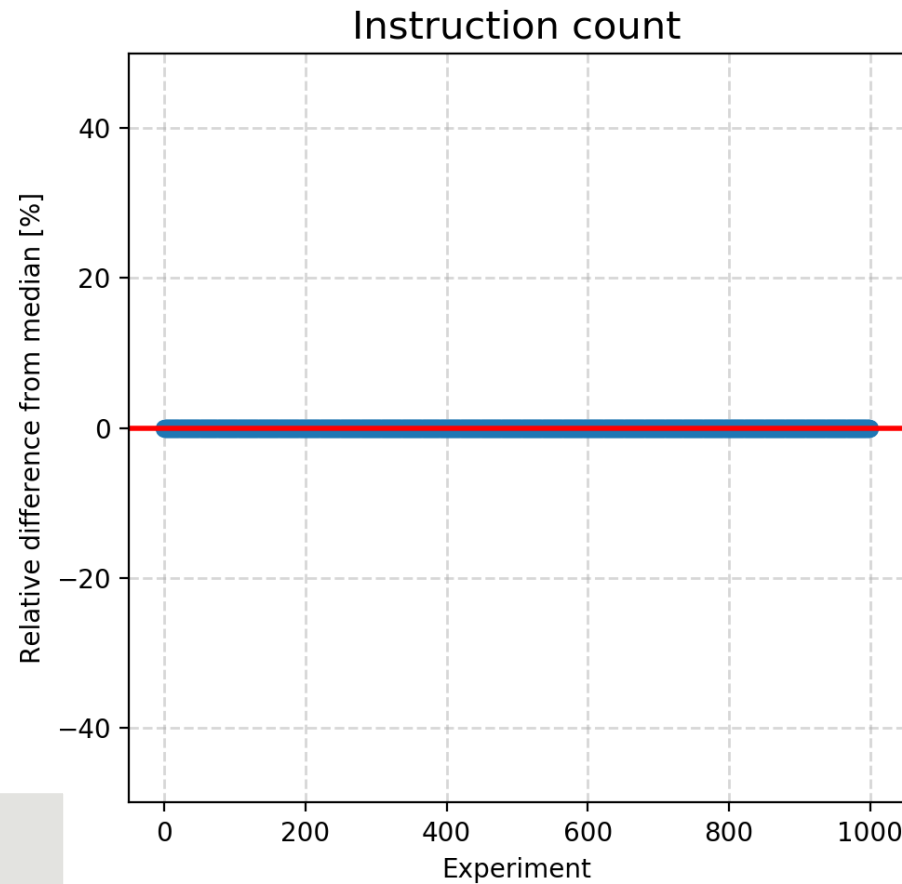
Lecture 5: Fast practical locks, lock-free, consensus, and scalable locks

Teaching assistant: Salvatore Di Girolamo

Motivational video: <https://www.youtube.com/watch?v=qx2dRIQXnbs>



Nondeterminism in [most] performance measurements!

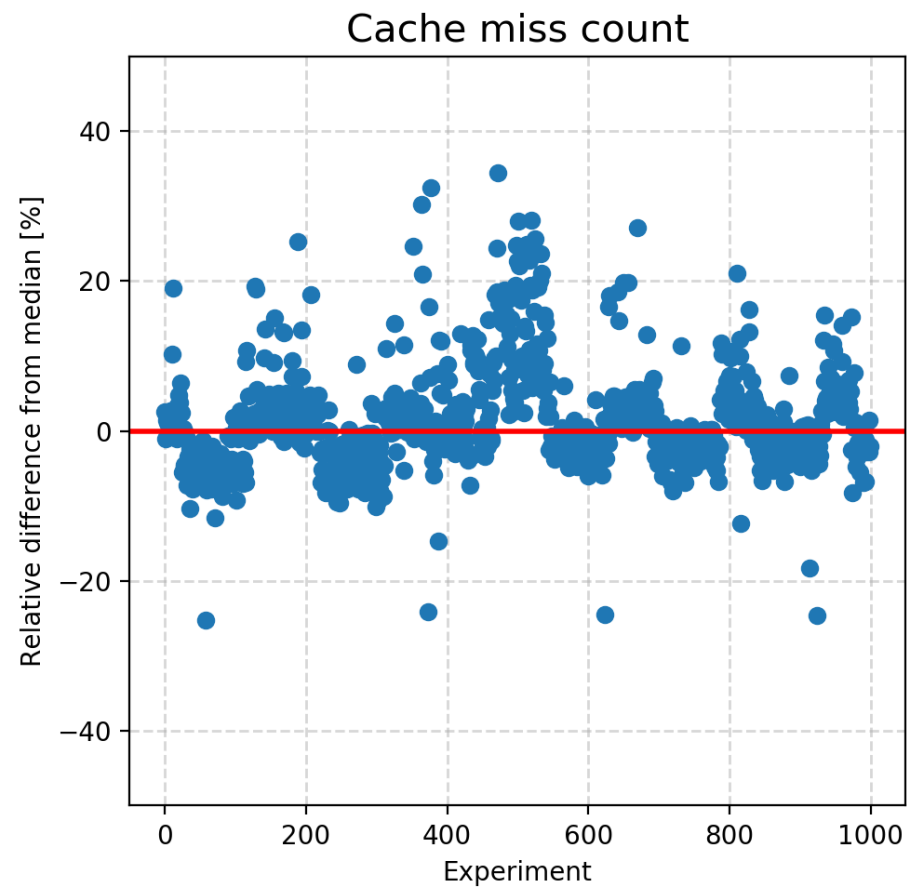
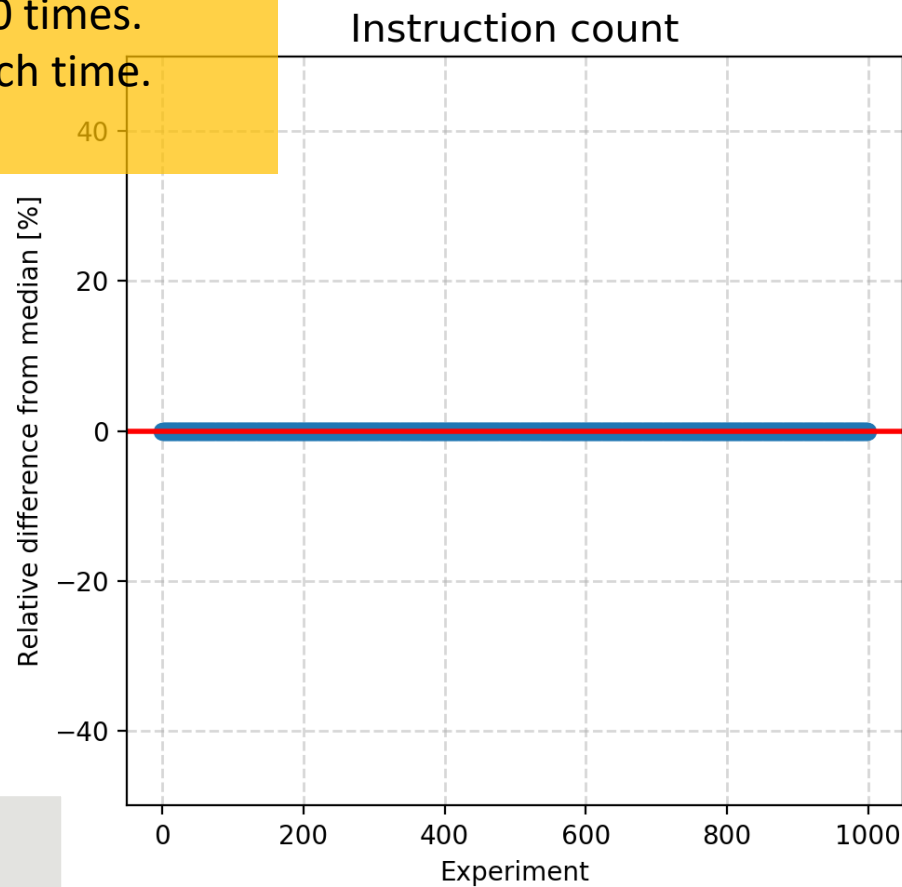


```
const int n=1000;  
volatile int a=0;  
for (int i=0; i<n; ++i)  
    a++;
```

Nondeterminism in [most] performance measurements!

Same code executed 1000 times.
Two metrics measured each time.

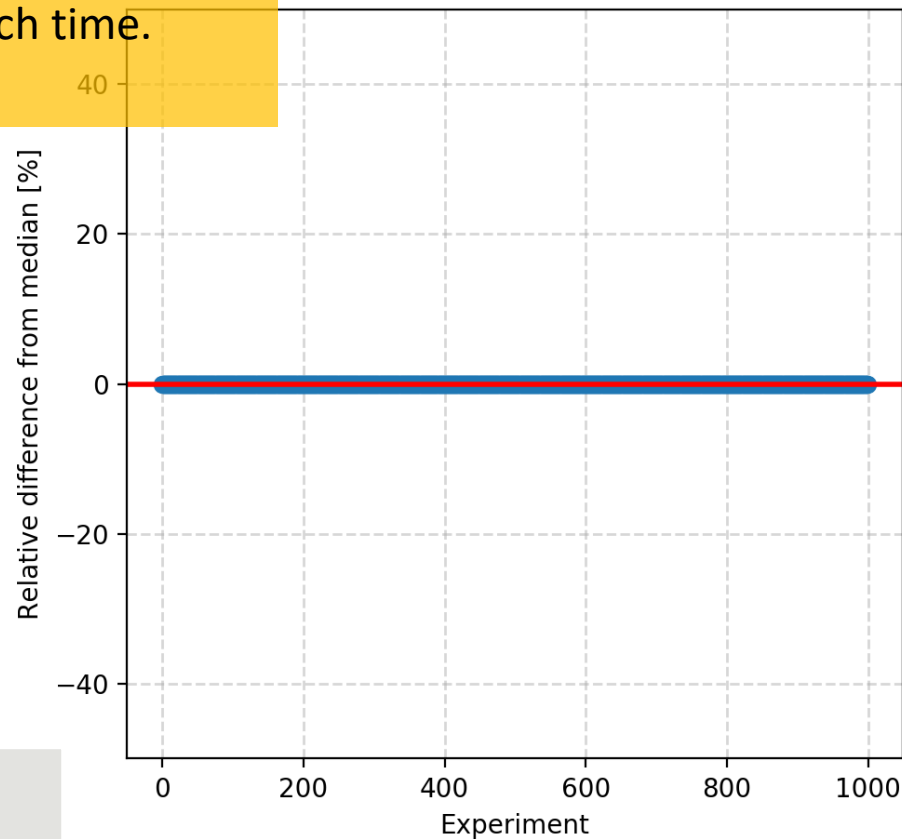
```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
    a++;
```



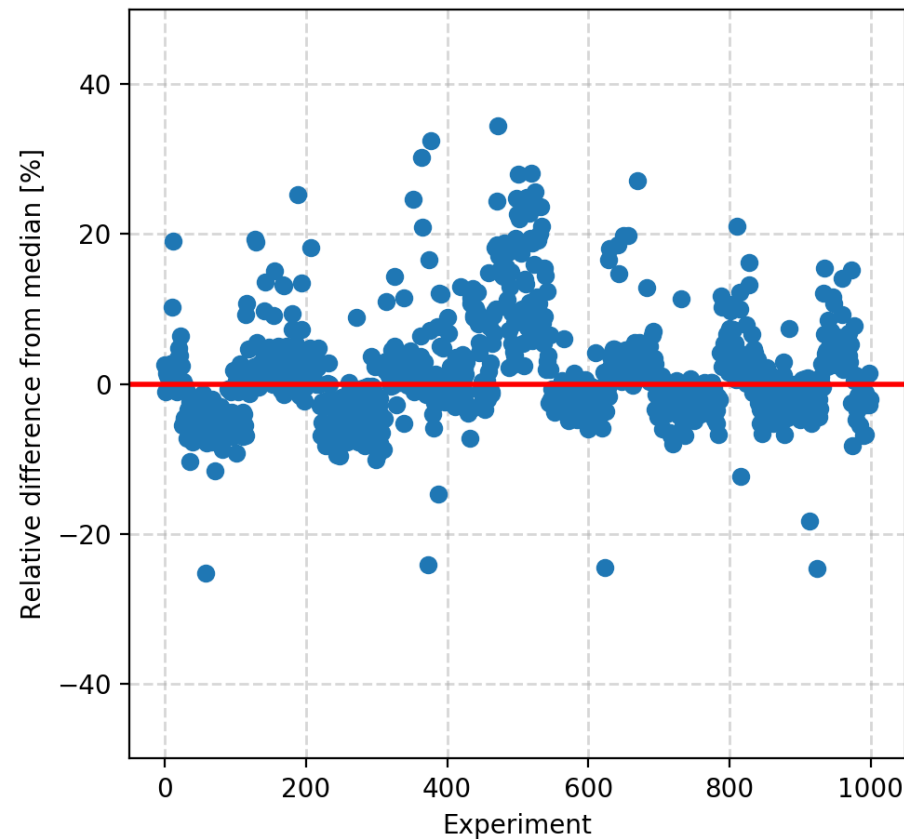
Nondeterminism in [most] performance measurements!

Same code executed 1000 times.
Two metrics measured each time.

Instruction count



Cache miss count



```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
    a++;
```

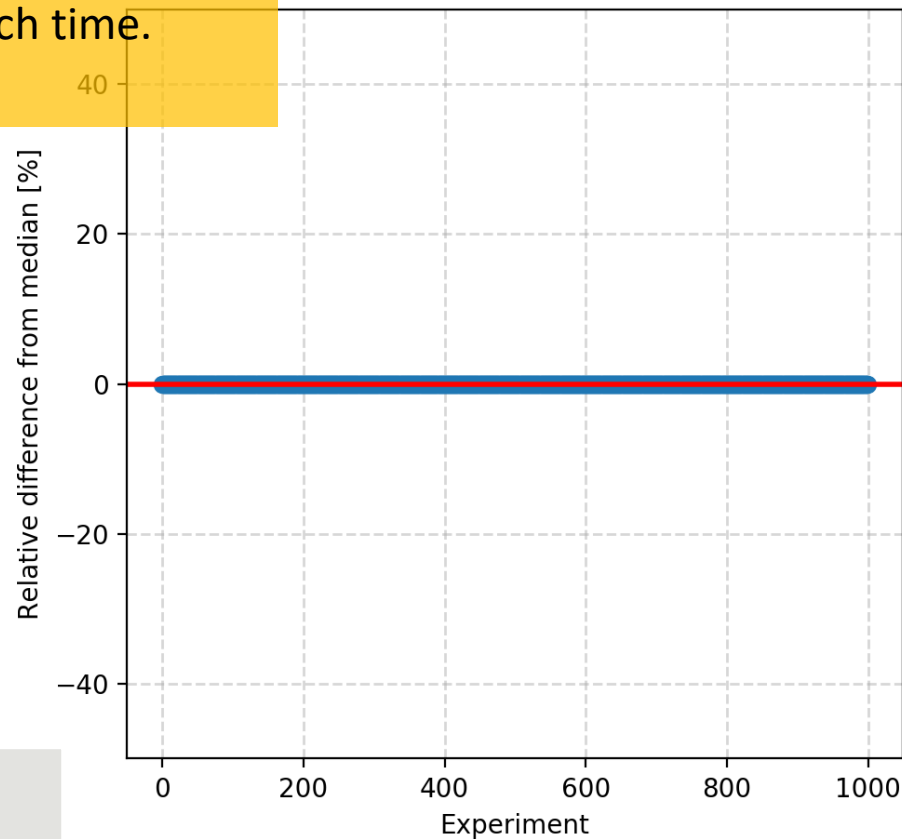
One is amazingly stable.

Nondeterminism in [most] performance measurements!

Same code executed 1000 times.
Two metrics measured each time.

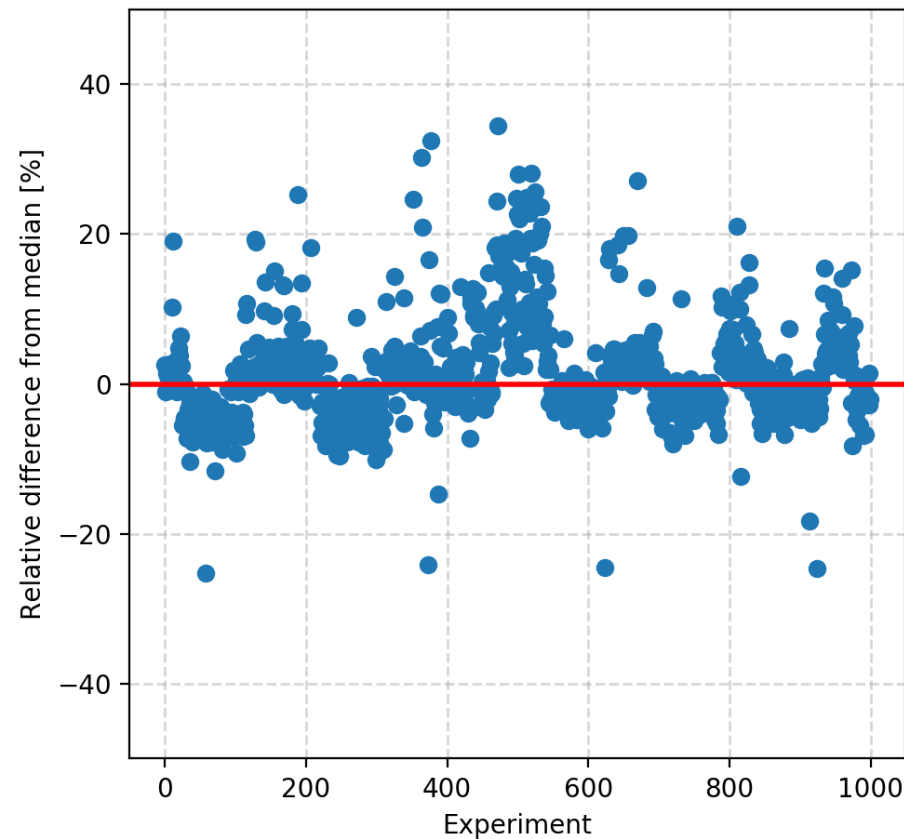
```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
    a++;
```

Instruction count



One is amazingly stable.

Cache miss count



The other—not at all!

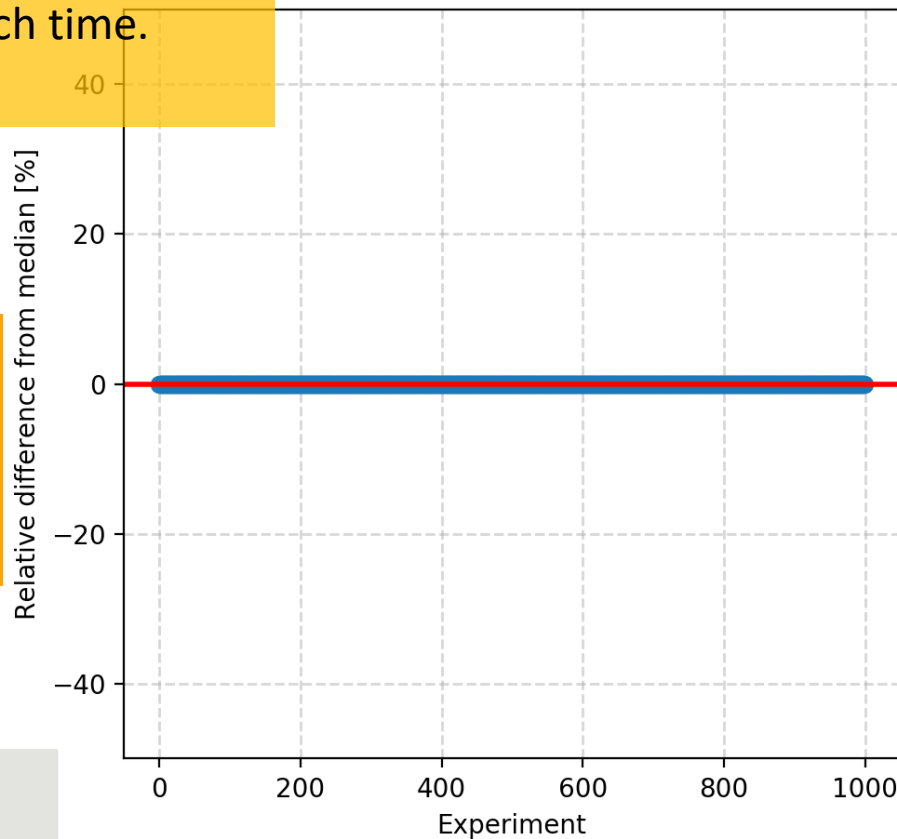
Nondeterminism in [most] performance measurements!

Same code executed 1000 times.
Two metrics measured each time.

How do we report measurements showing high variation?

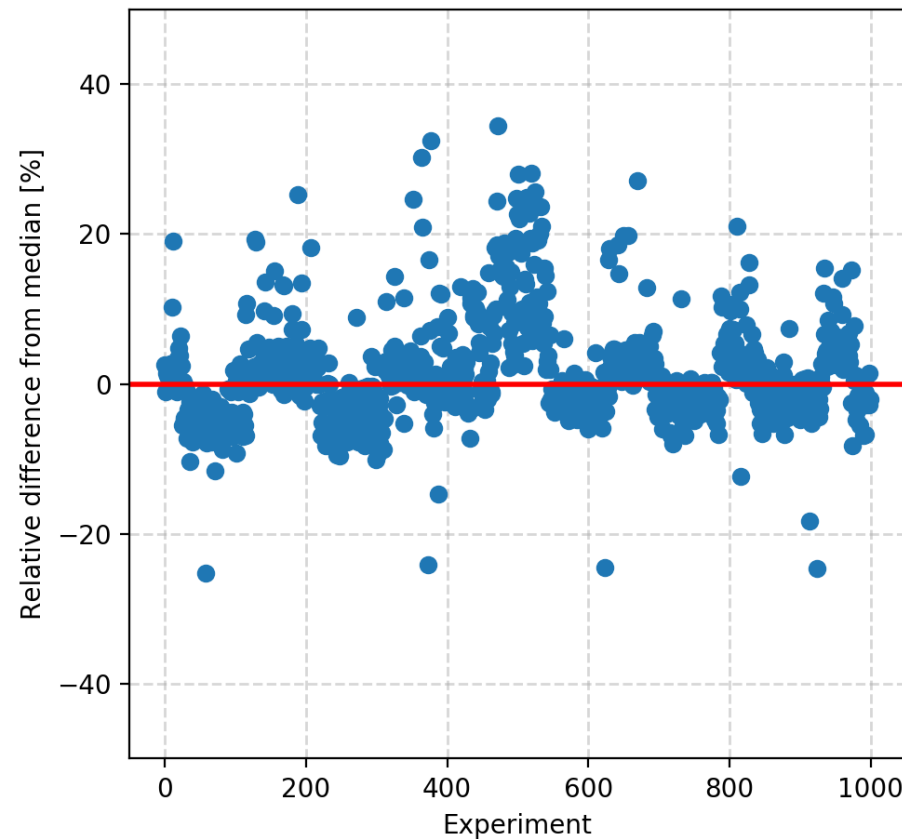
```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
    a++;
```

Instruction count



One is amazingly stable.

Cache miss count

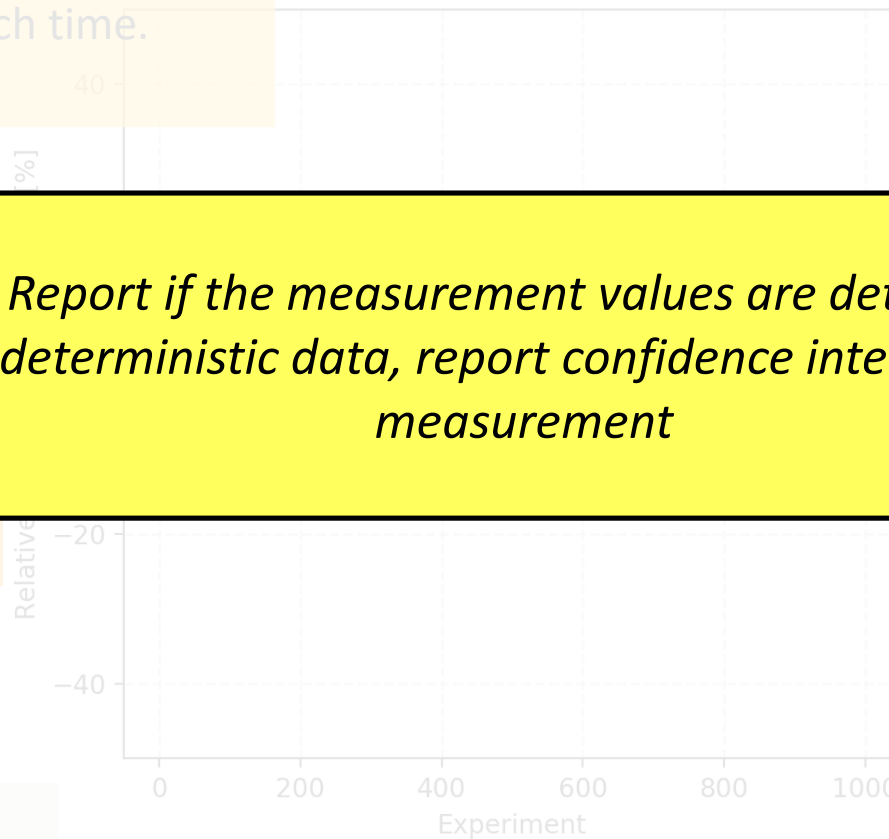


The other—not at all!

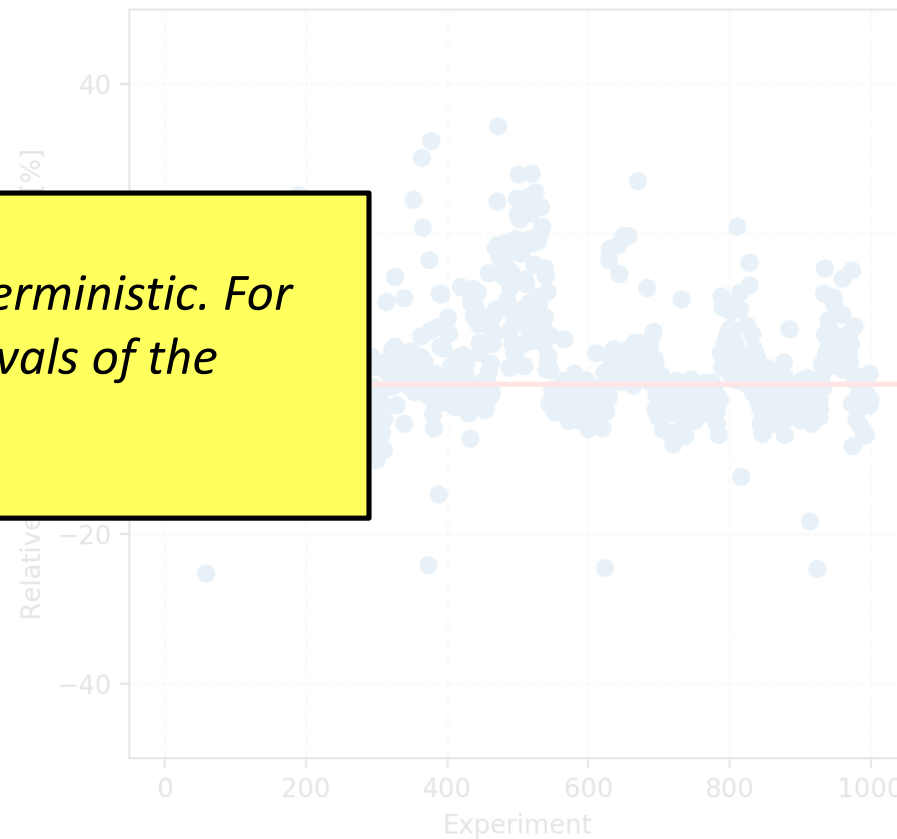
Nondeterminism in [most] performance measurements!

Same code executed 1000 times.
Two metrics measured each time.

Instruction count



Cache miss count



Rule 5: Report if the measurement values are deterministic. For nondeterministic data, report confidence intervals of the measurement

How do we report measurements showing high variability?

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
    a++;
```

One is amazingly stable.

The other—not at all!

Administrivia

- **Intermediate project presentation: next Monday 10/29 during lecture**
 - Report will be due in January!
Starting to write early is very helpful --- write – rewrite – rewrite (no joke!)
 - Coordinate your talk! You have 10 minutes (8 talk + 2 Q&A)
What will you be speaking about?
Focus on the key aspects (time is tight)!
Who will be speaking (up to you).
Engage the audience 😊
 - Send slides by Sunday night (11:59pm CH time) to Salvatore!
We will use a single (windows) laptop to avoid delays when switching
Expect only Windows (powerpoint) or a PDF viewer
The order of talks will be randomized for fairness

Review of last lecture

- **Memory models in practical parallel programming**

- Synchronized programming
- How locks synchronize processes *and* memory!

- **Proving program correctness**

- Pre-/postconditions – sequential
- Lifting to parallel

How to prove locked programs correct (nearly trivial)

- **Lock implementation**

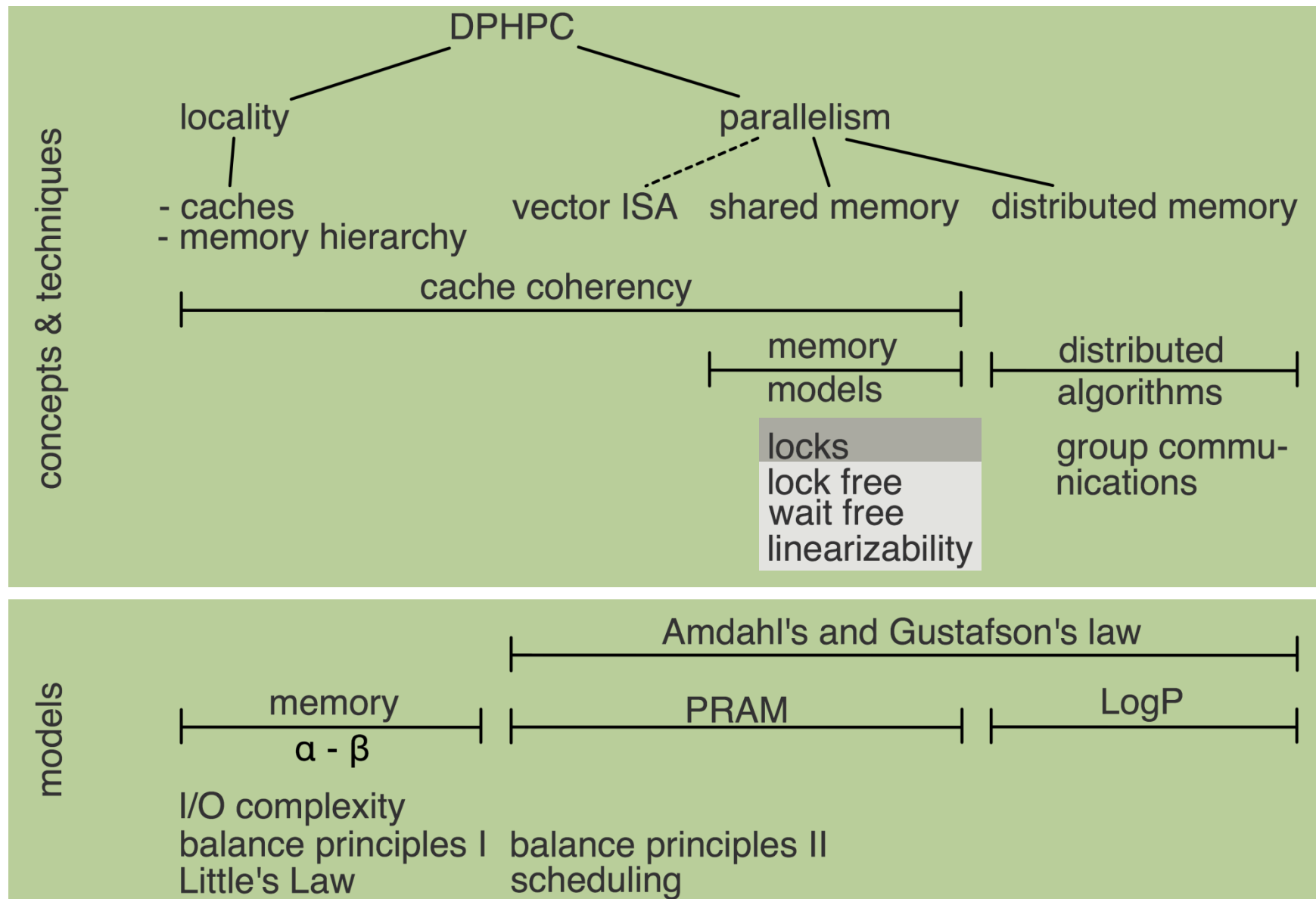
- Peterson lock – proof of correctness (using read/write histories, program and visibility orders)

With x86 memory model!

- Lock performance

Simple x86 – how much does memory model correctness cost?

DPHPC Overview



Goals of this lecture

- **Fast and scalable practical locks!**
 - Based on atomic operations
 - Why do we need atomic operations?
- **Recap lock-free and wait-free programming**
 - Proof that wait-free consensus is impossible without atomics
 - Valence argument: a proof technique similar to showing that atomics are needed for locks*
- **Locks in practical setting**
 - How to block?
 - When to block?
 - How long to block?
 - Simple proof of competitiveness*
- **Case study: large-scale distributed memory locking**
 - Problems and outline to next class

Back to Peterson in Practice ... on x86

- Implement and run our little counter on x86
- 100000 iterations
 - $1.6 \cdot 10^{-6}\%$ errors
 - What is the problem?

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I'm interested
    victim = tid; // other goes first
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```

Peterson in Practice ... on x86

- Implement and run our little counter on x86
- 100000 iterations
 - $1.6 \cdot 10^{-6}\%$ errors
 - What is the problem?

*No sequential
consistency
for $W(v)$ and
 $R(flag[j])$*

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I'm interested
    victim = tid; // other goes first
    asm("mfence");
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```

Peterson in Practice ... on x86

- Implement and run our little counter on x86
- Many iterations
 - $1.6 \cdot 10^{-6}\%$ errors
 - What is the problem?
No sequential consistency for $W(v)$ and $R(flag[j])$
 - Still $1.3 \cdot 10^{-6}\%$
Why?

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I'm interested
    victim = tid; // other goes first
    asm("mfence");
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```

Peterson in Practice ... on x86

- Implement and run our little counter on x86

- Many iterations

- $1.6 \cdot 10^{-6}\%$ errors
- What is the problem?

No sequential consistency for $W(v)$ and

$R(flag[j])$

- Still $1.3 \cdot 10^{-6}\%$

Why?

Reads may slip into CR!

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I'm interested
    victim = tid; // other goes first
    asm("mfence");
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    asm("mfence");
    flag[tid] = 0; // I'm not interested
}
```

Peterson in Practice ... on x86

- Implement and run our little counter on x86
- Many iterations
 - $1.6 \cdot 10^{-6}\%$ errors
 - What is the problem?
No sequential consistency for $W(v)$ and $R(flag[j])$
 - Still $1.3 \cdot 10^{-6}\%$
Why?
Reads may slip into CR!

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I'm interested
    victim = tid; // other goes first
    asm("mfence");
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    asm("mfence");
    flag[tid] = 0; // I'm not interested
}
```

The compiler may inline this function 😊

Correct Peterson Lock on x86

- Unoptimized (naïve sprinkling of mfences)
- Performance:
 - No mfence
375ns
 - mfence in lock
379ns
 - mfence in unlock
404ns
 - Two mfence
427ns (+14%)

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I'm interested
    victim = tid; // other goes first
    asm("mfence");
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    asm("mfence");
    flag[tid] = 0; // I'm not interested
}
```

Hardware Support?

- **Hardware atomic operations:**
 - Test&Set
 - Atomic swap
 - Fetch&Op
 - Compare&Swap
 - Load-linked/Store-Conditional LL/SC (or load-acquire (LDA) store-release (STL) on ARM)
 - Intel TSX (transactional synchronization extensions)

Hardware Support?

- **Hardware atomic operations:**
 - Test&Set
Write const to memory while returning the old value
 - Atomic swap
 - Fetch&Op
 - Compare&Swap
 - Load-linked/Store-Conditional LL/SC (or load-acquire (LDA) store-release (STL) on ARM)
 - Intel TSX (transactional synchronization extensions)

Hardware Support?

- **Hardware atomic operations:**

- Test&Set

- Write const to memory while returning the old value*

- Atomic swap

- Atomically exchange memory and register*

- Fetch&Op

- Compare&Swap

- Load-linked/Store-Conditional LL/SC (or load-acquire (LDA) store-release (STL) on ARM)

- Intel TSX (transactional synchronization extensions)

Hardware Support?

- **Hardware atomic operations:**
 - Test&Set
Write const to memory while returning the old value
 - Atomic swap
Atomically exchange memory and register
 - Fetch&Op
Get value and apply operation to memory location
 - Compare&Swap
- Load-linked/Store-Conditional LL/SC (or load-acquire (LDA) store-release (STL) on ARM)
- Intel TSX (transactional synchronization extensions)

Hardware Support?

- **Hardware atomic operations:**
 - Test&Set
Write const to memory while returning the old value
 - Atomic swap
Atomically exchange memory and register
 - Fetch&Op
Get value and apply operation to memory location
 - Compare&Swap
Compare two values and swap memory with register if equal
 - Load-linked/Store-Conditional LL/SC (or load-acquire (LDA) store-release (STL) on ARM)
- Intel TSX (transactional synchronization extensions)

Hardware Support?

- **Hardware atomic operations:**

- Test&Set

- Write const to memory while returning the old value*

- Atomic swap

- Atomically exchange memory and register*

- Fetch&Op

- Get value and apply operation to memory location*

- Compare&Swap

- Compare two values and swap memory with register if equal*

- Load-linked/Store-Conditional LL/SC (or load-acquire (LDA) store-release (STL) on ARM)

- Loads value from memory, allows operations, commits only if no other updates committed → mini-TM*

- Intel TSX (transactional synchronization extensions)

Hardware Support?

- **Hardware atomic operations:**

- Test&Set

- Write const to memory while returning the old value*

- Atomic swap

- Atomically exchange memory and register*

- Fetch&Op

- Get value and apply operation to memory location*

- Compare&Swap

- Compare two values and swap memory with register if equal*

- Load-linked/Store-Conditional LL/SC (or load-acquire (LDA) store-release (STL) on ARM)

- Loads value from memory, allows operations, commits only if no other updates committed → mini-TM*

- Intel TSX (transactional synchronization extensions)

- Hardware-TM (roll your own atomic operations)*

Relative Power of Synchronization

- **Design-Problem I: Multi-core Processor**
 - Which atomic operations are useful?
- **Design-Problem II: Complex Application**
 - What atomic should I use?
- **Generally hard to answer ☹**
 - Depends on too many systems details (access patterns, CC implementation, contention, algorithm ...)
- **Concept of “consensus number” C if a primitive can be used to solve the “consensus problem” in a finite number of steps (even if threads stop)**
 - atomic registers have $C=1$ (thus locks have $C=1!$)
 - TAS, Swap, Fetch&Op have $C=2$
 - CAS, LL/SC, TM have $C=\infty$

Test-and-Set Locks

- **Test-and-Set semantics**
 - Memoize old value
 - Set fixed value TASval (true)
 - Return old value
- **After execution:**
 - Post-condition is a fixed (constant) value!

```
bool TestAndSet (bool *flag) {  
    bool old = *flag;  
    *flag = true;  
    return old;  
} // all atomic!
```

Test-and-Set Locks

- Assume TASval indicates “locked”
- Write something else to indicate “unlocked”
- TAS until return value is \neq TASval (1 in this example)

- When will the lock be granted?
- Does this work well in practice?

```
bool TestAndSet (bool *flag) {  
    bool old = *flag;  
    *flag = true;  
    return old;  
} // all atomic!
```

```
volatile int lck = 0;  
  
void lock() {  
    while (TestAndSet(&lck) == 1);  
}  
  
void unlock() {  
    lck = 0;  
}
```

Cacheline contention (or: why I told you about MESI and friends)

- On x86, the XCHG instruction is used to implement TAS
 - x86 lock is implicit in xchg!
- **Cacheline is read and written**
 - Ends up in exclusive state, invalidates other copies
 - Cacheline is “thrown” around uselessly
 - High load on memory subsystem

x86 lock is essentially a full memory barrier ☹️

```
movl    $1, %eax
xchg    %eax, (%ebx)
```

Test-and-Test-and-Set (TATAS) Locks

- Spinning in TAS is not a good idea
- Spin on cache line in shared state
 - All threads at the same time, no cache coherency/memory traffic
- **Danger!**
 - Efficient but use with great care!
 - Generalizations are very dangerous

```
volatile int lck = 0;

void lock() {
    do {
        while (lck == 1);
    } while (TestAndSet(&lck) == 1);
}

void unlock() {
    lck = 0;
}
```

Warning: Even Experts get it wrong!

- Example: Double-Checked Locking

Warning: Even Experts get it wrong!

■ Example: Double-Checked Locking

1997

Double-Checked Locking

An Optimization Pattern for Efficiently
Initializing and Accessing Thread-safe Objects

Douglas C. Schmidt
schmidt@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

Tim Harrison
harrison@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

This paper appeared in a chapter in the book “Pattern Languages of Program Design 3” ISBN, edited by Robert Martin, Frank Buschmann, and Dirke Riehle published by Addison-Wesley, 1997.

Abstract

This paper shows how the canonical implementation [1] of the Singleton pattern does not work correctly in the presence of preemptive multi-tasking or true parallelism. To solve this problem, we present the Double-Checked Locking optimization pattern. This pattern is useful for reducing contention and synchronization overhead whenever “critical sections” of code should be executed just once. In addition, Double-Checked Locking illustrates how changes in underlying forces (i.e., adding multi-threading and parallelism to the common Singleton use-case) can impact the form and content of patterns used to develop concurrent software.

context of concurrency. To illustrate this, consider how the canonical implementation [1] of the Singleton pattern behaves in multi-threaded environments.

The Singleton pattern ensures a class has only one instance and provides a global point of access to that instance [1]. Dynamically allocating Singletons in C++ programs is common since the order of initialization of global static objects in C++ programs is not well-defined and is therefore non-portable. Moreover, dynamic allocation avoids the cost of initializing a Singleton if it is never used.

Defining a Singleton is straightforward:

```
class Singleton
{
public:
    static Singleton *instance (void)
    {
        if (instance_ == 0)
            // Critical section.
            instance_ = new Singleton;
        return instance_;
    }
}
```

Warning: Even Experts get it wrong!

- Example: Double-Checked Locking

1997

Double-Checked Locking

An Optimization Pattern for Efficiently
 Initializing and Accessing Thread-safe Objects

Douglas C. Schmidt
 schmidt@cs.wustl.edu
 Dept. of Computer Science
 Wash. U., St. Louis

Tim Harrison
 harrison@cs.wustl.edu
 Dept. of Computer Science
 Wash. U., St. Louis

This paper appeared in a chapter in the book "Pattern Languages of Program Design 3" ISBN, edited by Robert Martin, Frank Buschmann, and Dirke Riehle published by Addison-Wesley, 1997.

Abstract

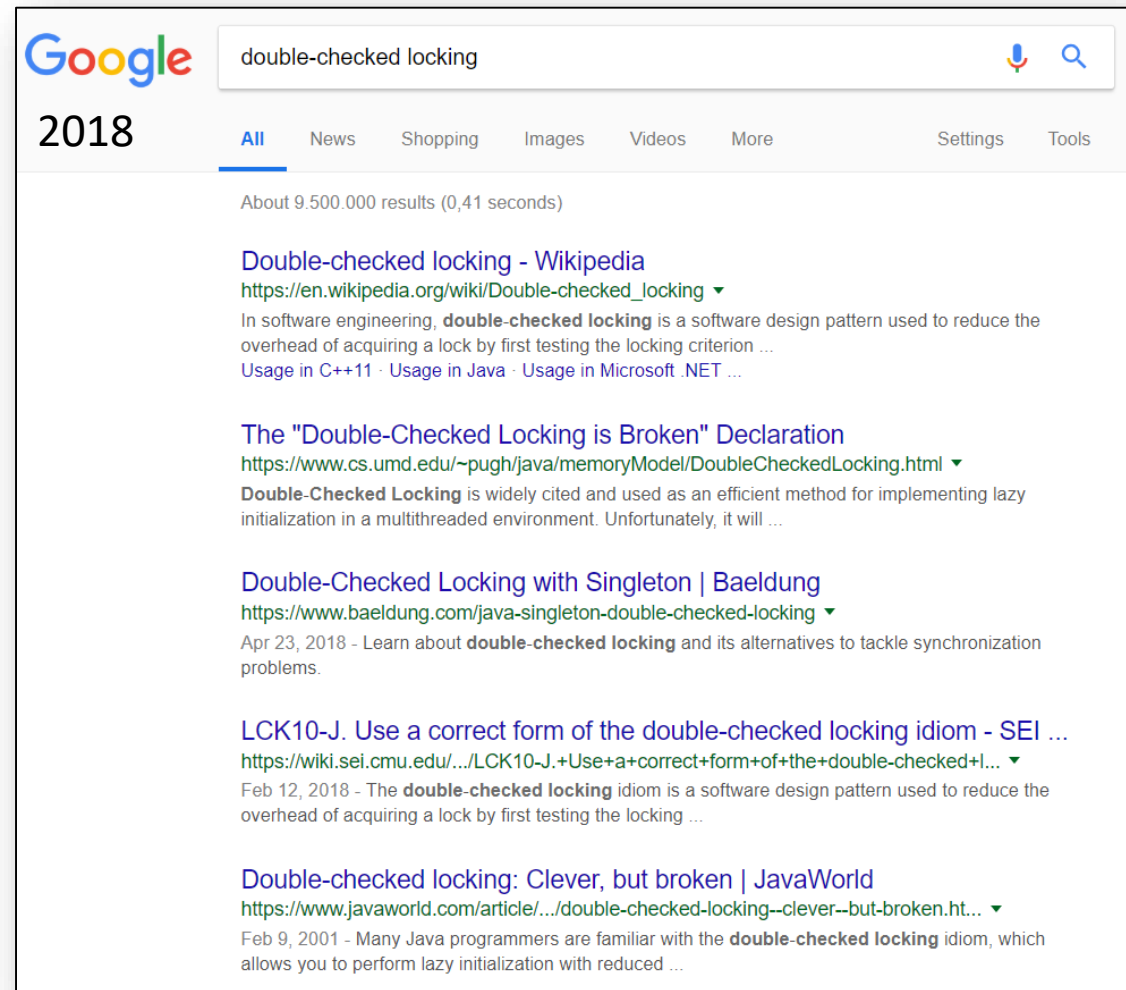
This paper shows how the canonical implementation [1] of the Singleton pattern does not work correctly in the presence of preemptive multi-tasking or true parallelism. To solve this problem, we present the Double-Checked Locking optimization pattern. This pattern is useful for reducing contention and synchronization overhead whenever "critical sections" of code should be executed just once. In addition, Double-Checked Locking illustrates how changes in underlying forces (i.e., adding multi-threading and parallelism to the common Singleton use-case) can impact the form and content of patterns used to develop concurrent software.

context of concurrency. To illustrate this, consider how the canonical implementation [1] of the Singleton pattern behaves in multi-threaded environments.

The Singleton pattern ensures a class has one instance and provides a global point of access to that instance [1]. Dynamically allocating Singletons in C++ programs is common since the order of initialization of global static objects in C++ programs is not well-defined and is therefore non-portable. Moreover, dynamic allocation avoids the cost of initializing a Singleton if it is never used.

Defining a Singleton is straightforward:

```
class Singleton
{
public:
    static Singleton *instance (void)
    {
        if (instance_ == 0)
            // Critical section.
            instance_ = new Singleton;
        return instance_;
    }
}
```



Google double-checked locking

2018

All News Shopping Images Videos More Settings Tools

About 9,500,000 results (0,41 seconds)

[Double-checked locking - Wikipedia](https://en.wikipedia.org/wiki/Double-checked_locking)
https://en.wikipedia.org/wiki/Double-checked_locking ▾
 In software engineering, **double-checked locking** is a software design pattern used to reduce the overhead of acquiring a lock by first testing the locking criterion ...
[Usage in C++11](#) · [Usage in Java](#) · [Usage in Microsoft .NET](#) ...

[The "Double-Checked Locking is Broken" Declaration](https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html)
<https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html> ▾
Double-Checked Locking is widely cited and used as an efficient method for implementing lazy initialization in a multithreaded environment. Unfortunately, it will ...

[Double-Checked Locking with Singleton | Baeldung](https://www.baeldung.com/java-singleton-double-checked-locking)
<https://www.baeldung.com/java-singleton-double-checked-locking> ▾
 Apr 23, 2018 - Learn about **double-checked locking** and its alternatives to tackle synchronization problems.

[LCK10-J. Use a correct form of the double-checked locking idiom - SEI ...](https://wiki.sei.cmu.edu/.../LCK10-J.+Use+a+correct+form+of+the+double-checked+I...)
<https://wiki.sei.cmu.edu/.../LCK10-J.+Use+a+correct+form+of+the+double-checked+I...> ▾
 Feb 12, 2018 - The **double-checked locking** idiom is a software design pattern used to reduce the overhead of acquiring a lock by first testing the locking ...

[Double-checked locking: Clever, but broken | JavaWorld](https://www.javaworld.com/article/.../double-checked-locking--clever--but-broken.ht...)
<https://www.javaworld.com/article/.../double-checked-locking--clever--but-broken.ht...> ▾
 Feb 9, 2001 - Many Java programmers are familiar with the **double-checked locking** idiom, which allows you to perform lazy initialization with reduced ...

Warning: Even Experts get it wrong!

- Example: Double-Checked Locking

1997

Double-Checked Locking

An Optimization Pattern for Efficiently
Initializing and Accessing Thread-safe Objects

Douglas C. Schmidt
schmidt@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

Tim Harrison
harrison@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

This paper appeared in a chapter in the book "Pattern Languages of Program Design 3" ISBN, edited by Robert Martin, Frank Buschmann, and Dirke Riehle published by Addison-Wesley, 1997.

Abstract

This paper shows how the canonical implementation [1] of the Singleton pattern does not work correctly in the presence of preemptive multi-tasking or true parallelism. To solve this problem, we present the Double-Checked Locking optimization pattern. This pattern is useful for reducing contention and synchronization overhead whenever "critical sections" of code should be executed just once. In addition, Double-Checked Locking illustrates how changes in underlying forces (i.e., adding multi-threading and parallelism to the common Singleton use-case) can impact the form and content of patterns used to develop concurrent software.

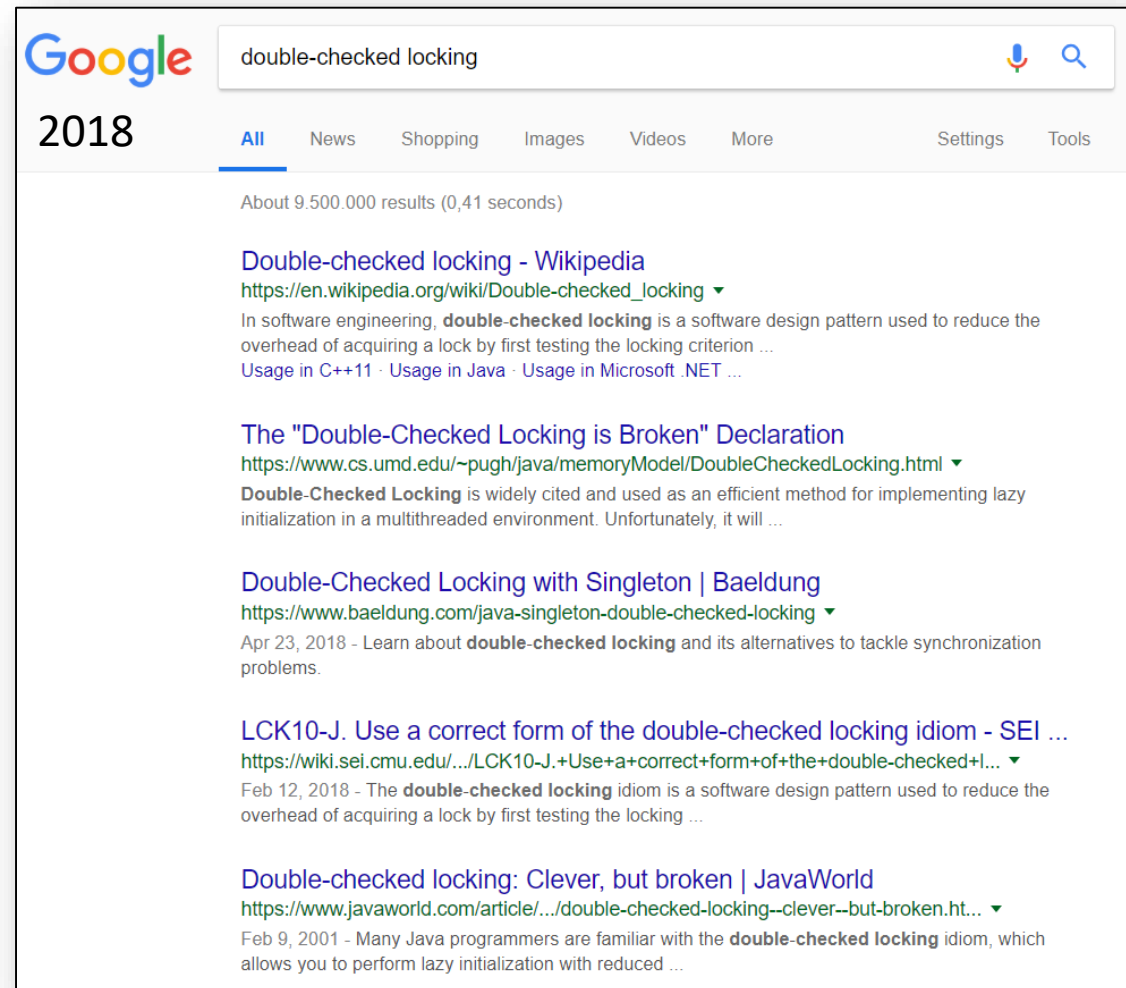
context of concurrency. To illustrate this, consider how the canonical implementation [1] of the Singleton pattern behaves in multi-threaded environments.

The Singleton pattern ensures a class has only one instance and provides a global point of access to that instance [1]. Dynamically allocating Singletons in C++ programs is common since the order of initialization of global static objects in C++ programs is not well-defined and is therefore non-portable. Moreover, dynamic allocation avoids the cost of initializing a Singleton if it is never used.

Defining a Singleton is straightforward:

```
class Singleton
{
public:
    static Singleton *instance (void)
    {
        if (instance_ == 0)
            // Critical section.
            instance_ = new Singleton;

        return instance_;
    }
}
```



Google double-checked locking

2018

All News Shopping Images Videos More Settings Tools

About 9,500,000 results (0,41 seconds)

[Double-checked locking - Wikipedia](https://en.wikipedia.org/wiki/Double-checked_locking)
https://en.wikipedia.org/wiki/Double-checked_locking ▾
 In software engineering, **double-checked locking** is a software design pattern used to reduce the overhead of acquiring a lock by first testing the locking criterion ...
[Usage in C++11](#) · [Usage in Java](#) · [Usage in Microsoft .NET](#) ...

[The "Double-Checked Locking is Broken" Declaration](https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html)
<https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html> ▾
Double-Checked Locking is widely cited and used as an efficient method for implementing lazy initialization in a multithreaded environment. Unfortunately, it will ...

[Double-Checked Locking with Singleton | Baeldung](https://www.baeldung.com/java-singleton-double-checked-locking)
<https://www.baeldung.com/java-singleton-double-checked-locking> ▾
 Apr 23, 2018 - Learn about **double-checked locking** and its alternatives to tackle synchronization problems.

[LCK10-J. Use a correct form of the double-checked locking idiom - SEI ...](https://wiki.sei.cmu.edu/.../LCK10-J.+Use+a+correct+form+of+the+double-checked+I...)
<https://wiki.sei.cmu.edu/.../LCK10-J.+Use+a+correct+form+of+the+double-checked+I...> ▾
 Feb 12, 2018 - The **double-checked locking** idiom is a software design pattern used to reduce the overhead of acquiring a lock by first testing the locking ...

[Double-checked locking: Clever, but broken | JavaWorld](https://www.javaworld.com/article/.../double-checked-locking--clever--but-broken.ht...)
<https://www.javaworld.com/article/.../double-checked-locking--clever--but-broken.ht...> ▾
 Feb 9, 2001 - Many Java programmers are familiar with the **double-checked locking** idiom, which allows you to perform lazy initialization with reduced ...

Problem: Memory ordering leads to race-conditions!

Contention?

- Do TATAS locks still have contention?

```
volatile int lck = 0;

void lock() {
    do {
        while (lck == 1);
    } while (TestAndSet(&lck) == 1);
}

void unlock() {
    lck = 0;
}
```

Contention?

- Do TATAS locks still have contention?
- When lock is released, k threads fight for cache line ownership
 - One gets the lock, all get the CL exclusively (serially!)
 - What would be a good solution? (think “collision avoidance”)

```
volatile int lck = 0;

void lock() {
    do {
        while (lck == 1);
    } while (TestAndSet(&lck) == 1);
}

void unlock() {
    lck = 0;
}
```

TAS Lock with Exponential Backoff

- Exponential backoff eliminates contention statistically

- Locks granted in unpredictable order
- Starvation possible but unlikely

How can we make it even less likely?

```
volatile int lck = 0;

void lock() {
    while (TestAndSet(&lck) == 1) {
        wait(time);
        time *= 2; // double waiting time
    }
}

void unlock() {
    lck = 0;
}
```

TAS Lock with Exponential Backoff

- **Exponential backoff eliminates contention statistically**

- Locks granted in unpredictable order
- Starvation possible but unlikely

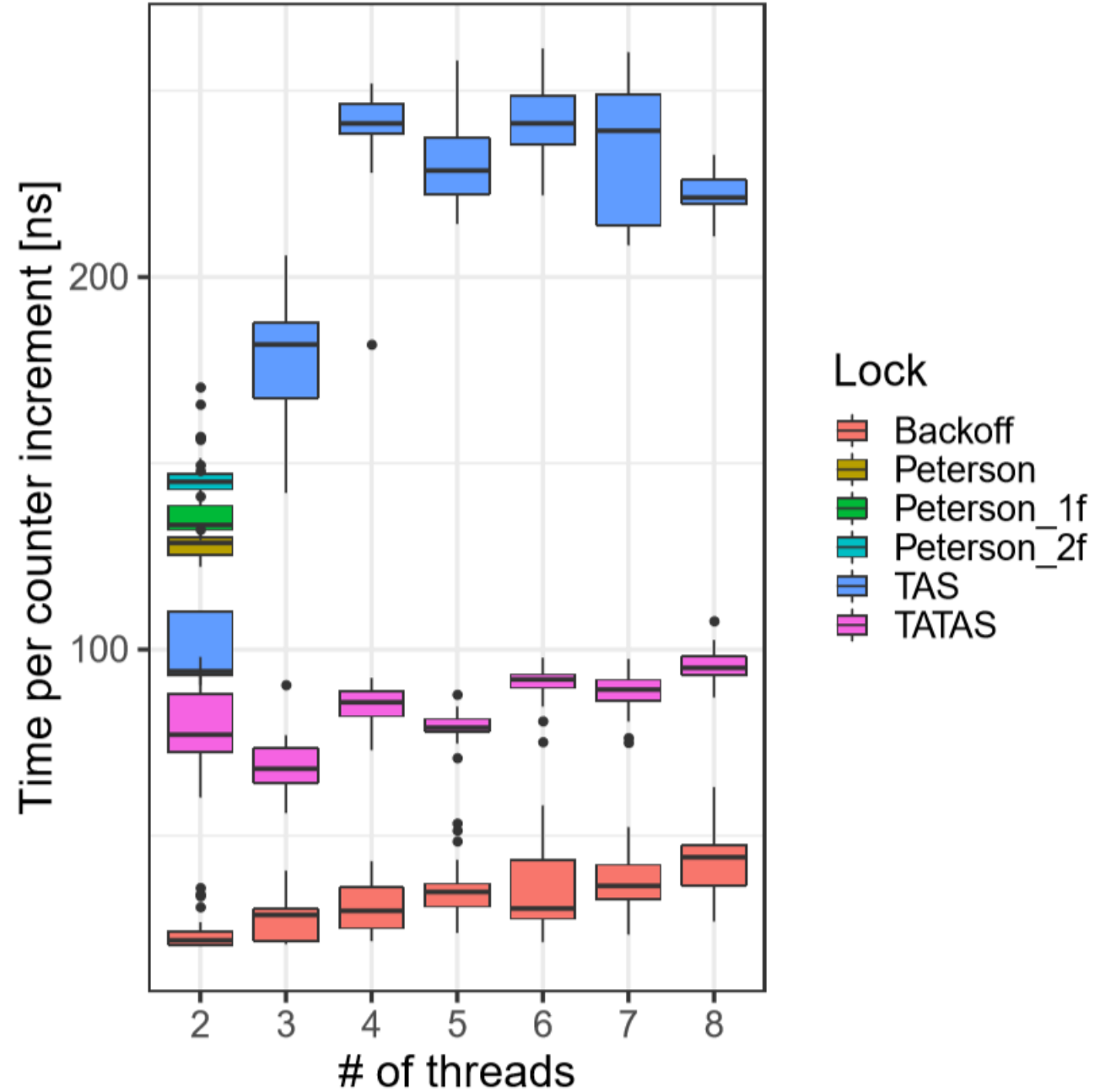
Maximum waiting time makes it less likely

```
volatile int lck = 0;
const int maxtime=1000;

void lock() {
    while (TestAndSet(&lck) == 1) {
        wait(time);
        time = min(time * 2, maxtime);
    }
}

void unlock() {
    lck = 0;
}
```

Perofmance of Locks



Improvements?

- **Are TAS locks perfect?**
 - What are the two biggest issues?

Improvements?

- **Are TAS locks perfect?**

- What are the two biggest issues?

- Cache coherency traffic (contending on same location with expensive atomics)

-- or --

- Critical section underutilization (waiting for backoff times will delay entry to CR)

- **What would be a fix for that?**

- How is this solved at airports and shops (often at least)?

Improvements?

- **Are TAS locks perfect?**

- What are the two biggest issues?
 - Cache coherency traffic (contending on same location with expensive atomics)

-- or --

- Critical section underutilization (waiting for backoff times will delay entry to CR)

- **What would be a fix for that?**

- How is this solved at airports and shops (often at least)?

- **Queue locks -- Threads enqueue**

- Learn from predecessor if it's their turn
- Each threads spins at a different location
- FIFO fairness

Array Queue Lock

- **Array to implement queue**
 - Tail-pointer shows next free queue position
 - Each thread spins on own location
CL padding!
 - `index[]` array can be put in TLS

```
volatile int array[n] = {1,0,...,0};
volatile int index[n] = {0,0,...,0};
volatile int tail = 0;

void lock() {
    index[tid] = GetAndInc(tail) % n;
    while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
    array[index[tid]] = 0; // I release my lock
    array[(index[tid] + 1) % n] = 1; // next one
}
```

Array Queue Lock

- **Array to implement queue**
 - Tail-pointer shows next free queue position
 - Each thread spins on own location
 - CL padding!*
 - index[] array can be put in TLS
- **So are we done now?**
 - What's wrong?

```
volatile int array[n] = {1,0,...,0};
volatile int index[n] = {0,0,...,0};
volatile int tail = 0;

void lock() {
    index[tid] = GetAndInc(tail) % n;
    while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
    array[index[tid]] = 0; // I release my lock
    array[(index[tid] + 1) % n] = 1; // next one
}
```

Array Queue Lock

- **Array to implement queue**
 - Tail-pointer shows next free queue position
 - Each thread spins on own location
 - CL padding!*
 - `index[]` array can be put in TLS
- **So are we done now?**
 - What's wrong?
 - Synchronizing M objects requires $\Theta(NM)$ storage
 - What do we do now?

```
volatile int array[n] = {1,0,...,0};
volatile int index[n] = {0,0,...,0};
volatile int tail = 0;

void lock() {
    index[tid] = GetAndInc(tail) % n;
    while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
    array[index[tid]] = 0; // I release my lock
    array[(index[tid] + 1) % n] = 1; // next one
}
```

CLH Lock (1993)

- List-based (same queue principle)
- Discovered twice by Craig, Landin, Hagersten 1993/94
- 2N+3M words
 - N threads, M locks
- Requires thread-local qnode pointer
 - Can be hidden!

```
typedef struct qnode {  
    struct qnode *prev;  
    int succ_blocked;  
} qnode;
```

```
qnode *lck = new qnode; // node owned by lock
```

```
void lock(qnode *lck, qnode *qn) {  
    qn->succ_blocked = 1;  
    qn->prev = FetchAndSet(lck, qn);  
    while (qn->prev->succ_blocked);  
}
```

```
void unlock(qnode **qn) {  
    qnode *pred = (*qn)->prev;  
    (*qn)->succ_blocked = 0;  
    *qn = pred;  
}
```

CLH Lock (1993)

- **Qnode objects represent thread state!**
 - `succ_blocked == 1` if waiting or acquired lock
 - `succ_blocked == 0` if released lock
- **List is implicit!**
 - One node per thread
 - Spin location changes
NUMA issues (cacheless)
- **Can we do better?**

```
typedef struct qnode {  
    struct qnode *prev;  
    int succ_blocked;  
} qnode;
```

```
qnode *lck = new qnode; // node owned by lock
```

```
void lock(qnode *lck, qnode *qn) {  
    qn->succ_blocked = 1;  
    qn->prev = FetchAndSet(lck, qn);  
    while (qn->prev->succ_blocked);  
}
```

```
void unlock(qnode **qn) {  
    qnode *pred = (*qn)->prev;  
    (*qn)->succ_blocked = 0;  
    *qn = pred;  
}
```

MCS Lock (1991)

- **Make queue explicit**
 - Acquire lock by appending to queue
 - Spin on own node until locked is reset
- **Similar advantages as CLH but**
 - Only $2N + M$ words
 - Spinning position is fixed!
Benefits cache-less NUMA
- **What are the issues?**

```

void lock(qnode *lck, qnode *qn) {
    qn->next = NULL;
    qnode *pred = FetchAndSet(lck, qn);
    if(pred != NULL) {
        qn->locked = 1;
        pred->next = qn;
        while(qn->locked);
    }
}

void unlock(qnode * lck, qnode *qn) {
    if(qn->next == NULL) { // if we're the last waiter
        if(CAS(lck, qn, NULL)) return;
        while(qn->next == NULL); // wait for pred arrival
    }
    qn->next->locked = 0; // free next waiter
    qn->next = NULL;
}
    
```

```

typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;

qnode *lck = NULL;
    
```

MCS Lock (1991)

- **Make queue explicit**
 - Acquire lock by appending to queue
 - Spin on own node until locked is reset
- **Similar advantages as CLH but**
 - Only $2N + M$ words
 - Spinning position is fixed!
Benefits cache-less NUMA
- **What are the issues?**
 - Releasing lock spins
 - More atomics!

```
void lock(qnode *lck, qnode *qn) {
    qn->next = NULL;
    qnode *pred = FetchAndSet(lck, qn);
    if(pred != NULL) {
        qn->locked = 1;
        pred->next = qn;
        while(qn->locked);
    }
}
```

```
void unlock(qnode * lck, qnode *qn) {
    if(qn->next == NULL) { // if we're the last waiter
        if(CAS(lck, qn, NULL)) return;
        while(qn->next == NULL); // wait for pred arrival
    }
    qn->next->locked = 0; // free next waiter
    qn->next = NULL;
}
```

```
typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;

qnode *lck = NULL;
```


Lessons Learned!

- **Key Lesson:**
 - Reducing memory (coherency) traffic is most important!
 - Not always straight-forward (need to reason about CL states)
- **MCS: 2006 Dijkstra Prize in distributed computing**
 - *“an outstanding paper on the principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade”*
 - *“probably the most influential practical mutual exclusion algorithm ever”*
 - *“vastly superior to all previous mutual exclusion algorithms”*
 - fast, fair, scalable → widely used, always compared against!

Time to Declare Victory?

- **Down to memory complexity of $2N+M$**
 - Probably close to optimal
- **Only local spinning**
 - Several variants with low expected contention
- **But: we assumed sequential consistency ☹️**
 - Reality causes trouble sometimes
 - Sprinkling memory fences may harm performance
 - Open research on minimally-synching algorithms!
Come and talk to me if you're interested

Fighting CPU waste: Condition Variables

- **Allow threads to yield CPU and leave the OS run queue**
 - Other threads can get them back on the queue!
- **cond_wait(cond, lock) – yield and go to sleep**
- **cond_signal(cond) – wake up sleeping threads**
- **Wait and signal are OS calls**
 - Often expensive, which one is more expensive?

Fighting CPU waste: Condition Variables

- **Allow threads to yield CPU and leave the OS run queue**
 - Other threads can get them back on the queue!
- **cond_wait(cond, lock) – yield and go to sleep**
- **cond_signal(cond) – wake up sleeping threads**
- **Wait and signal are OS calls**
 - Often expensive, which one is more expensive?
Wait, because it has to perform a full context switch

When to Spin and When to Block?

- **Spinning consumes CPU cycles but is cheap**
 - “Steals” CPU from other threads
- **Blocking has high one-time cost and is then free**
 - Often hundreds of cycles (trap, save TCB ...)
 - Wakeup is also expensive (latency)
Also cache-pollution
- **Strategy:**
 - Poll for a while and then block
But what is a “while”??

When to Spin and When to Block?

When to Spin and When to Block?

- **Optimal time depends on the future**

- When will the active thread leave the CR?
- Can compute optimal offline schedule

Q: What is the optimal offline schedule (assuming we know the future, i.e., when the lock will become available)?

When to Spin and When to Block?

- **Optimal time depends on the future**

- When will the active thread leave the CR?
- Can compute optimal offline schedule

Q: What is the optimal offline schedule (assuming we know the future, i.e., when the lock will become available)?

- Actual problem is an online problem

- **Competitive algorithms**

- An algorithm is c -competitive if for a sequence of actions x and a constant a holds:

$$C(x) \leq c * C_{opt}(x) + a$$

- What would a good spinning algorithm look like and what is the competitiveness?

Competitive Spinning

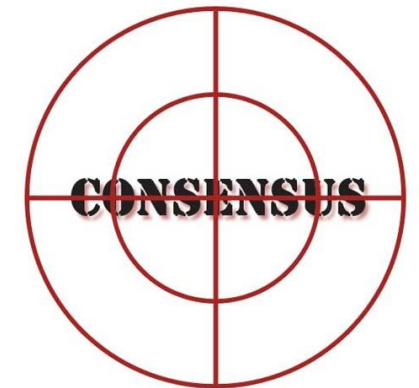
- If T is the overhead to process a wait, then a locking algorithm that spins for time T before it blocks is 2-competitive!
 - Karlin, Manasse, McGeoch, Owicki: “Competitive Randomized Algorithms for Non-Uniform Problems”, SODA 1989
- If randomized algorithms are used, then $e/(e-1)$ -competitiveness (~ 1.58) can be achieved
 - See paper above!

Remember: lock-free vs. wait-free

- **A lock-free method**
 - guarantees that infinitely often **some** method call finishes in a finite number of steps
- **A wait-free method**
 - guarantees that **each** method call finishes in a finite number of steps (implies lock-free)
- **Synchronization instructions are not equally powerful!**
 - Indeed, they form an infinite hierarchy; no instruction (primitive) in level x can be used for lock-/wait-free implementations of primitives in level $z > x$.

Concept: Consensus Number

- Each level of the hierarchy has a “consensus number” assigned.
 - Is the maximum number of threads for which primitives in level x can solve the consensus problem
- **The consensus problem:**
 - Has single function: $\text{decide}(v)$
 - Each thread calls it at most once, the function returns a value that meets two conditions:
 - consistency: all threads get the same value*
 - validity: the value is some thread's input*
 - Simplification: binary consensus (inputs in $\{0,1\}$)



Understanding Consensus

- **Can a particular class solve n-thread consensus wait-free?**
 - A class C solves n-thread consensus if there exists a consensus protocol using **any number** of objects of class C and **any number** of atomic registers
 - The protocol has to be wait-free (bounded number of steps per thread)
 - The consensus number of a class C is the largest n for which that class solves n-thread consensus (may be infinite)
 - Assume we have a class D whose objects can be constructed from objects out of class C. If class C has consensus number n, what does class D have?

Starting simple ...

- **Binary consensus with two threads (A, B)!**
 - Each thread moves until it decides on a value
 - May update shared objects
 - Protocol state = state of threads + state of shared objects
 - Initial state = state before any thread moved
 - Final state = state after all threads finished
 - States form a tree, wait-free property guarantees a finite tree

Example with two threads and two moves each!

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
 - I.e., they cannot be used to solve even two-thread consensus! Really?

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
 - I.e., they cannot be used to solve even two-thread consensus! Really?
- **Proof outline:**
 - Assume arbitrary consensus protocol, thread A, B
 - Run until it reaches critical state where next action determines outcome (show that it must have a critical state first)
 - Show all options using atomic registers and show that they cannot be used to determine one outcome for all possible executions!
 - 1) *Any thread reads (other thread runs solo until end)*
 - 2) *Threads write to different registers (order doesn't matter)*
 - 3) *Threads write to same register (solo thread can start after each write)*

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
- **Corollary: It is impossible to construct a wait-free implementation of any object with consensus number of >1 using atomic registers**
 - “perhaps one of the most striking impossibility results in Computer Science” (Herlihy, Shavit)
 - → We need hardware atomics or Transactional Memory!
- **Proof technique borrowed from:**

[Impossibility of distributed consensus with one ... - ACM Digita...](#)

dl.acm.org/citation.cfm?id=214121 ▼

by MJ Fischer - 1985 - [Cited by 4189](#) - [Related articles](#)

Apr 1, 1985 - The **consensus** problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable ...

- **Very influential paper, always worth a read!**
 - Nicely shows proof techniques that are central to parallel and distributed computing!

Other Atomic Operations

- **Simple RMW operations (Test&Set, Fetch&Op, Swap, basically all functions where the op commutes or overwrites) have consensus number 2!**
 - Similar proof technique (bivalence argument)
- **CAS and TM have consensus number ∞**
 - Constructive proof!

Compare and Set/Swap Consensus

```
const int first = -1
volatile int thread = -1;
int proposed[n];

int decide(v) {
    proposed[tid] = v;
    if(CAS(thread, first, tid))
        return v; // I won!
    else
        return proposed[thread]; // thread won
}
```



- **CAS provides an infinite consensus number**
 - Machines providing CAS are **asynchronous** computation equivalents of the Turing Machine
 - I.e., any concurrent object can be implemented in a wait-free manner (not necessarily fast!)

Now you know everything 😊

- **Not really ... ;-)**
 - We'll argue more about **performance** now!
- **But you have all the tools for:**
 - Efficient locks
 - Efficient lock-based algorithms
 - Efficient lock-free algorithms (or even wait-free)
 - Reasoning about parallelism!
- **What now?**
 - A different class of problems
 - Impact on wait-free/lock-free on actual performance is not well understood*
 - Relevant to HPC, applies to shared and distributed memory
 - *Group communications*

Case study: Fast Large-scale Locking in Practice

Case study: Fast Large-scale Locking in Practice

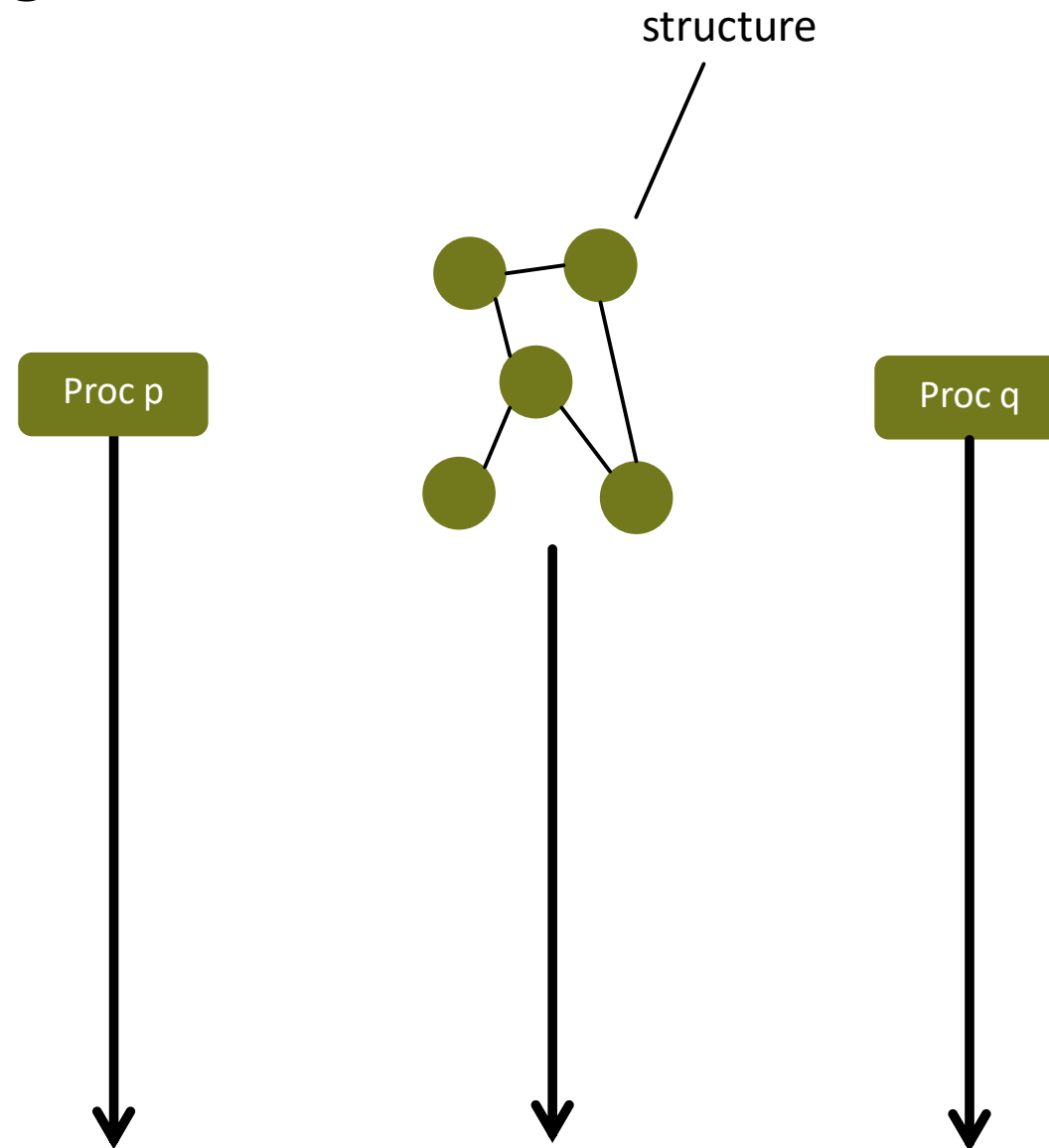
Proc p



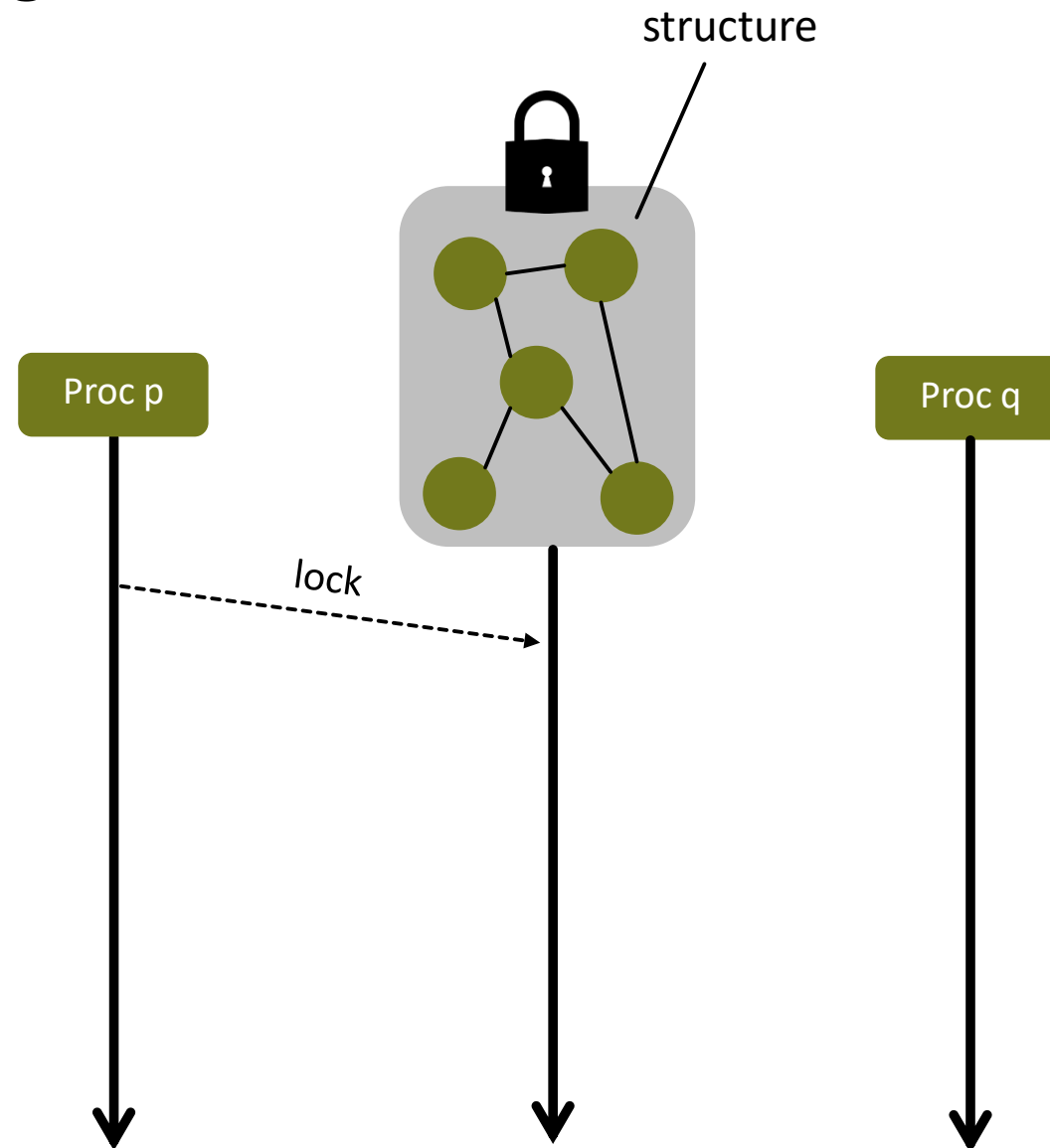
Proc q



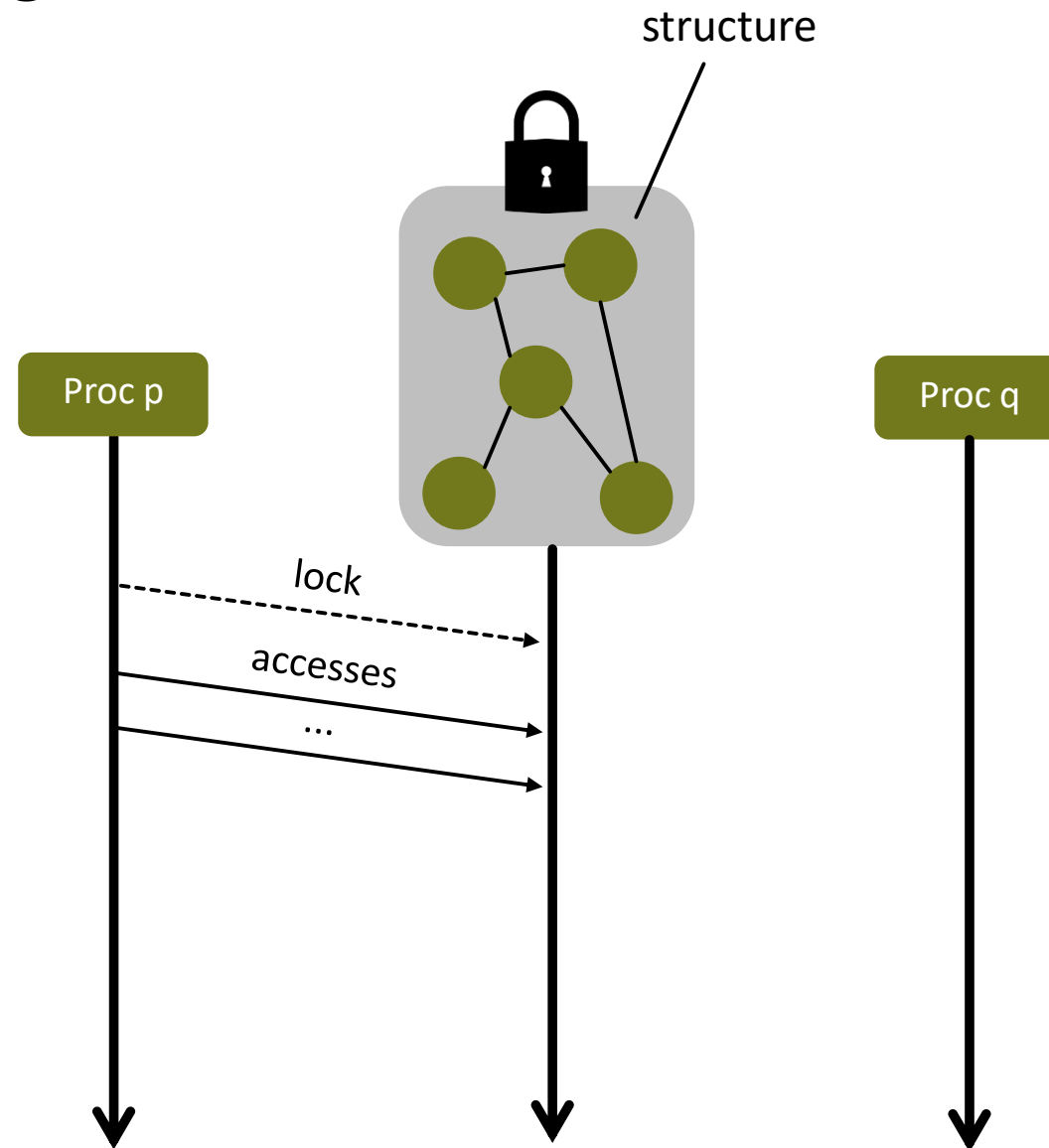
Case study: Fast Large-scale Locking in Practice



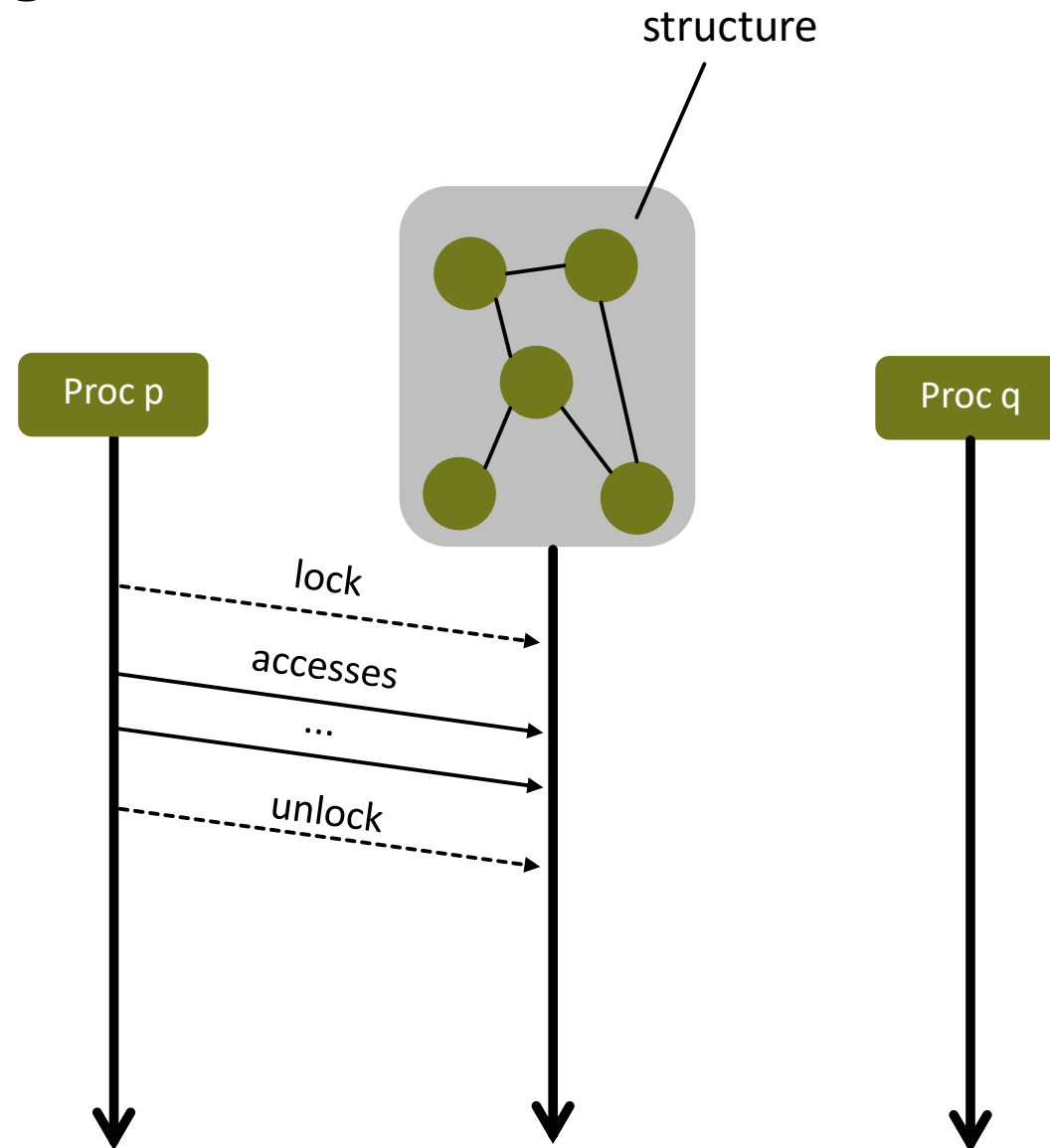
Case study: Fast Large-scale Locking in Practice



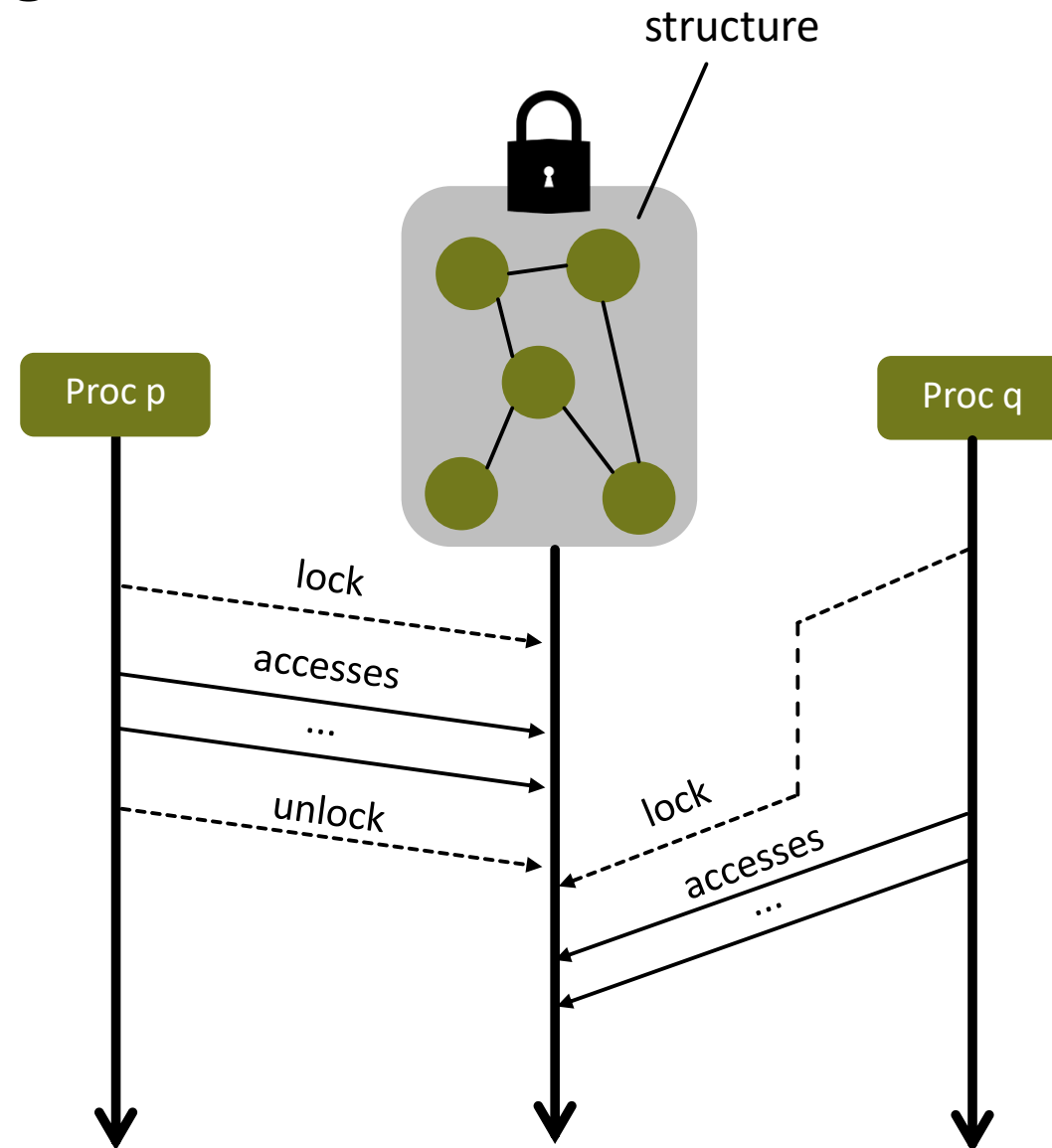
Case study: Fast Large-scale Locking in Practice



Case study: Fast Large-scale Locking in Practice

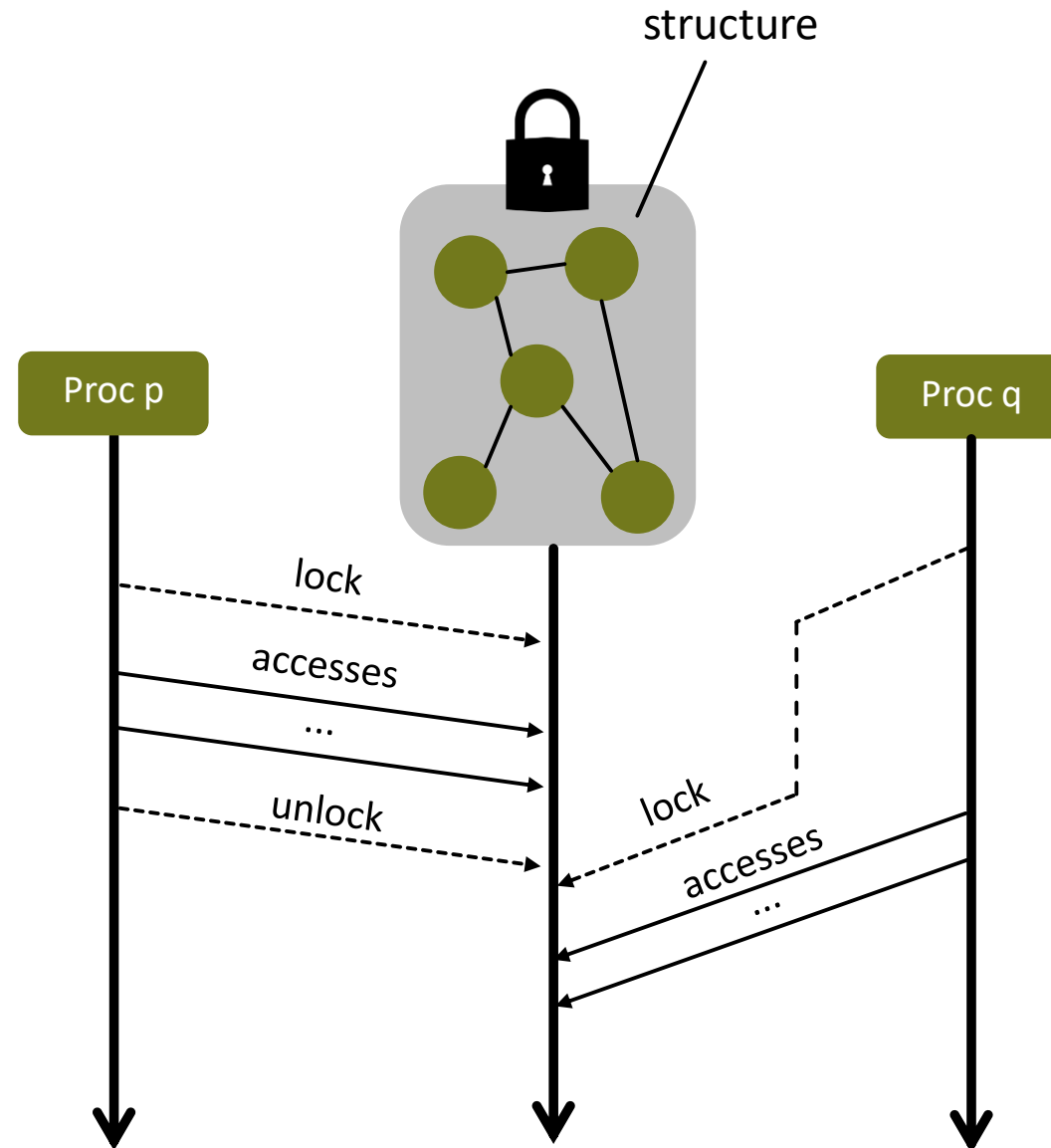


Case study: Fast Large-scale Locking in Practice



Case study: Fast Large-scale Locking in Practice

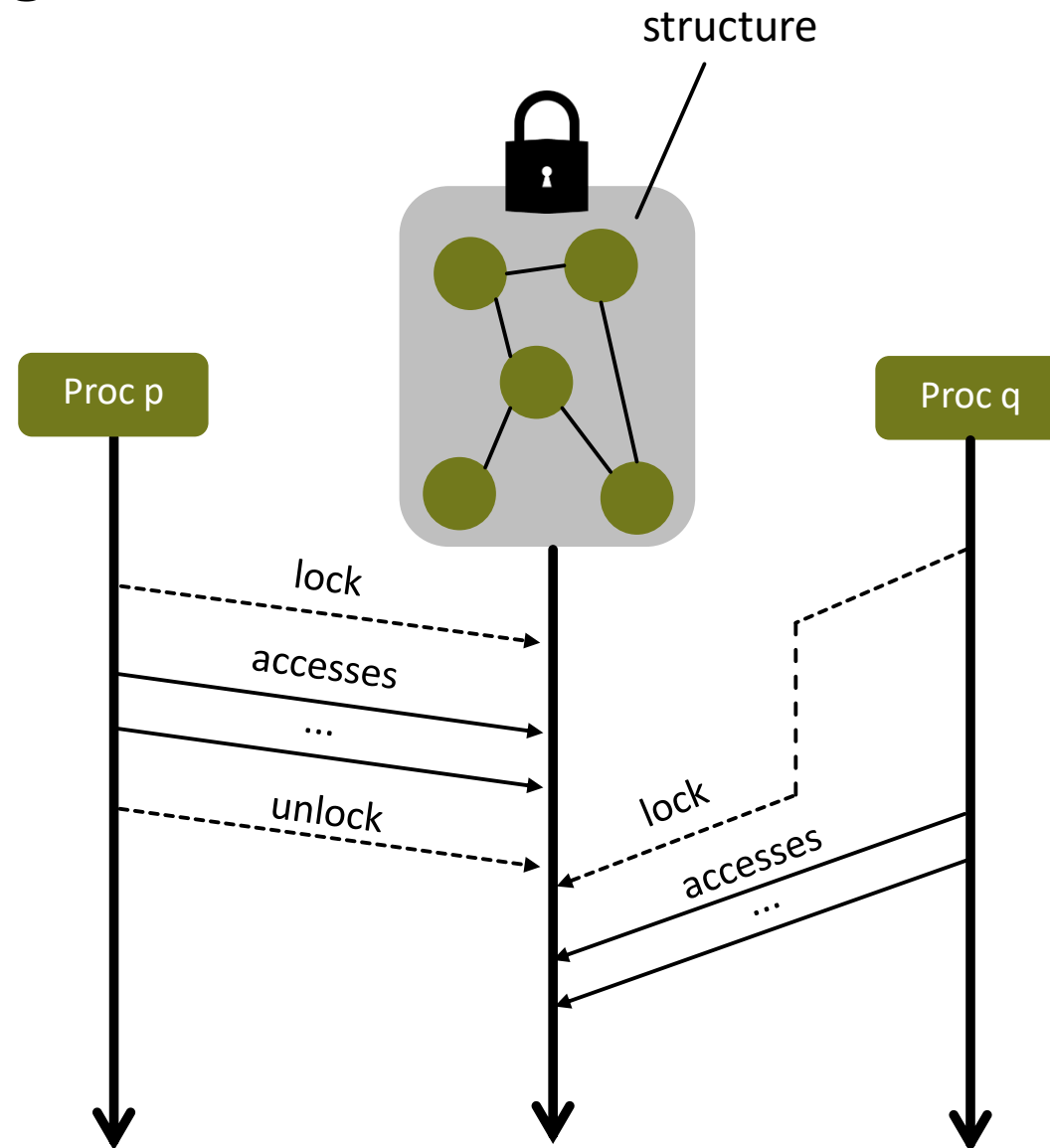
 Inuitive semantics



Case study: Fast Large-scale Locking in Practice

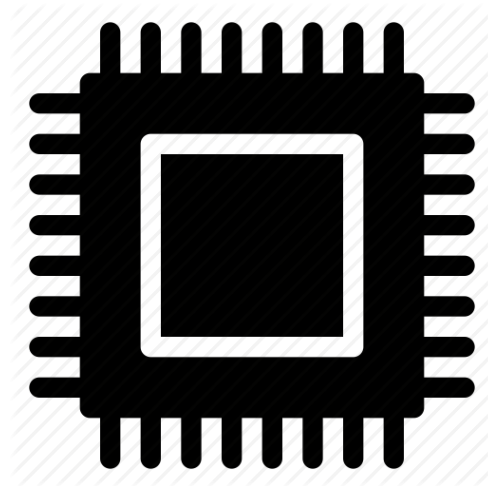
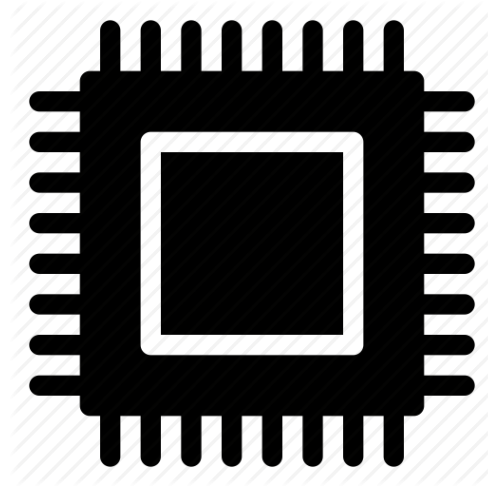
✓ Inuitive semantics

✗ Various performance penalties

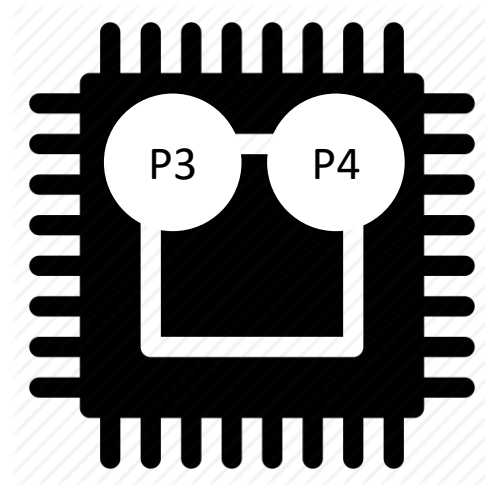
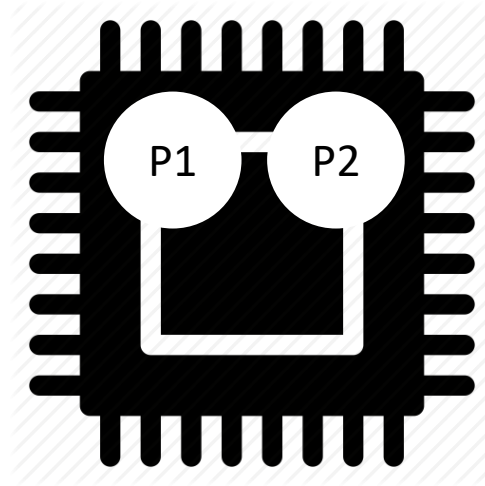


Locks: Challenges

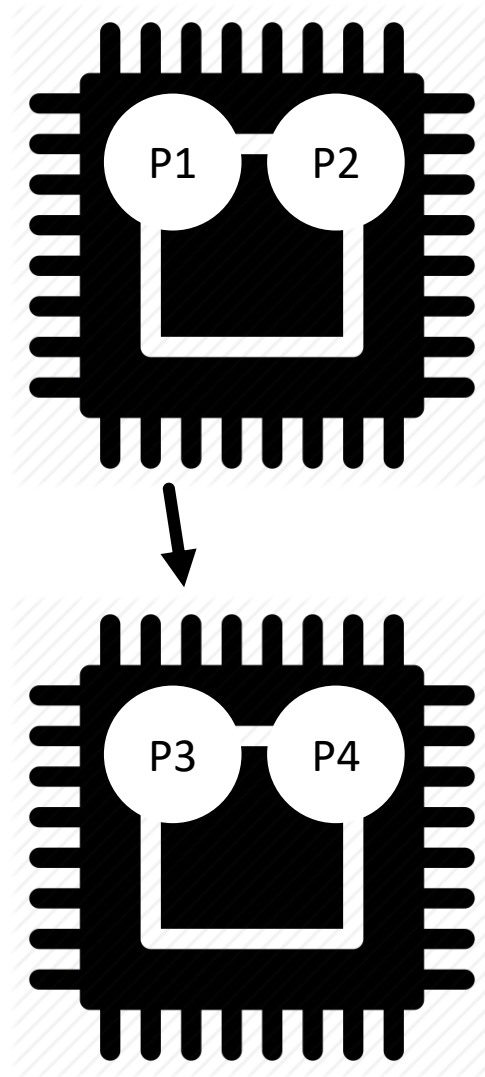
Locks: Challenges



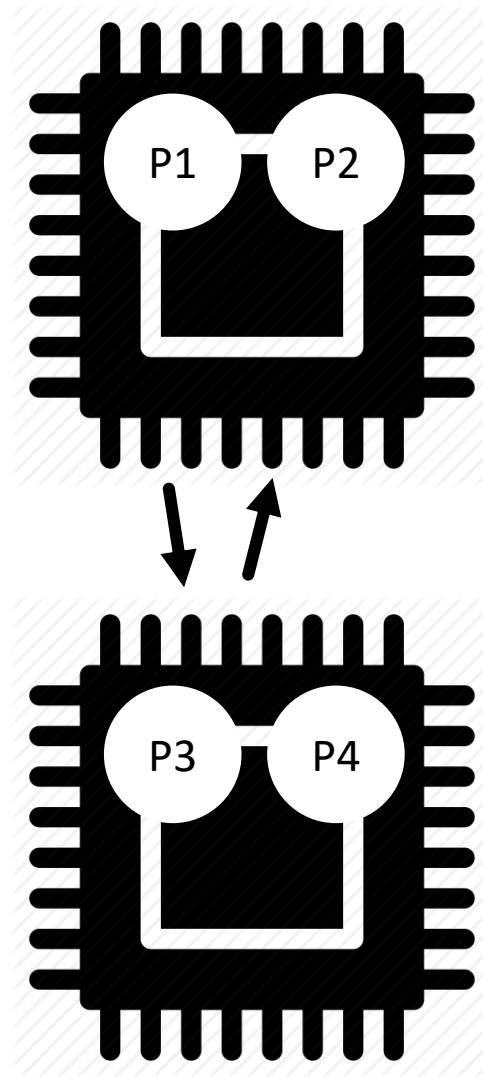
Locks: Challenges



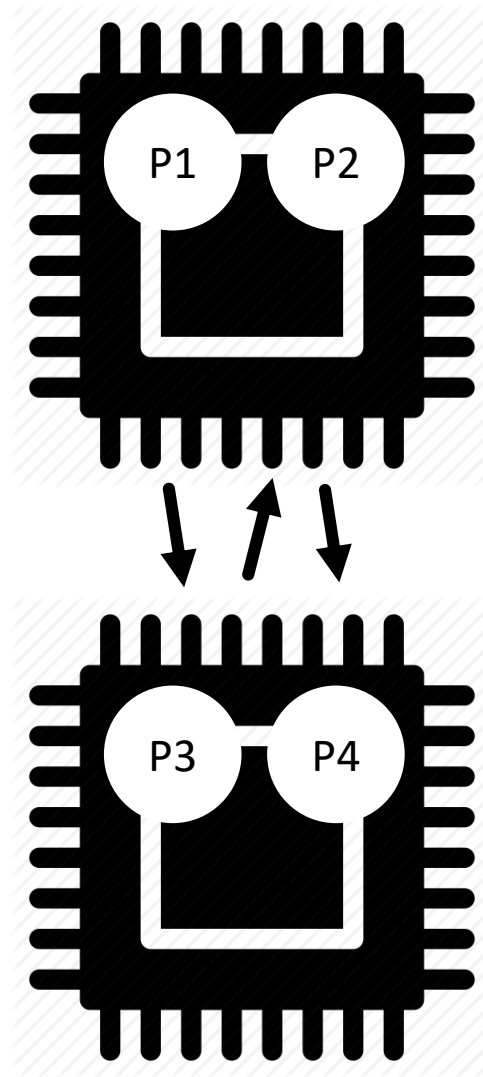
Locks: Challenges



Locks: Challenges



Locks: Challenges



Locks: Challenges



Locks: Challenges



Locks: Challenges



We need intra- and inter-node topology-awareness



We need to cover arbitrary topologies



Locks: Challenges

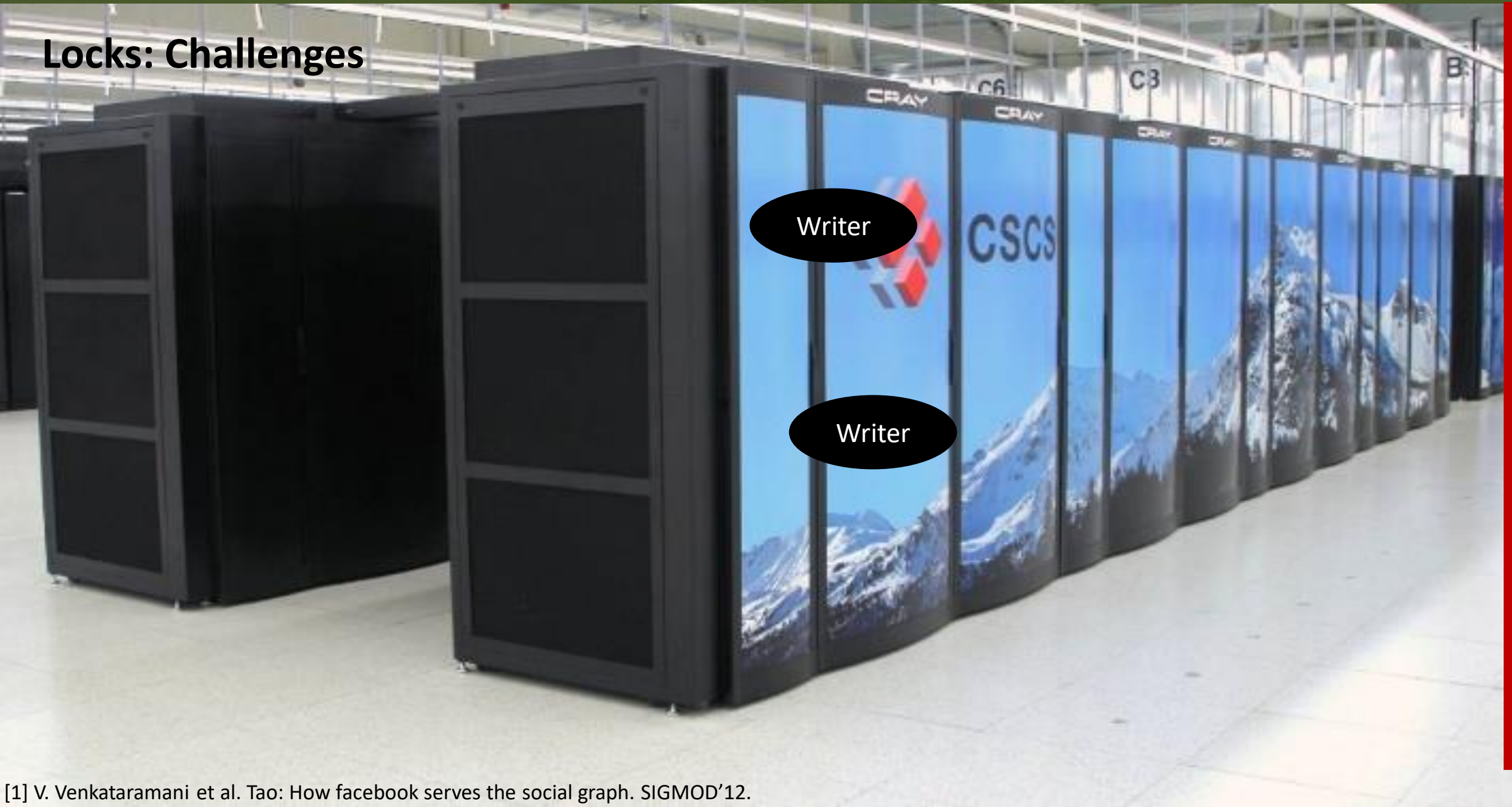
Locks: Challenges



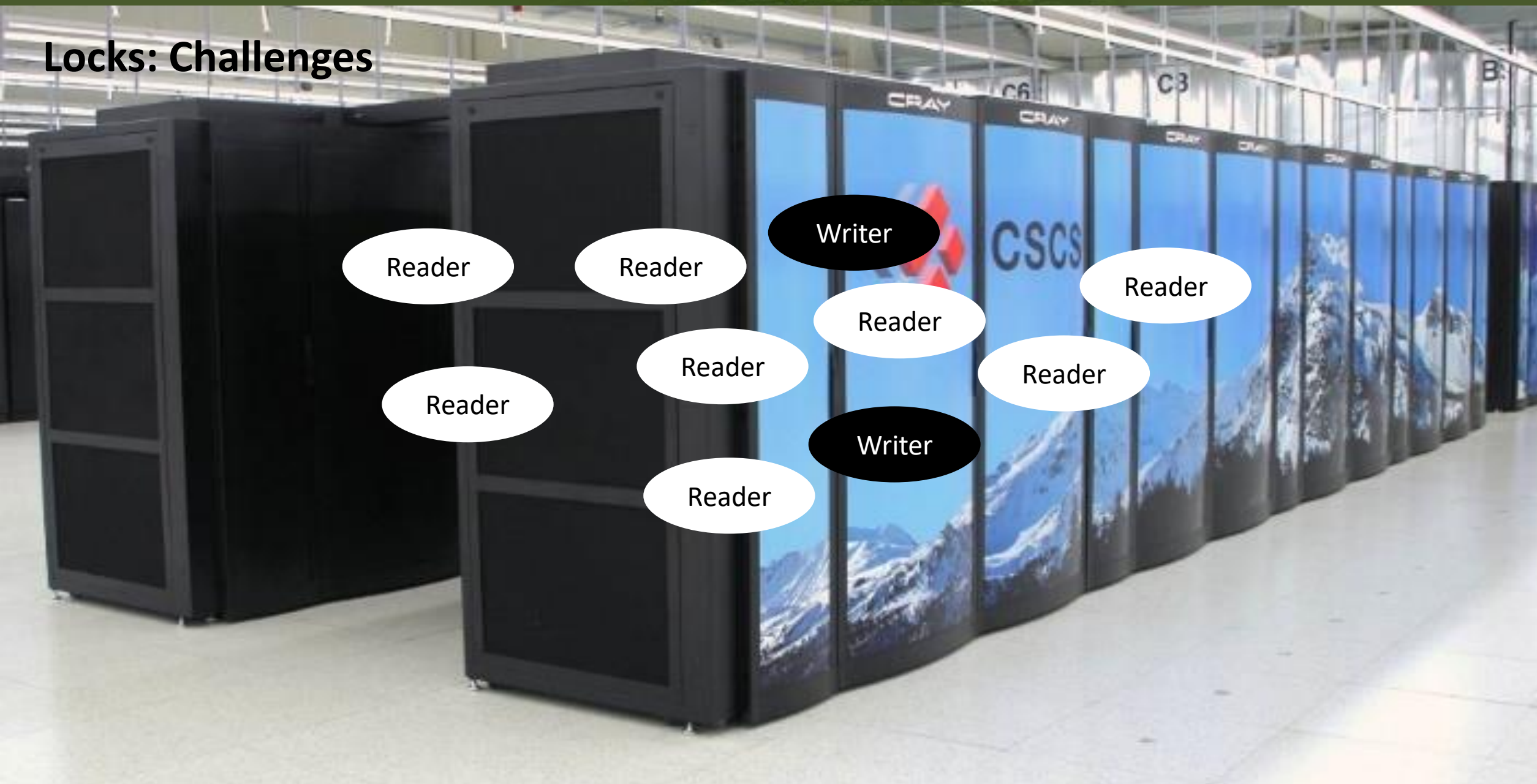
Locks: Challenges



Locks: Challenges



Locks: Challenges



[1] V. Venkataramani et al. Tao: How facebook serves the social graph. SIGMOD'12.

Locks: Challenges



We need to distinguish between readers and writers

Reader

Reader

Writer

Reader

Reader

Locks: Challenges



We need to distinguish between readers and writers

Reader

Reader

Reader

Writer

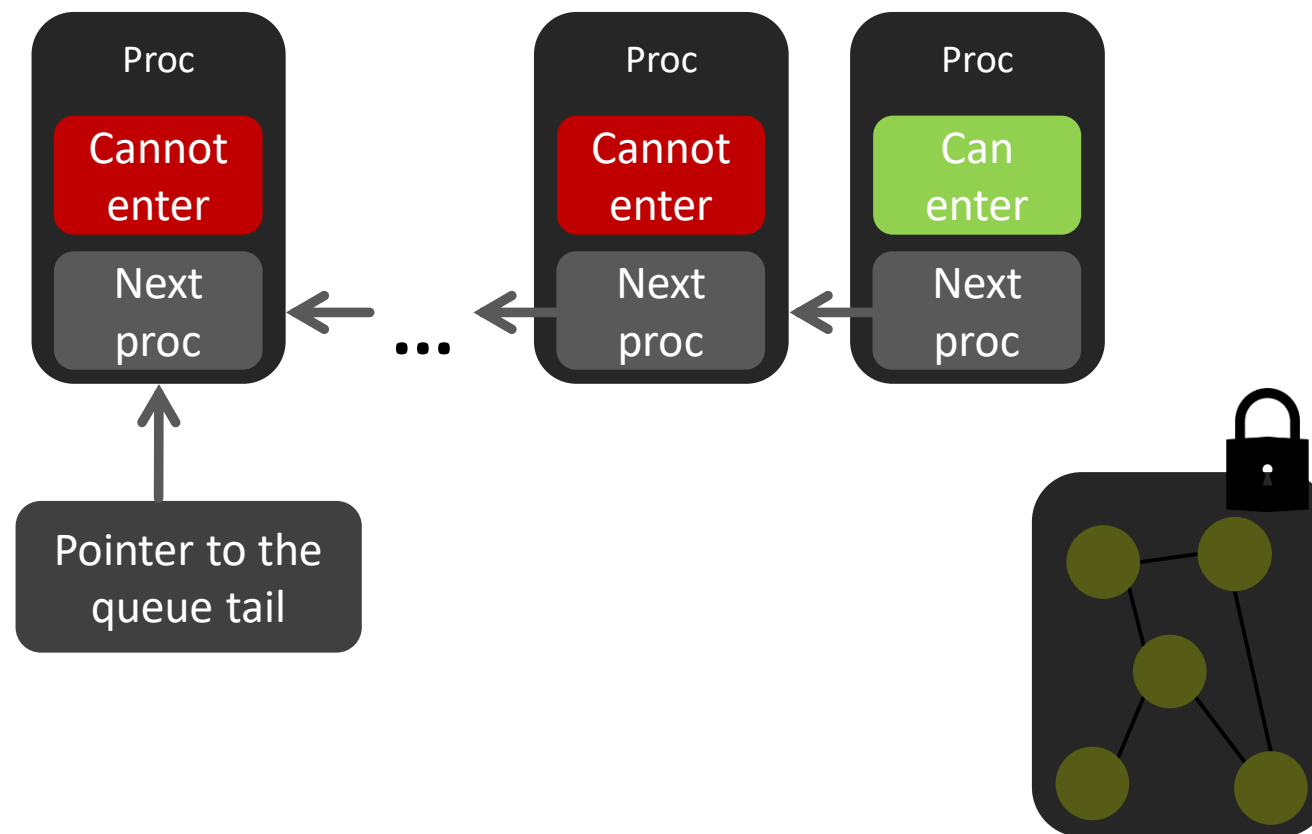


We need flexible performance for both types of processes

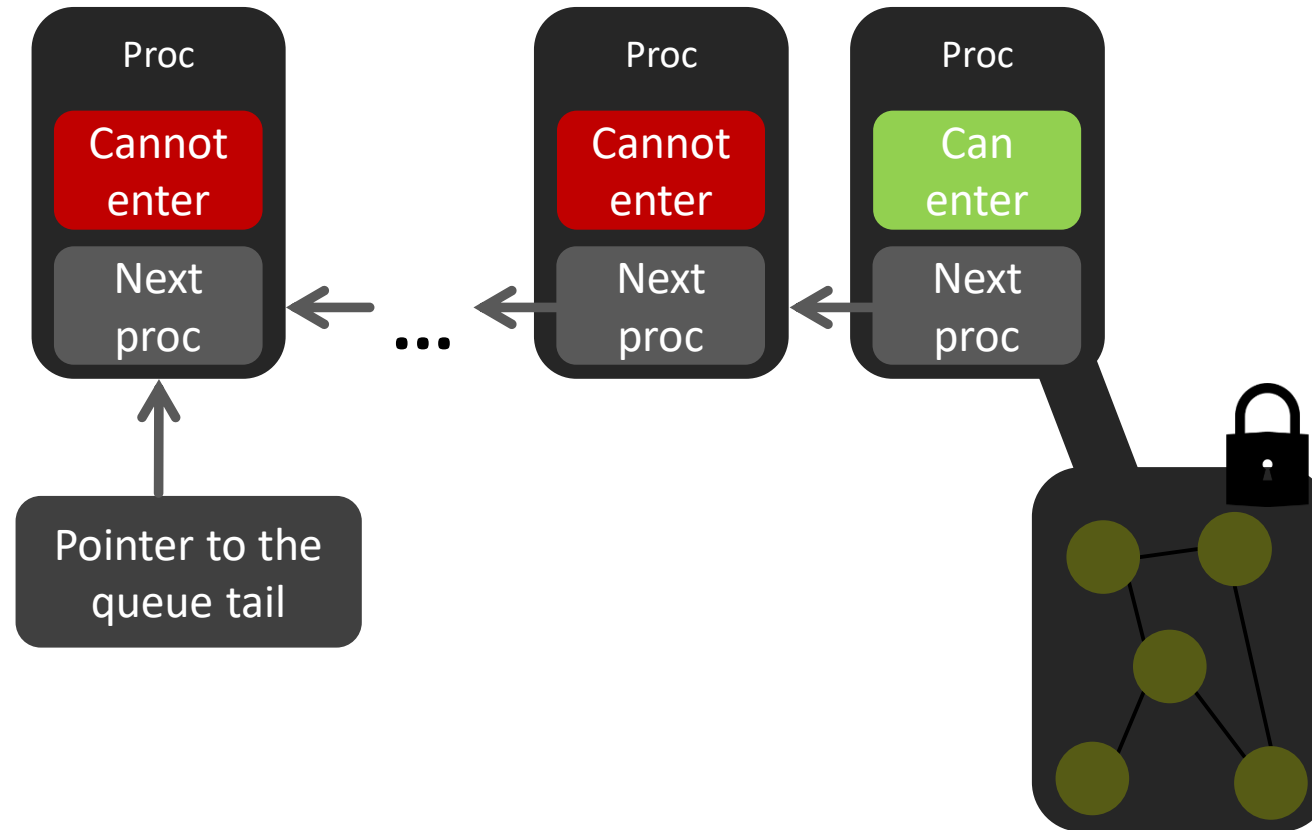


What will we use in the
design?

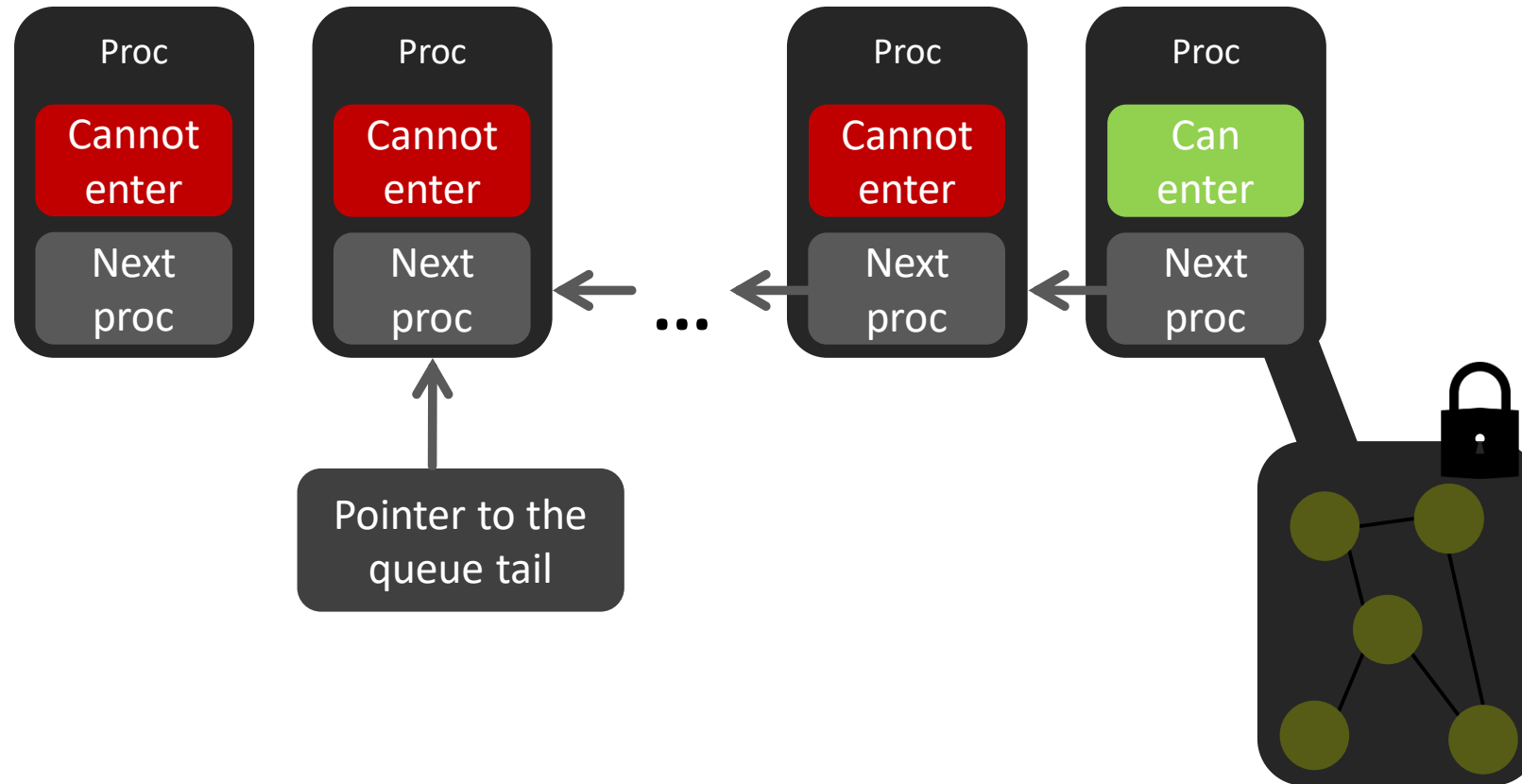
Ingredient 1 - MCS Locks



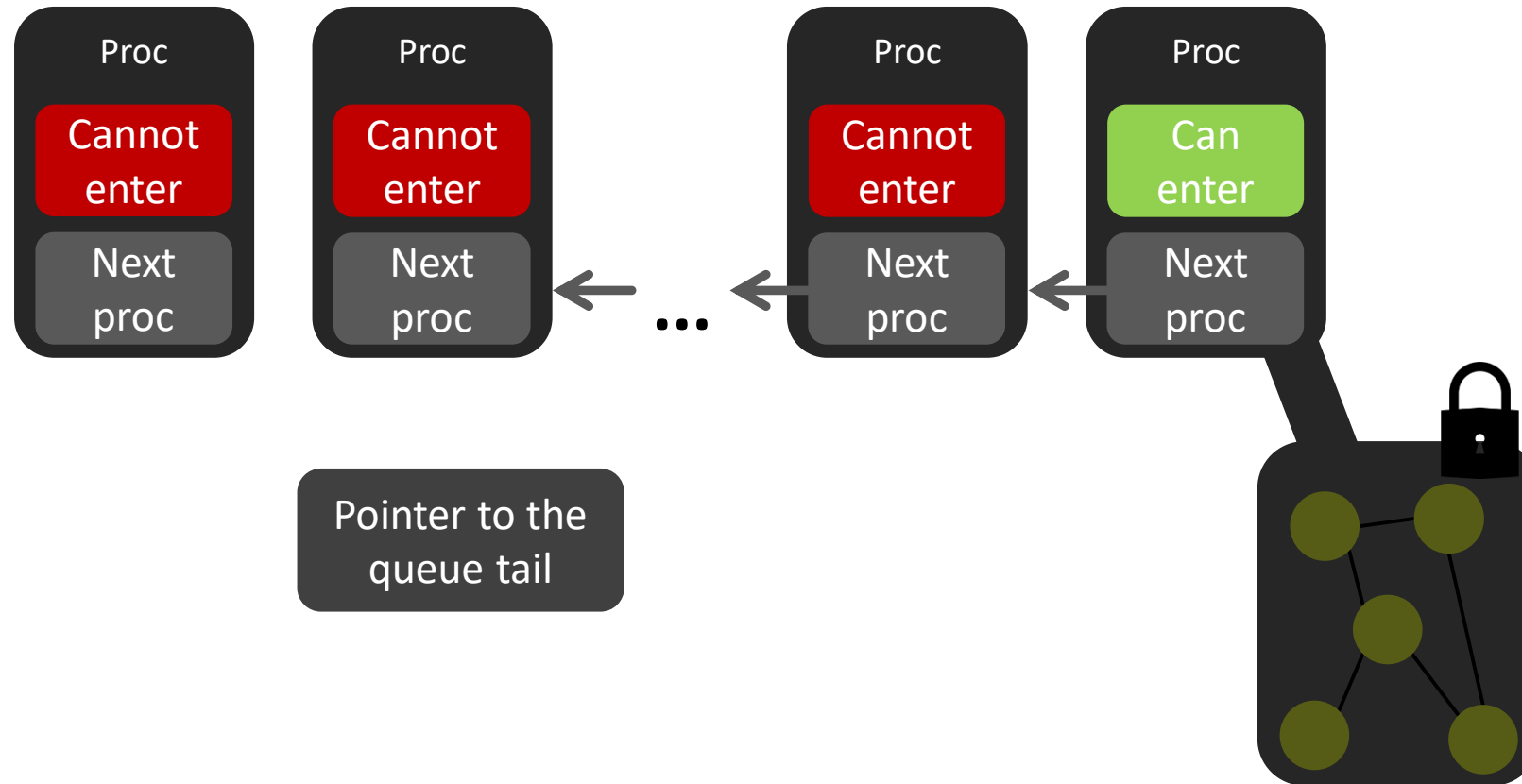
Ingredient 1 - MCS Locks



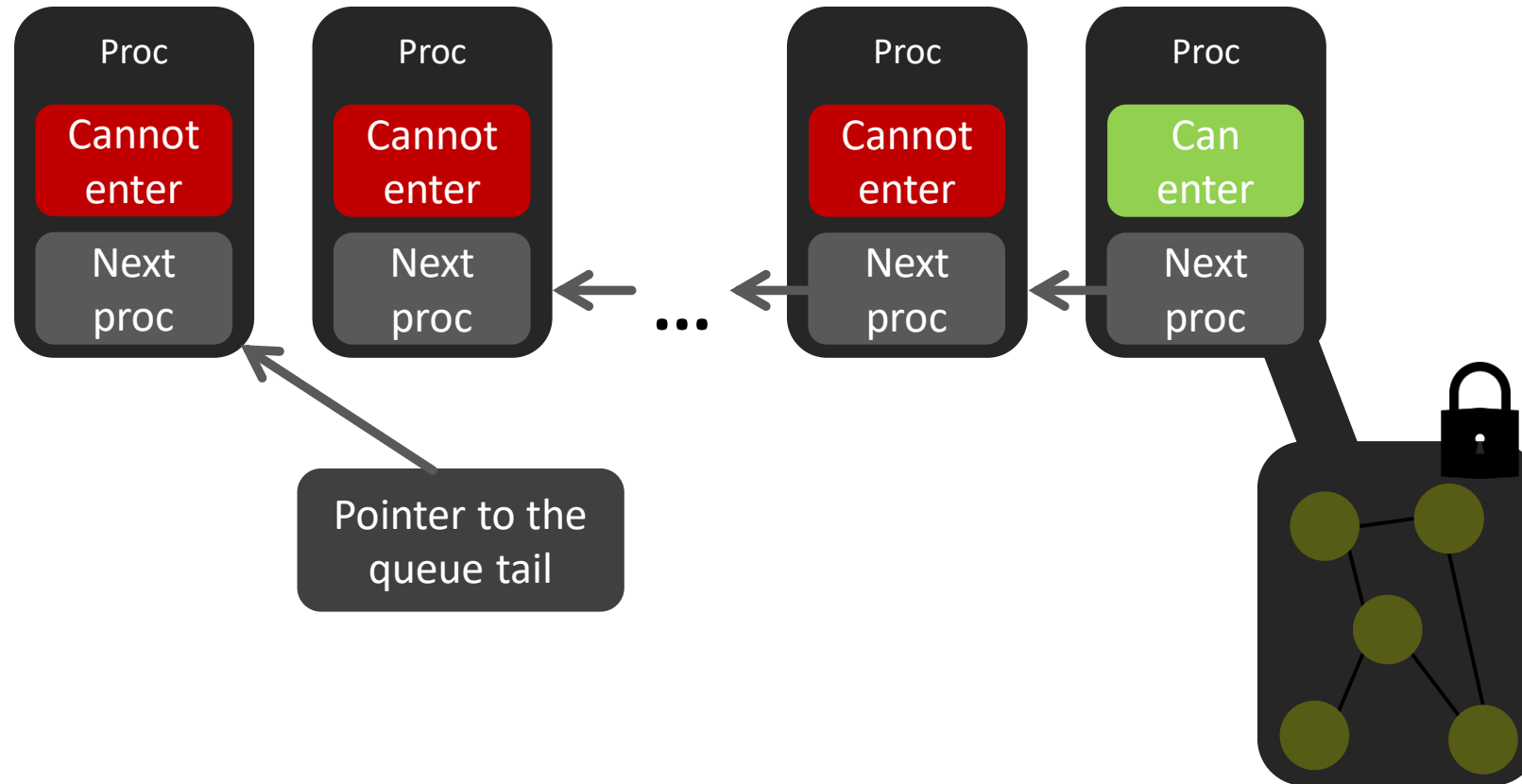
Ingredient 1 - MCS Locks



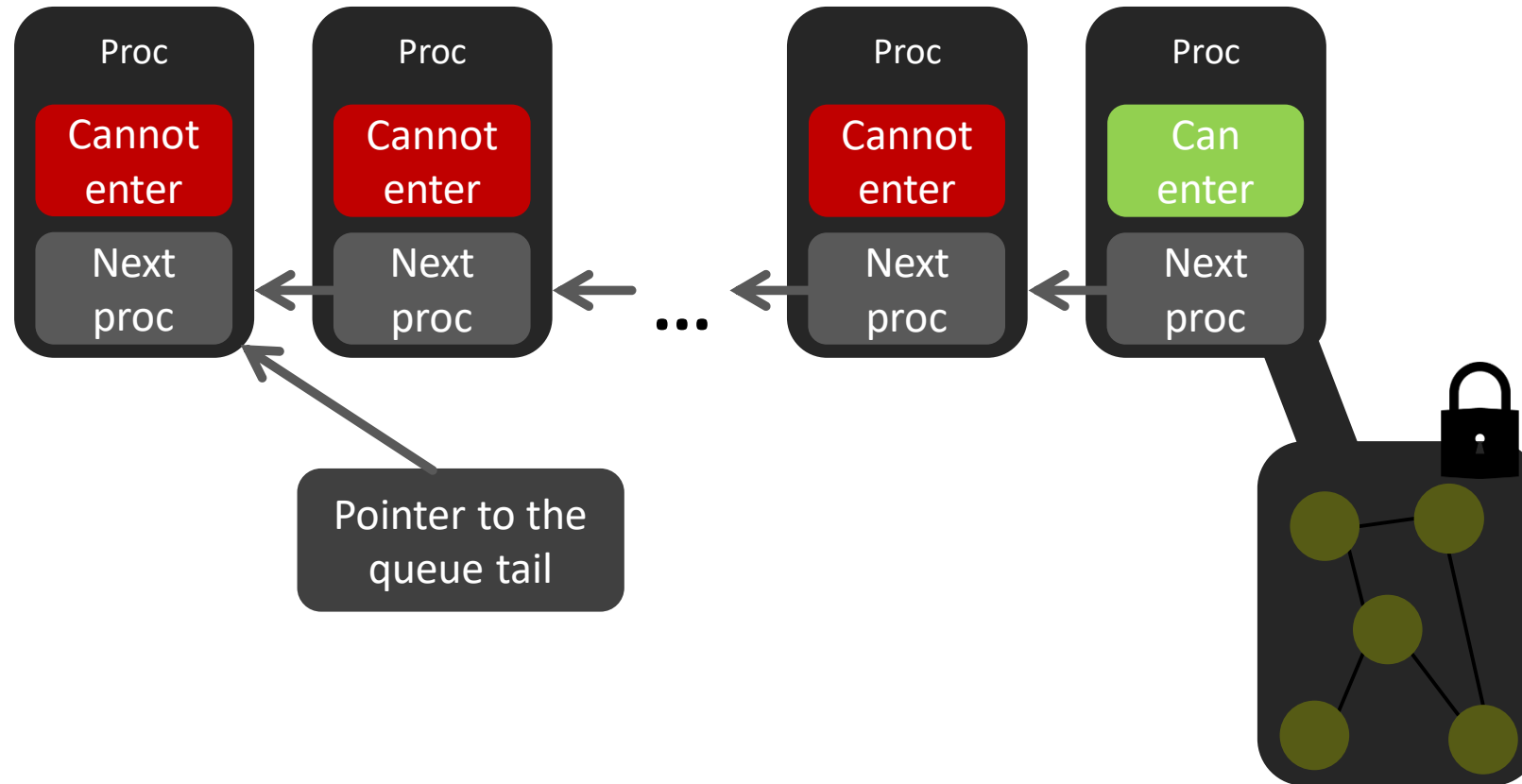
Ingredient 1 - MCS Locks



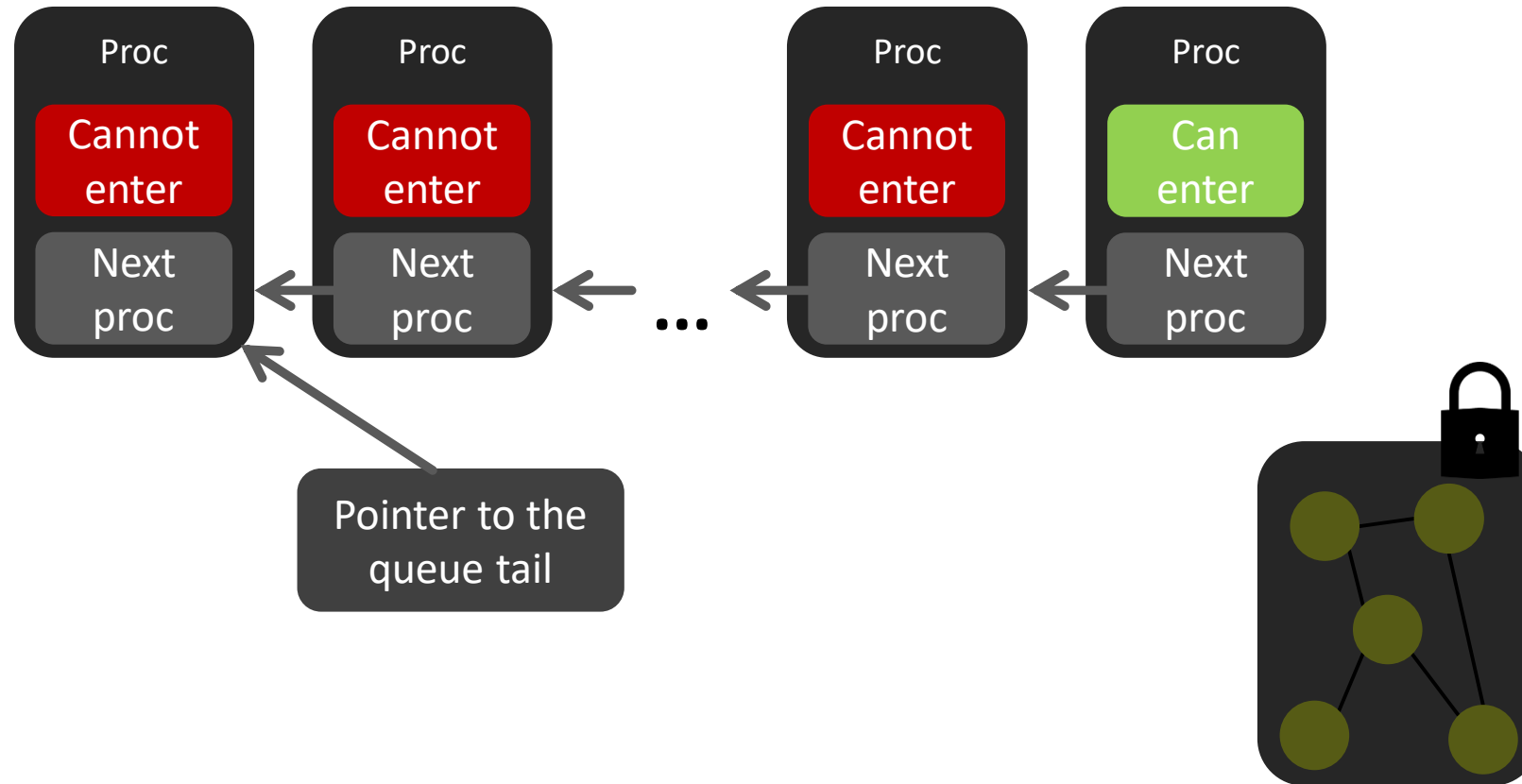
Ingredient 1 - MCS Locks



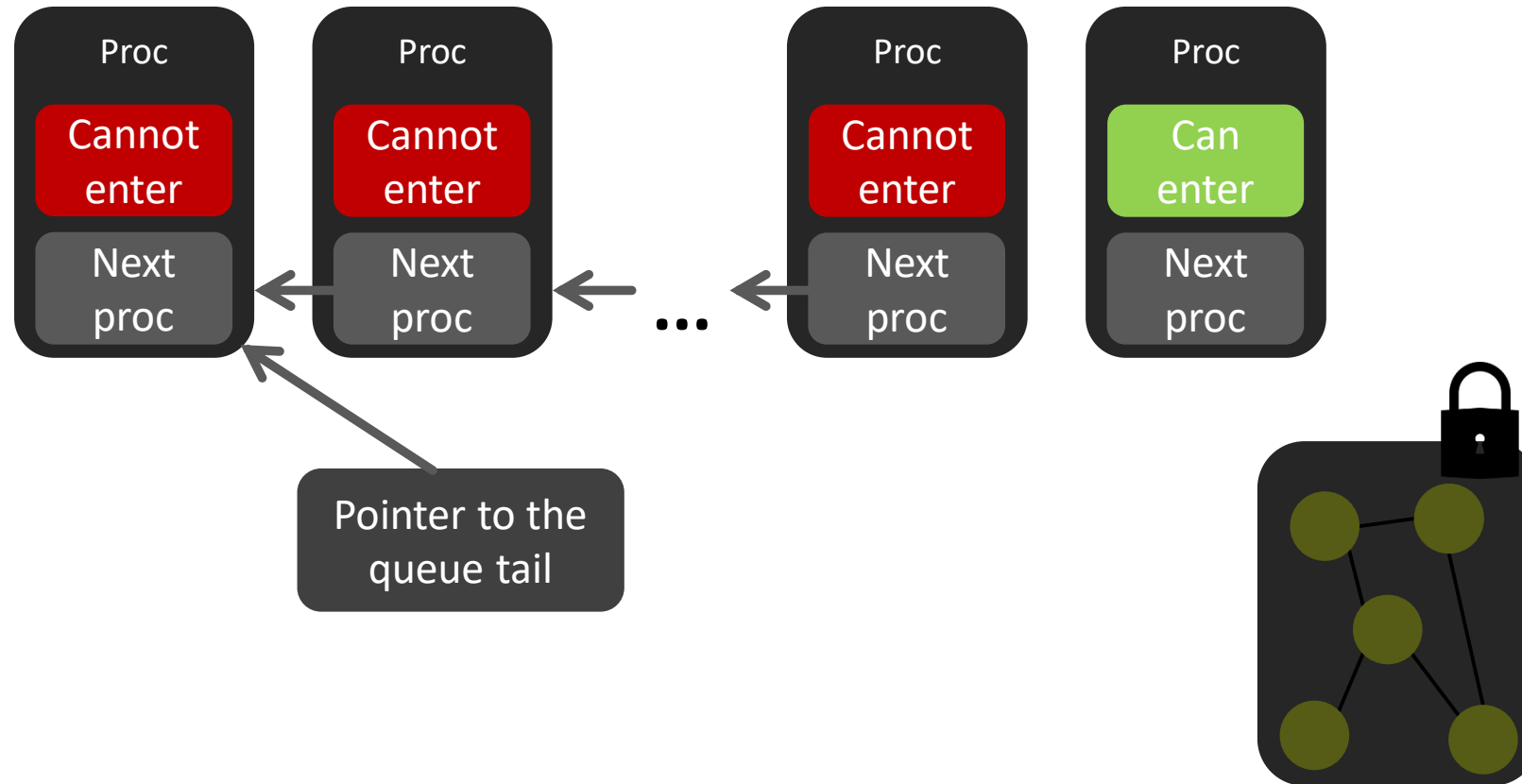
Ingredient 1 - MCS Locks



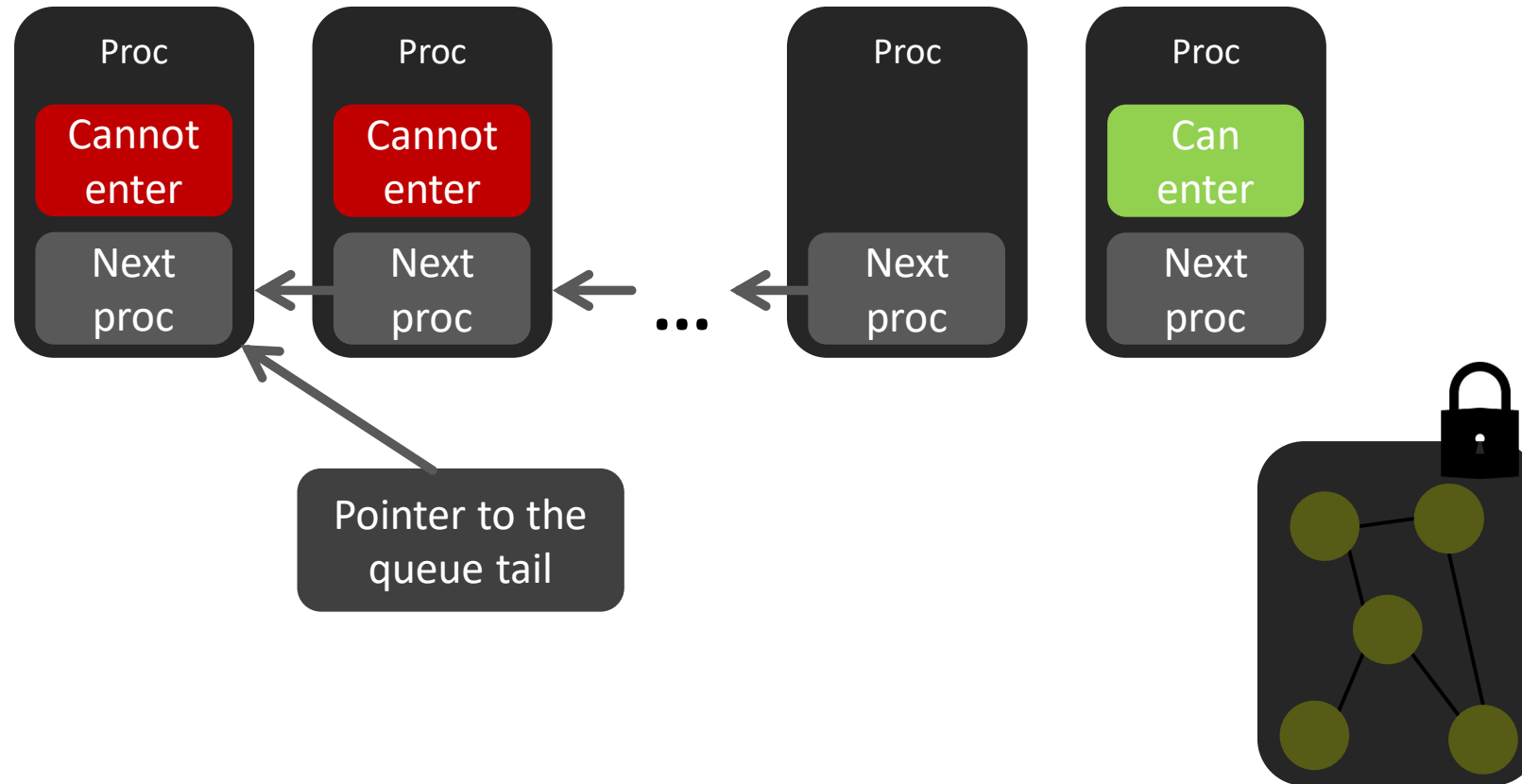
Ingredient 1 - MCS Locks



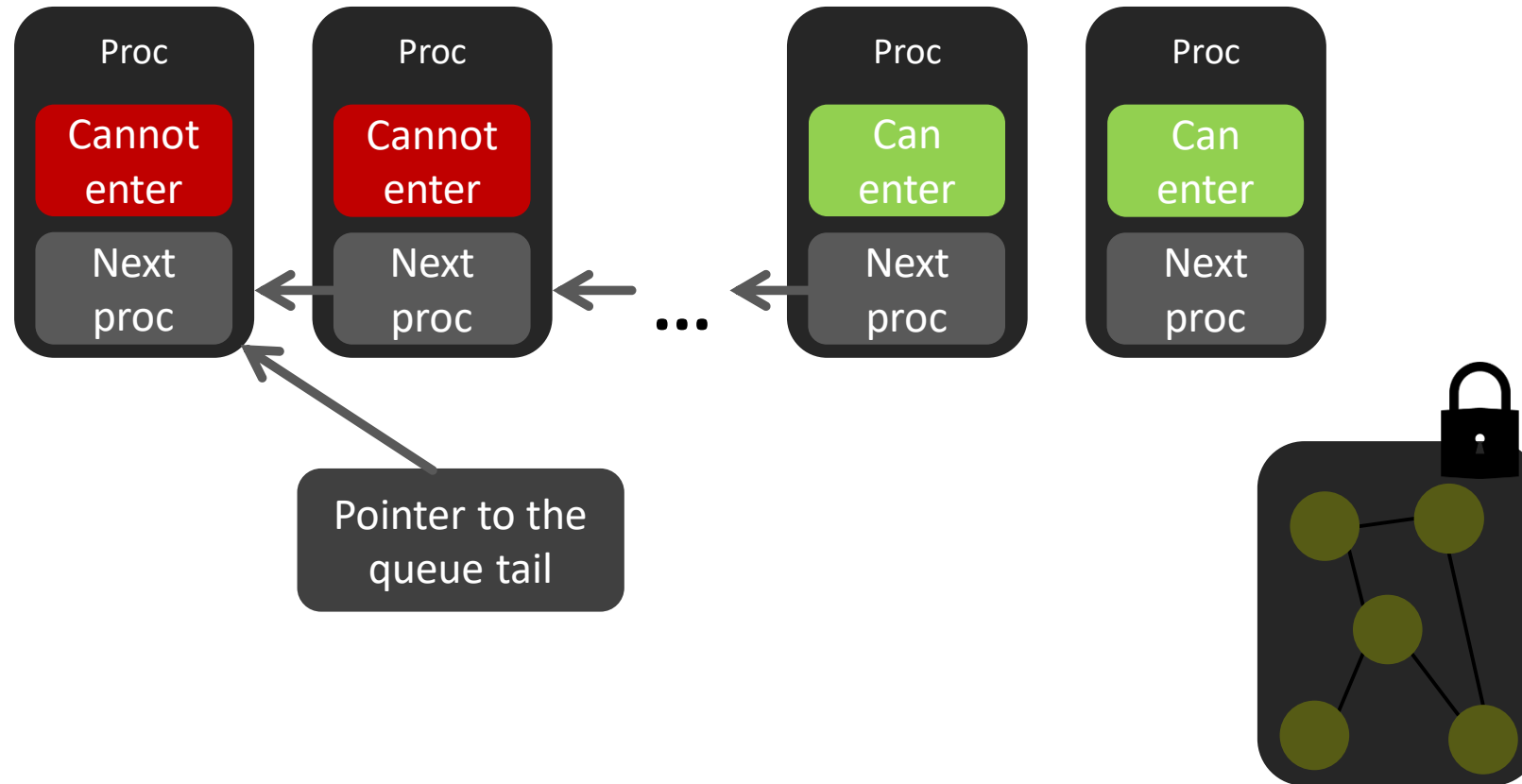
Ingredient 1 - MCS Locks



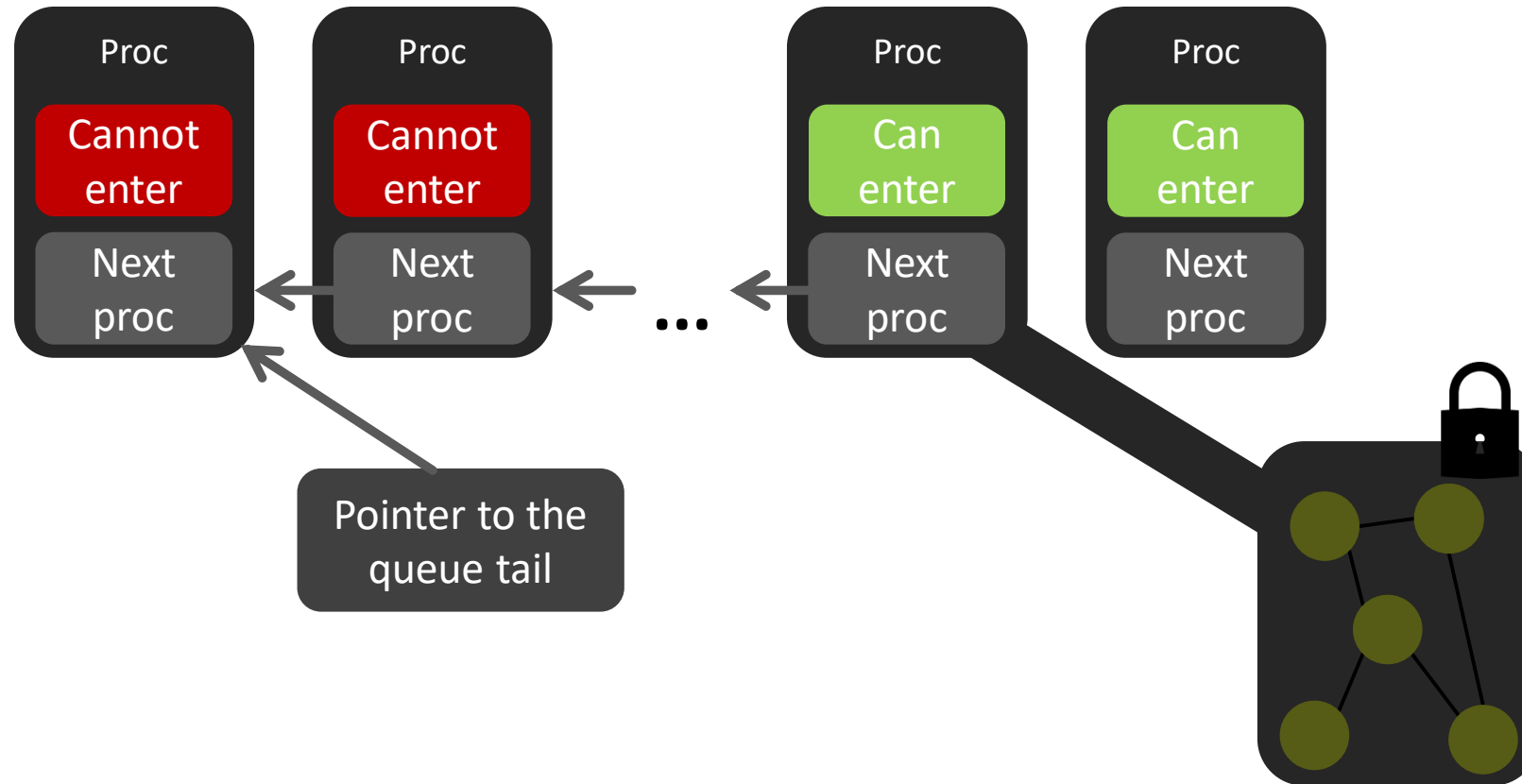
Ingredient 1 - MCS Locks



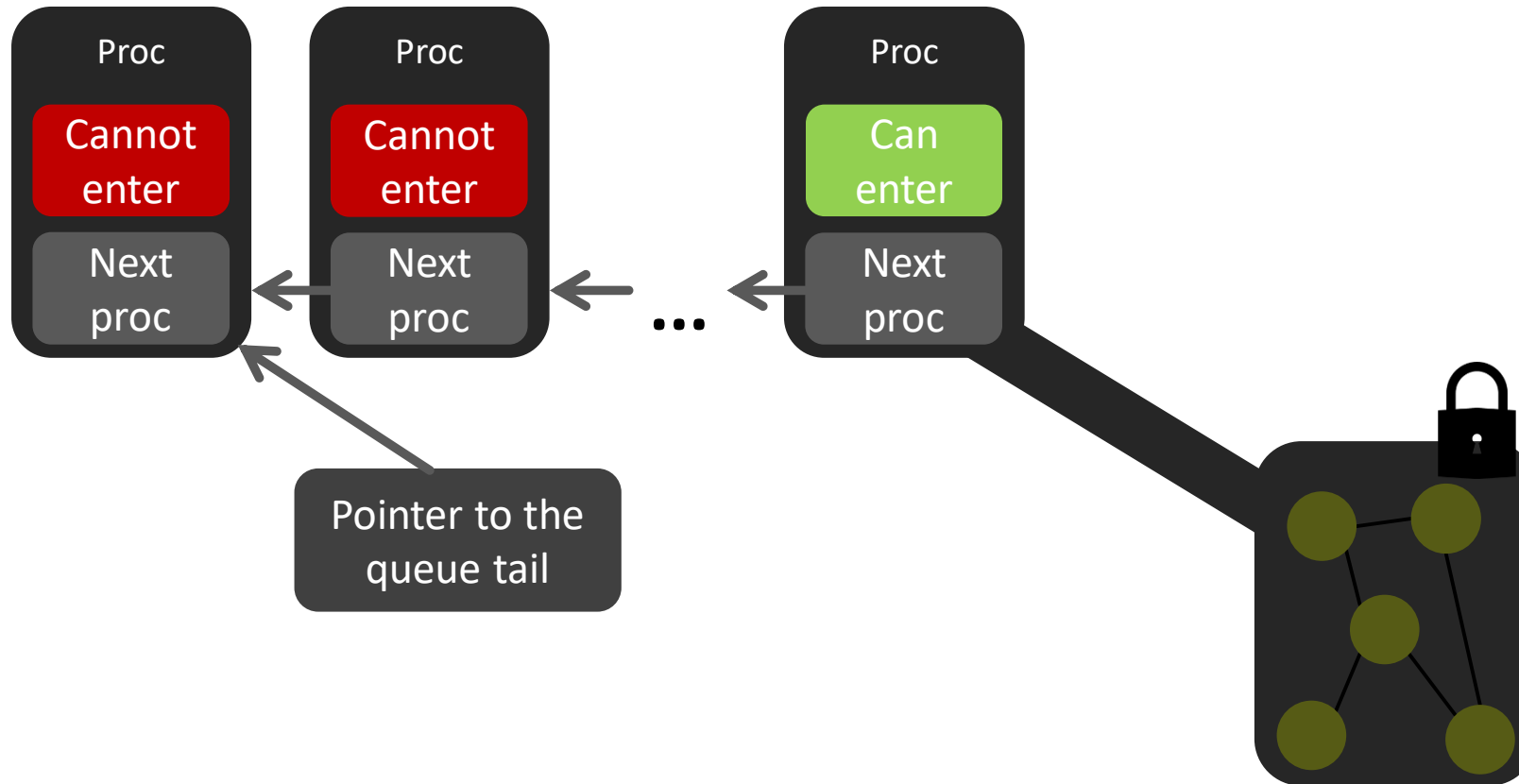
Ingredient 1 - MCS Locks



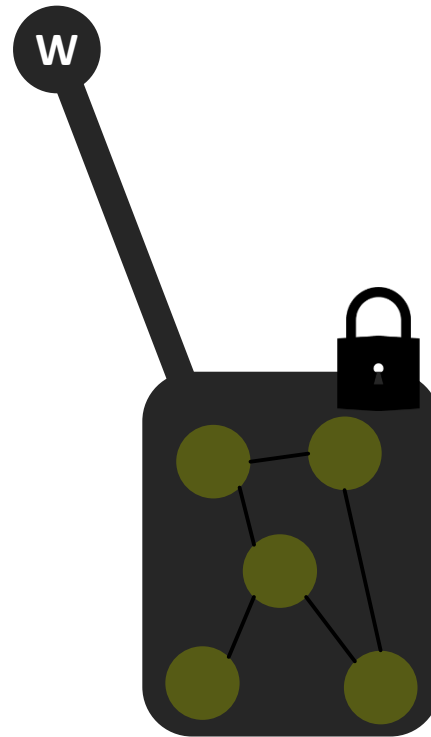
Ingredient 1 - MCS Locks



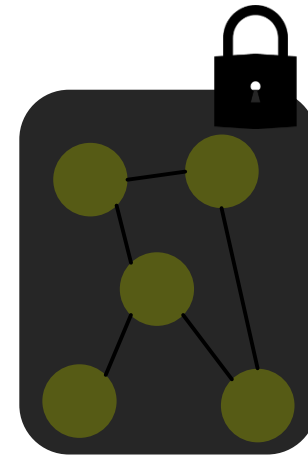
Ingredient 1 - MCS Locks



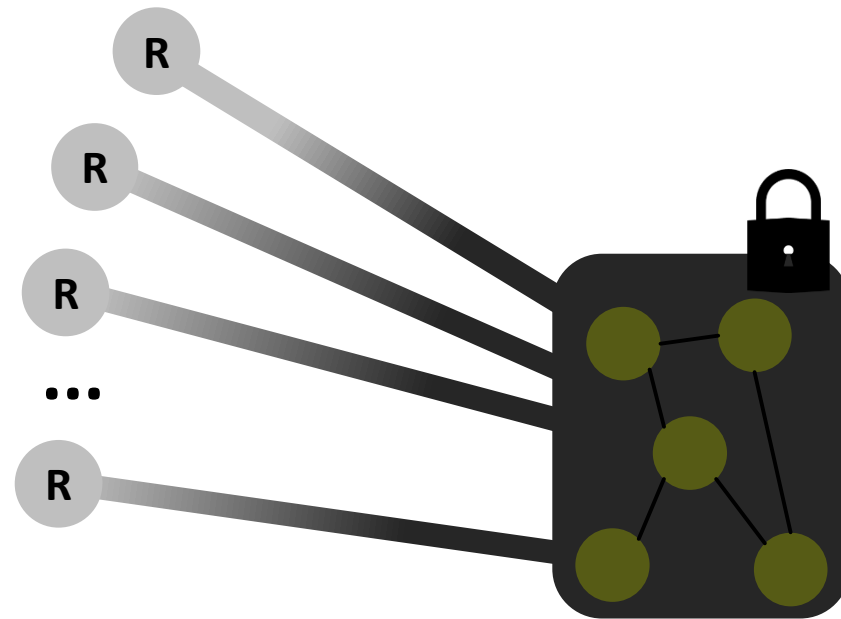
Ingredient 2 - Reader-Writer Locks



Ingredient 2 - Reader-Writer Locks



Ingredient 2 - Reader-Writer Locks





How to manage the design
complexity?



How to manage the design complexity?



What mechanism to use for efficient implementation?



How to manage the design complexity?



How to ensure tunable performance?



What mechanism to use for efficient implementation?



How to manage the design complexity?



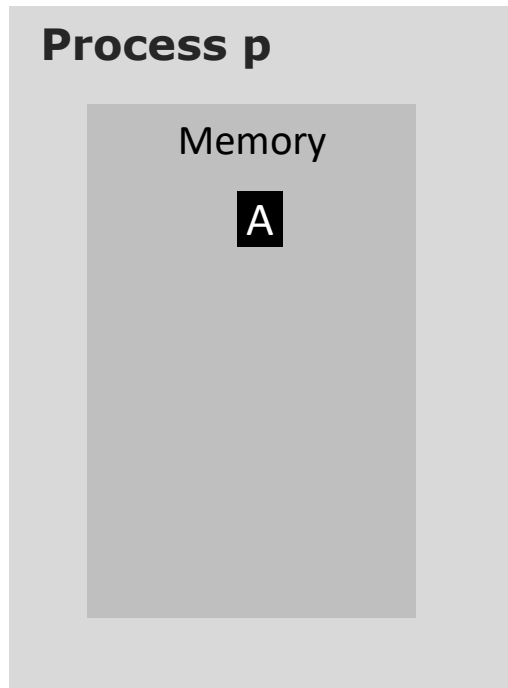
How to ensure tunable performance?



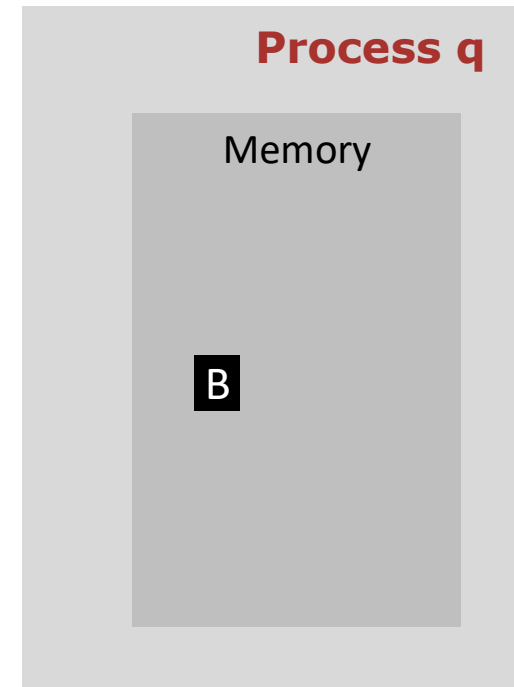
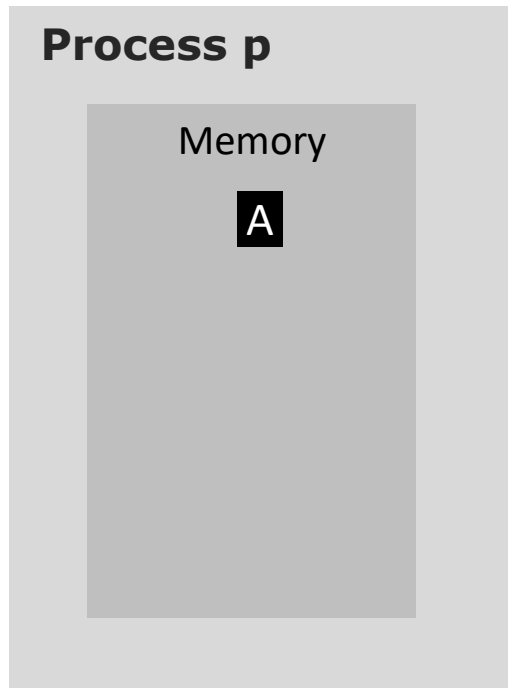
What mechanism to use for efficient implementation?

REMOTE MEMORY ACCESS (RMA) PROGRAMMING

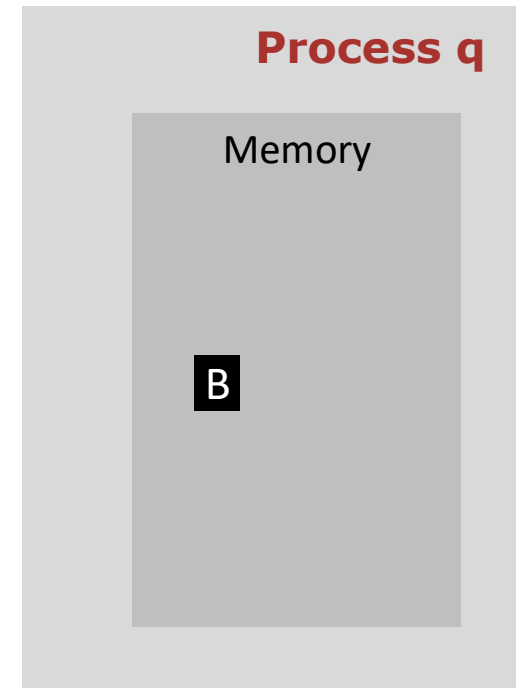
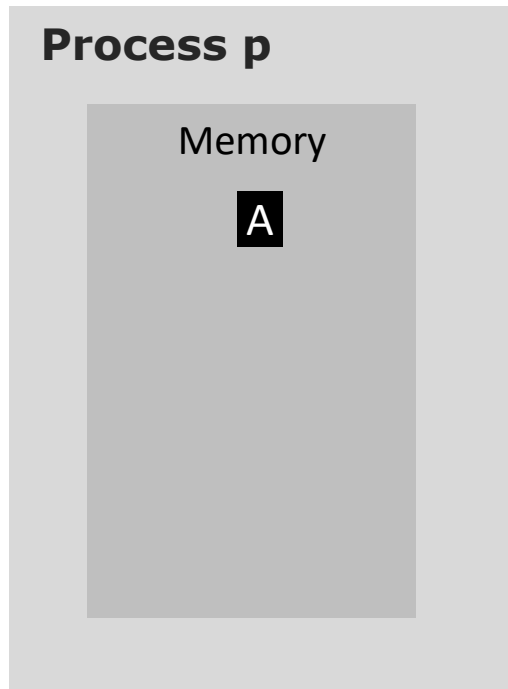
REMOTE MEMORY ACCESS (RMA) PROGRAMMING



REMOTE MEMORY ACCESS (RMA) PROGRAMMING

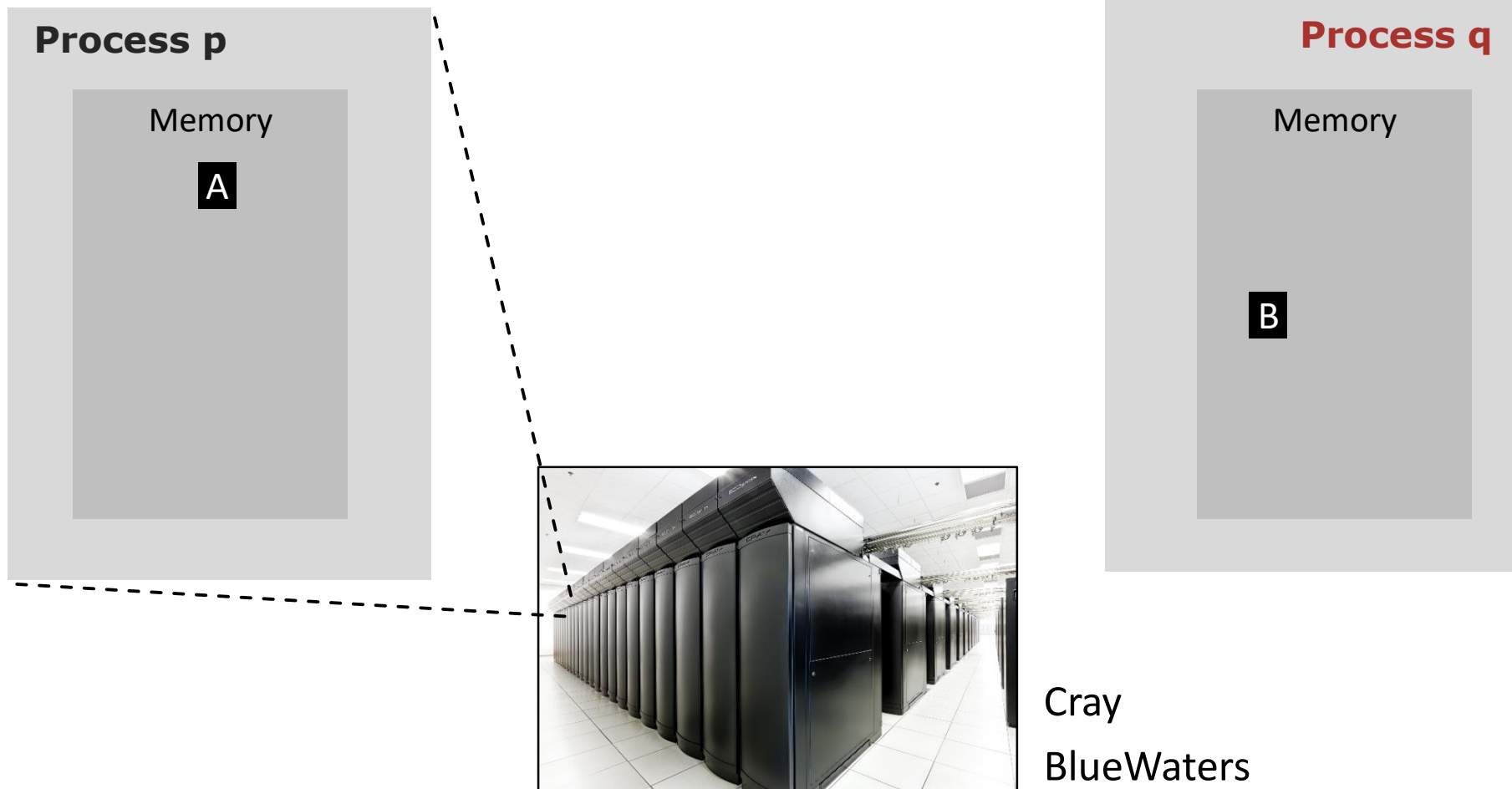


REMOTE MEMORY ACCESS (RMA) PROGRAMMING

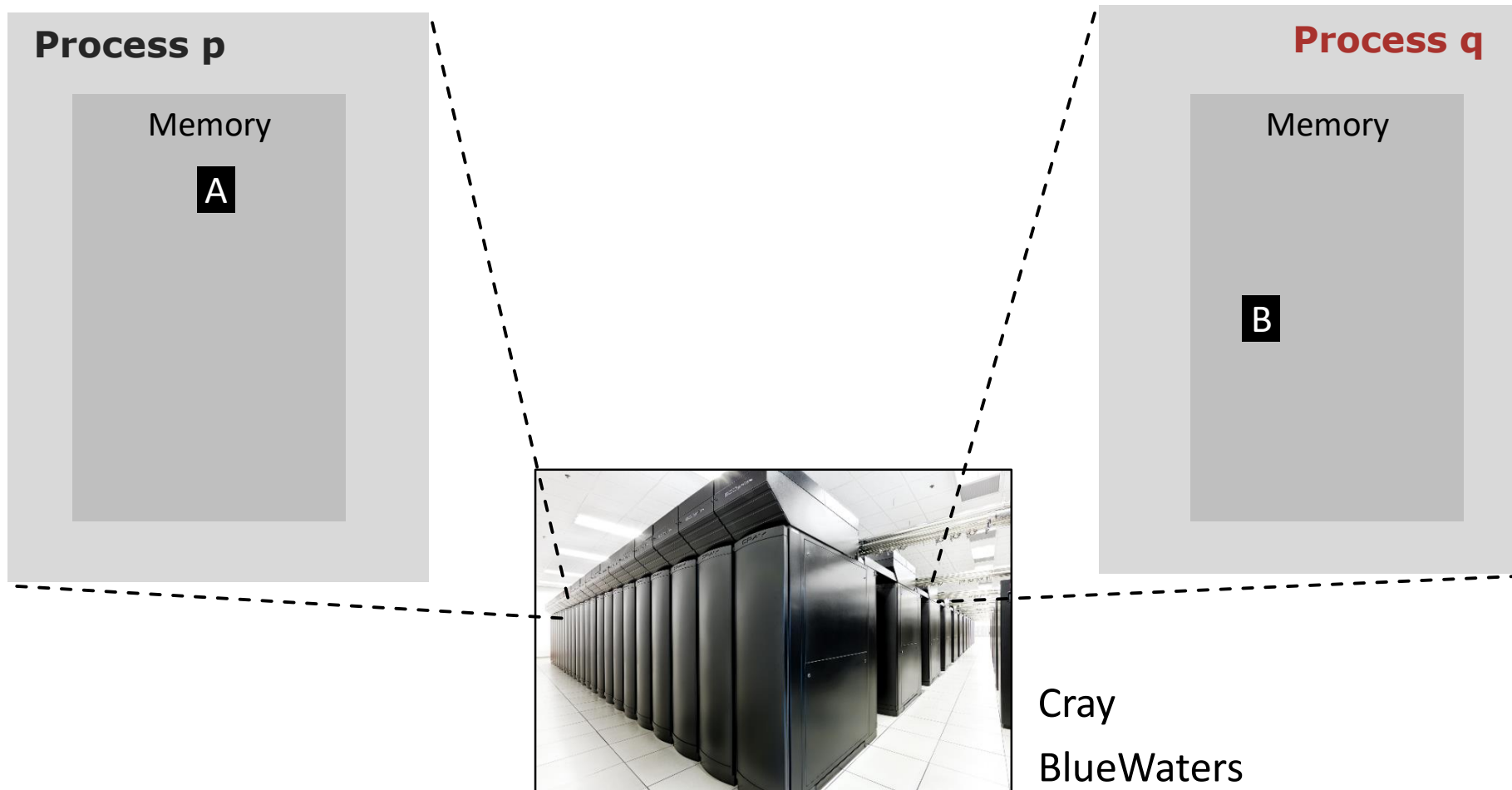


Cray
BlueWaters

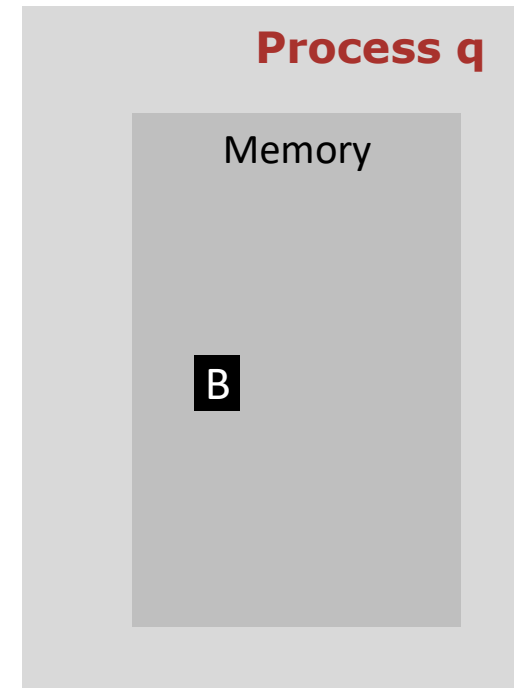
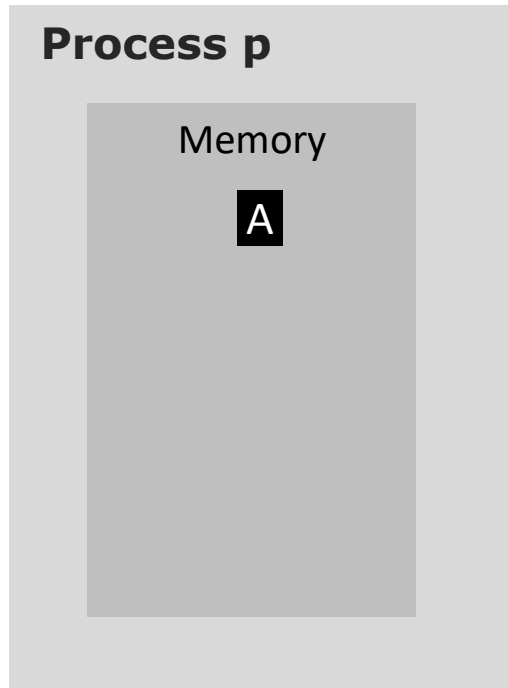
REMOTE MEMORY ACCESS (RMA) PROGRAMMING



REMOTE MEMORY ACCESS (RMA) PROGRAMMING

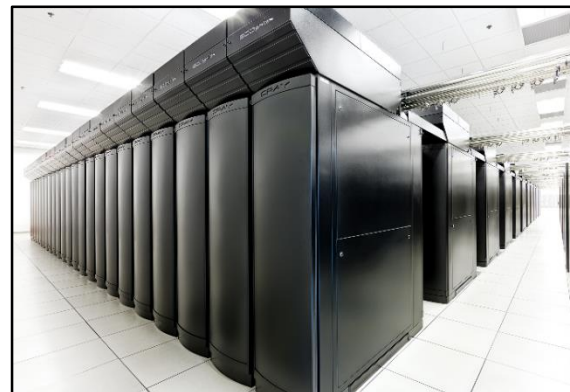
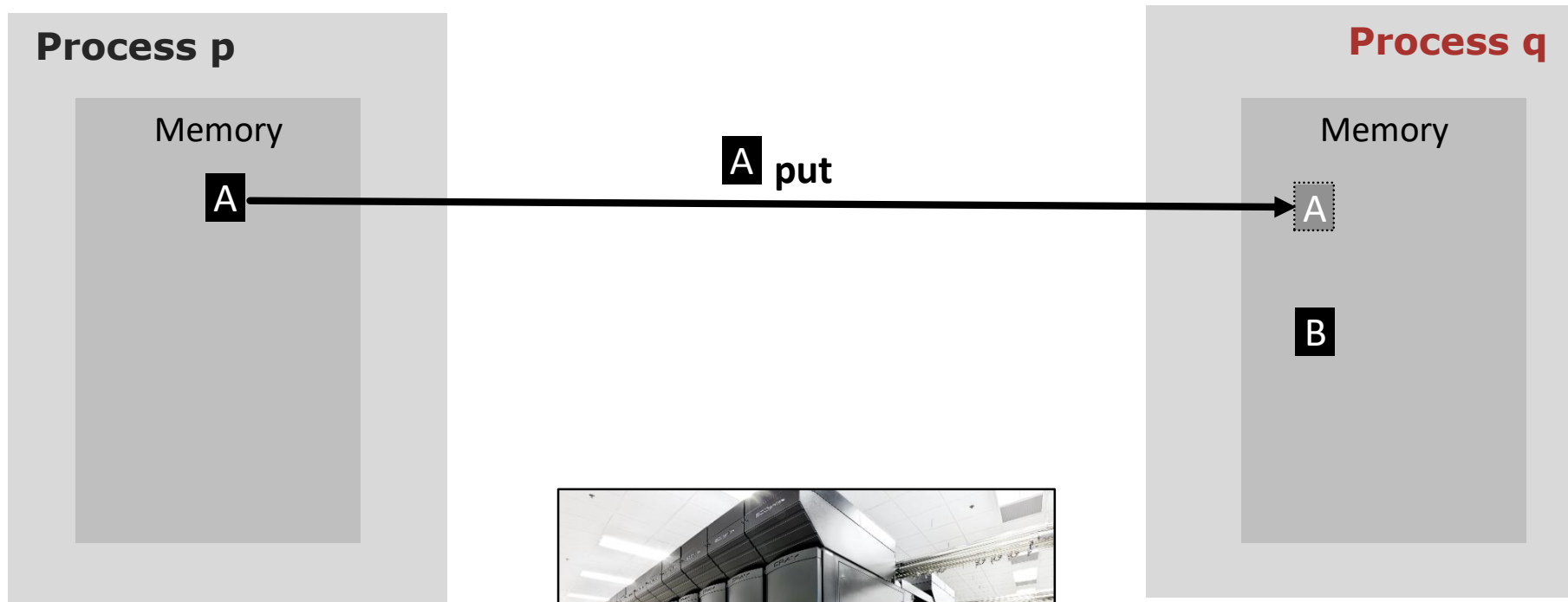


REMOTE MEMORY ACCESS (RMA) PROGRAMMING



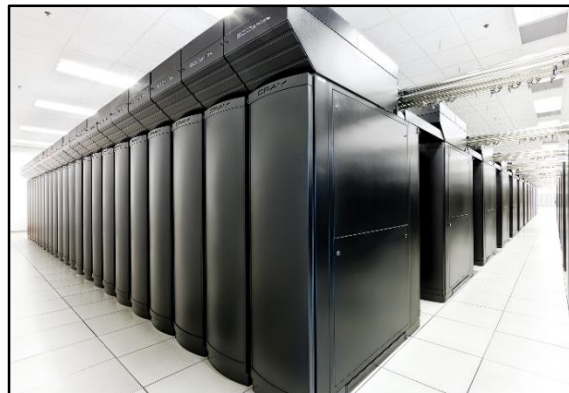
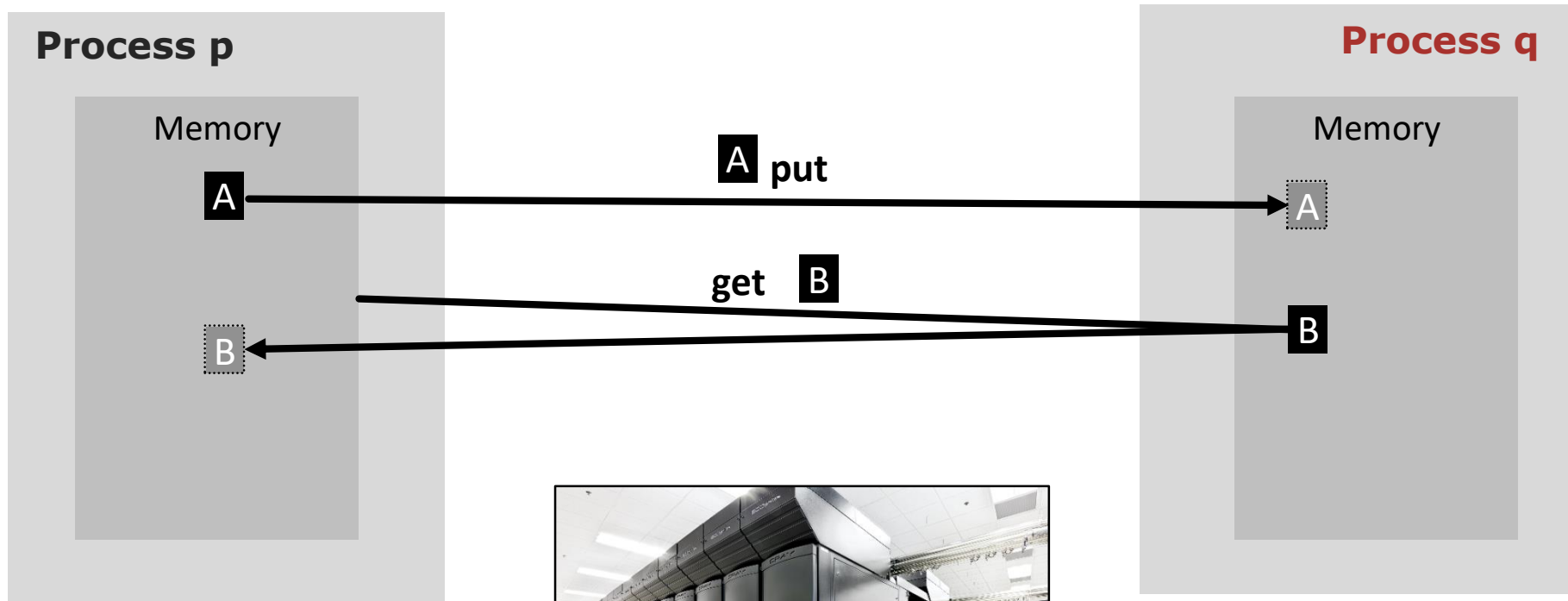
Cray
BlueWaters

REMOTE MEMORY ACCESS (RMA) PROGRAMMING



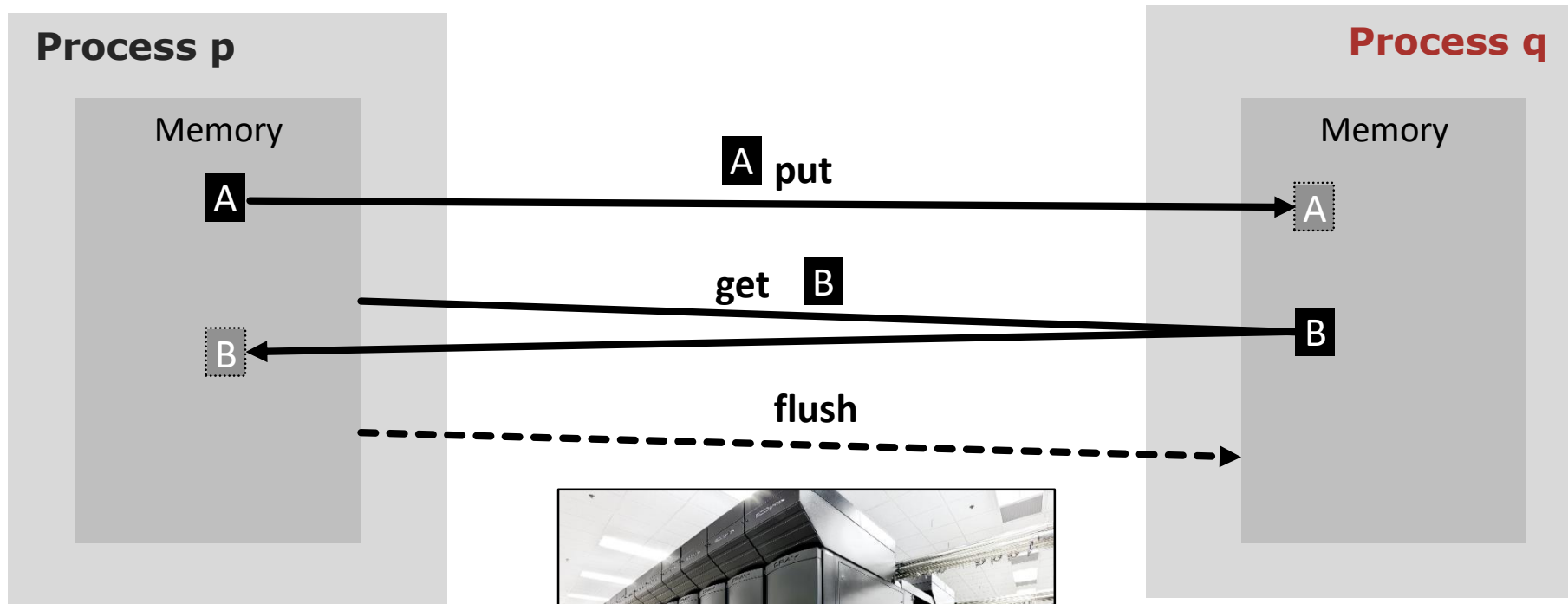
Cray
BlueWaters

REMOTE MEMORY ACCESS (RMA) PROGRAMMING



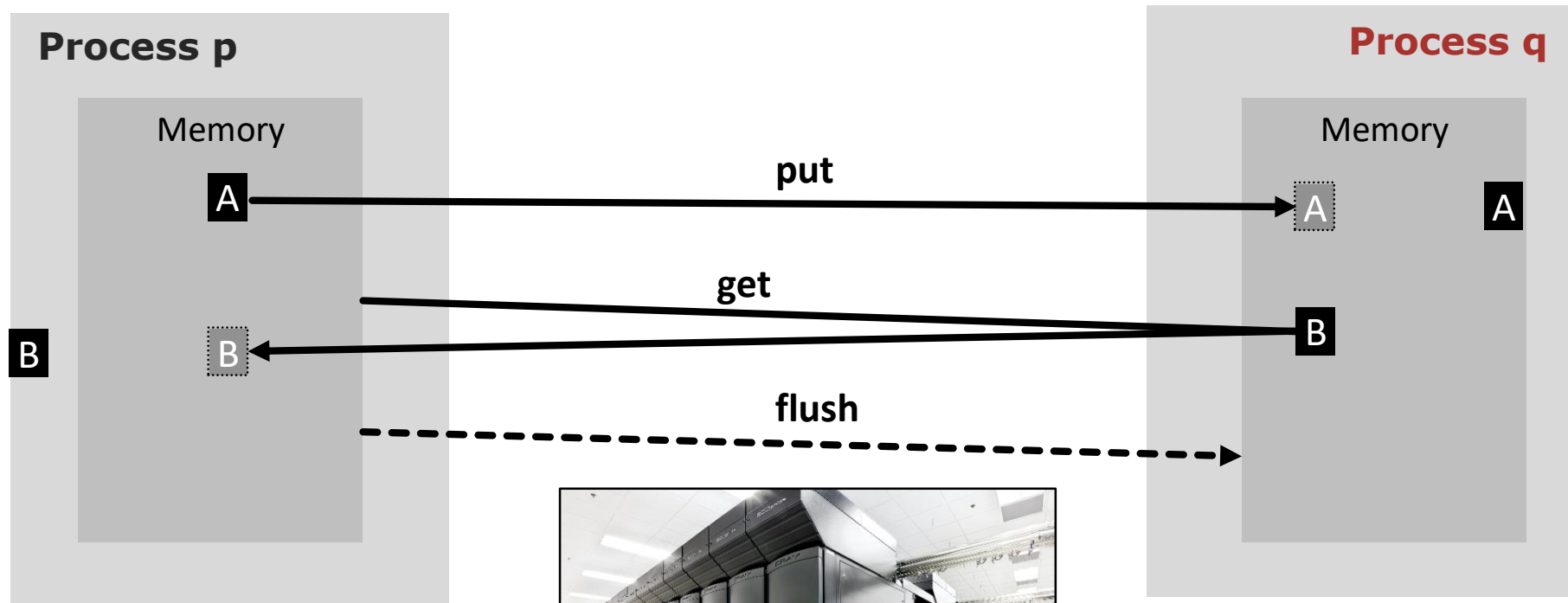
Cray
BlueWaters

REMOTE MEMORY ACCESS (RMA) PROGRAMMING



Cray
BlueWaters

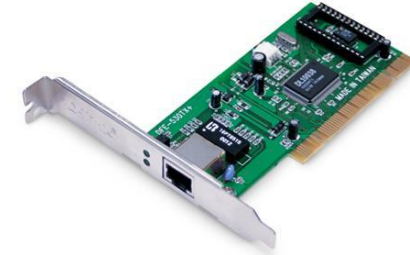
REMOTE MEMORY ACCESS (RMA) PROGRAMMING



Cray
BlueWaters

REMOTE MEMORY ACCESS PROGRAMMING

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).



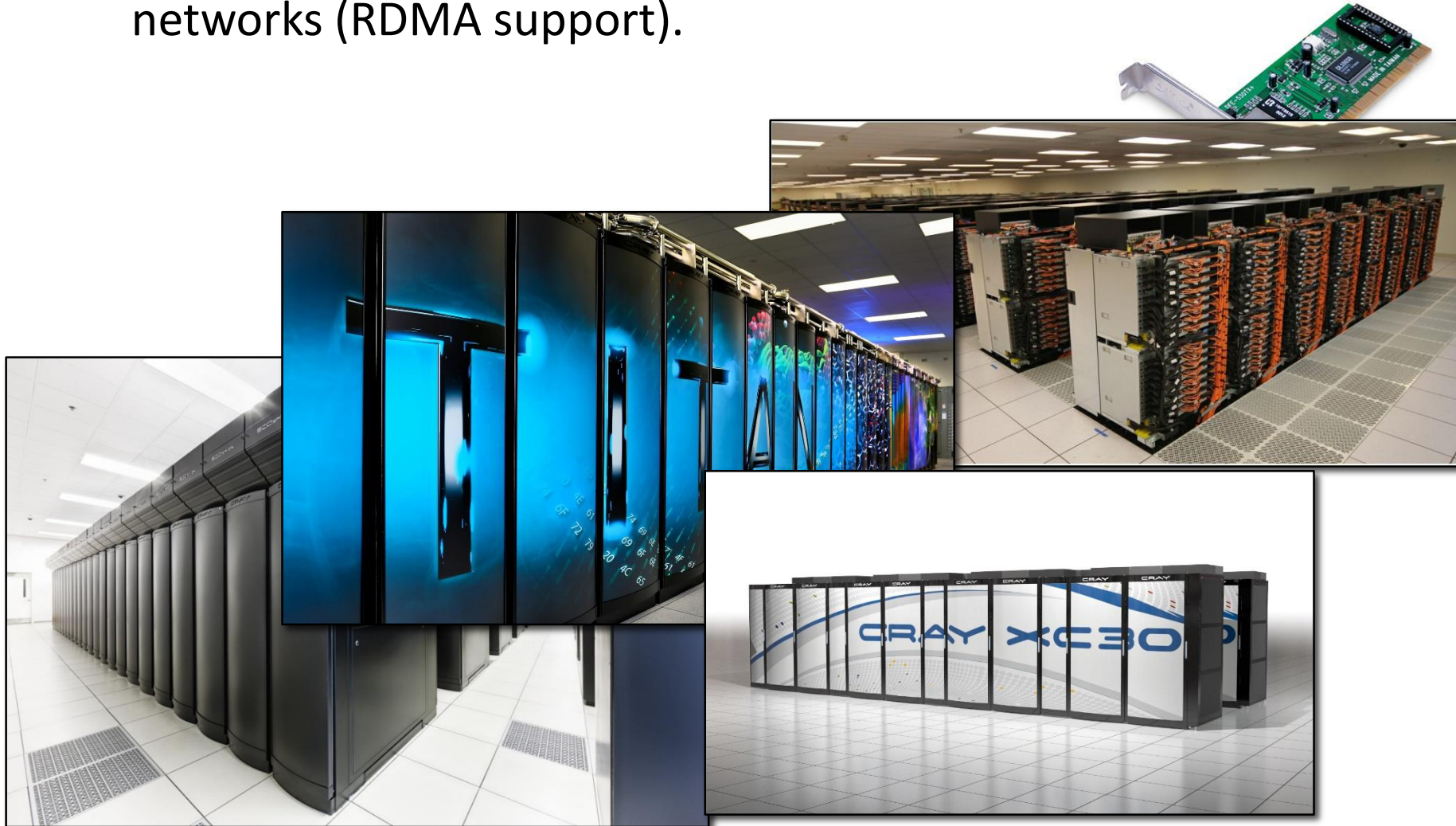
REMOTE MEMORY ACCESS PROGRAMMING

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).



REMOTE MEMORY ACCESS PROGRAMMING

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).



REMOTE MEMORY ACCESS PROGRAMMING

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).

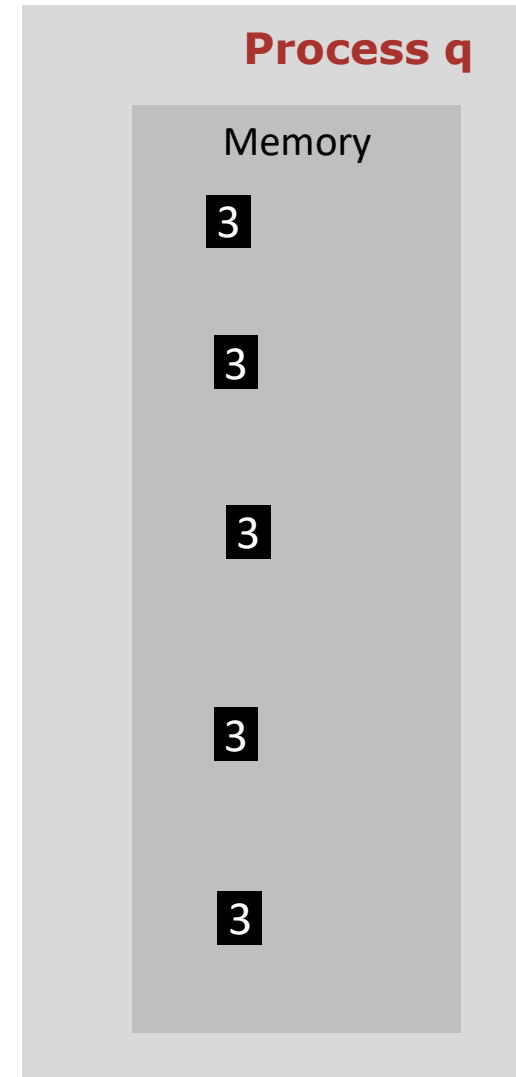
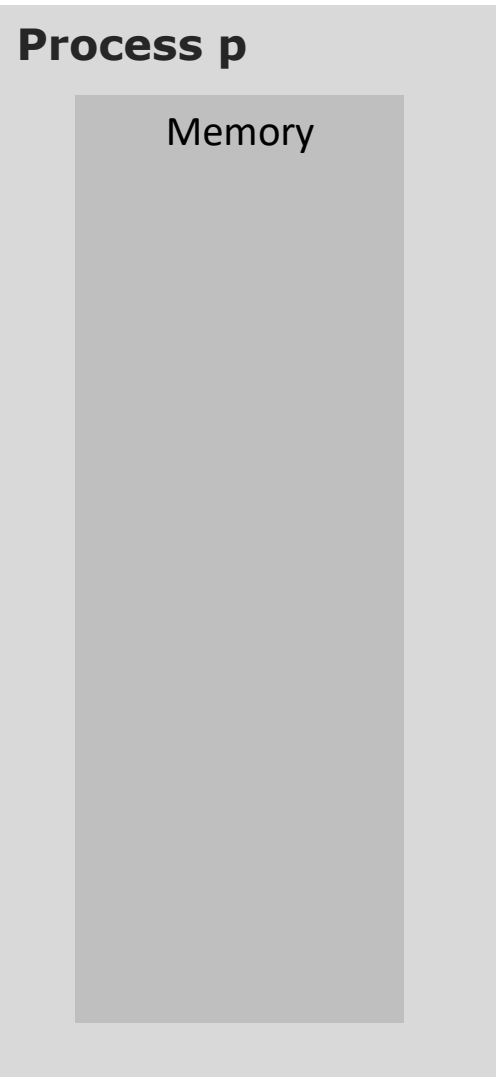


REMOTE MEMORY ACCESS PROGRAMMING

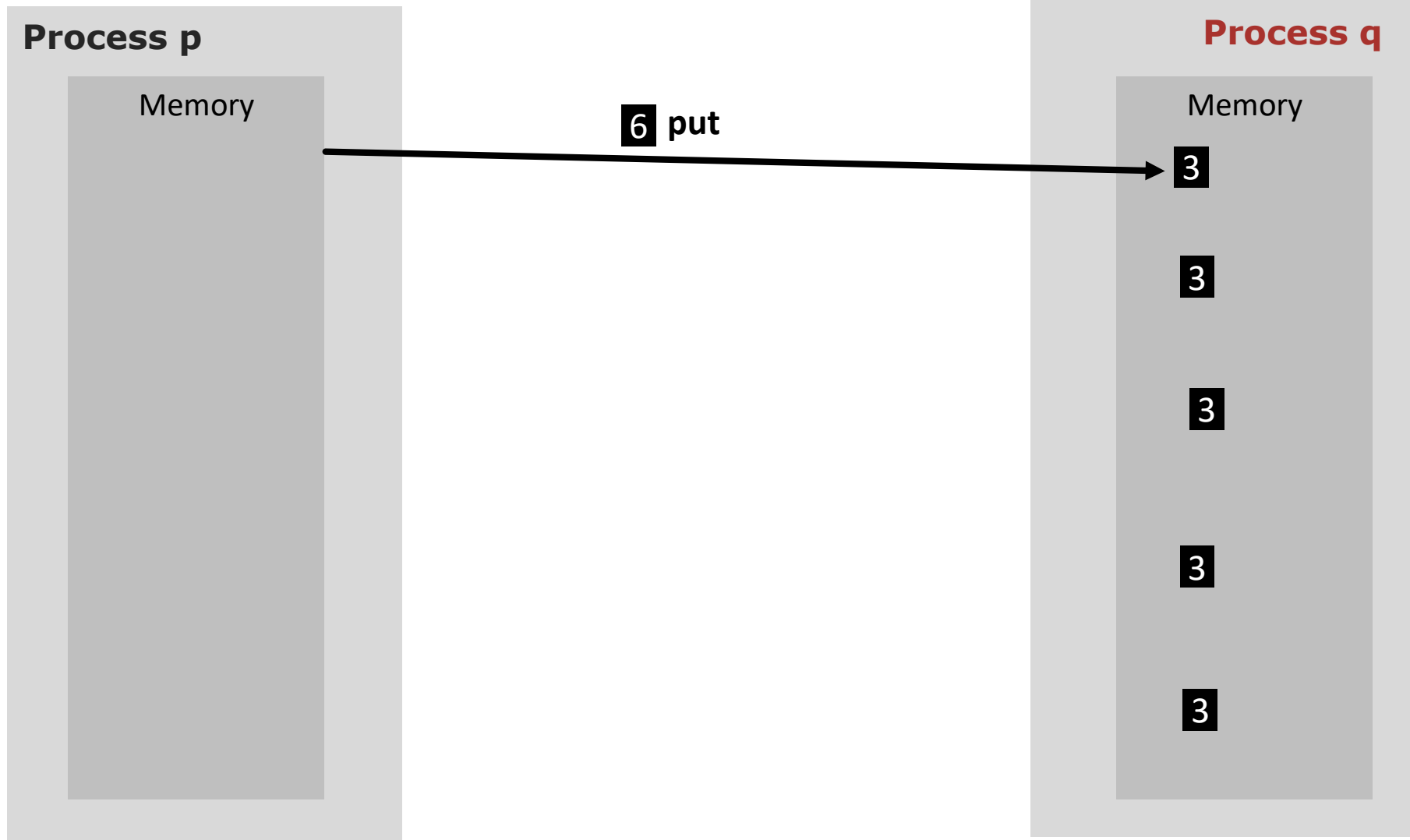
- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).



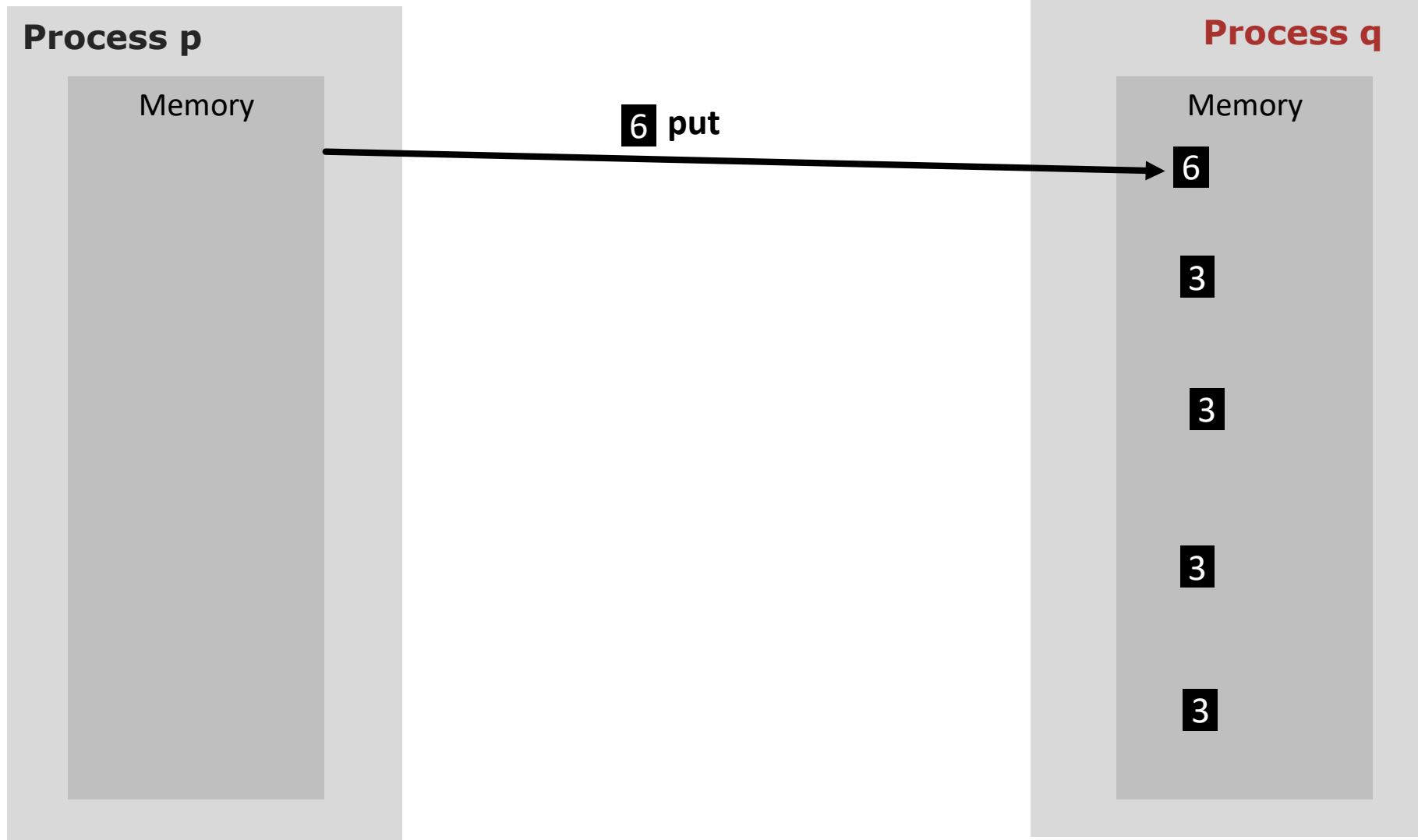
RMA-RW - Required Operations



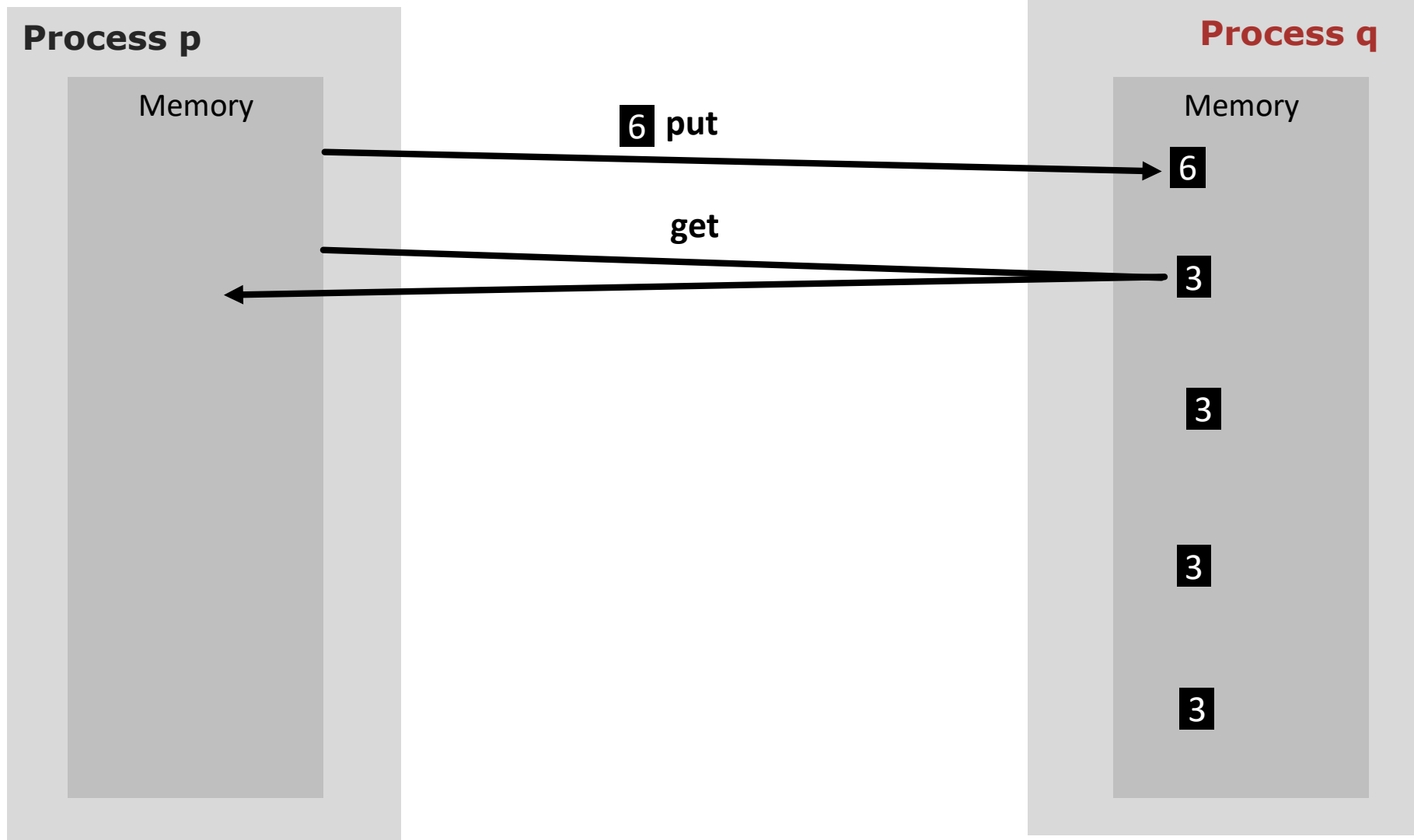
RMA-RW - Required Operations



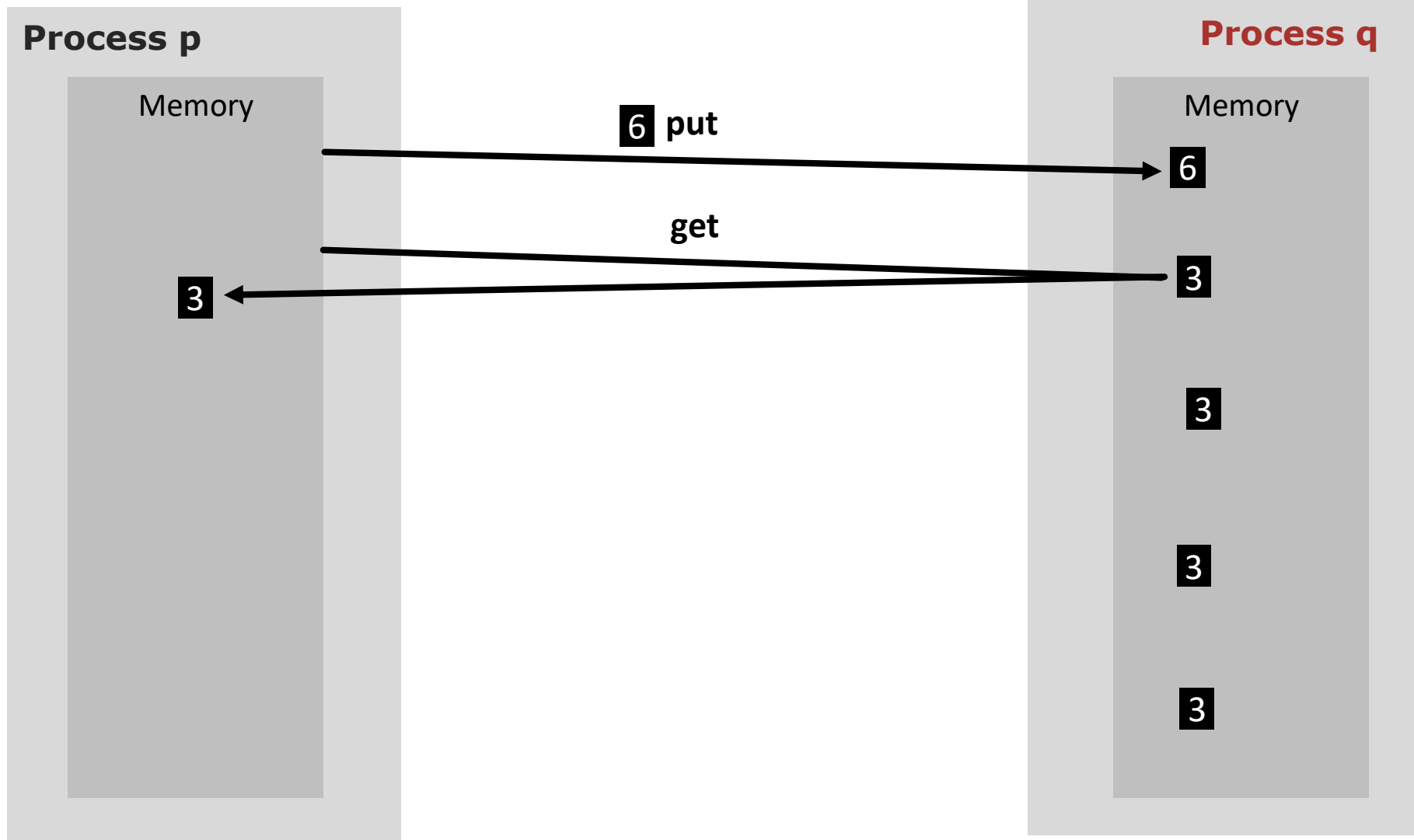
RMA-RW - Required Operations



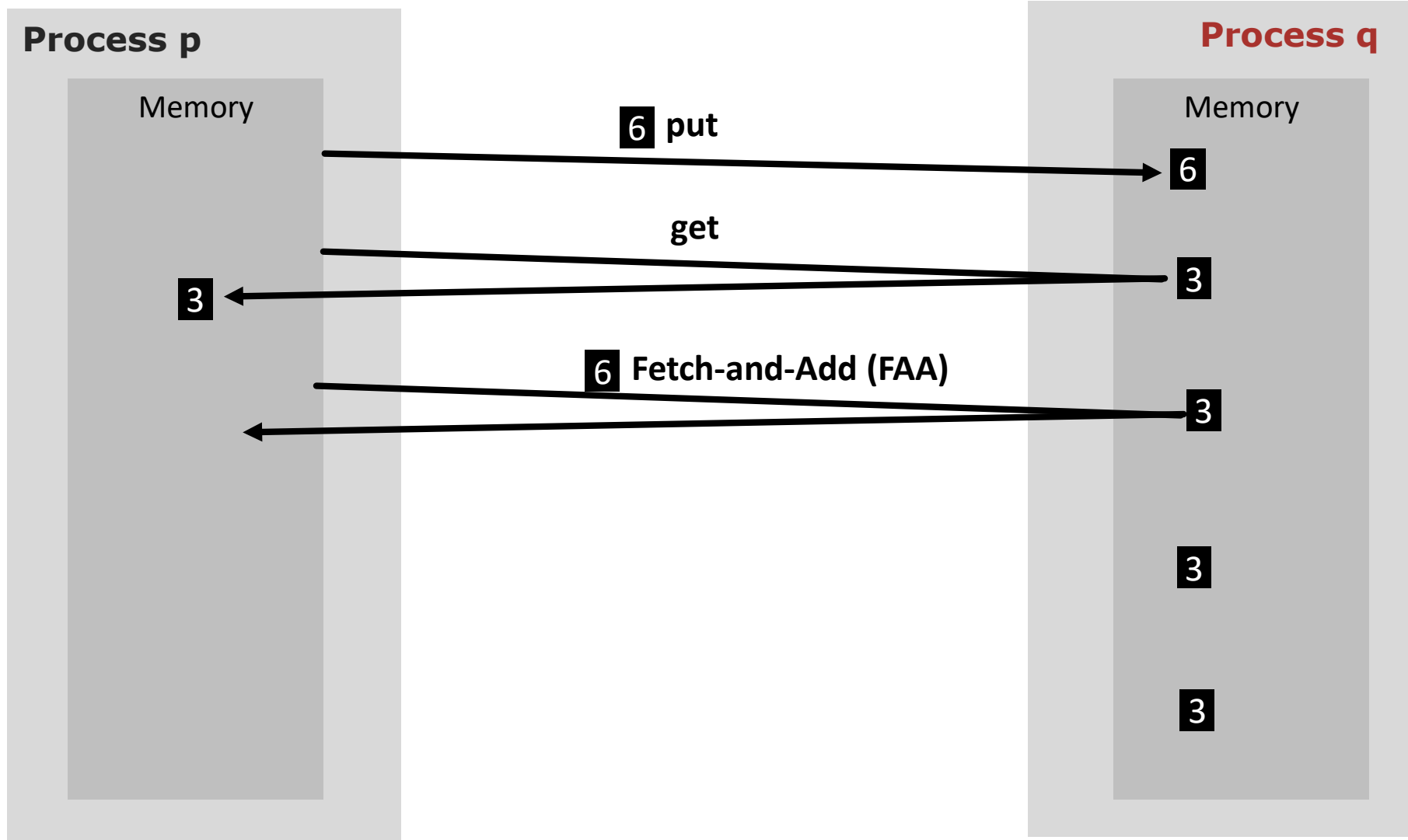
RMA-RW - Required Operations



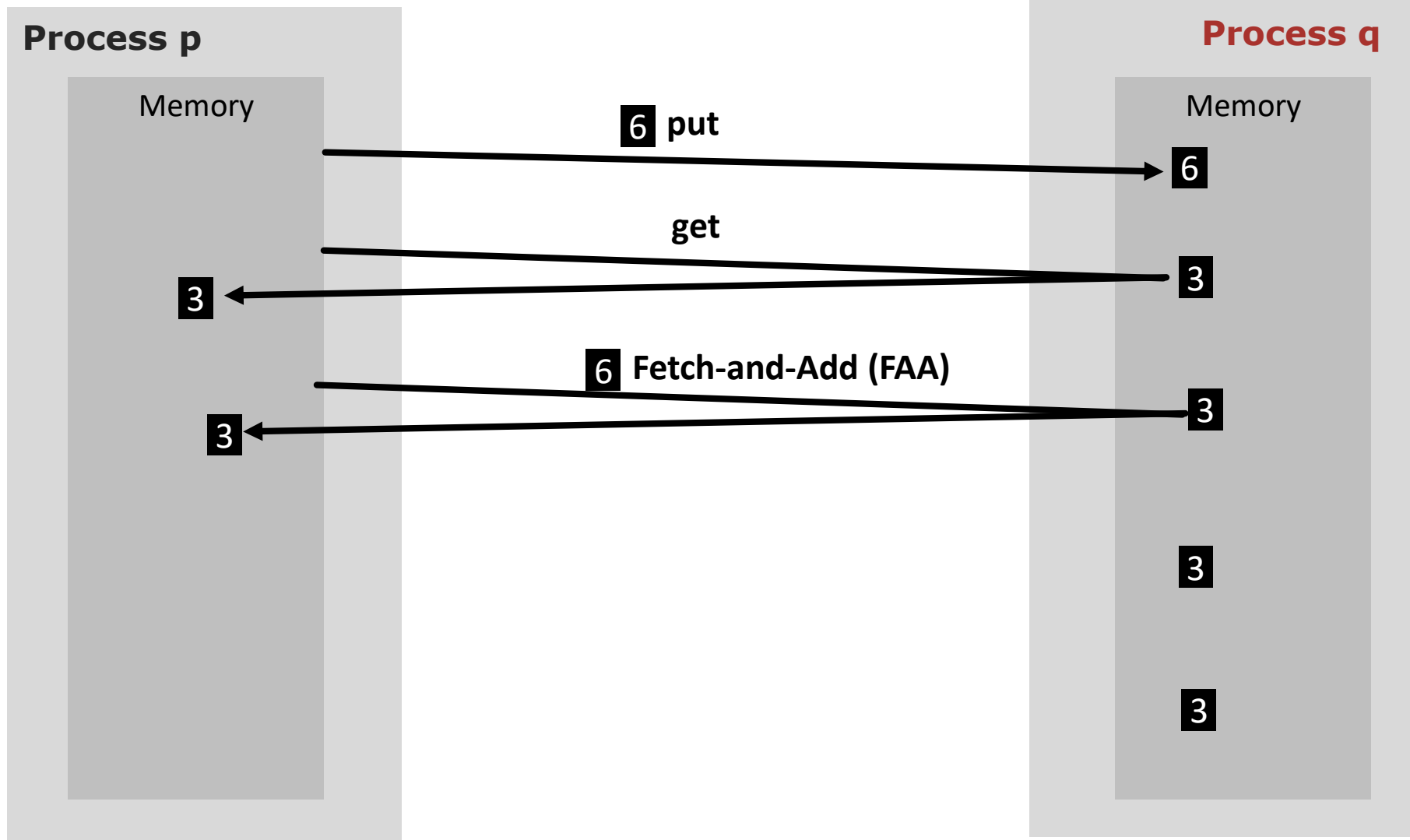
RMA-RW - Required Operations



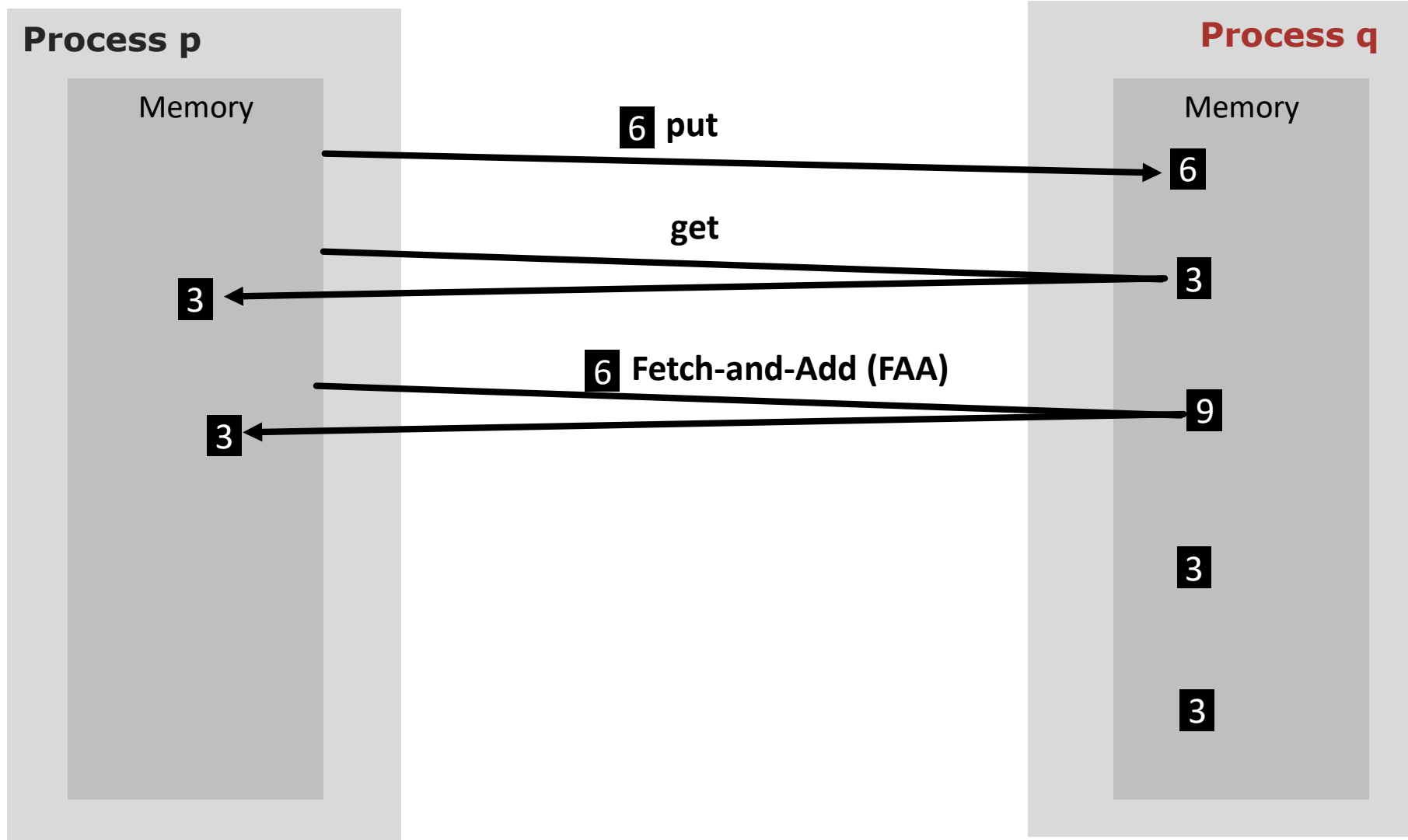
RMA-RW - Required Operations



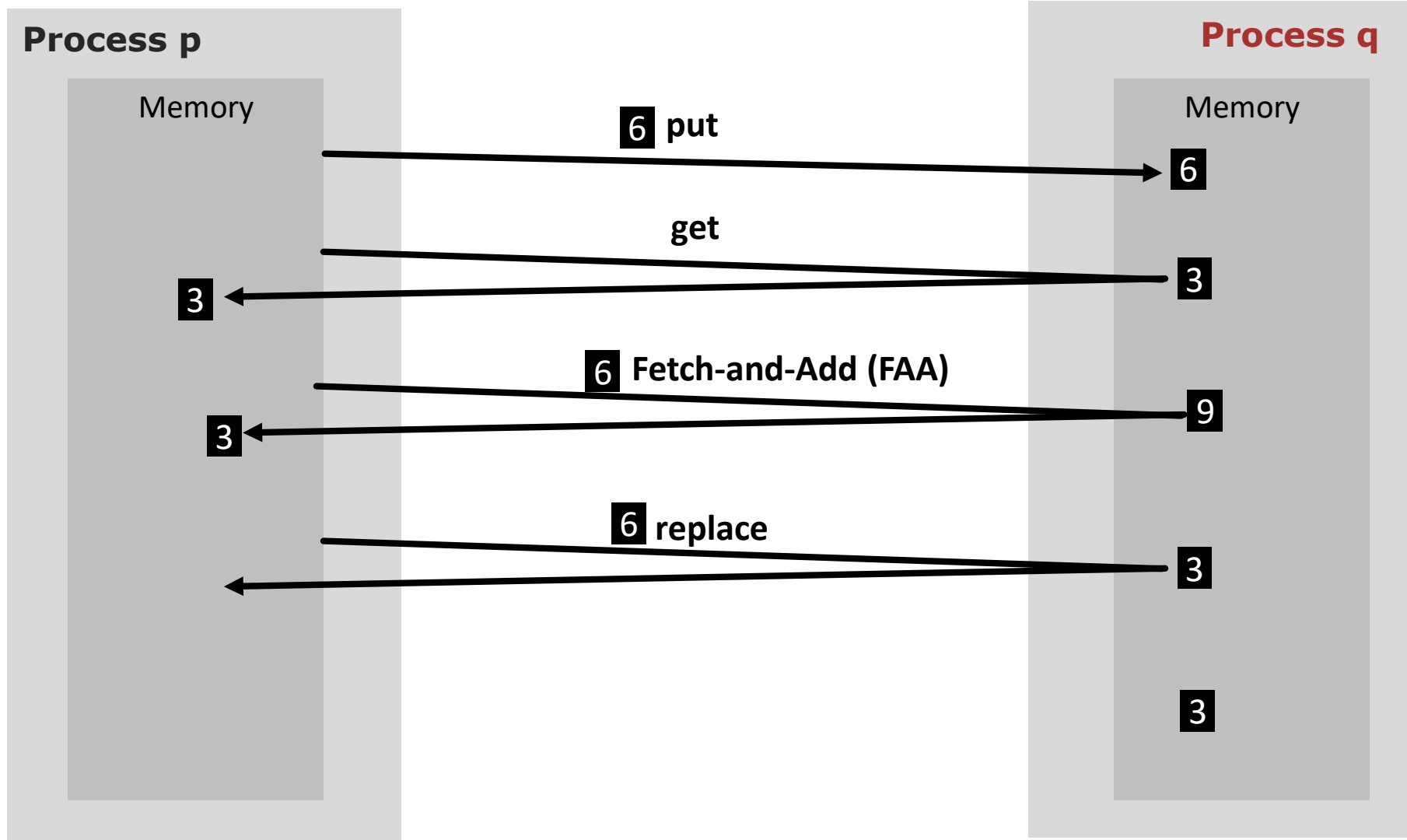
RMA-RW - Required Operations



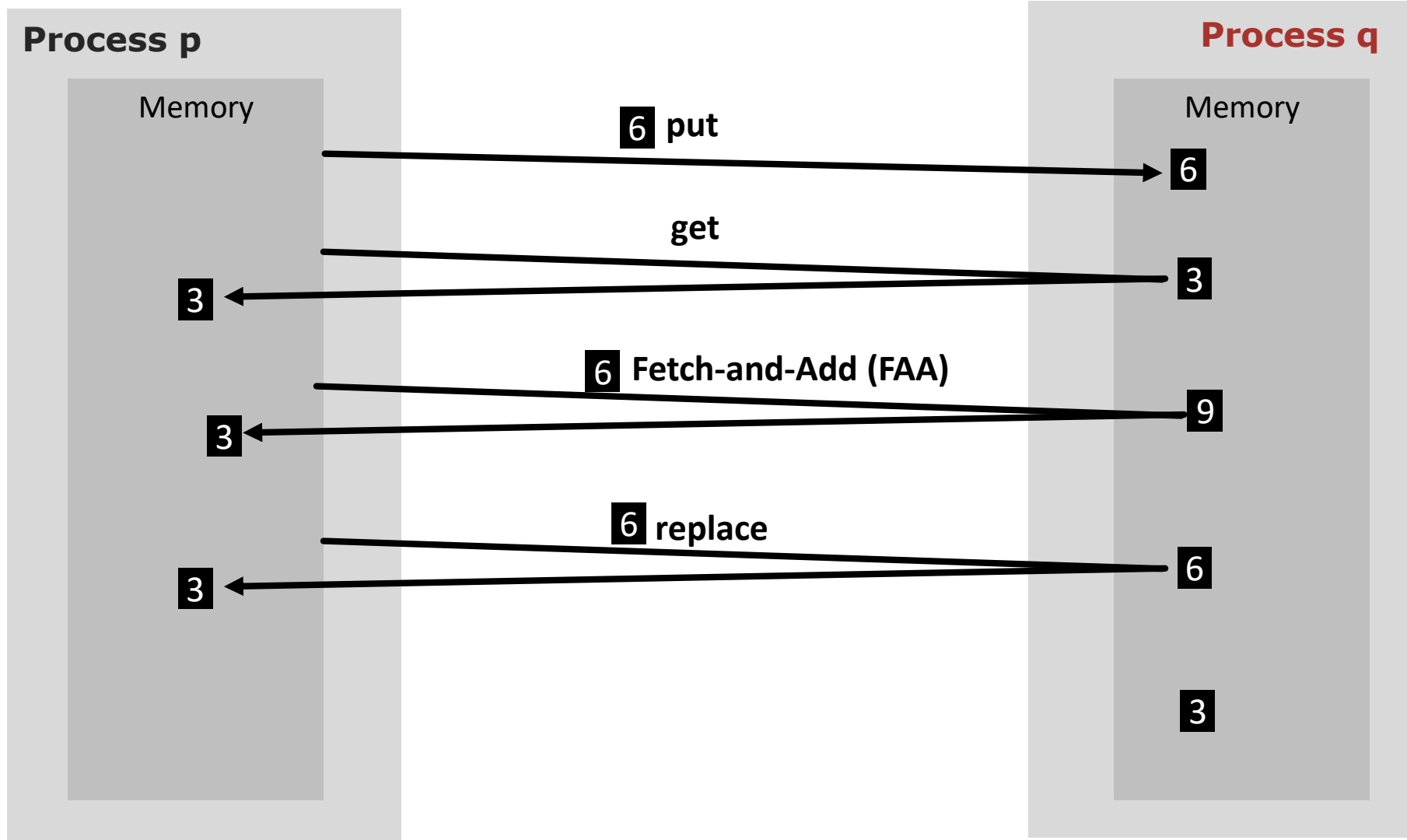
RMA-RW - Required Operations



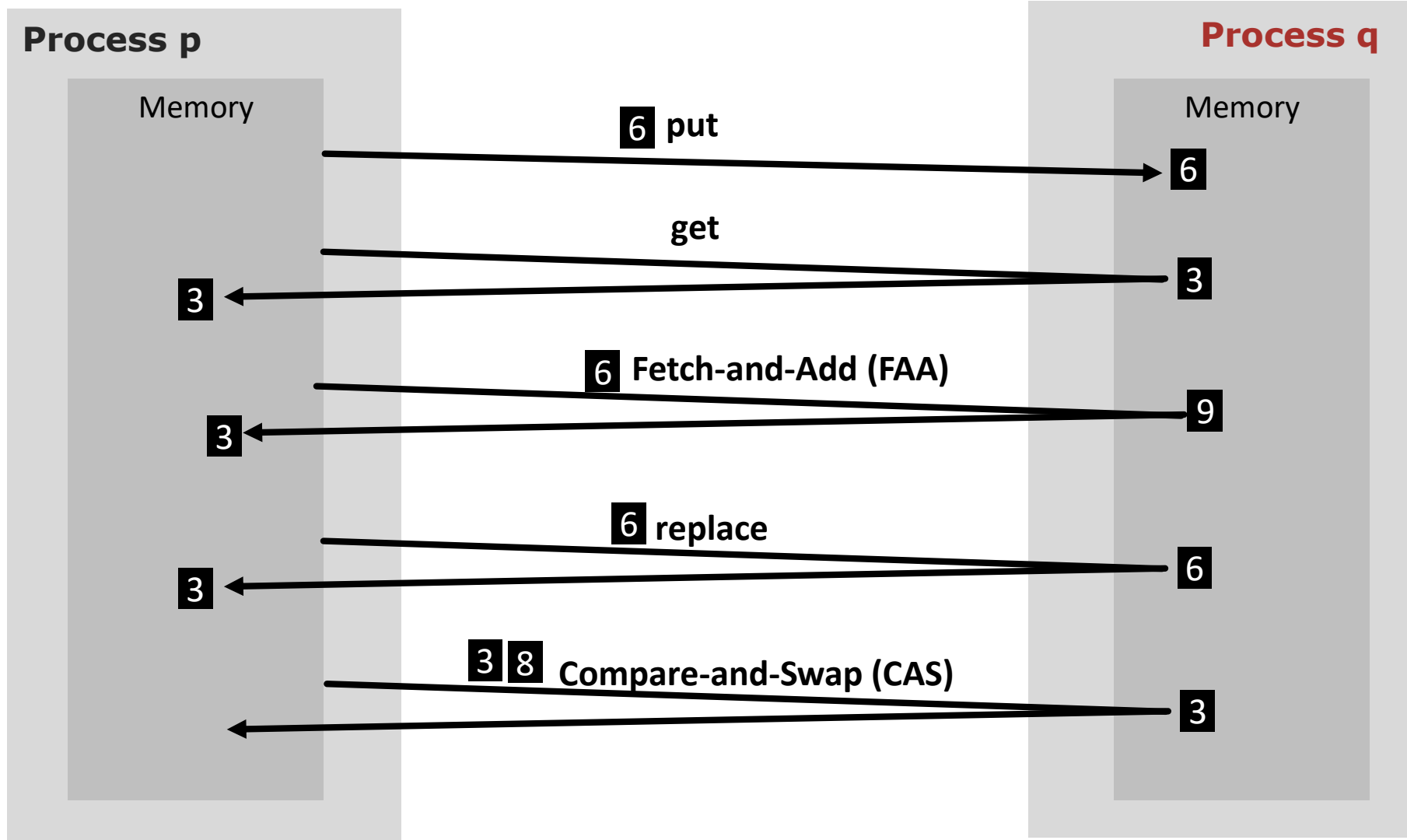
RMA-RW - Required Operations



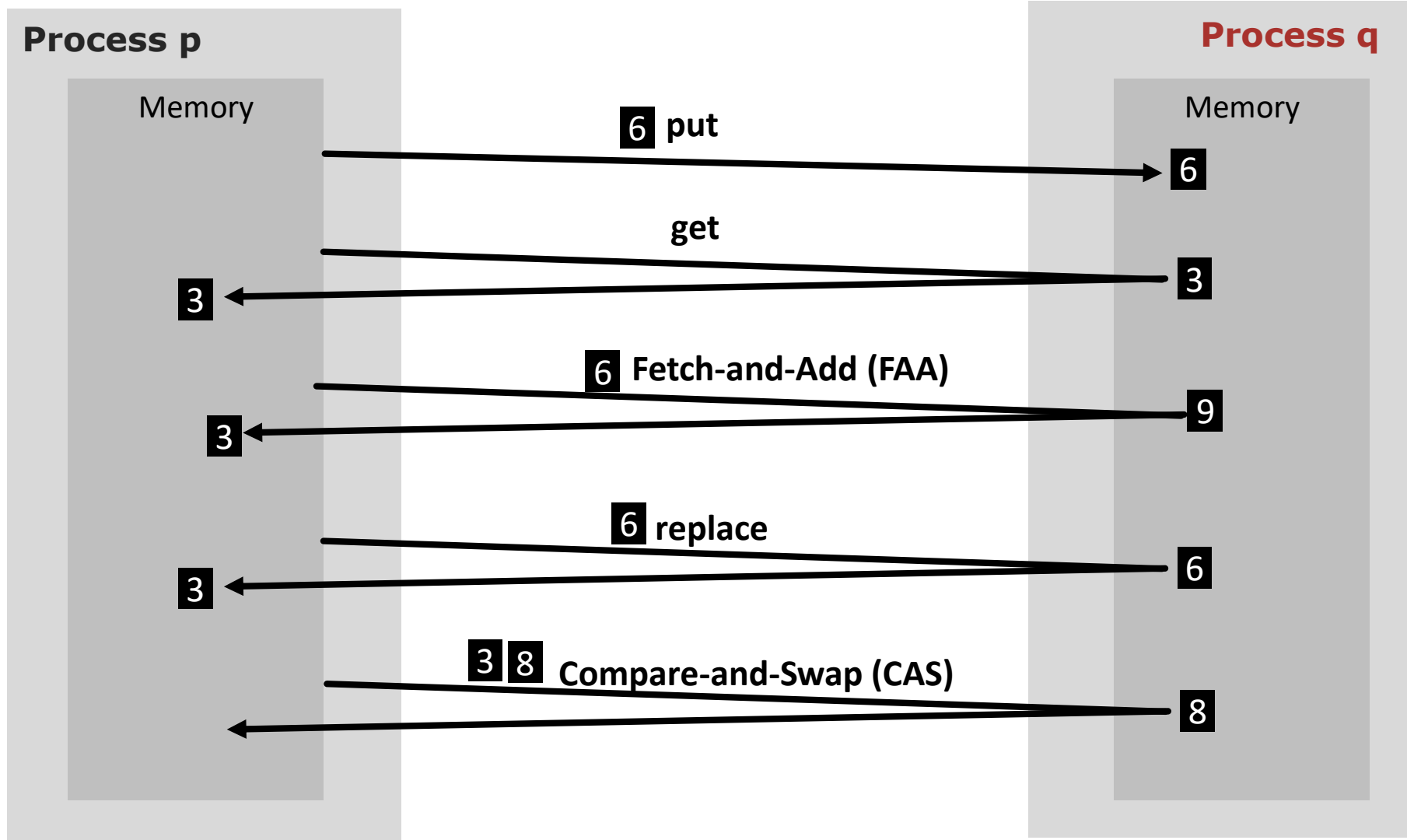
RMA-RW - Required Operations



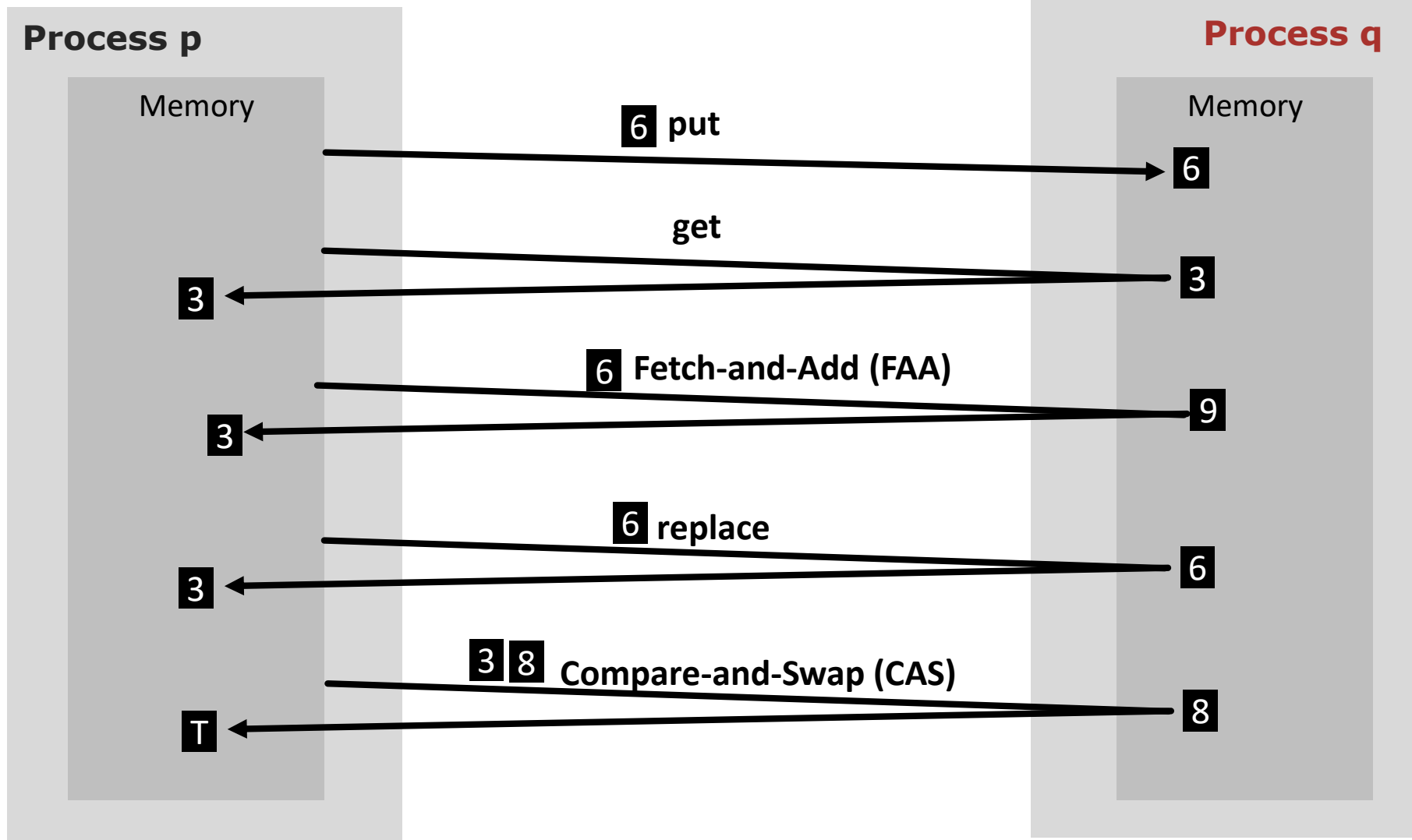
RMA-RW - Required Operations



RMA-RW - Required Operations



RMA-RW - Required Operations



MPI RMA primer ([much] more in the recitation sessions)

- **Windows expose memory**

- Created explicitly

- **Remote accesses**

- Put, get
- Atomics

Accumulate (also atomic Put)

Get_accumulate (also atomic Get)

Fetch and op (faster single-word get_accumulate)

Compare and swap

- **Synchronization**

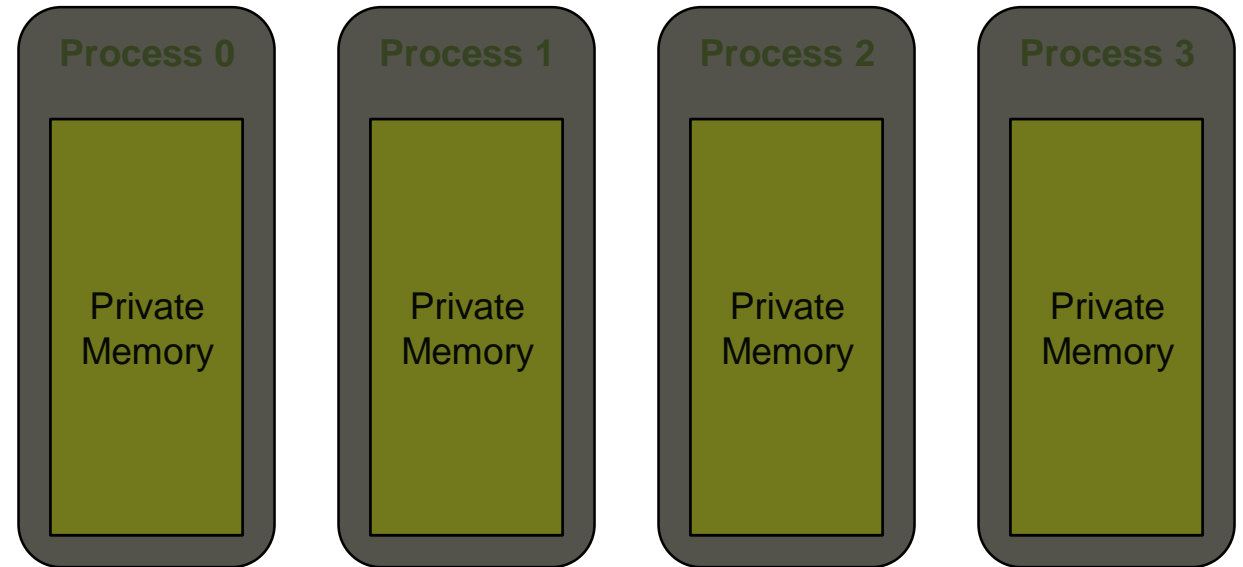
- Two modes: passive and active target

We use passive target today, similar to shared memory!

Synchronization: flush, flush_local

- **Memory model**

- Unified (coherent) and separate (not coherent) view - it's complicated but versatile



MPI RMA primer ([much] more in the recitation sessions)

- **Windows expose memory**

- Created explicitly

- **Remote accesses**

- Put, get
- Atomics

Accumulate (also atomic Put)

Get_accumulate (also atomic Get)

Fetch and op (faster single-word get_accumulate)

Compare and swap

- **Synchronization**

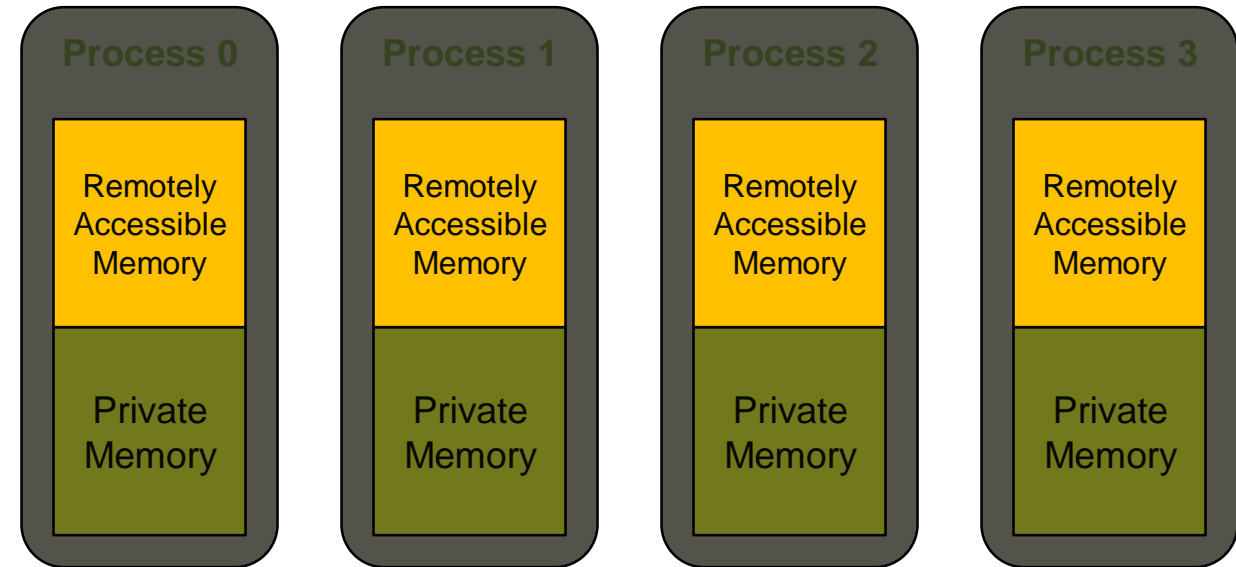
- Two modes: passive and active target

We use passive target today, similar to shared memory!

Synchronization: flush, flush_local

- **Memory model**

- Unified (coherent) and separate (not coherent) view - it's complicated but versatile



MPI RMA primer ([much] more in the recitation sessions)

- **Windows expose memory**

- Created explicitly

- **Remote accesses**

- Put, get
- Atomics

Accumulate (also atomic Put)

Get_accumulate (also atomic Get)

Fetch and op (faster single-word get_accumulate)

Compare and swap

- **Synchronization**

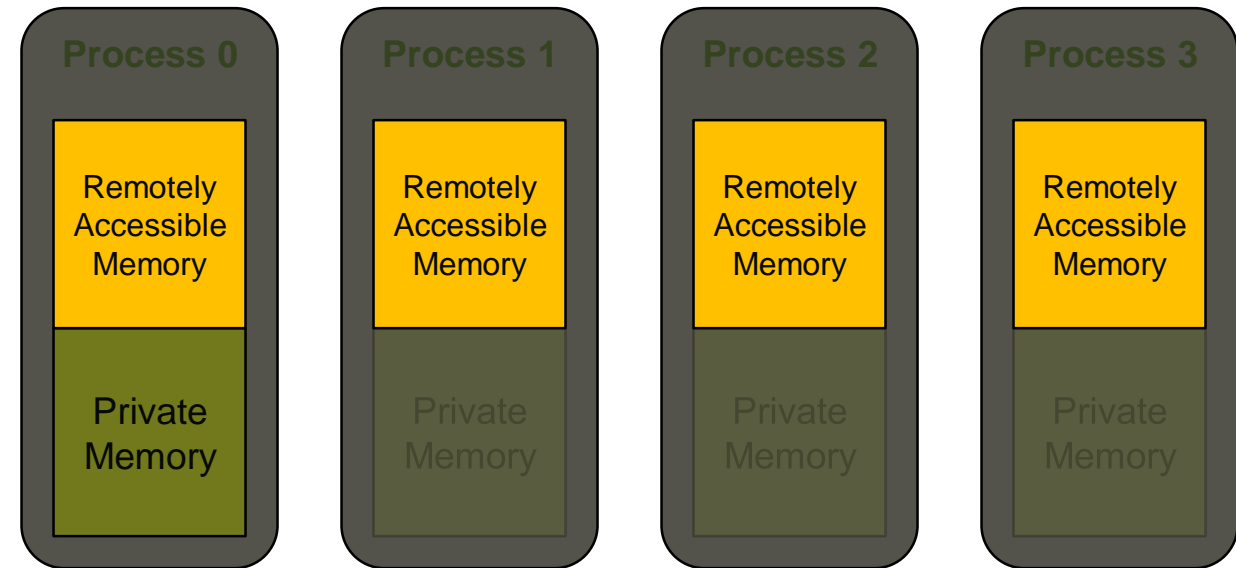
- Two modes: passive and active target

We use passive target today, similar to shared memory!

Synchronization: flush, flush_local

- **Memory model**

- Unified (coherent) and separate (not coherent) view - it's complicated but versatile



MPI RMA primer ([much] more in the recitation sessions)

- **Windows expose memory**

- Created explicitly

- **Remote accesses**

- Put, get
- Atomics

Accumulate (also atomic Put)

Get_accumulate (also atomic Get)

Fetch and op (faster single-word get_accumulate)

Compare and swap

- **Synchronization**

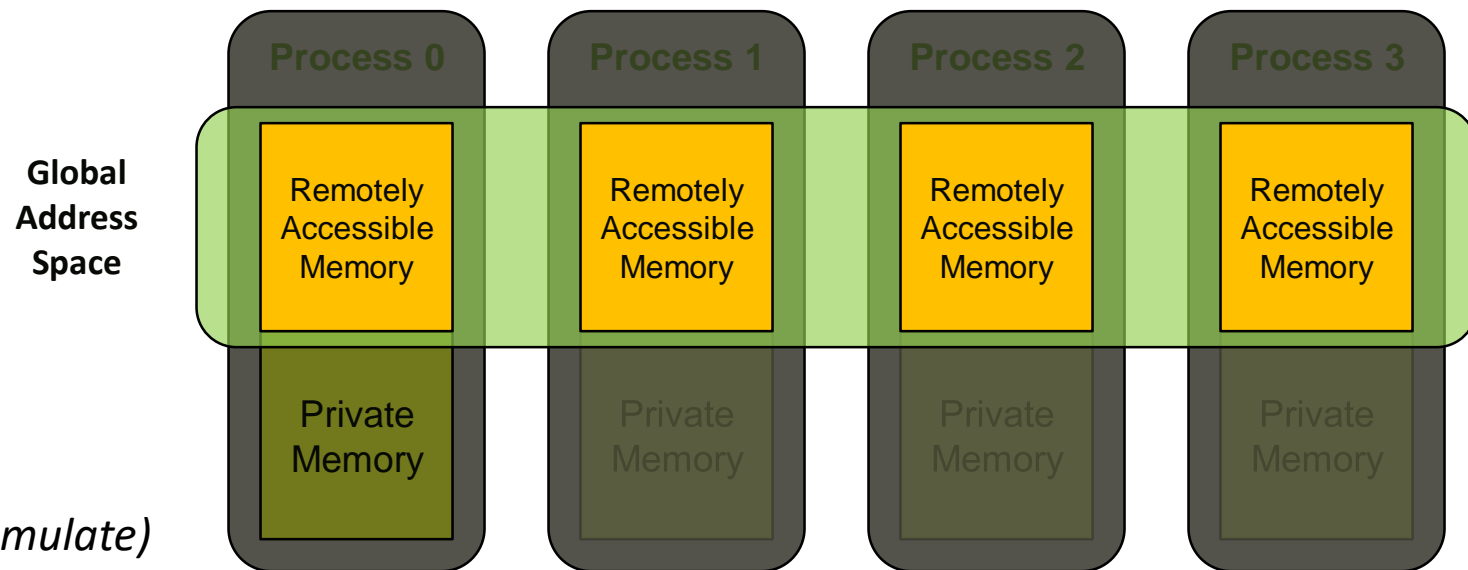
- Two modes: passive and active target

We use passive target today, similar to shared memory!

Synchronization: flush, flush_local

- **Memory model**

- Unified (coherent) and separate (not coherent) view - it's complicated but versatile



MPI RMA primer ([much] more in the recitation sessions)

- **Windows expose memory**

- Created explicitly

- **Remote accesses**

- Put, get
- Atomics

Accumulate (also atomic Put)

Get_accumulate (also atomic Get)

Fetch and op (faster single-word get_accumulate)

Compare and swap

- **Synchronization**

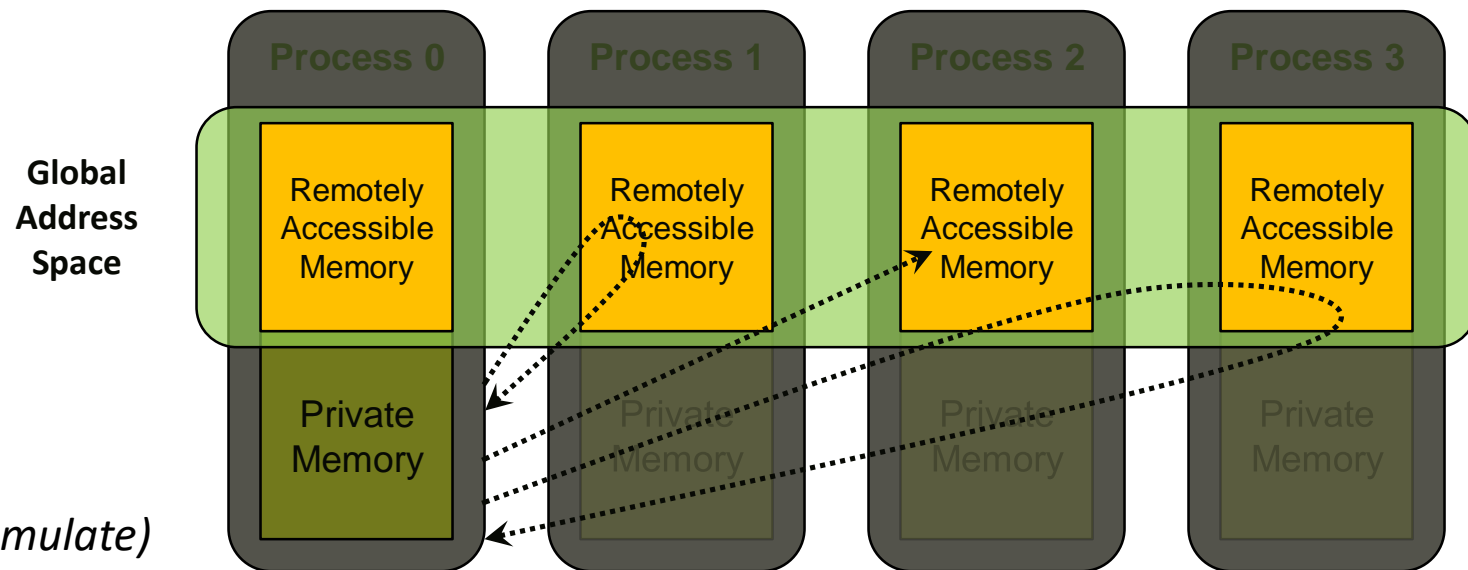
- Two modes: passive and active target

We use passive target today, similar to shared memory!

Synchronization: flush, flush_local

- **Memory model**

- Unified (coherent) and separate (not coherent) view - it's complicated but versatile





How to manage the design complexity?



How to ensure tunable performance?



What mechanism to use for efficient implementation?



How to manage the design complexity?



How to ensure tunable performance?



What mechanism to use for efficient implementation?



How to manage the design complexity?



How to ensure tunable performance?



What mechanism to use for efficient implementation?



How to manage the design complexity?



How to manage the design complexity?





How to manage the design complexity?





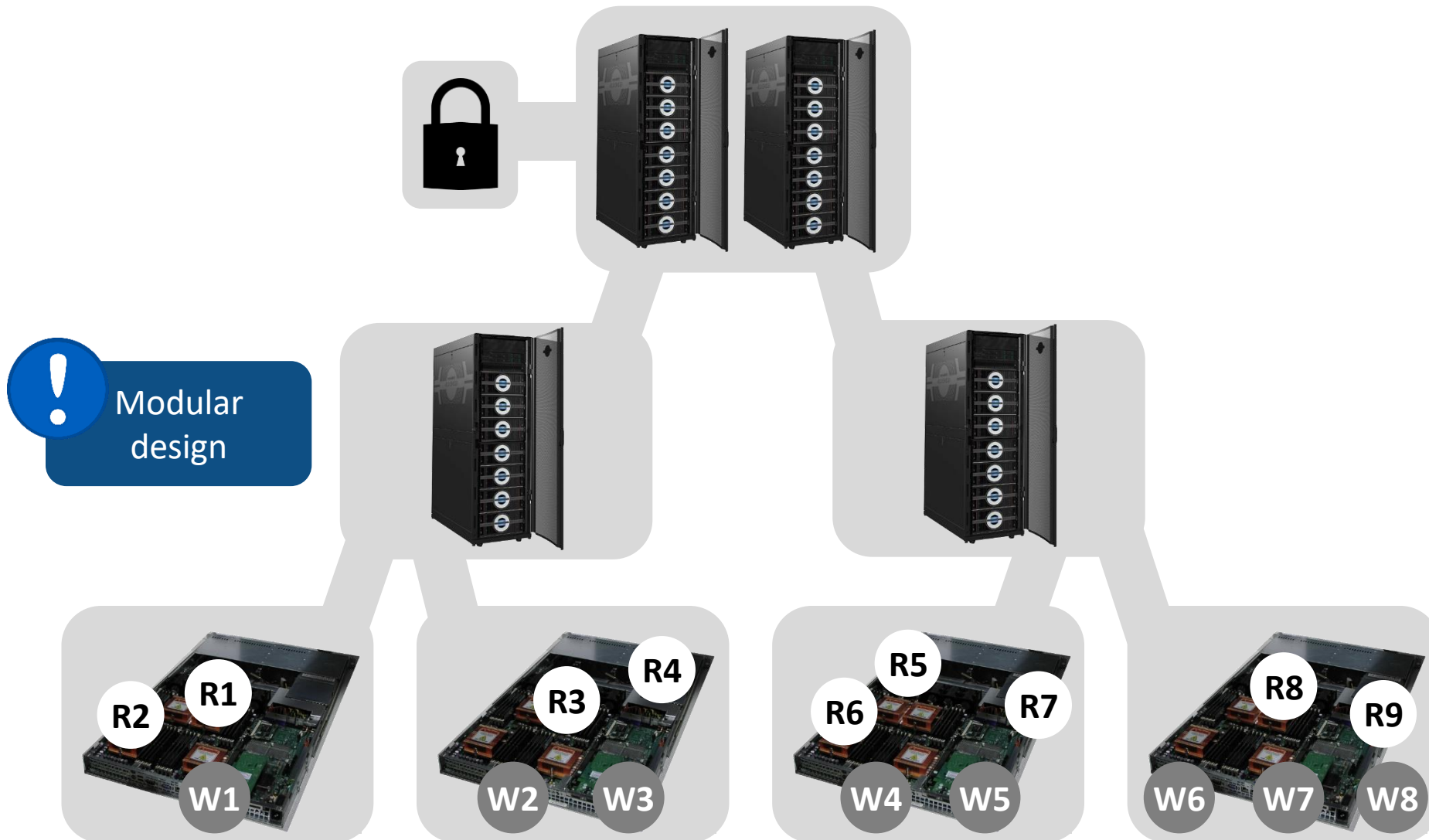
How to manage the design complexity?



Modular design



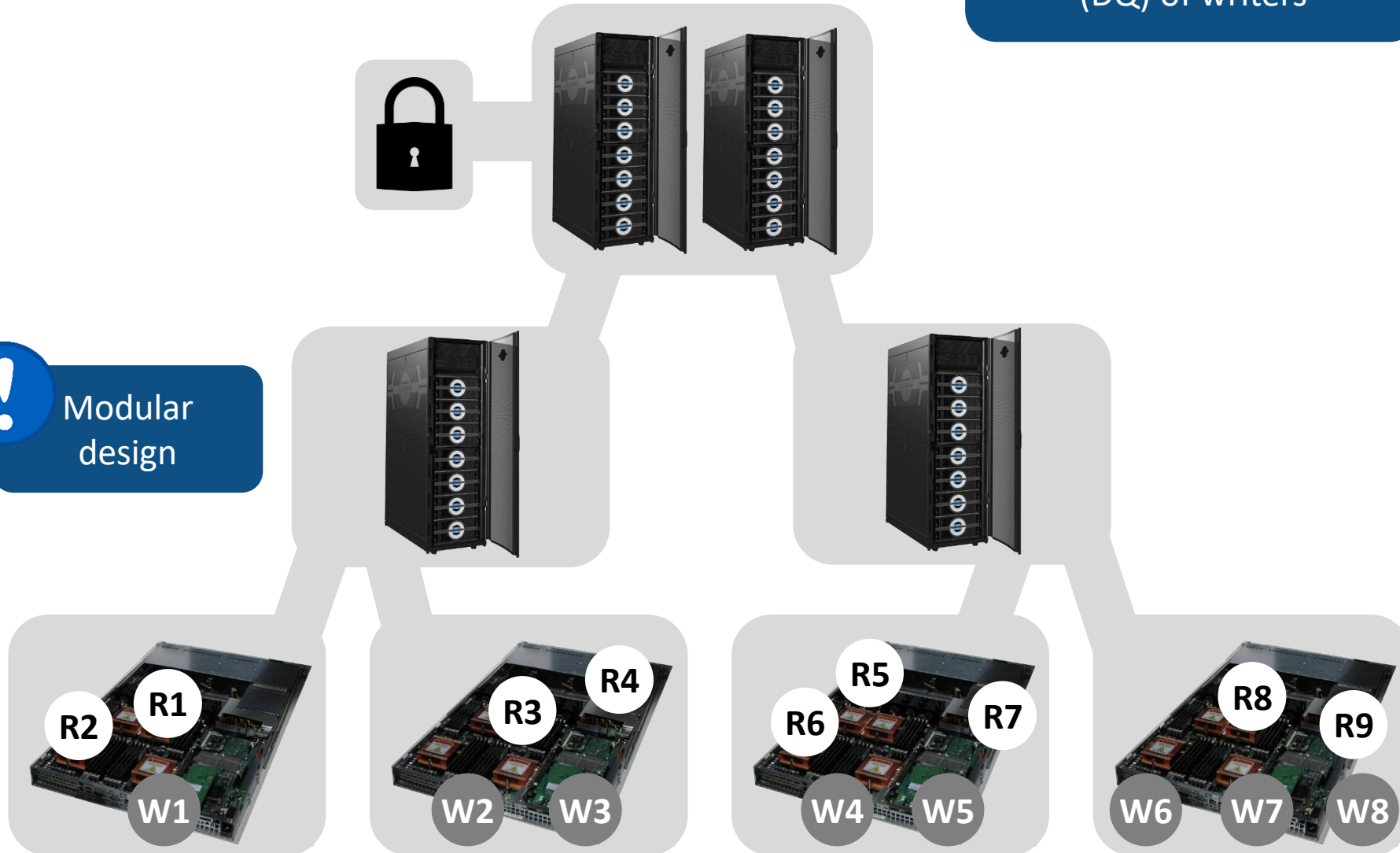
? How to manage the design complexity?



? How to manage the design complexity?

! Each element has its own distributed MCS queue (DQ) of writers

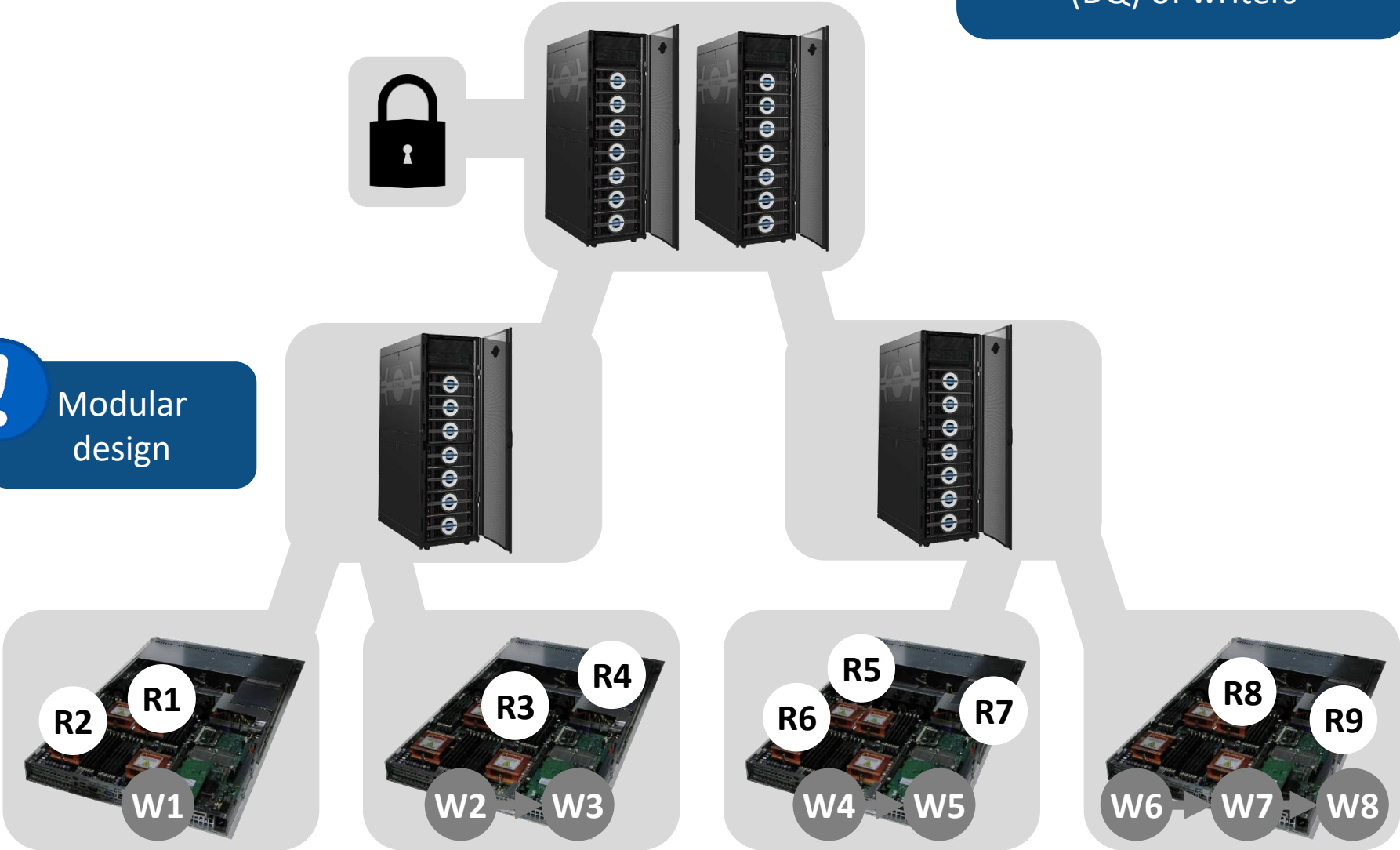
! Modular design



? How to manage the design complexity?

! Each element has its own distributed MCS queue (DQ) of writers

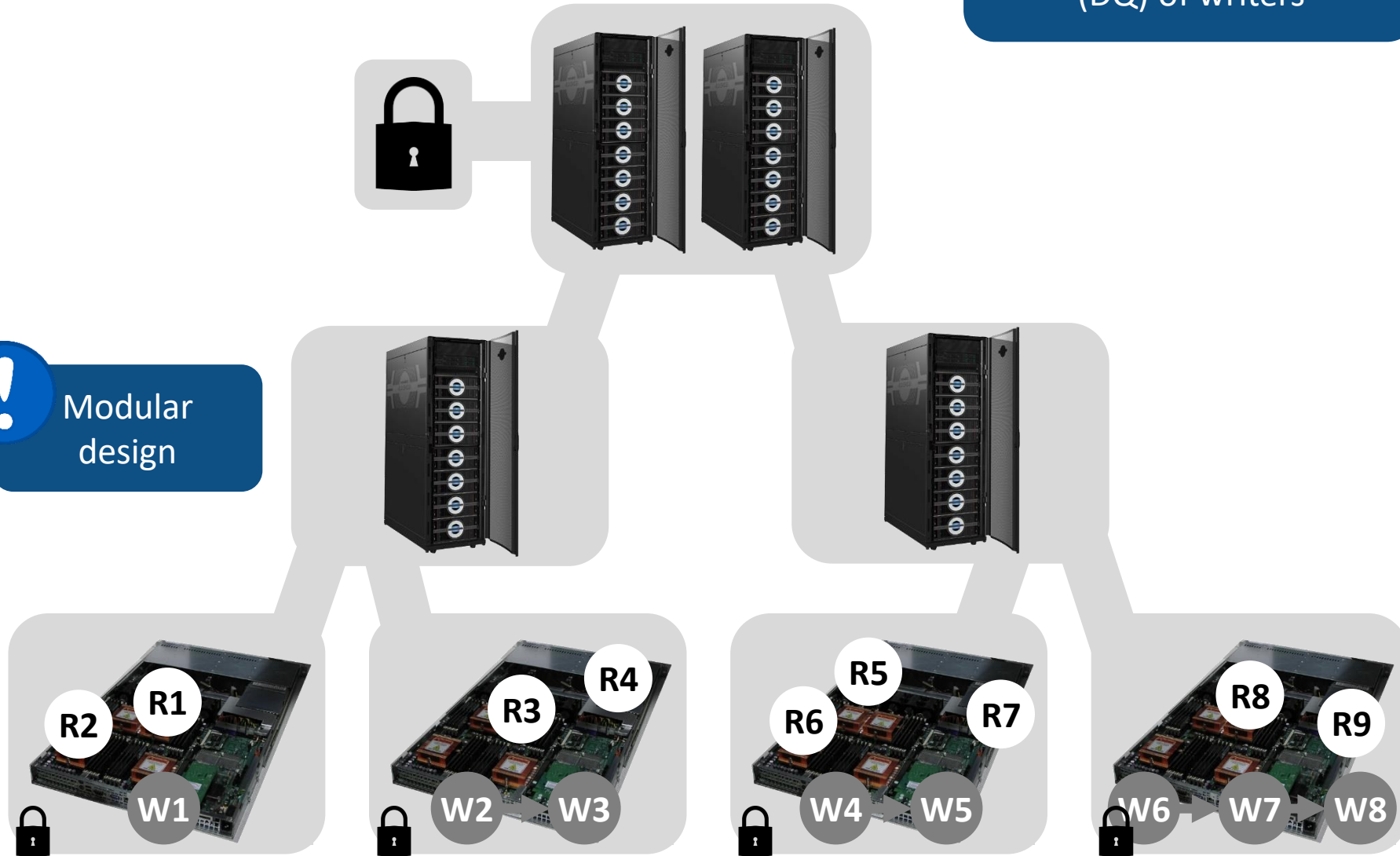
! Modular design



? How to manage the design complexity?

! Each element has its own distributed MCS queue (DQ) of writers

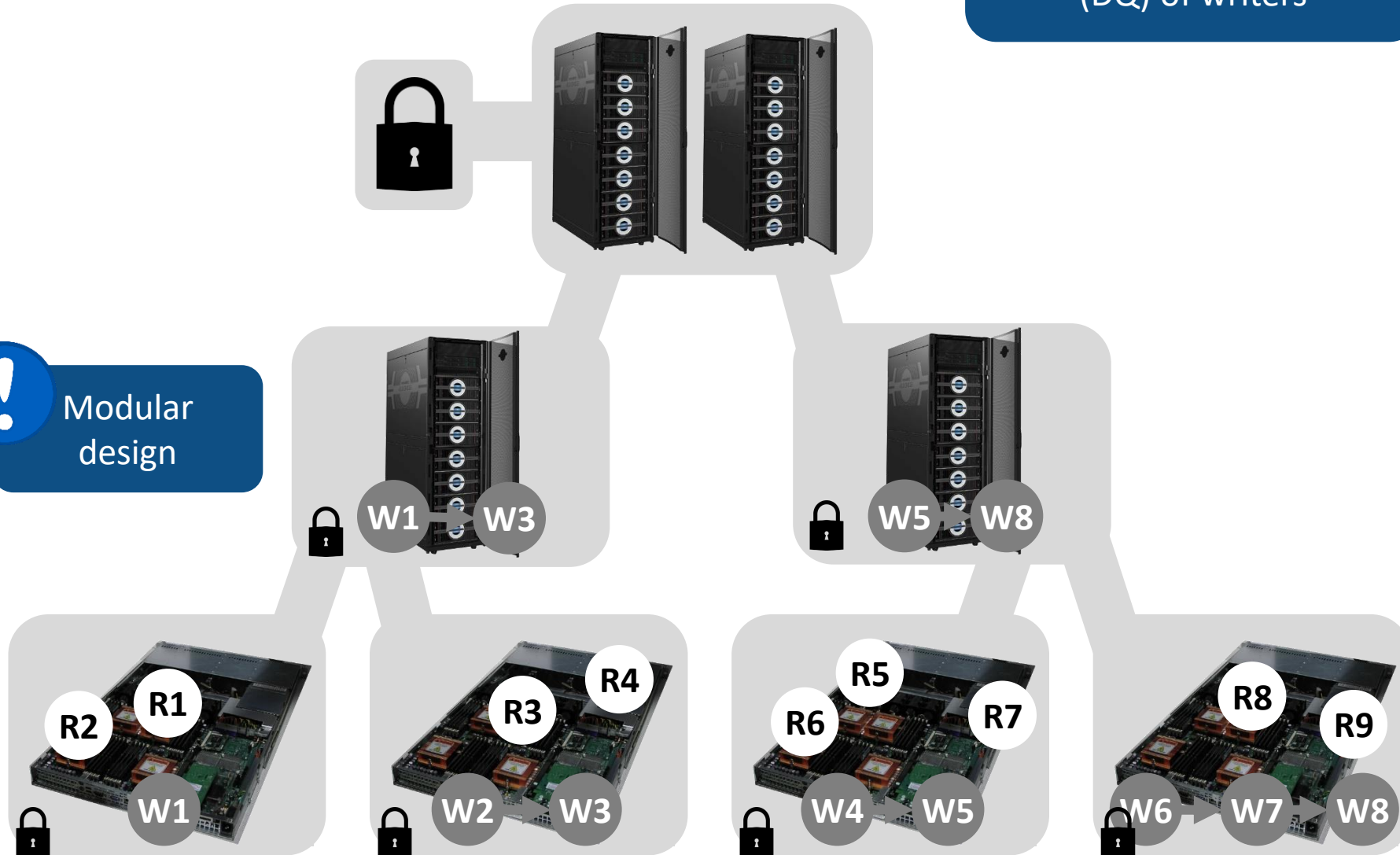
! Modular design



? How to manage the design complexity?

! Each element has its own distributed MCS queue (DQ) of writers

! Modular design





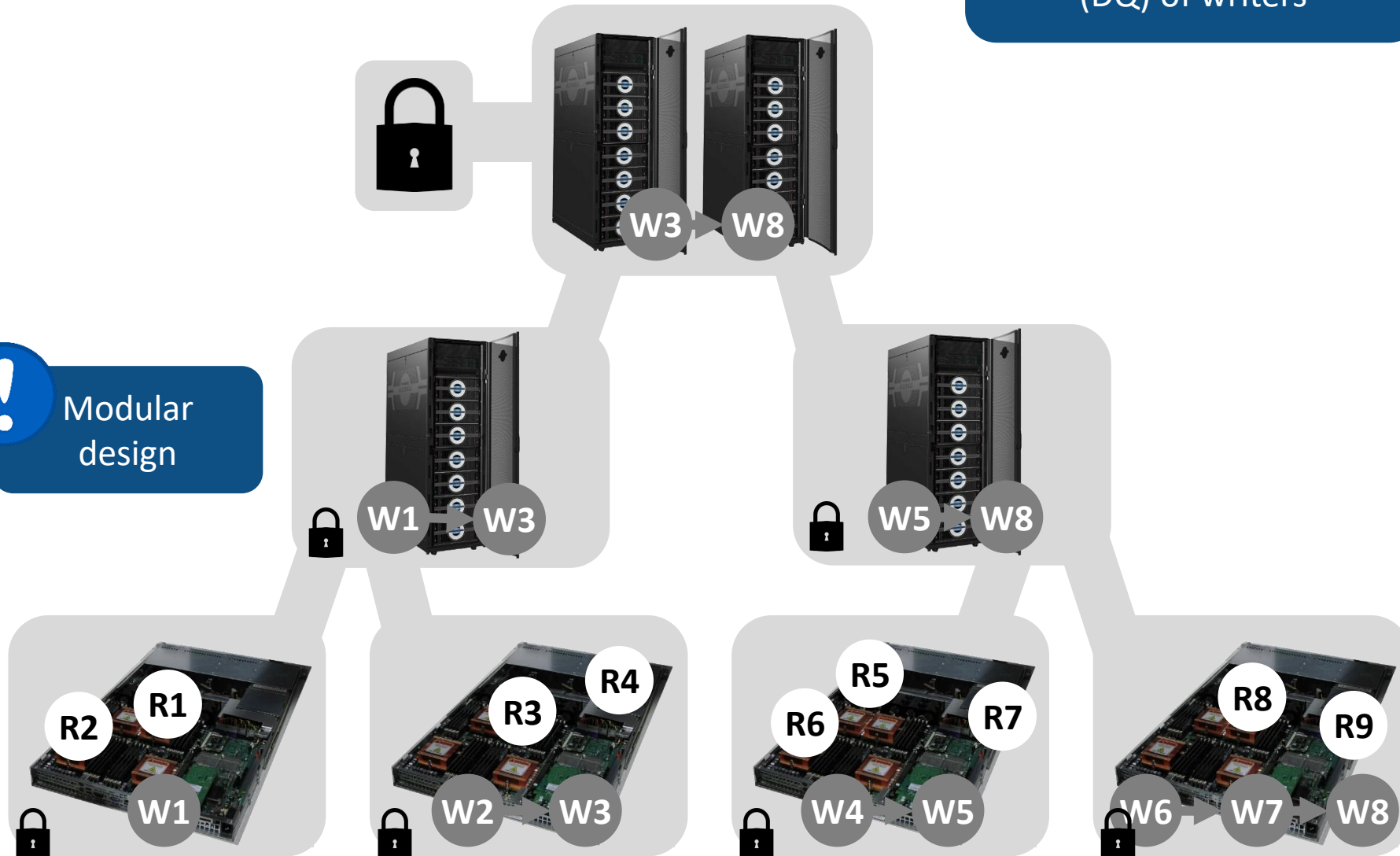
How to manage the design complexity?



Each element has its own distributed MCS queue (DQ) of writers



Modular design

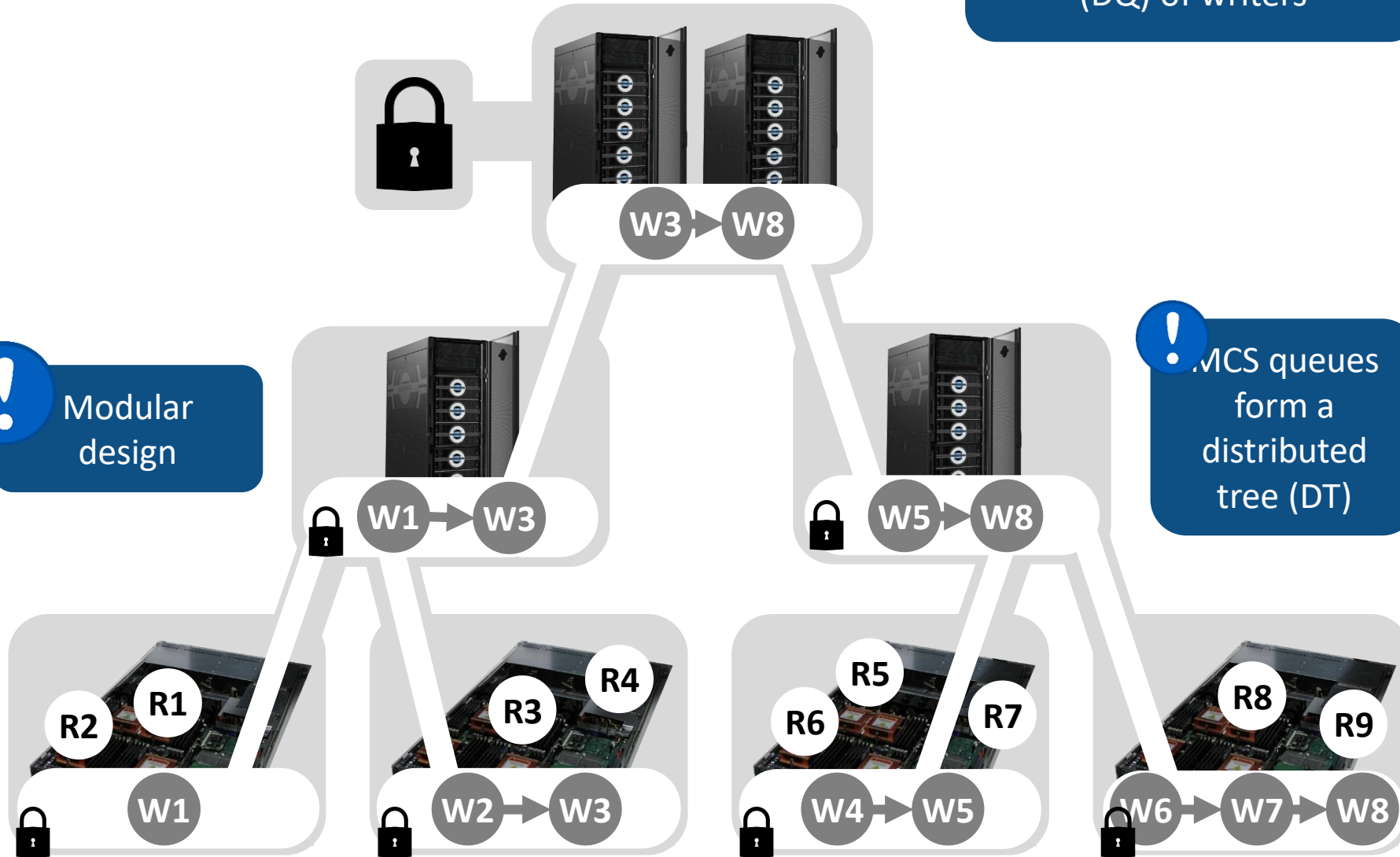


? How to manage the design complexity?

! Each element has its own distributed MCS queue (DQ) of writers

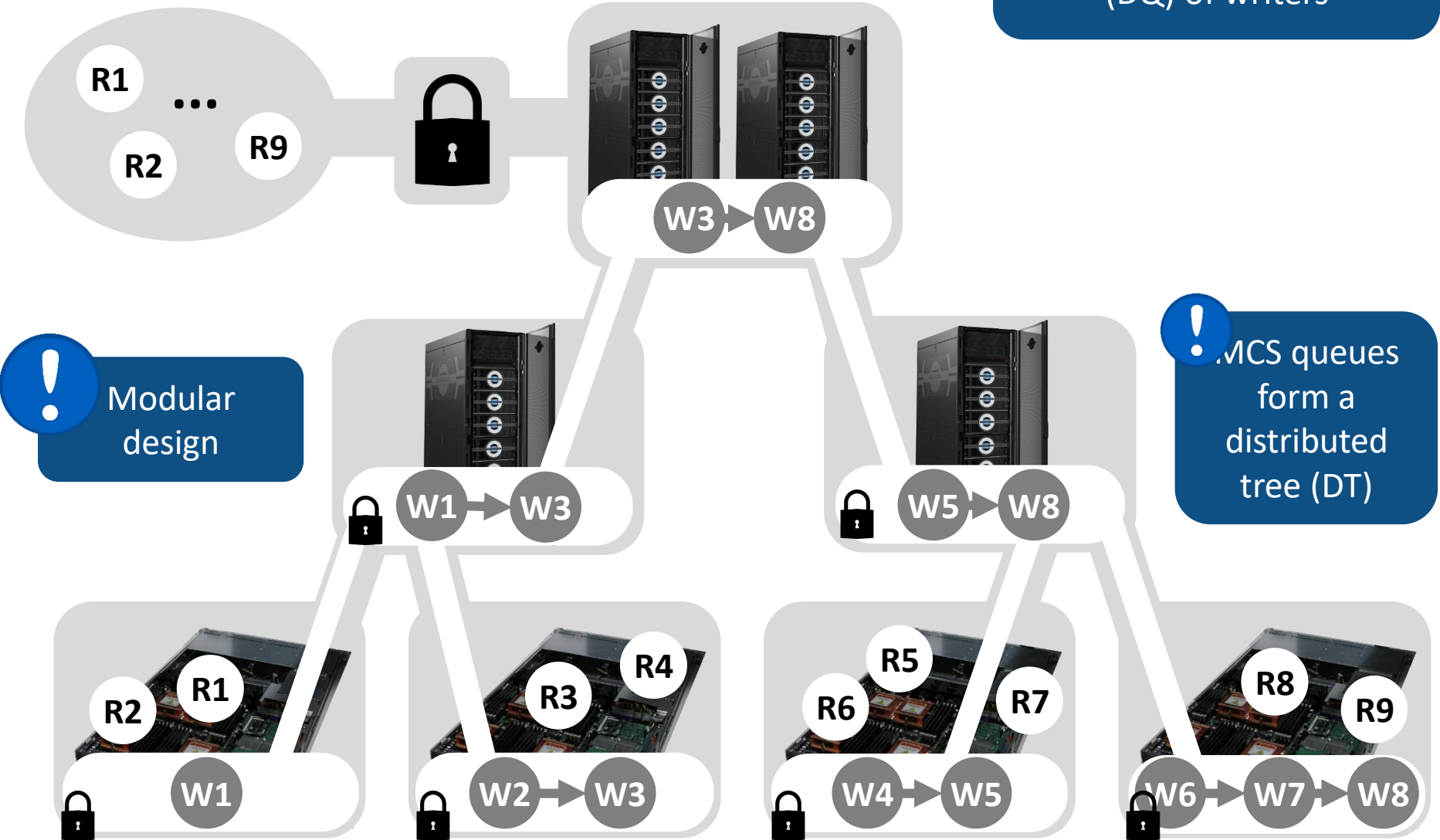
! Modular design

! MCS queues form a distributed tree (DT)



? How to manage the design complexity?

! Each element has its own distributed MCS queue (DQ) of writers

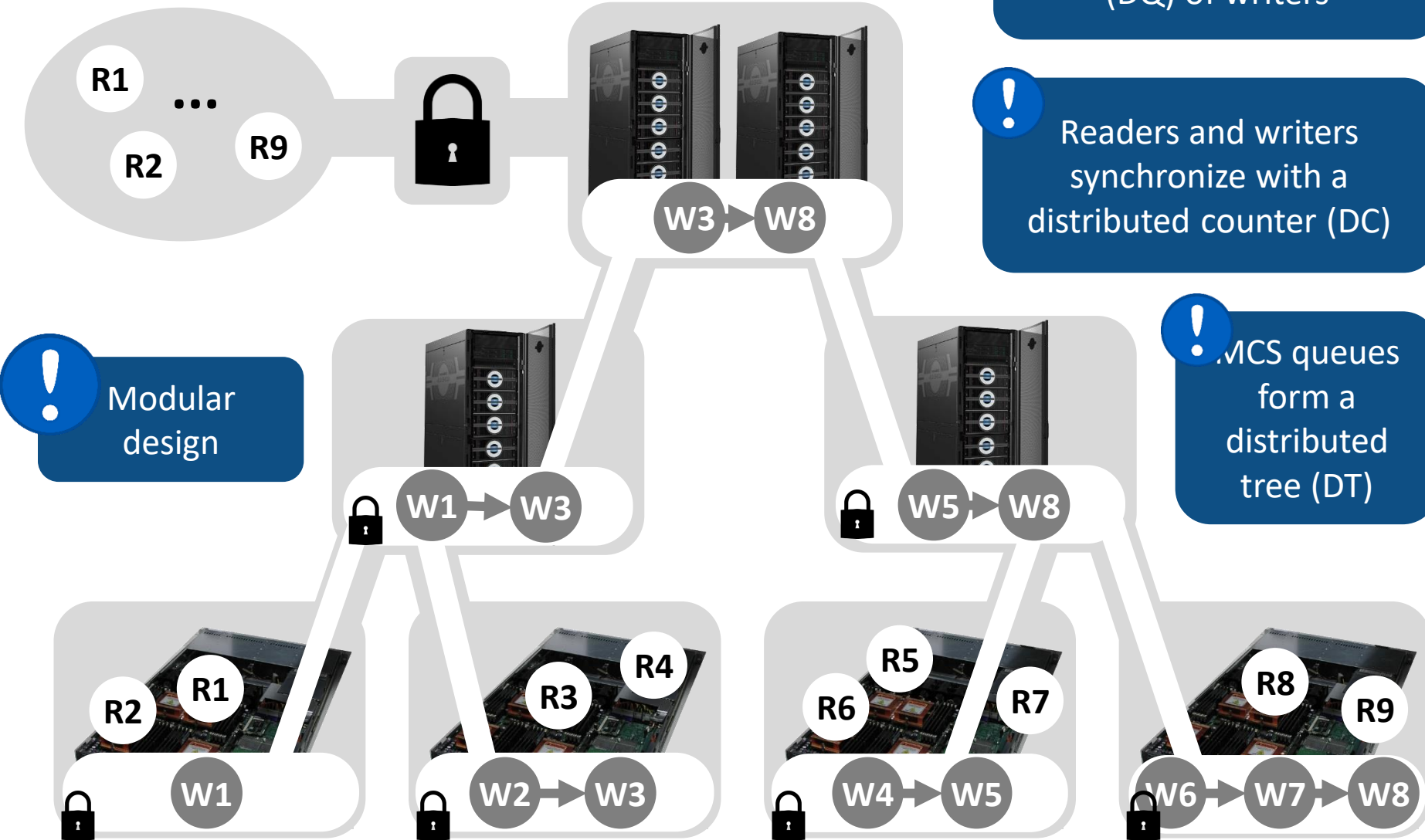


? How to manage the design complexity?

! Each element has its own distributed MCS queue (DQ) of writers

! Readers and writers synchronize with a distributed counter (DC)

! MCS queues form a distributed tree (DT)



! Modular design

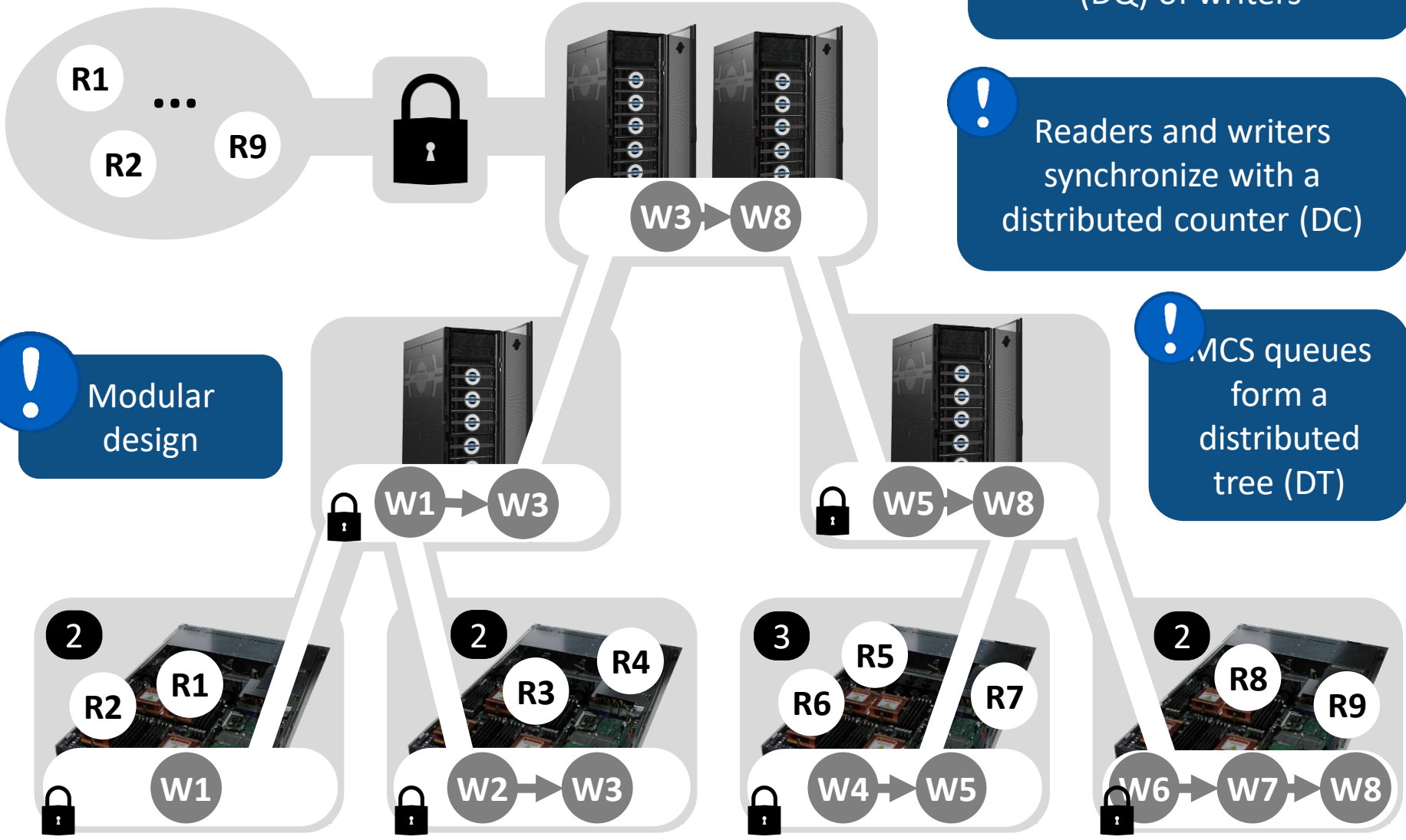
? How to manage the design complexity?

! Each element has its own distributed MCS queue (DQ) of writers

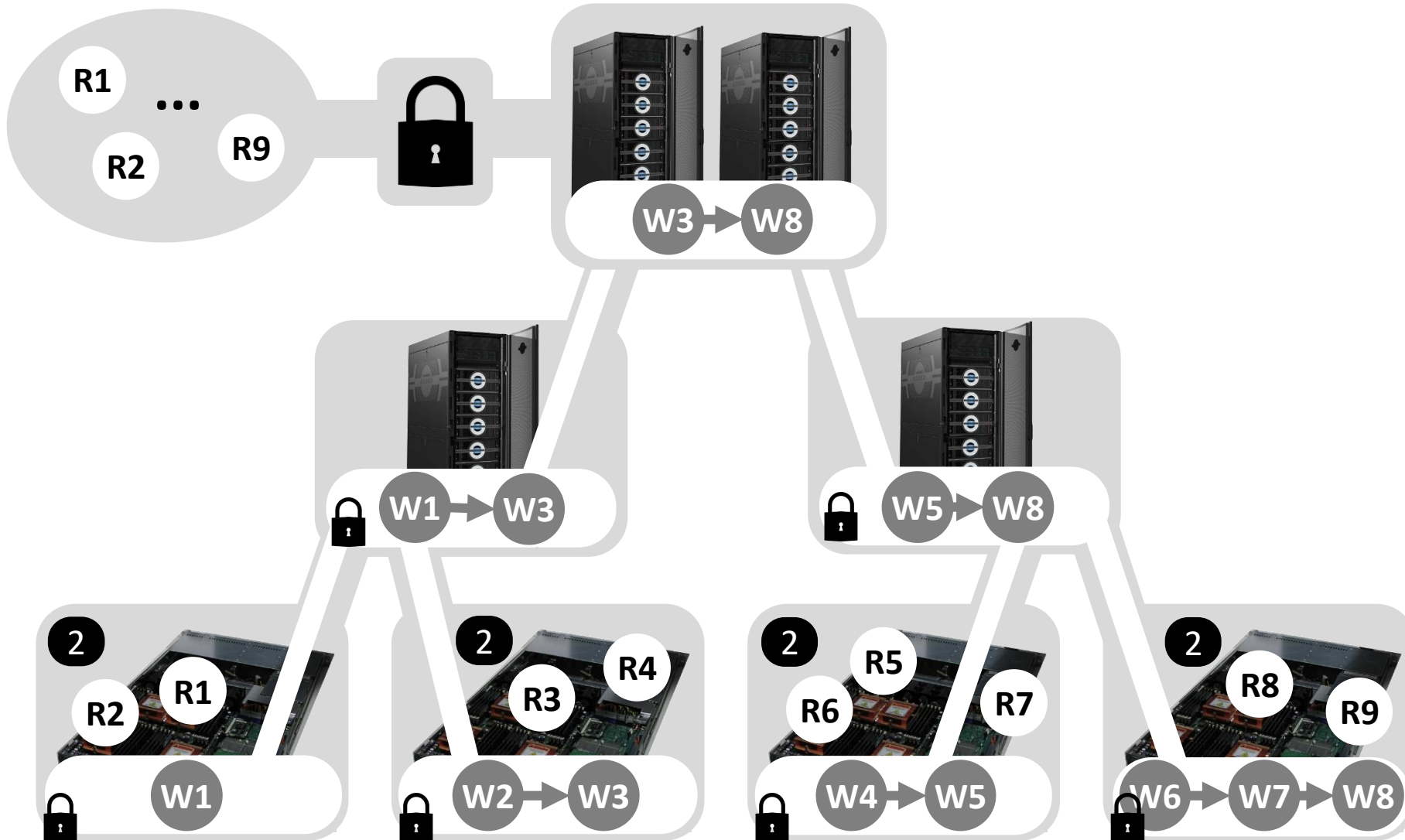
! Readers and writers synchronize with a distributed counter (DC)

! MCS queues form a distributed tree (DT)

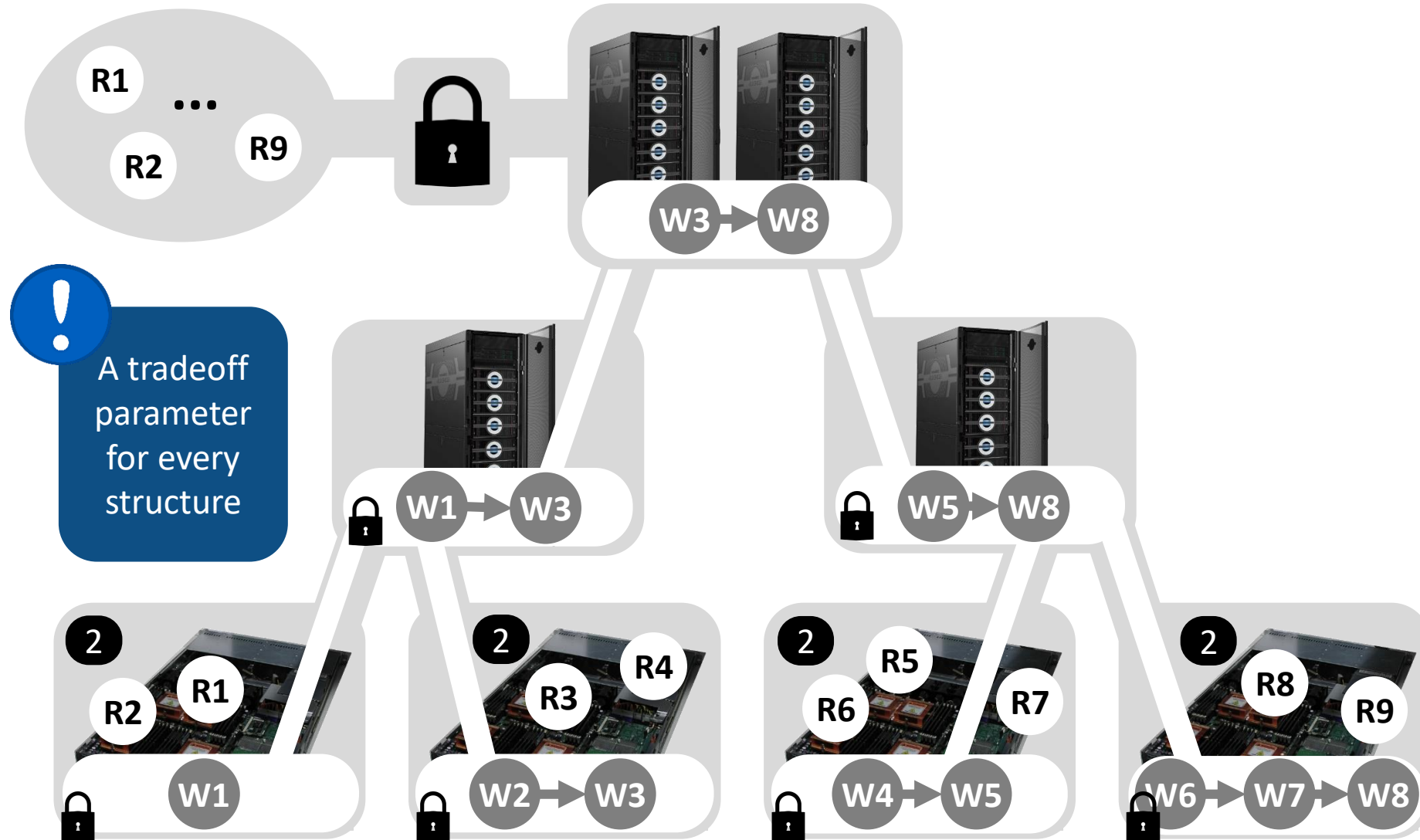
! Modular design



How to ensure tunable performance?

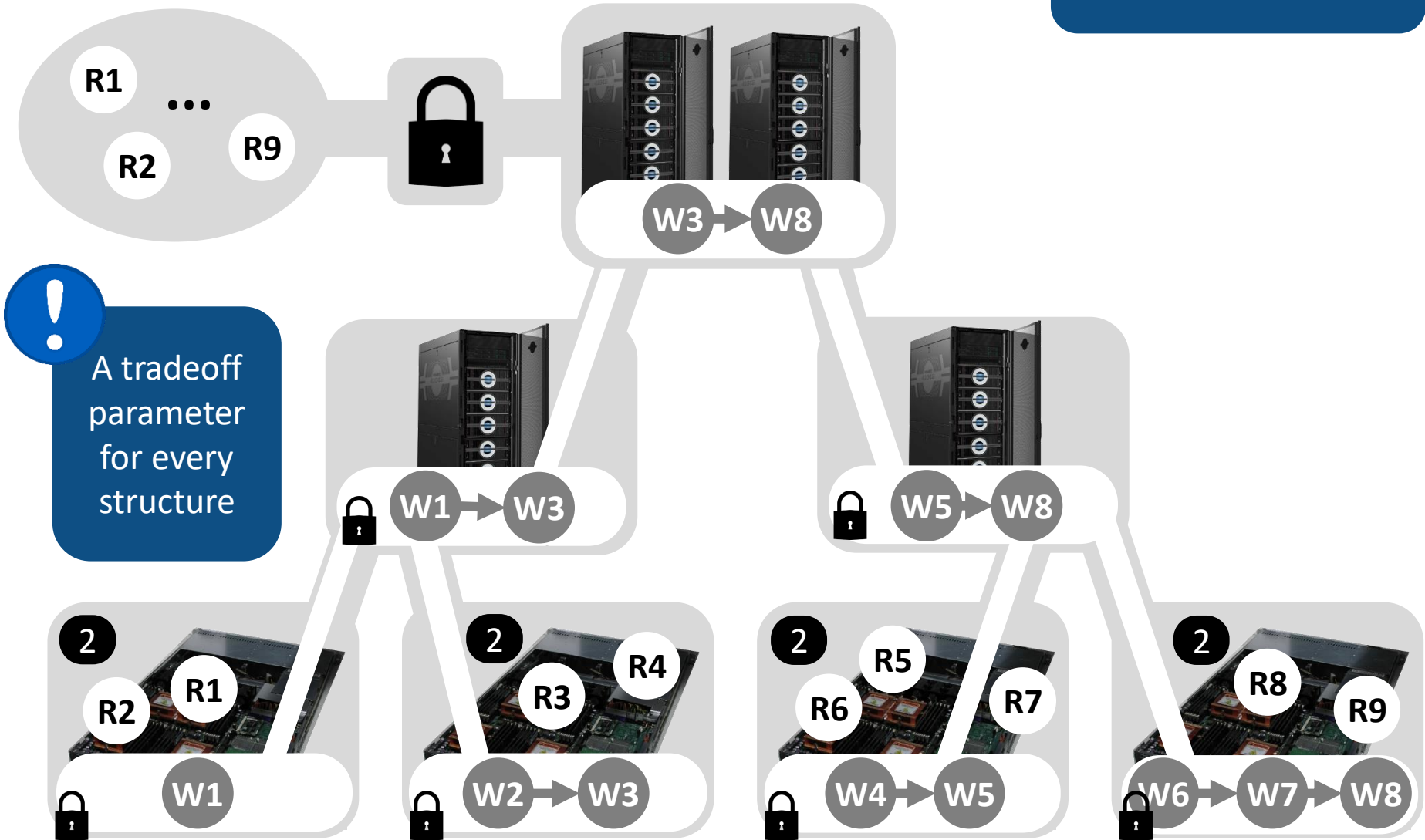


? How to ensure tunable performance?



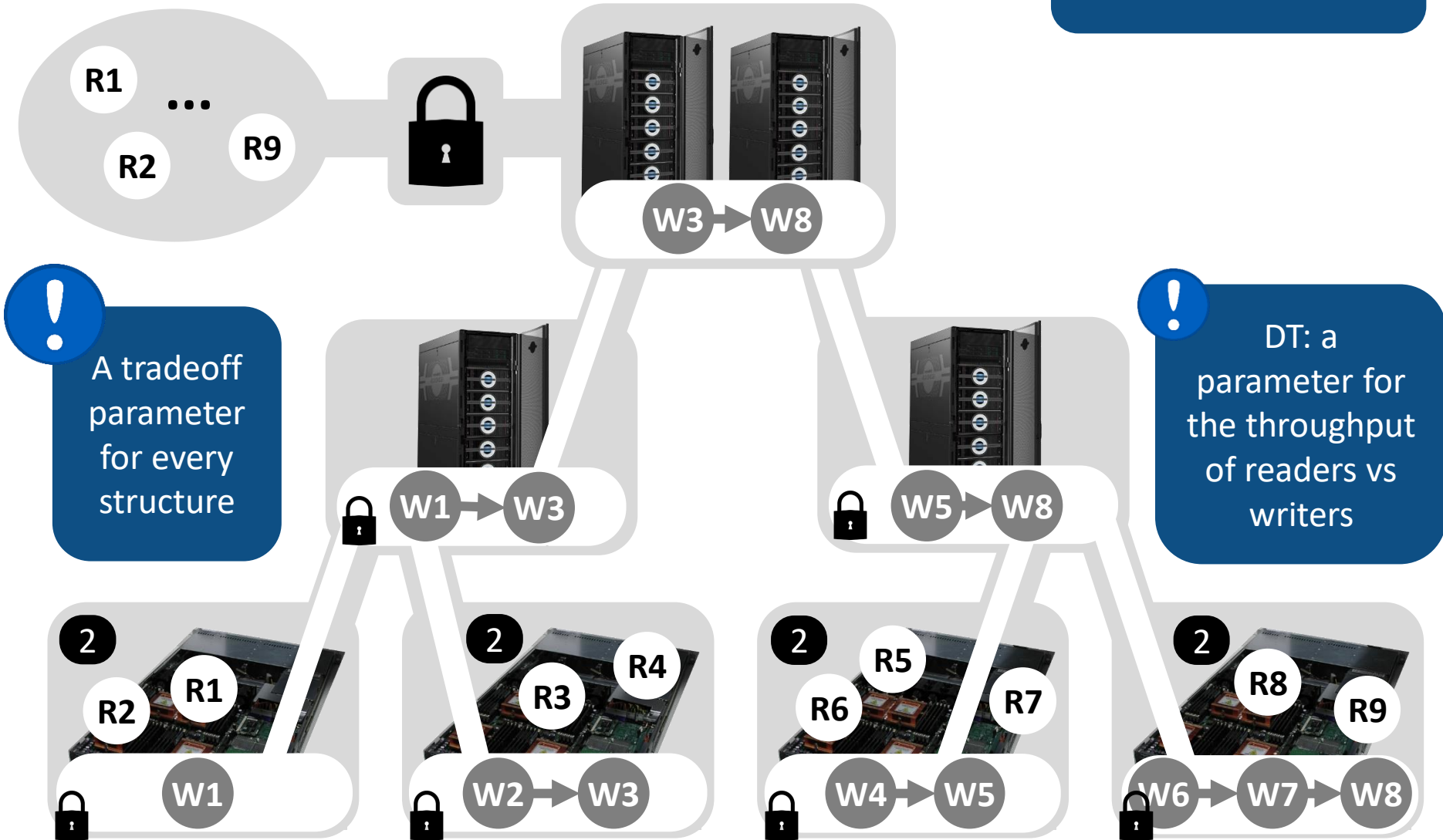
How to ensure tunable performance?

Each DQ: fairness vs throughput of writers



How to ensure tunable performance?

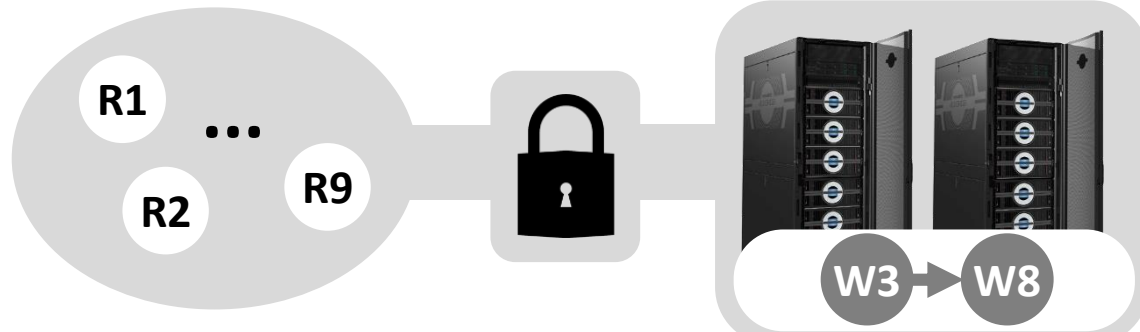
Each DQ: fairness vs throughput of writers



A tradeoff parameter for every structure

DT: a parameter for the throughput of readers vs writers

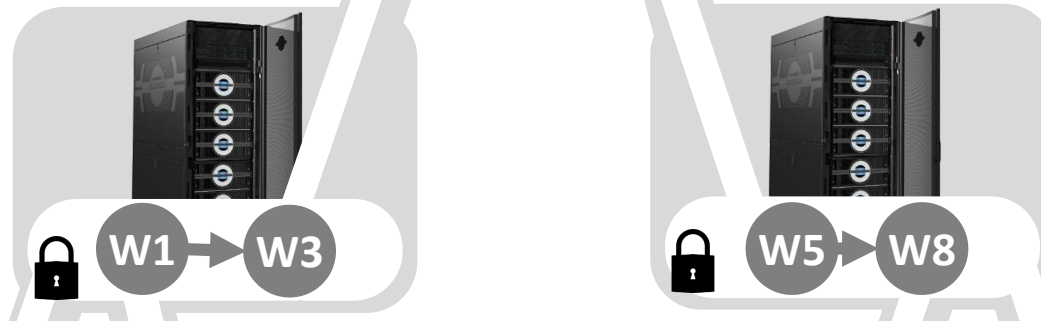
? How to ensure tunable performance?



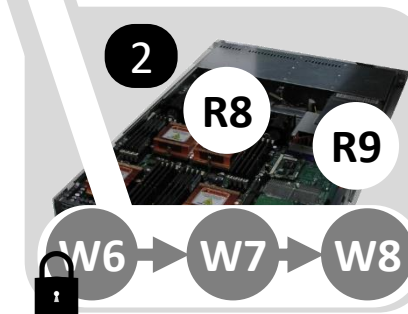
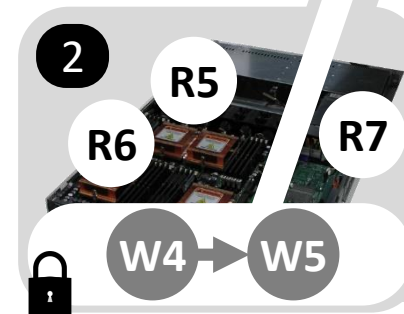
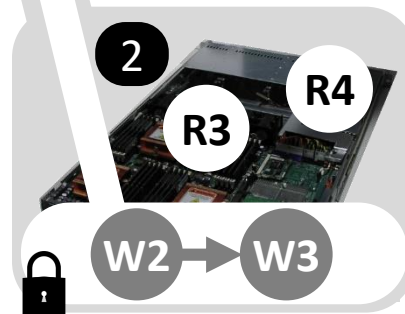
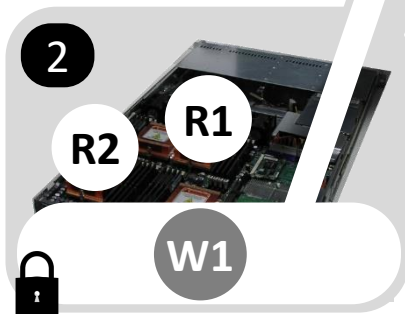
! Each DQ: fairness vs throughput of writers

! DC: a parameter for the latency of readers vs writers

! A tradeoff parameter for every structure



! DT: a parameter for the throughput of readers vs writers



Distributed MCS Queues (DQs) - Throughput vs Fairness



Distributed MCS Queues (DQs) - Throughput vs Fairness



! Each DQ: The maximum number of lock passings within a DQ at level i , before it is passed to another $T_{L,i}$ DQ at i .

Distributed MCS Queues (DQs) - Throughput vs Fairness



! Each DQ: The maximum number of lock passings within a DQ at level i , before it is passed to another $T_{L,i}$ DQ at i .

Distributed MCS Queues (DQs) - Throughput vs Fairness



! Each DQ: The maximum number of lock passings within a DQ at level i , before it is passed to another $T_{L,i}$ DQ at i .

Distributed MCS Queues (DQs) - Throughput vs Fairness

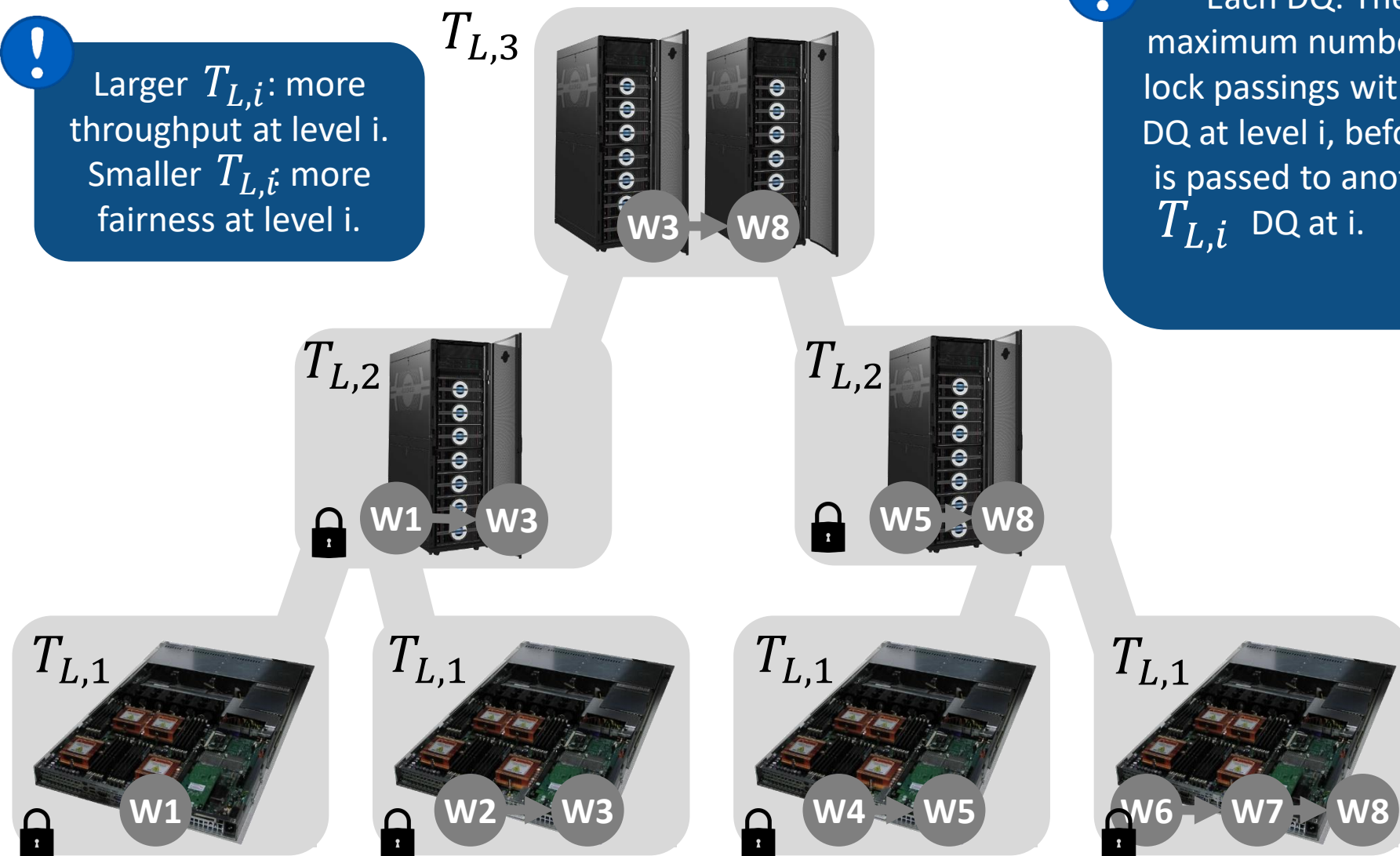


! Each DQ: The maximum number of lock passings within a DQ at level i , before it is passed to another $T_{L,i}$ DQ at i .

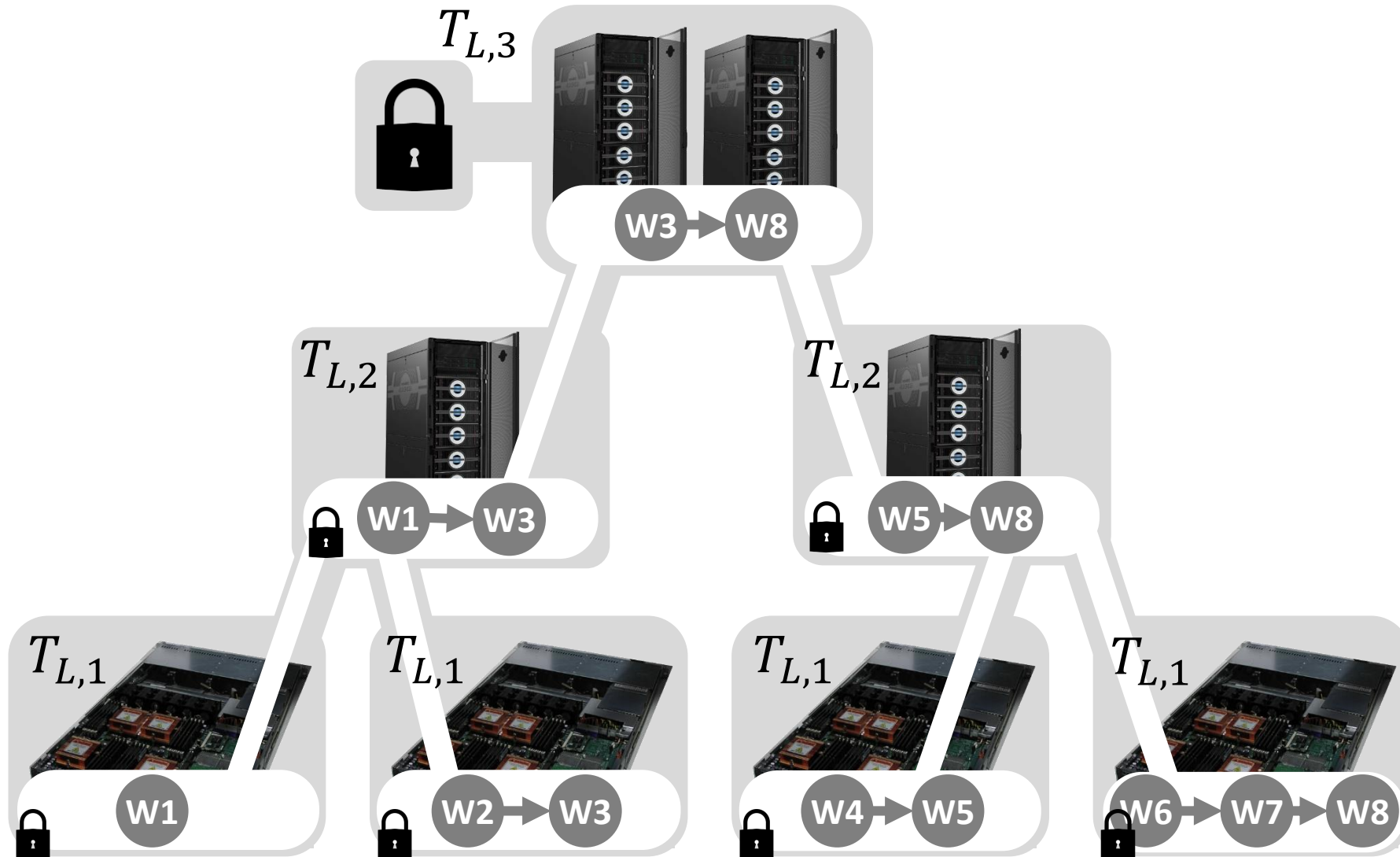
Distributed MCS Queues (DQs) - Throughput vs Fairness

! Larger $T_{L,i}$: more throughput at level i .
Smaller $T_{L,i}$: more fairness at level i .

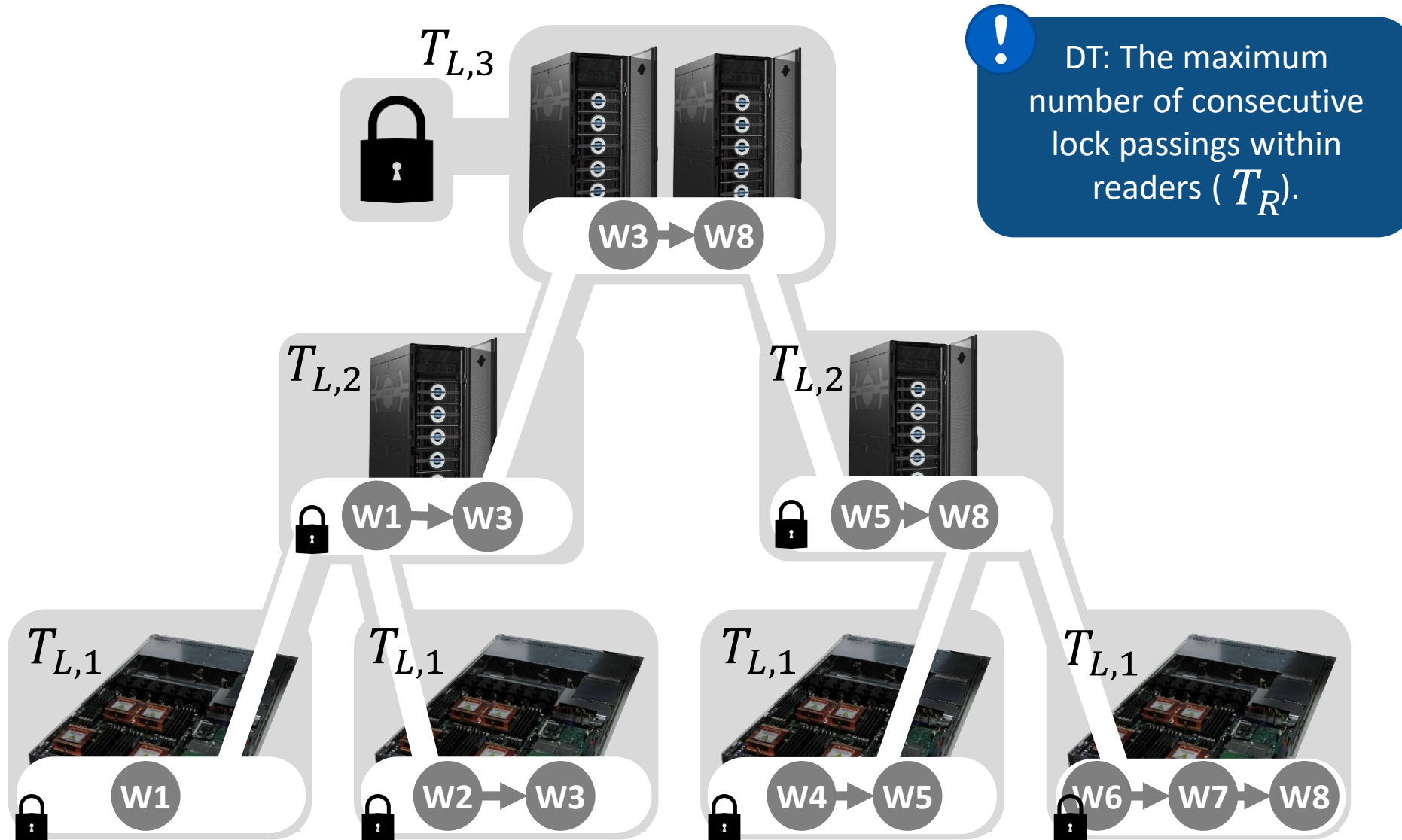
! Each DQ: The maximum number of lock passings within a DQ at level i , before it is passed to another $T_{L,i}$ DQ at i .



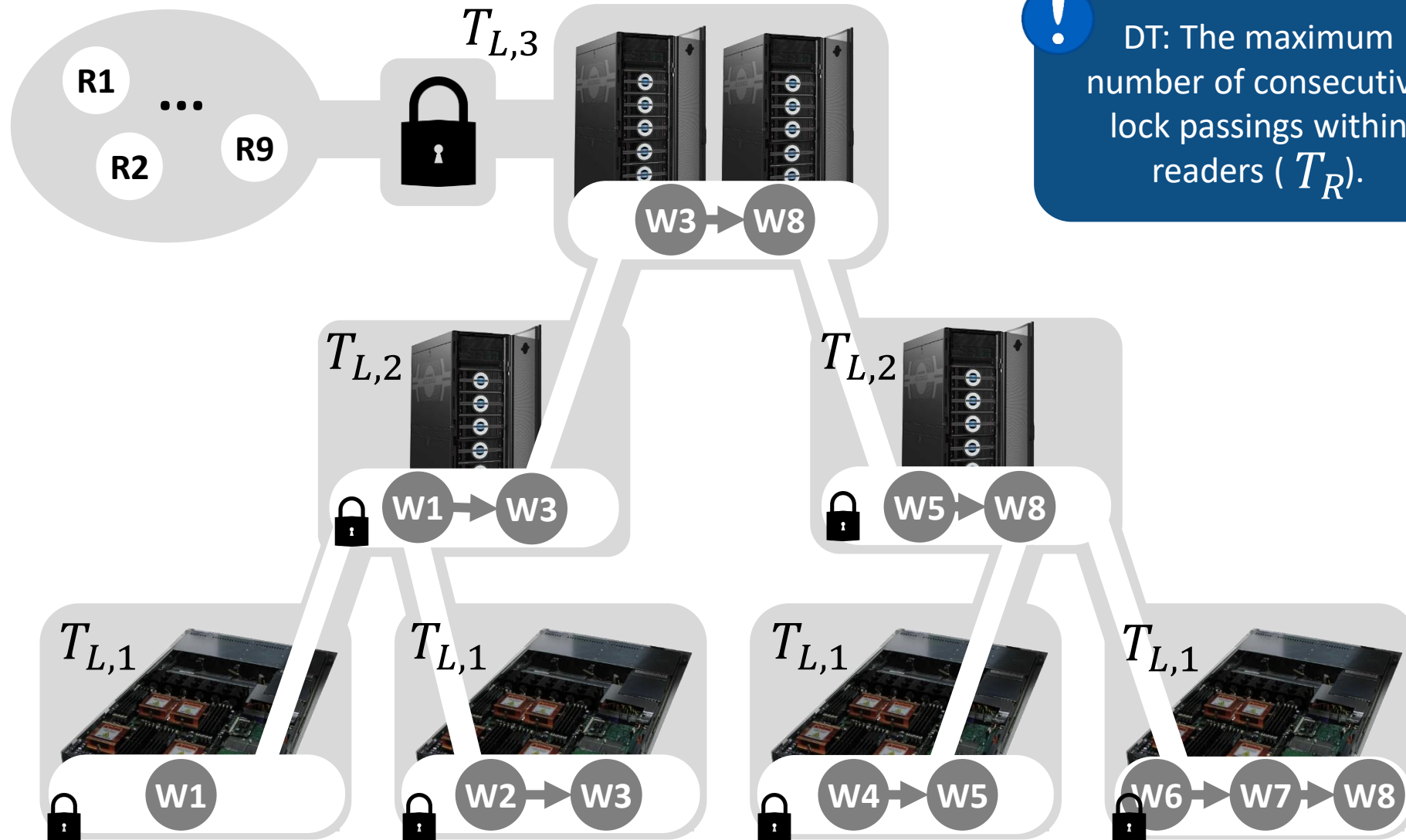
Distributed Tree of Queues (DT) - Throughput of readers vs writers



Distributed Tree of Queues (DT) - Throughput of readers vs writers



Distributed Tree of Queues (DT) - Throughput of readers vs writers



! DT: The maximum number of consecutive lock passings within readers (T_R).

Distributed Counter (DC) - Latency of readers vs writers



Distributed Counter (DC) - Latency of readers vs writers



DC: every k th compute node hosts a partial counter, all of which constitute the DC.

! $k = T_{DC}$

Distributed Counter (DC) - Latency of readers vs writers



DC: every k th compute node hosts a partial counter, all of which constitute the DC.



$$k = T_{DC}$$

$b|x|y$

Distributed Counter (DC) - Latency of readers vs writers



DC: every k th compute node hosts a partial counter, all of which constitute the DC.

! $k = T_{DC}$

A writer holds the lock $b|x|y$

Distributed Counter (DC) - Latency of readers vs writers



DC: every k th compute node hosts a partial counter, all of which constitute the DC.

! $k = T_{DC}$

A writer holds the lock $b|x|y$

Readers that arrived at the CS

Distributed Counter (DC) - Latency of readers vs writers



DC: every k th compute node hosts a partial counter, all of which constitute the DC.

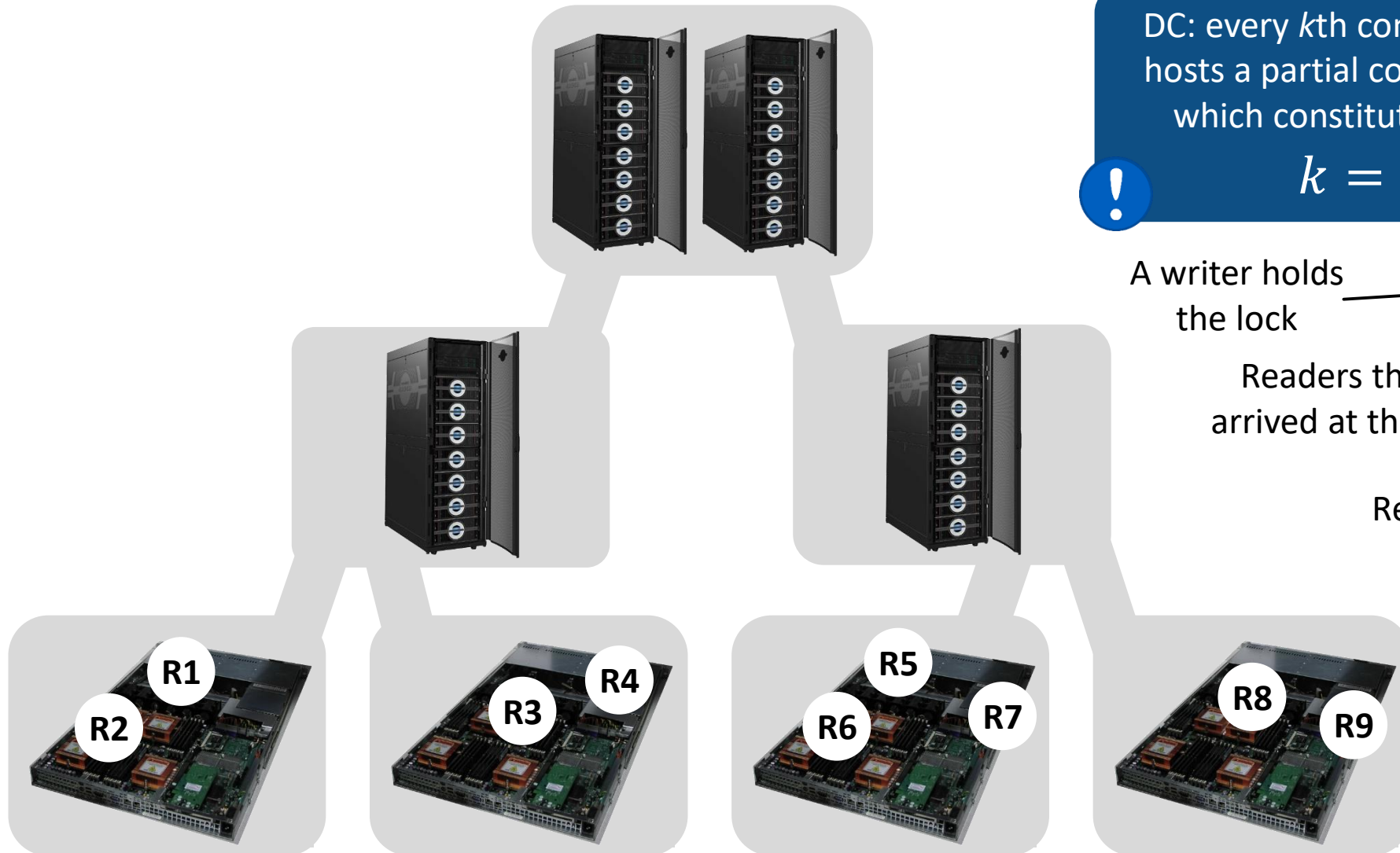
! $k = T_{DC}$

A writer holds the lock — **b|x|y**

Readers that arrived at the CS — **x**

Readers that left the CS — **y**

Distributed Counter (DC) - Latency of readers vs writers



DC: every k th compute node hosts a partial counter, all of which constitute the DC.

! $k = T_{DC}$

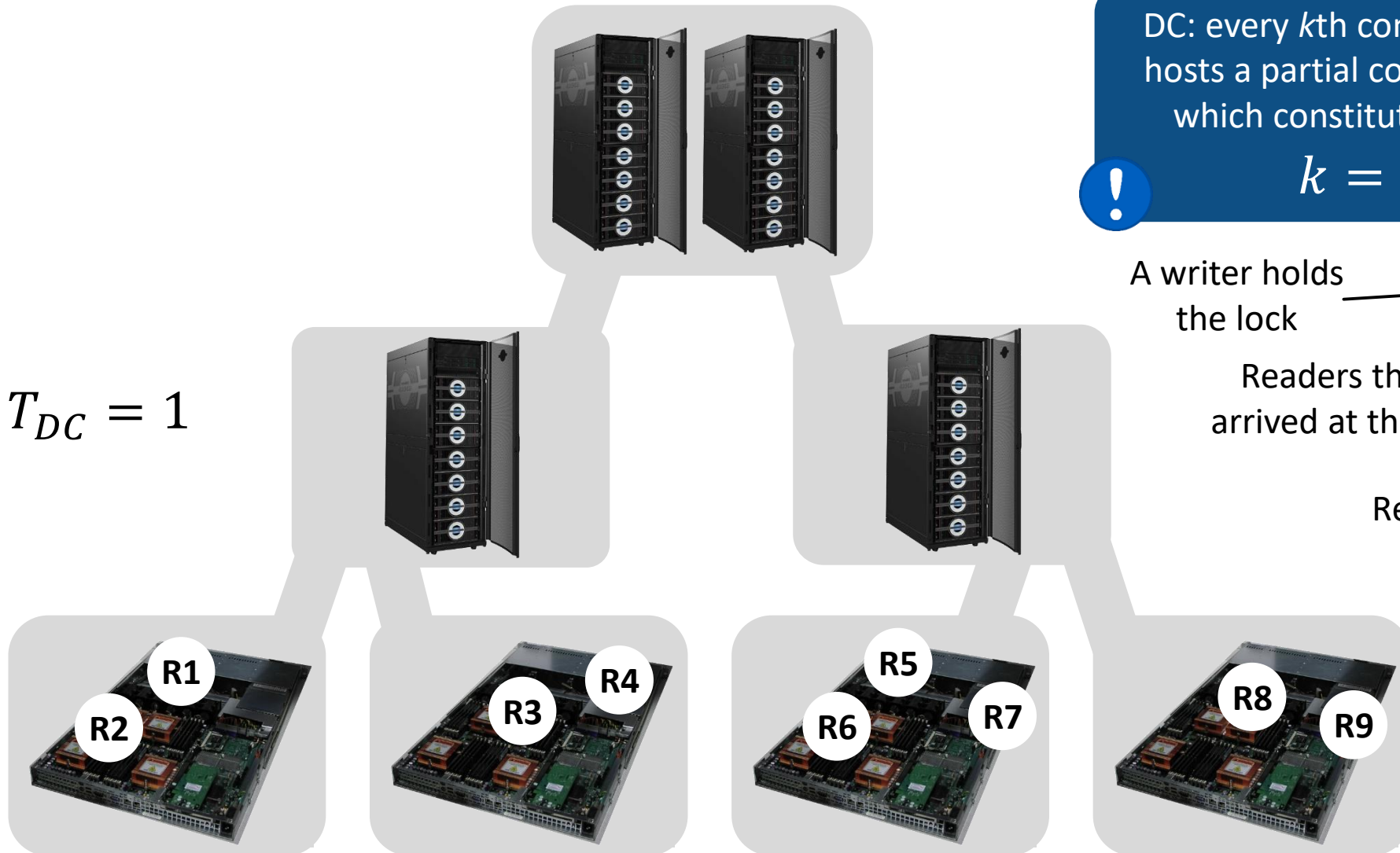
A writer holds the lock **b|x|y**

Readers that arrived at the CS

Readers that left the CS

Distributed Counter (DC) - Latency of readers vs writers

$$T_{DC} = 1$$



DC: every k th compute node hosts a partial counter, all of which constitute the DC.

! $k = T_{DC}$

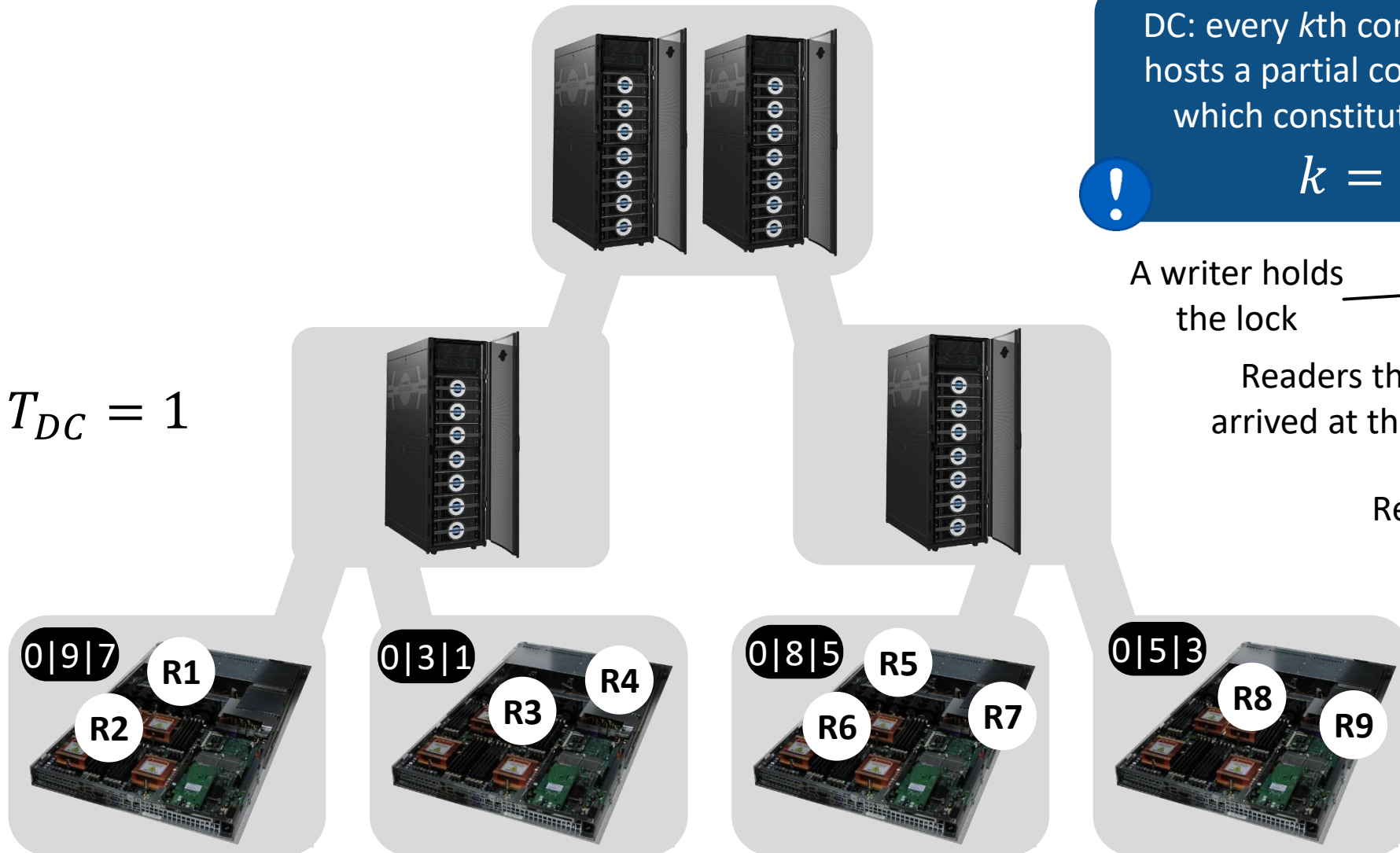
A writer holds the lock $b|x|y$

Readers that arrived at the CS

Readers that left the CS

Distributed Counter (DC) - Latency of readers vs writers

$$T_{DC} = 1$$



DC: every k th compute node hosts a partial counter, all of which constitute the DC.

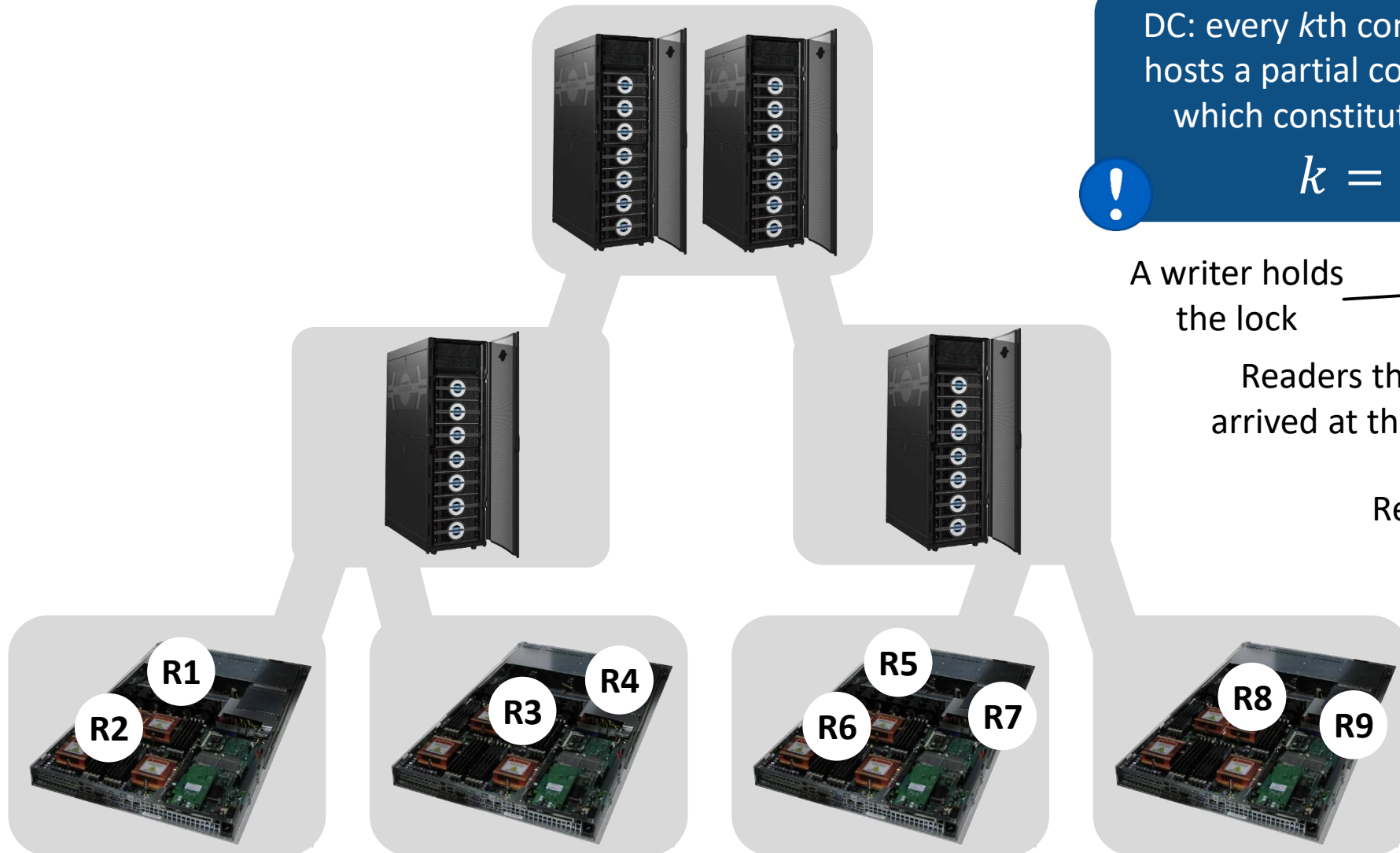
! $k = T_{DC}$

A writer holds the lock $b|x|y$

Readers that arrived at the CS

Readers that left the CS

Distributed Counter (DC) - Latency of readers vs writers



DC: every k th compute node hosts a partial counter, all of which constitute the DC.

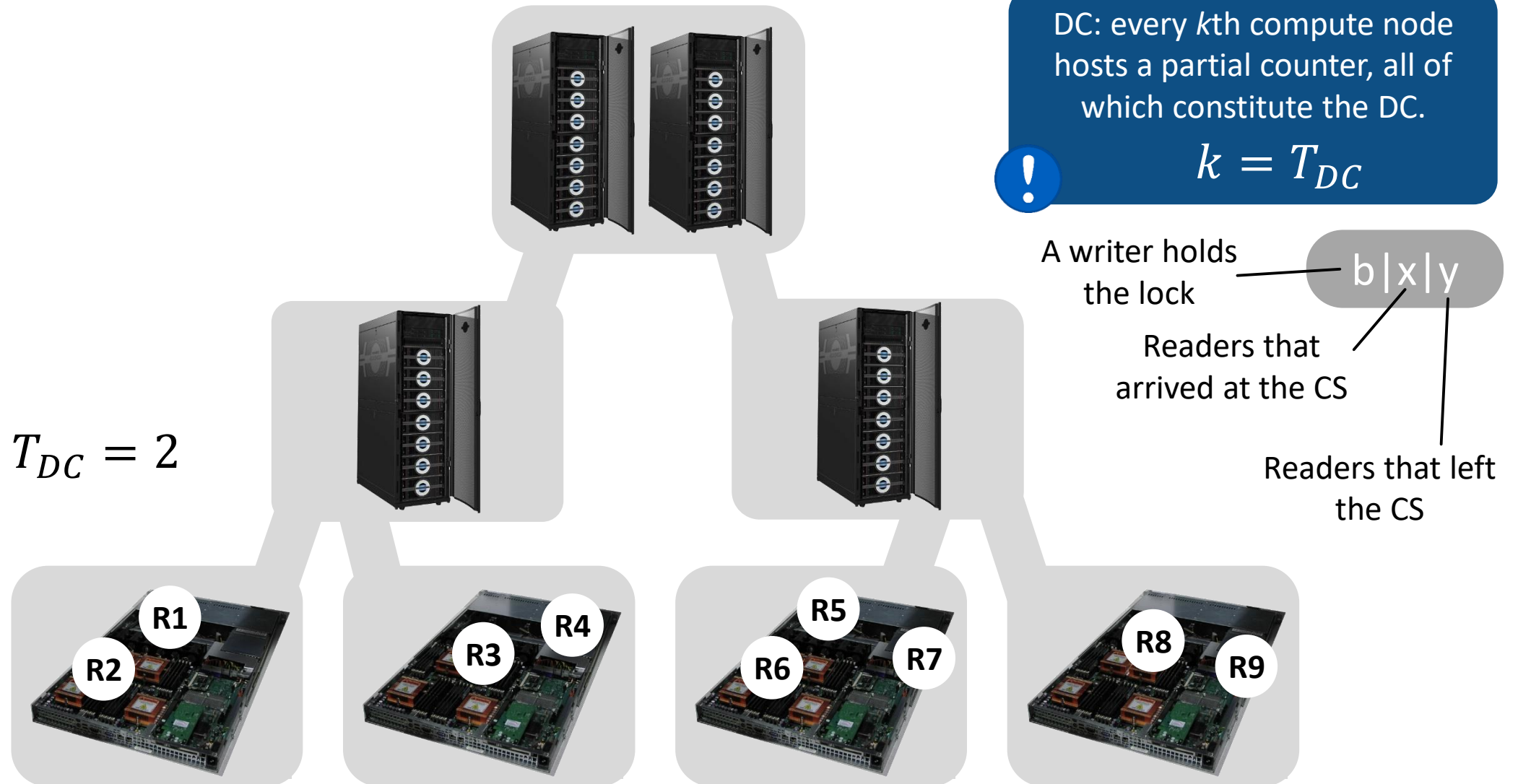
! $k = T_{DC}$

A writer holds the lock — **b|x|y**

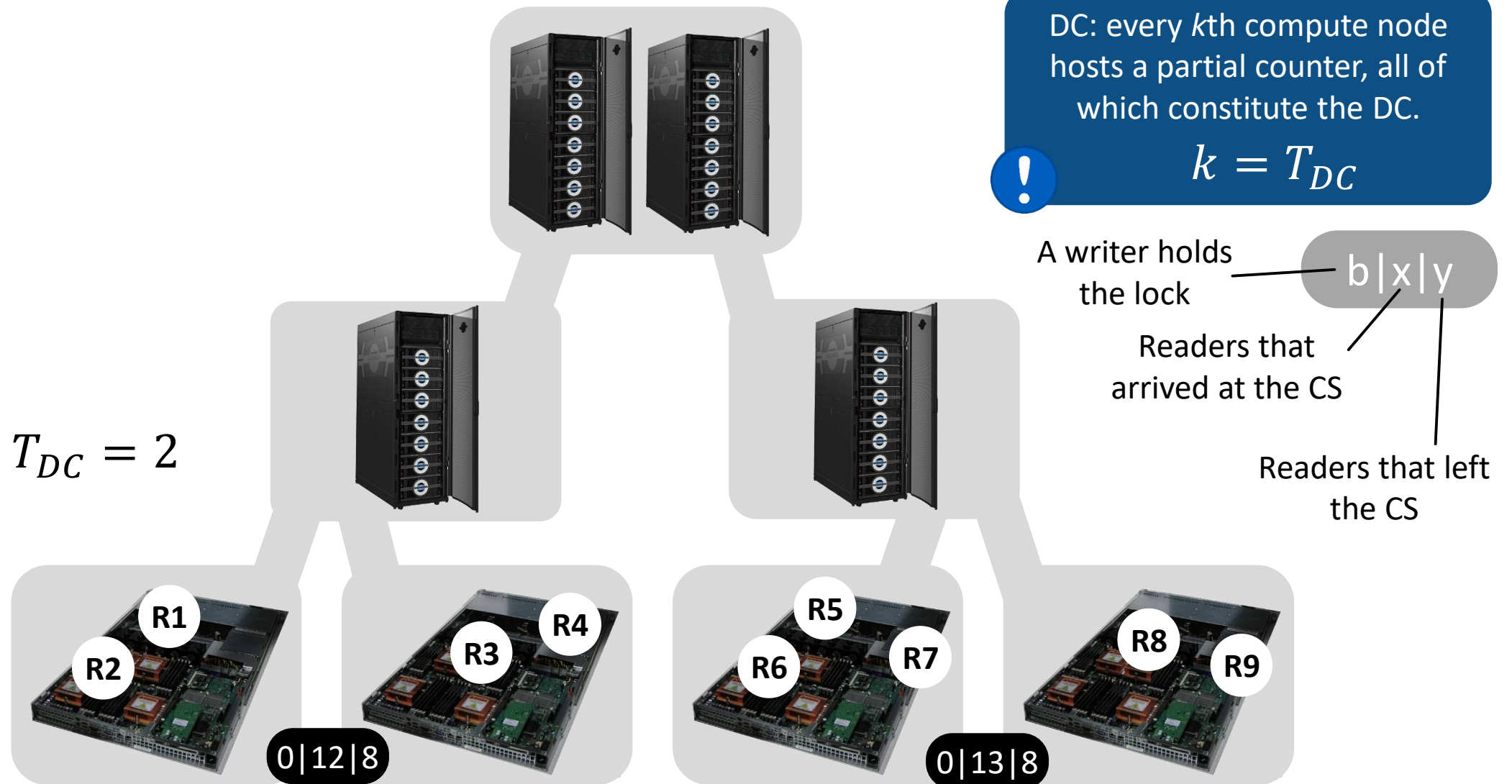
Readers that arrived at the CS — **x**

Readers that left the CS — **y**

Distributed Counter (DC) - Latency of readers vs writers

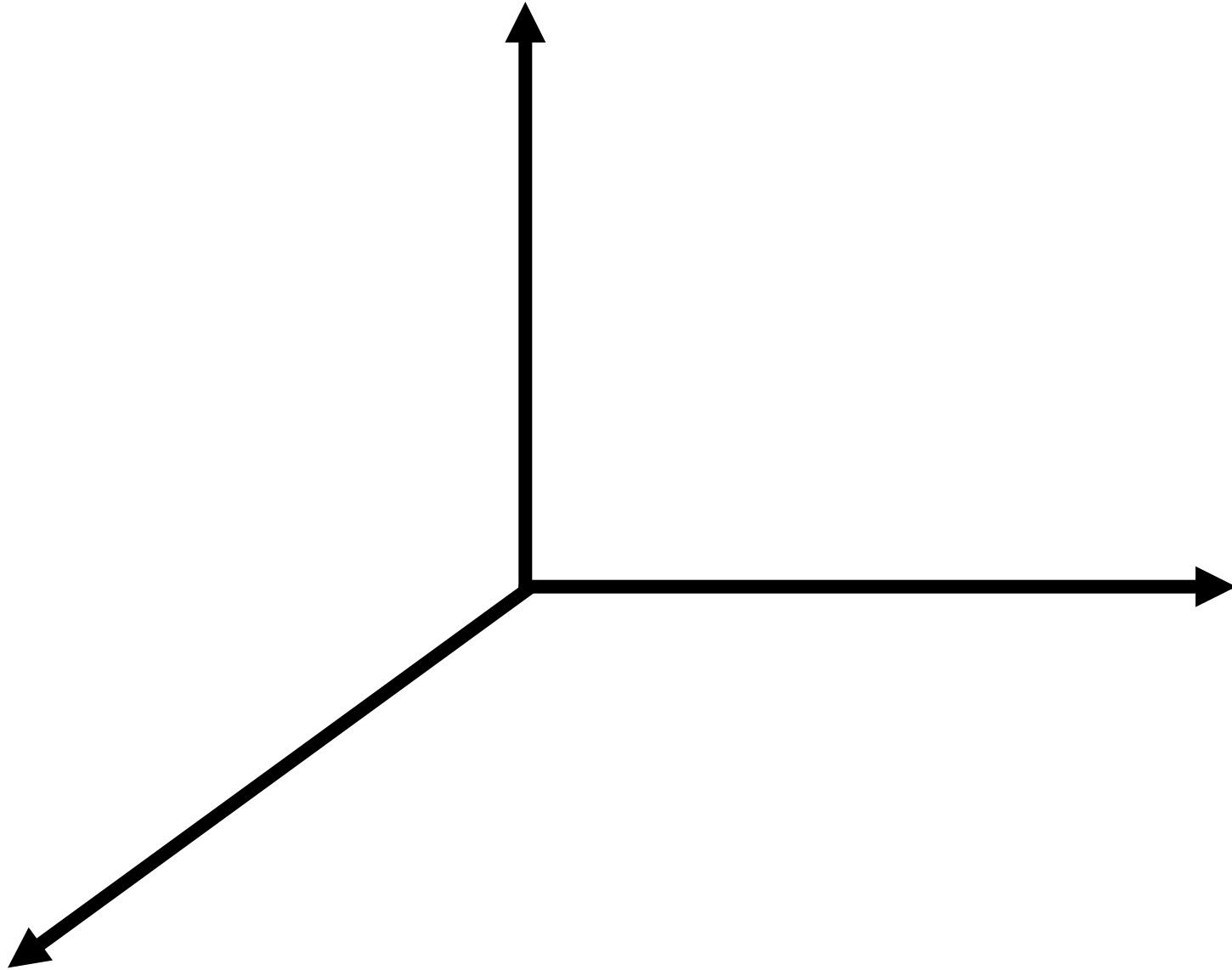


Distributed Counter (DC) - Latency of readers vs writers

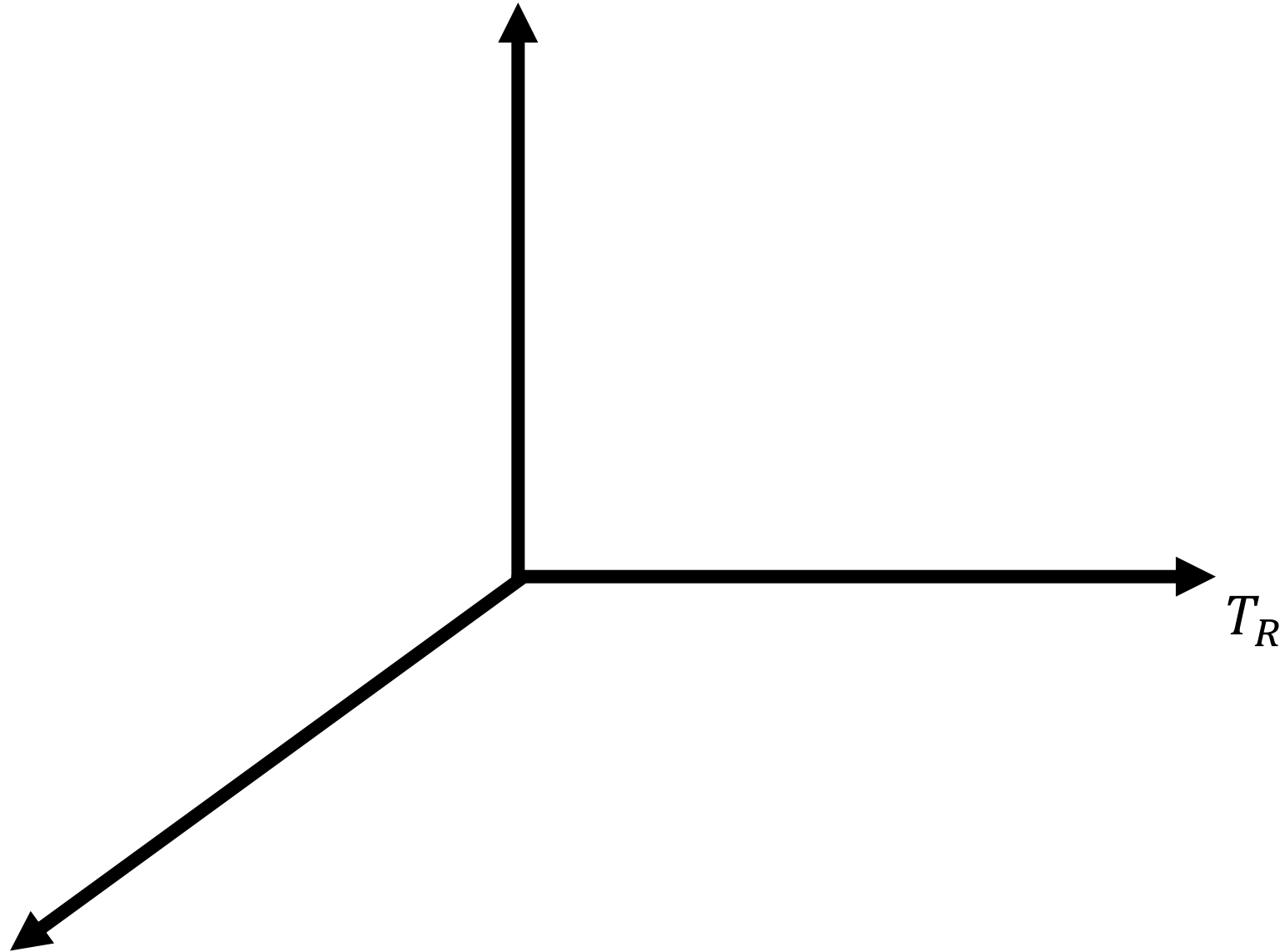


Design space

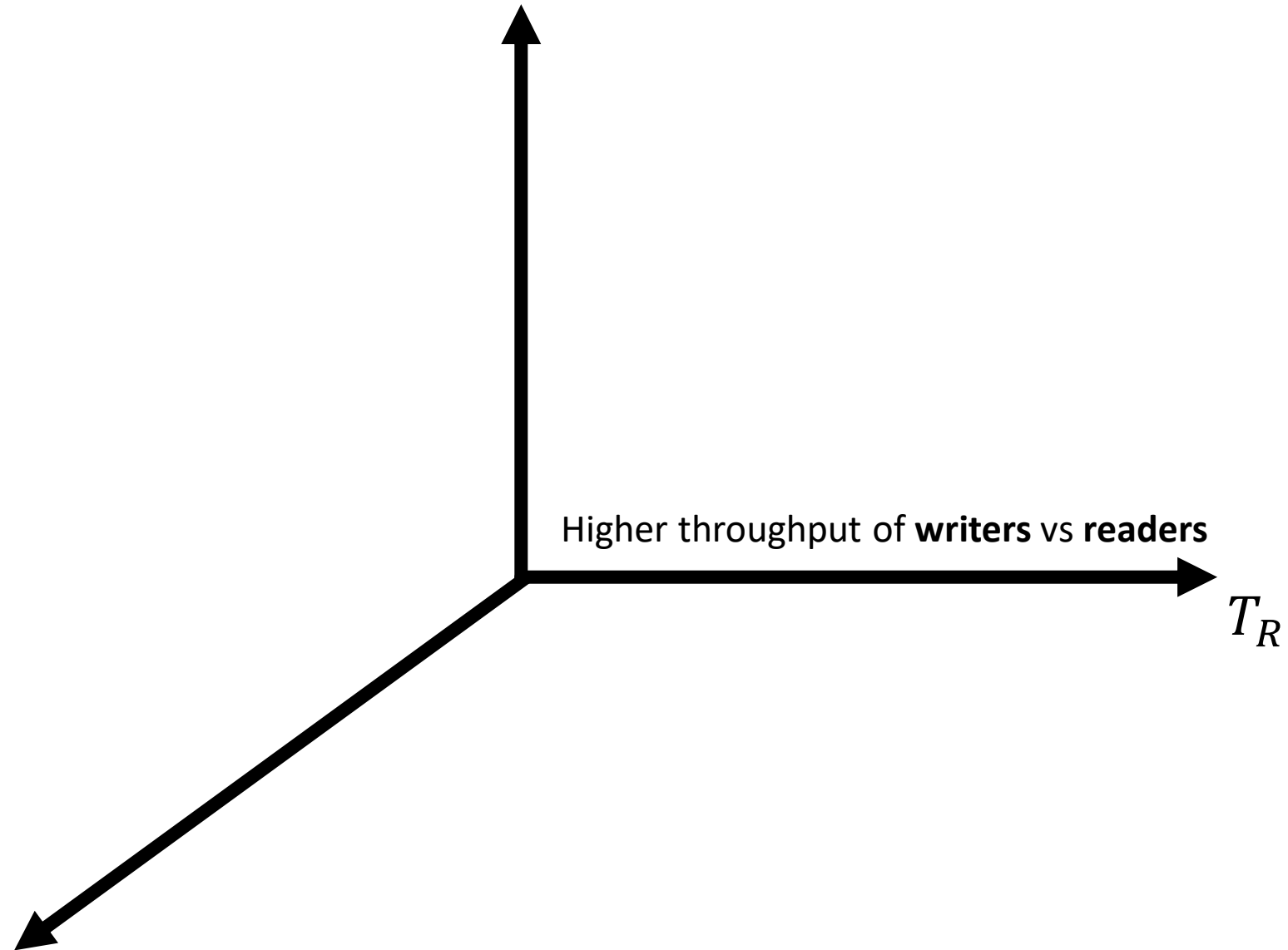
Design space



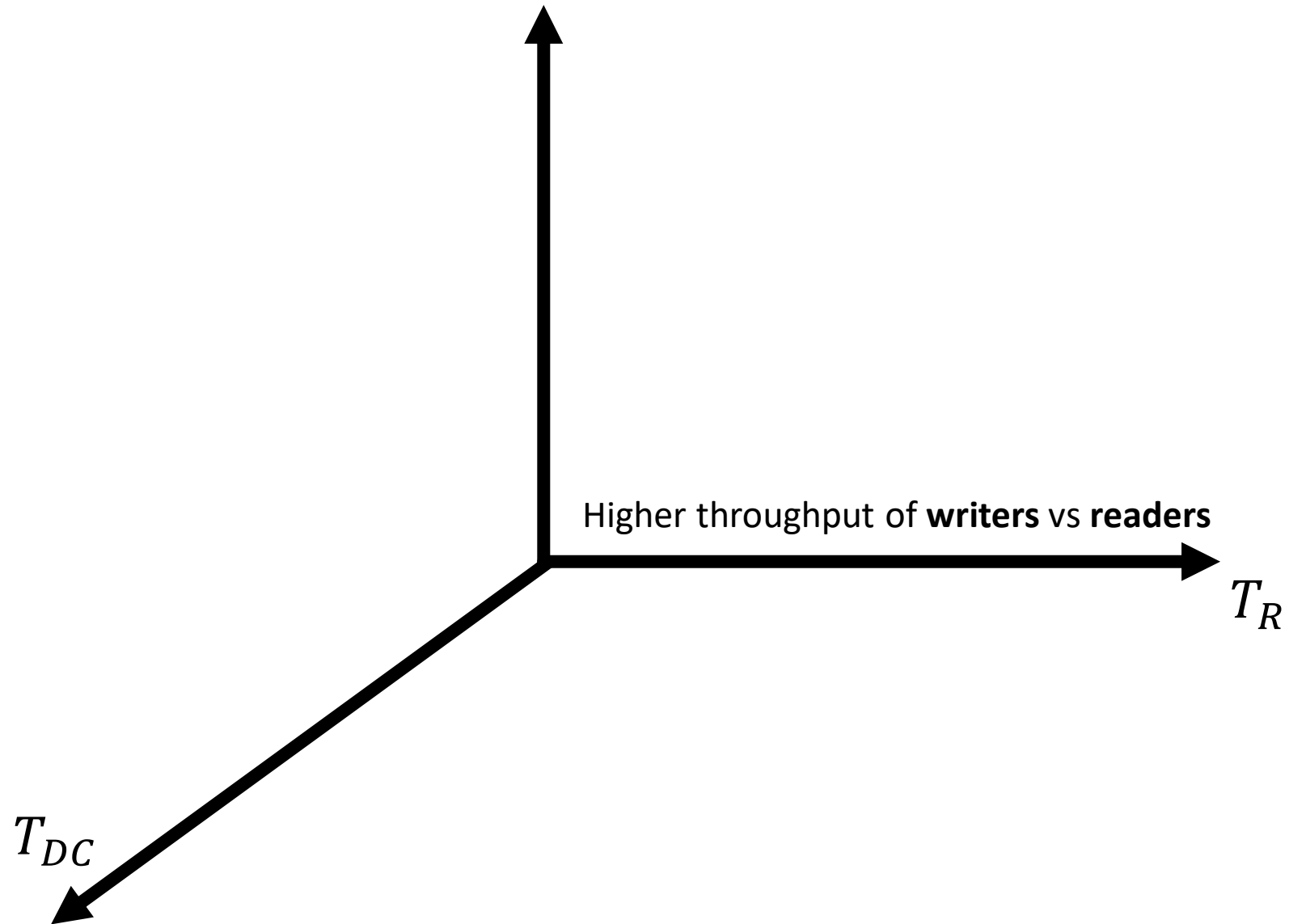
Design space



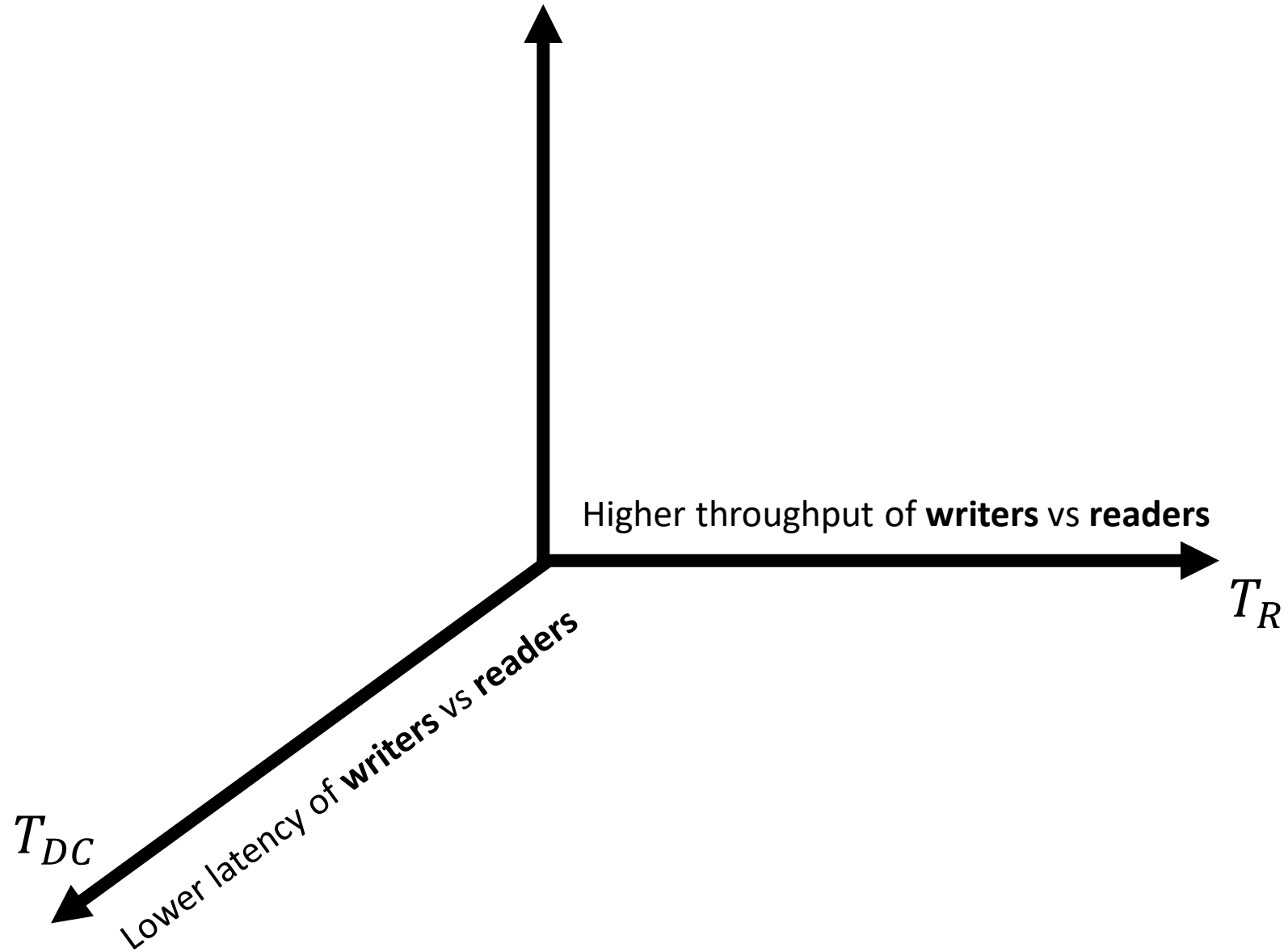
Design space



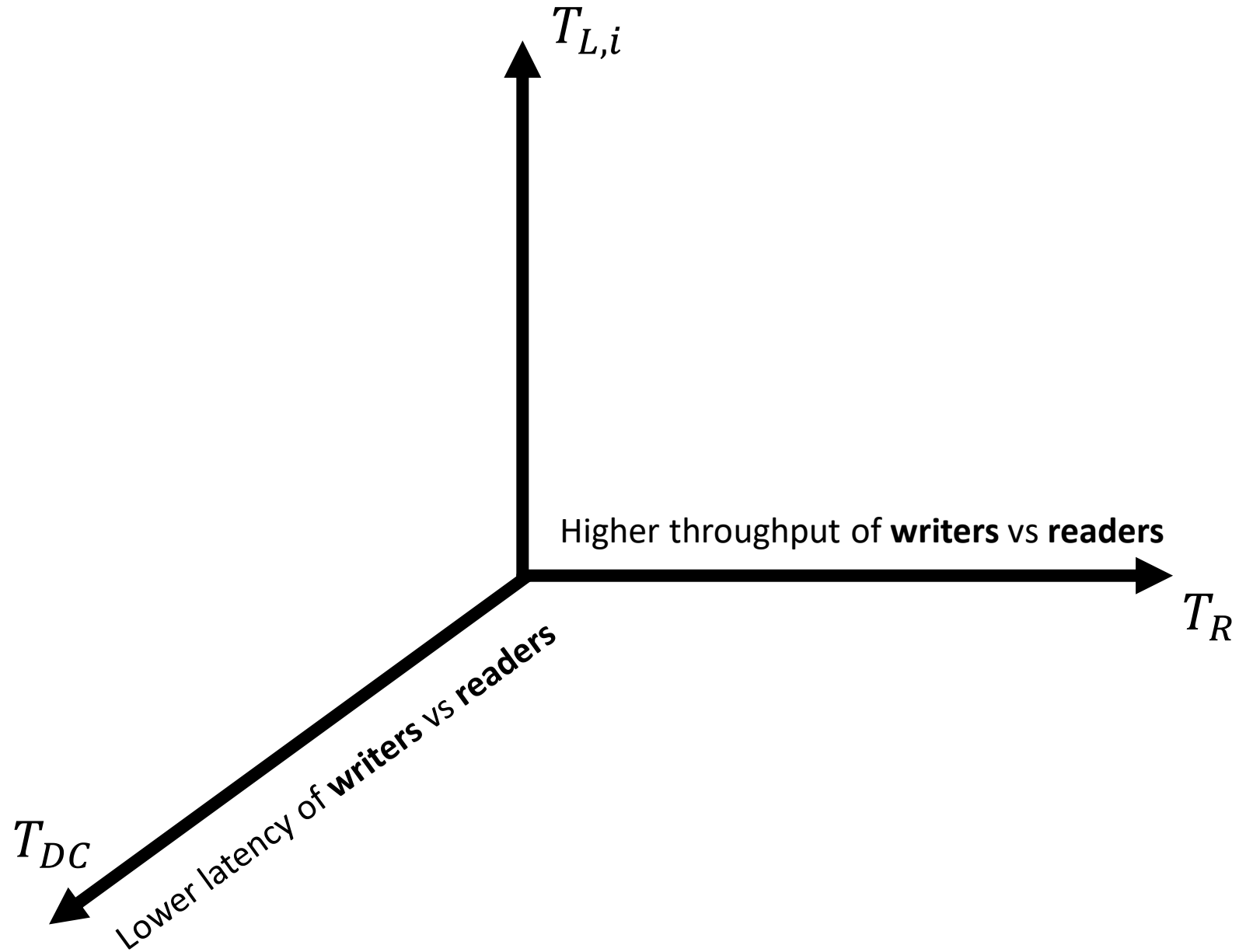
Design space



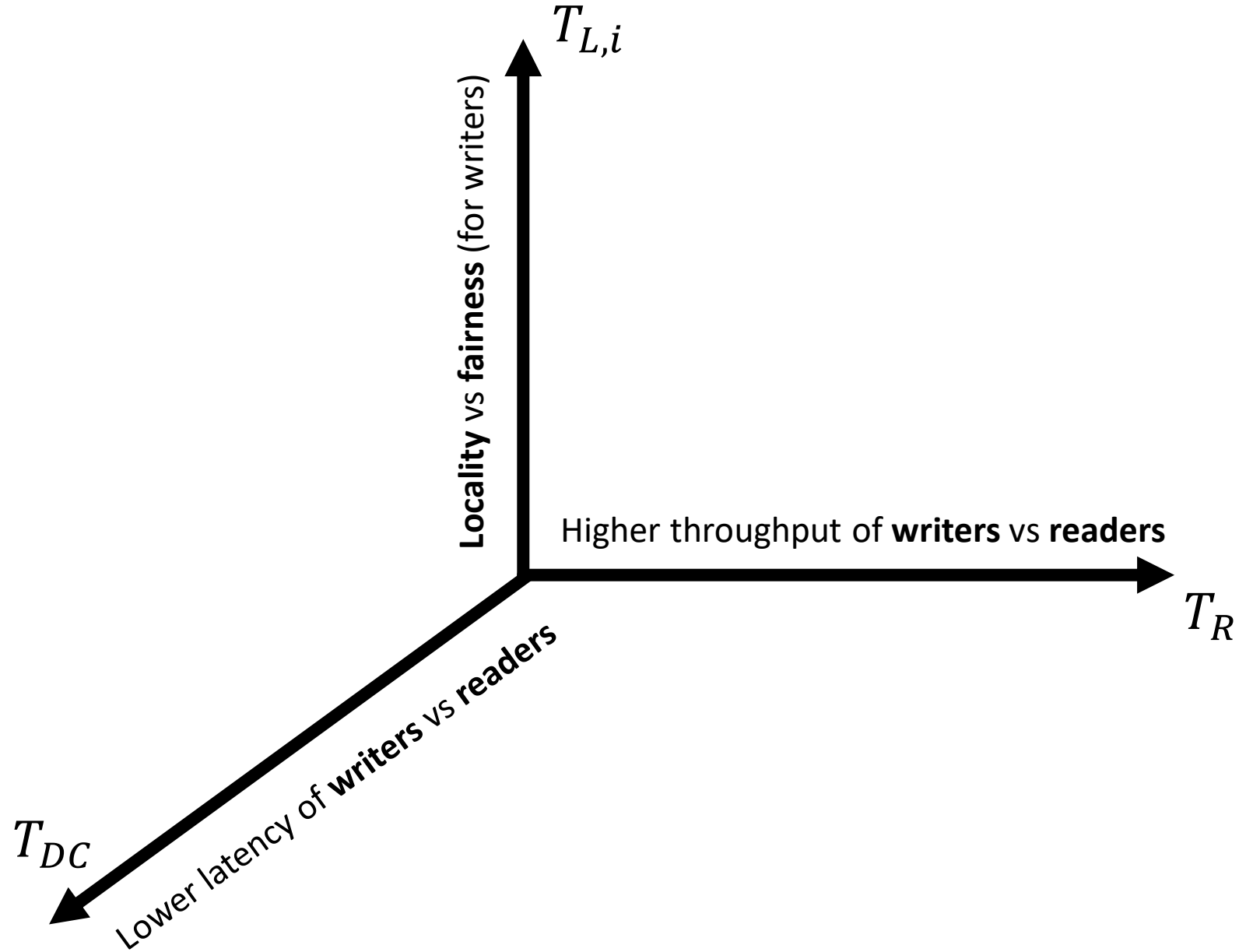
Design space



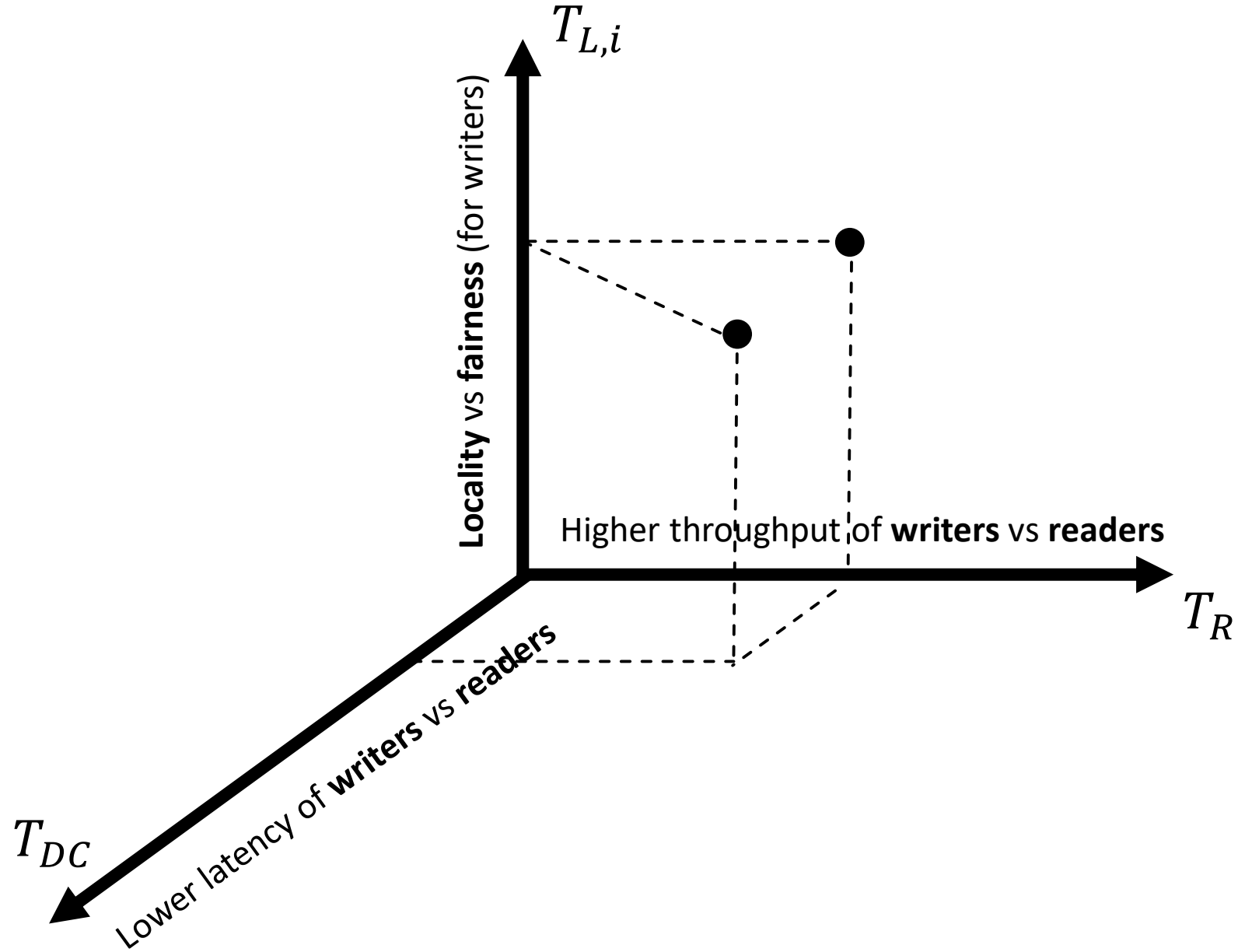
Design space



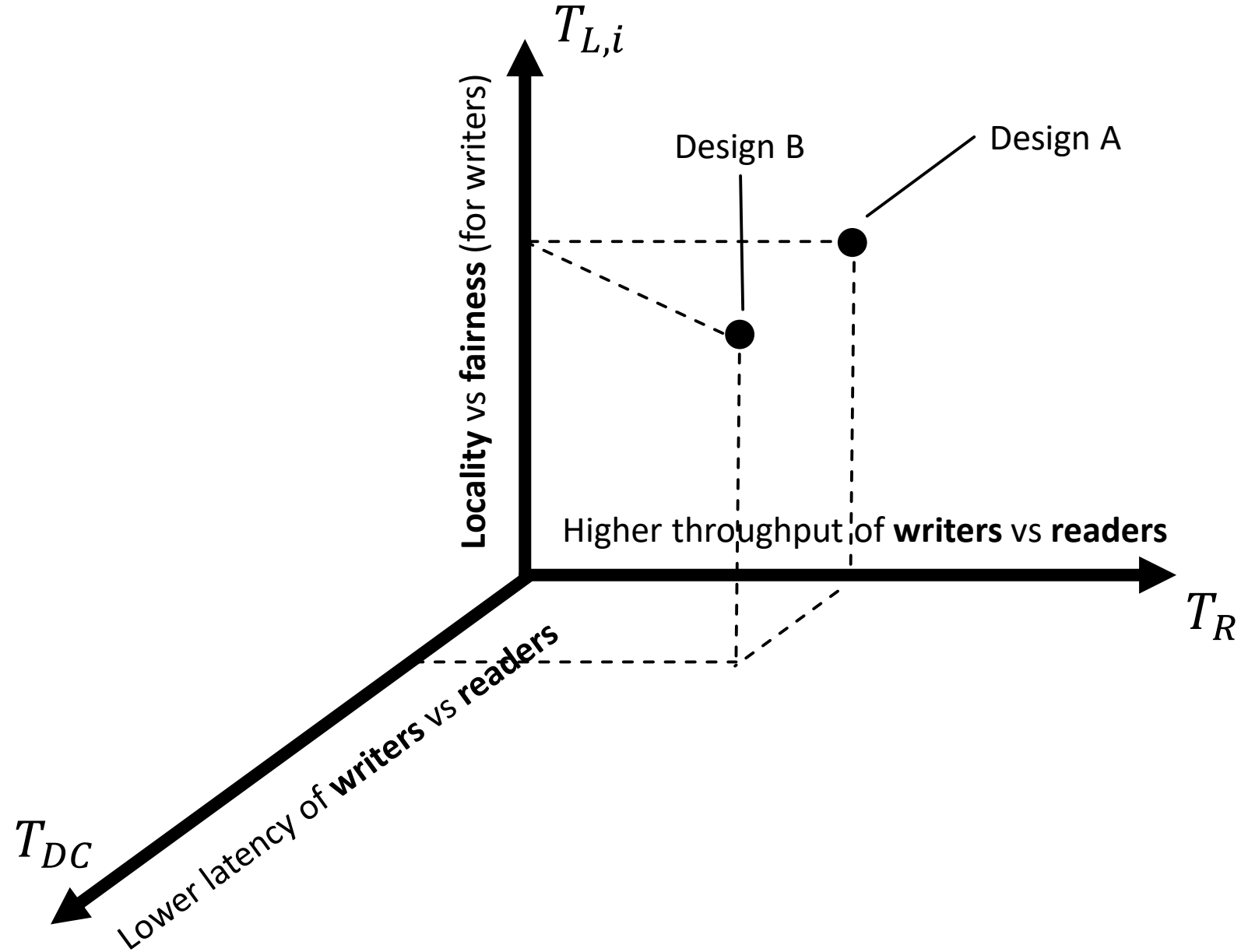
Design space



Design space



Design space



Lock Acquire by Readers



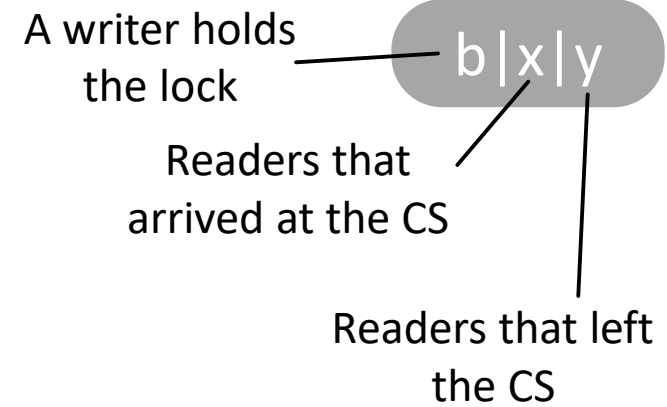
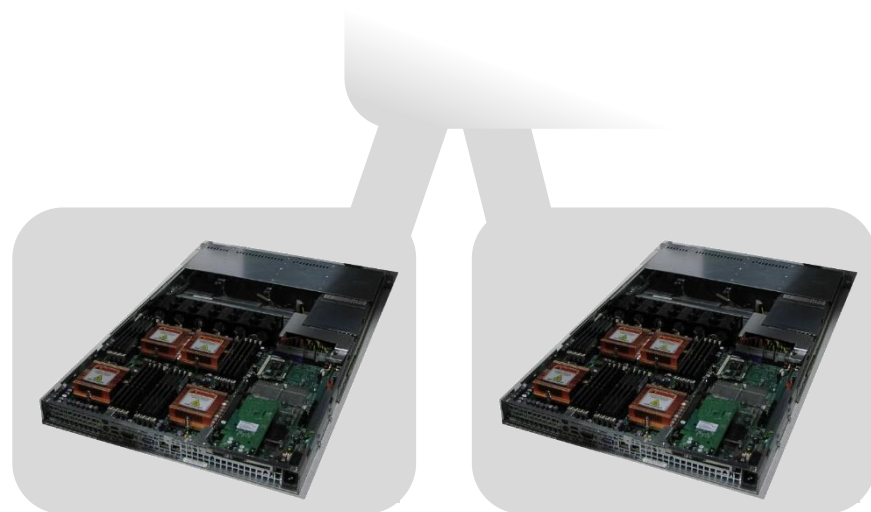
Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



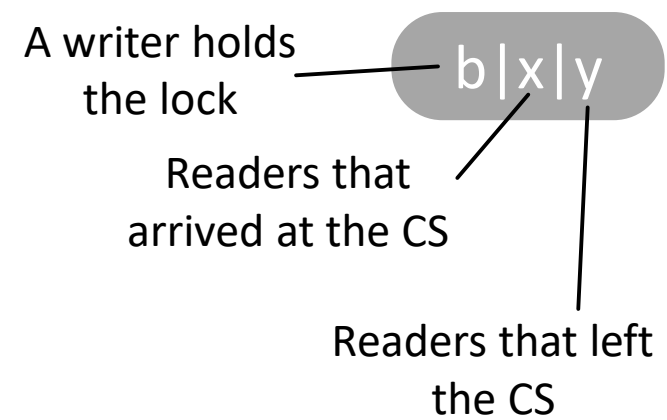
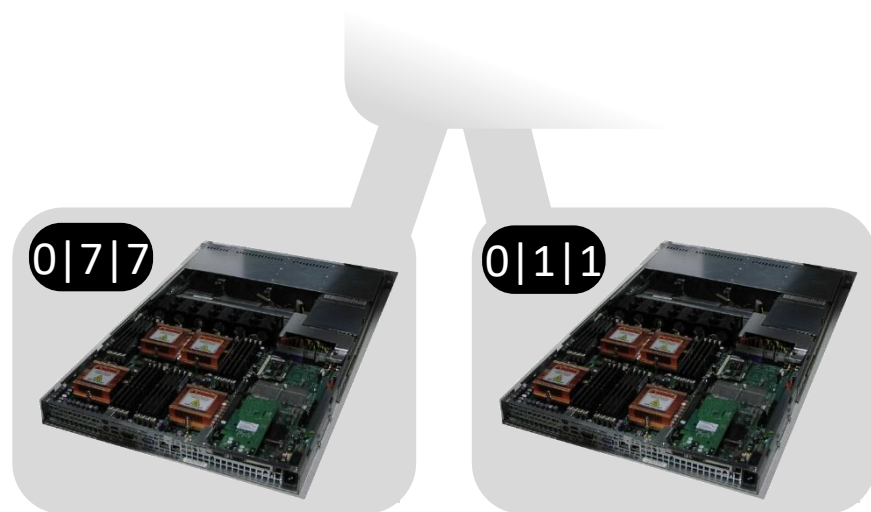
Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



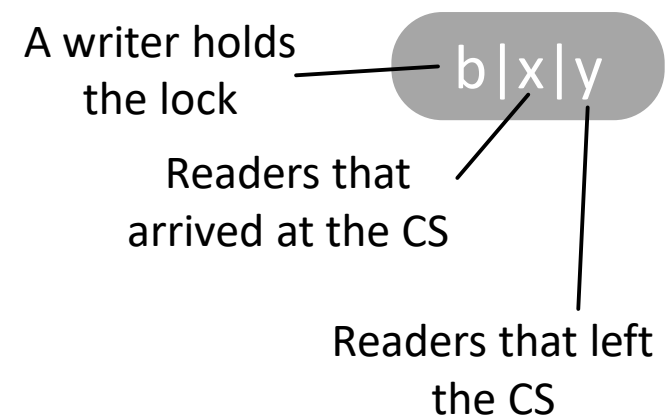
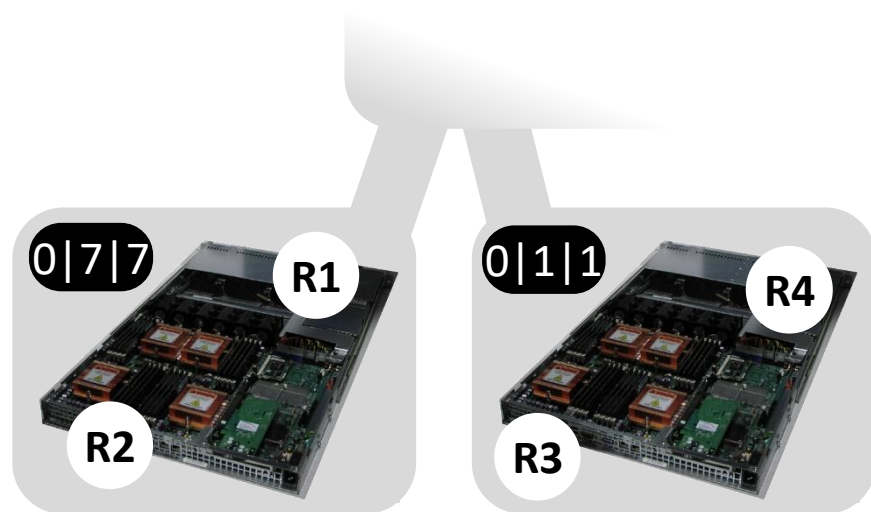
Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



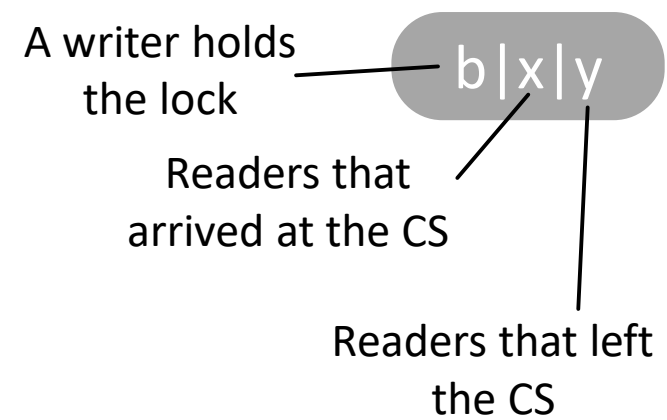
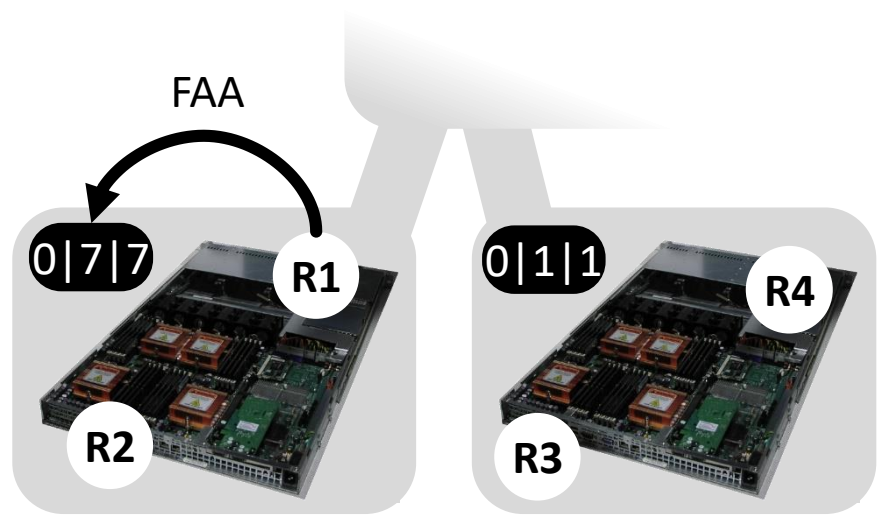
Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



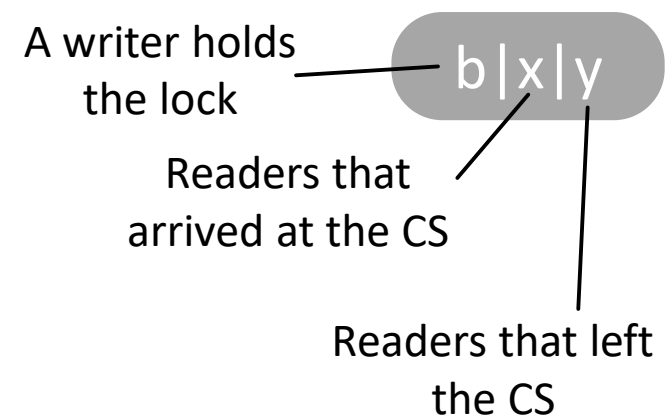
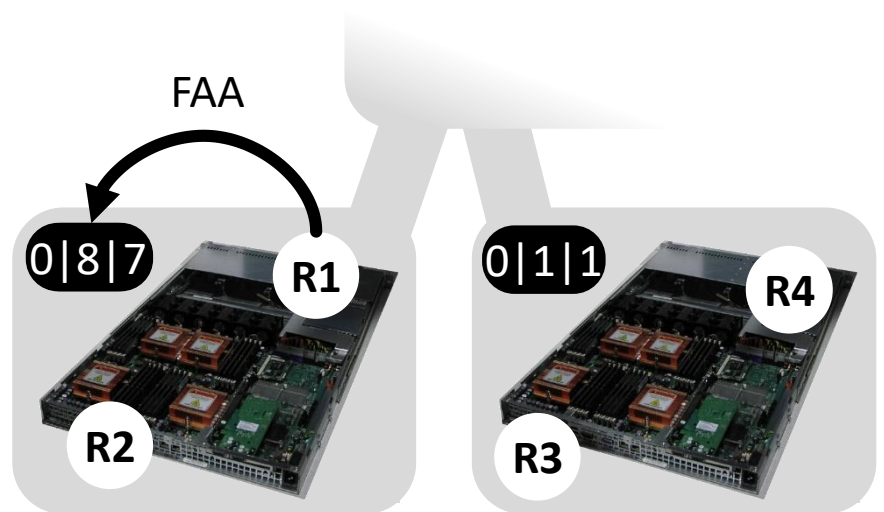
Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



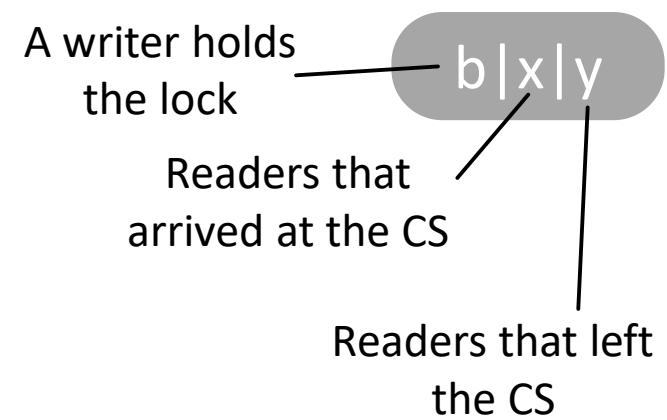
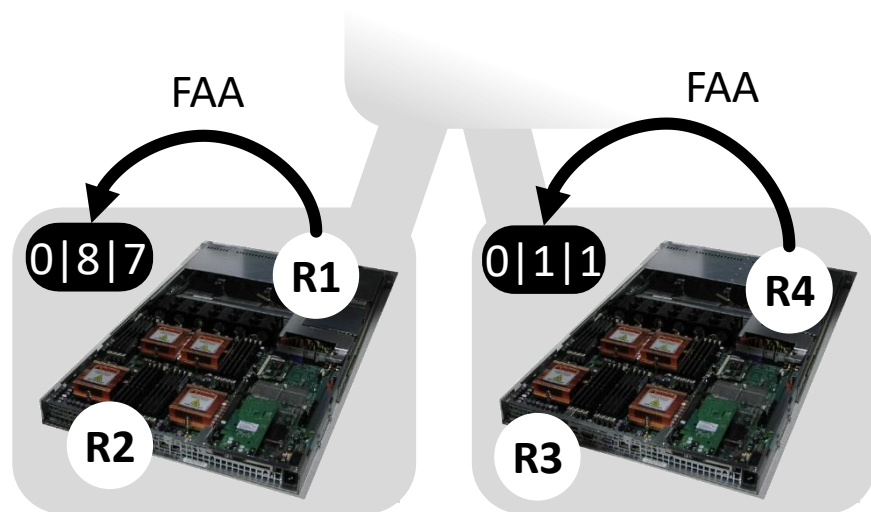
Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



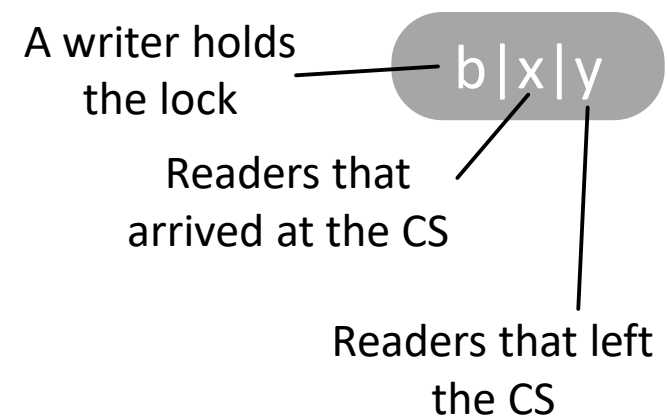
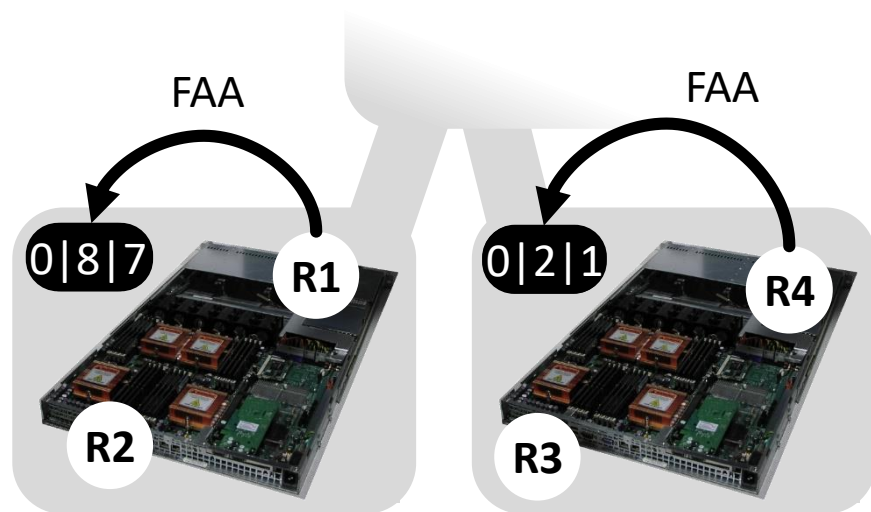
Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



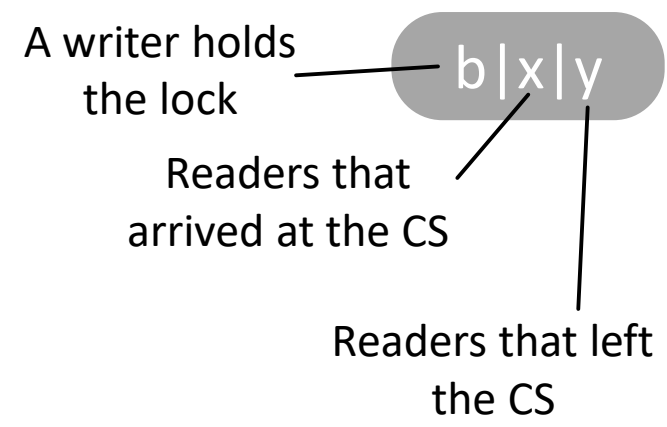
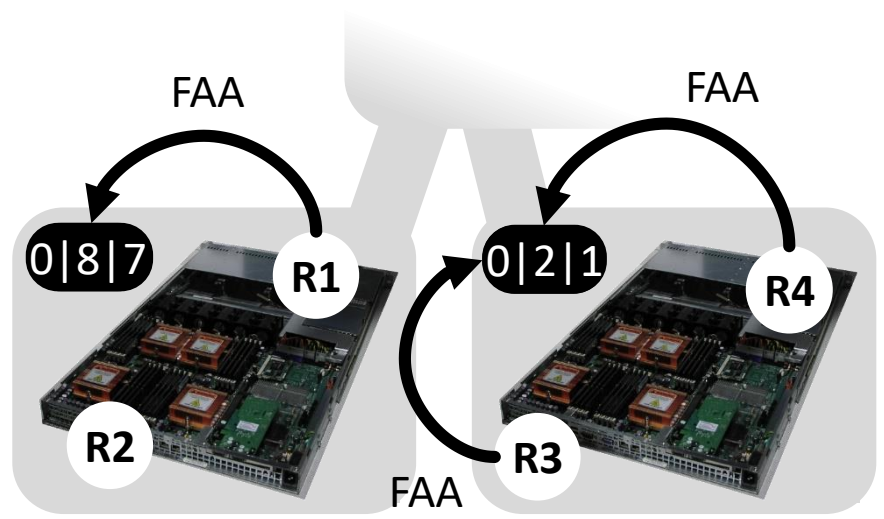
Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



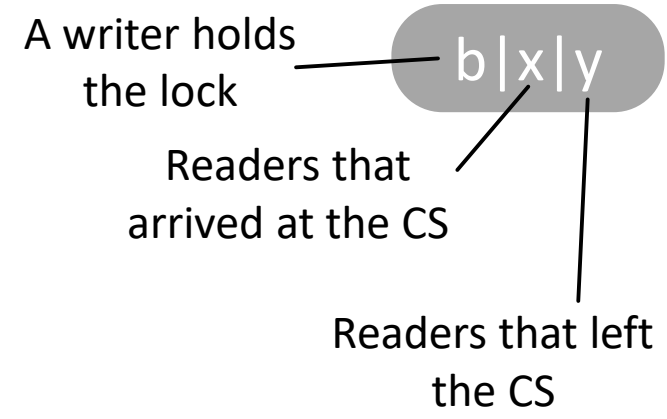
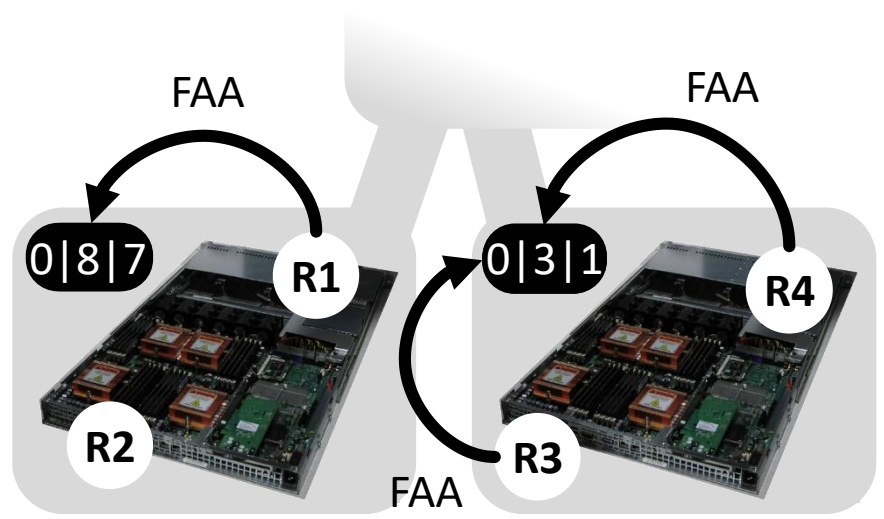
Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



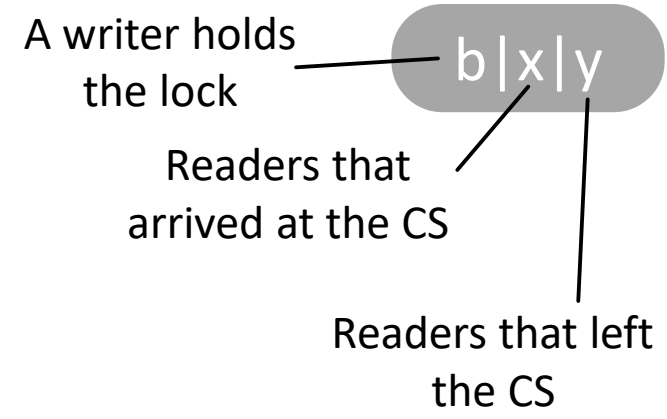
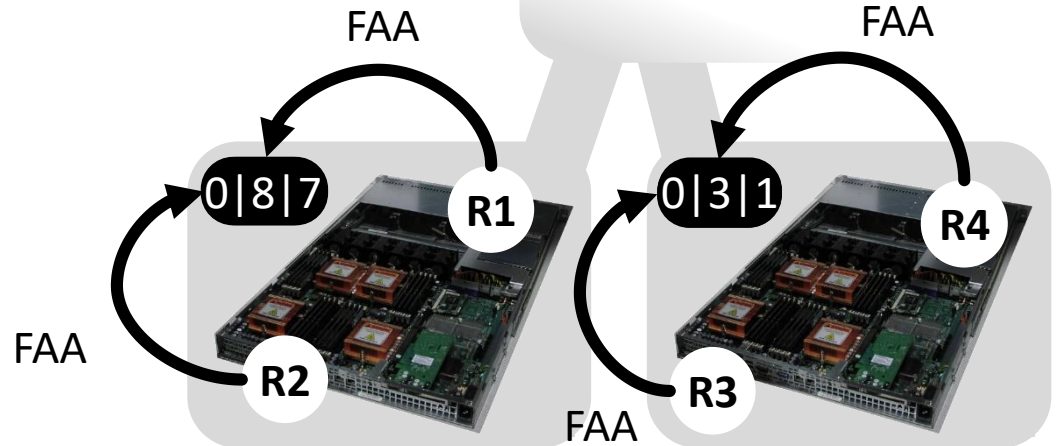
Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



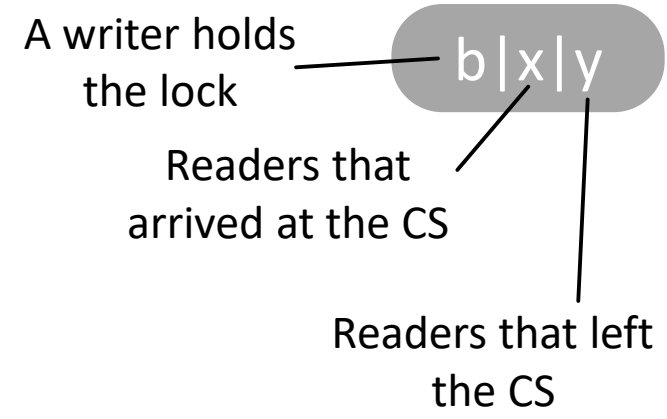
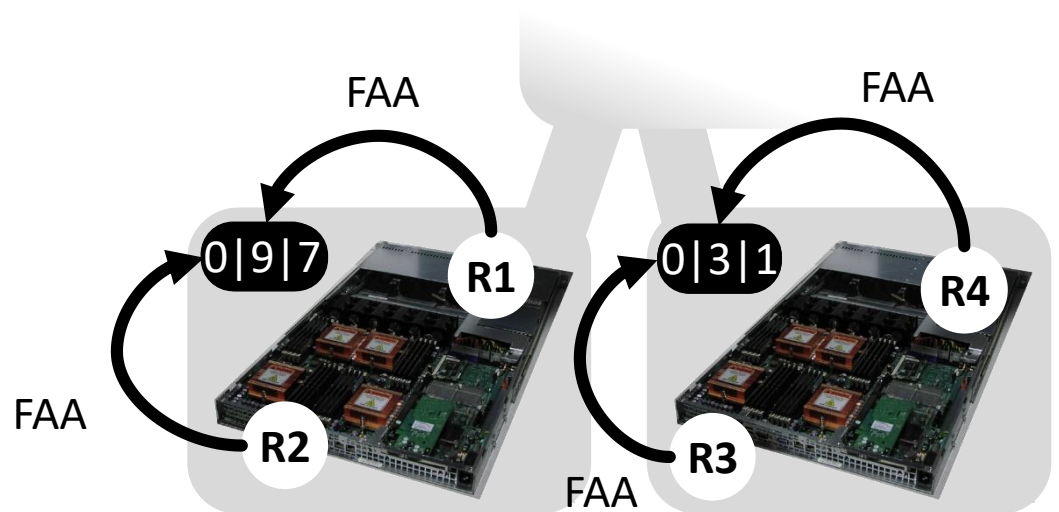
Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation

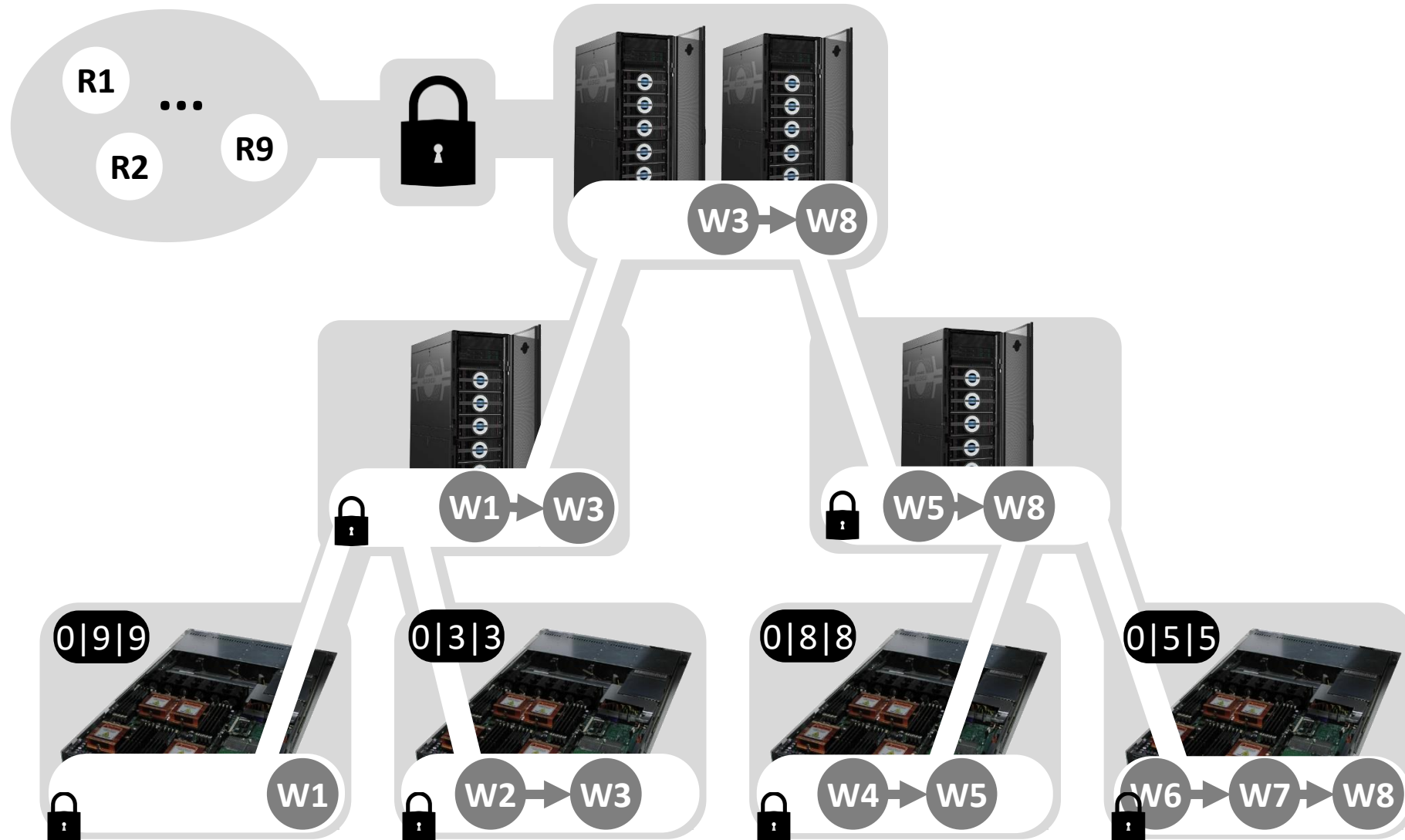


Lock Acquire by Readers

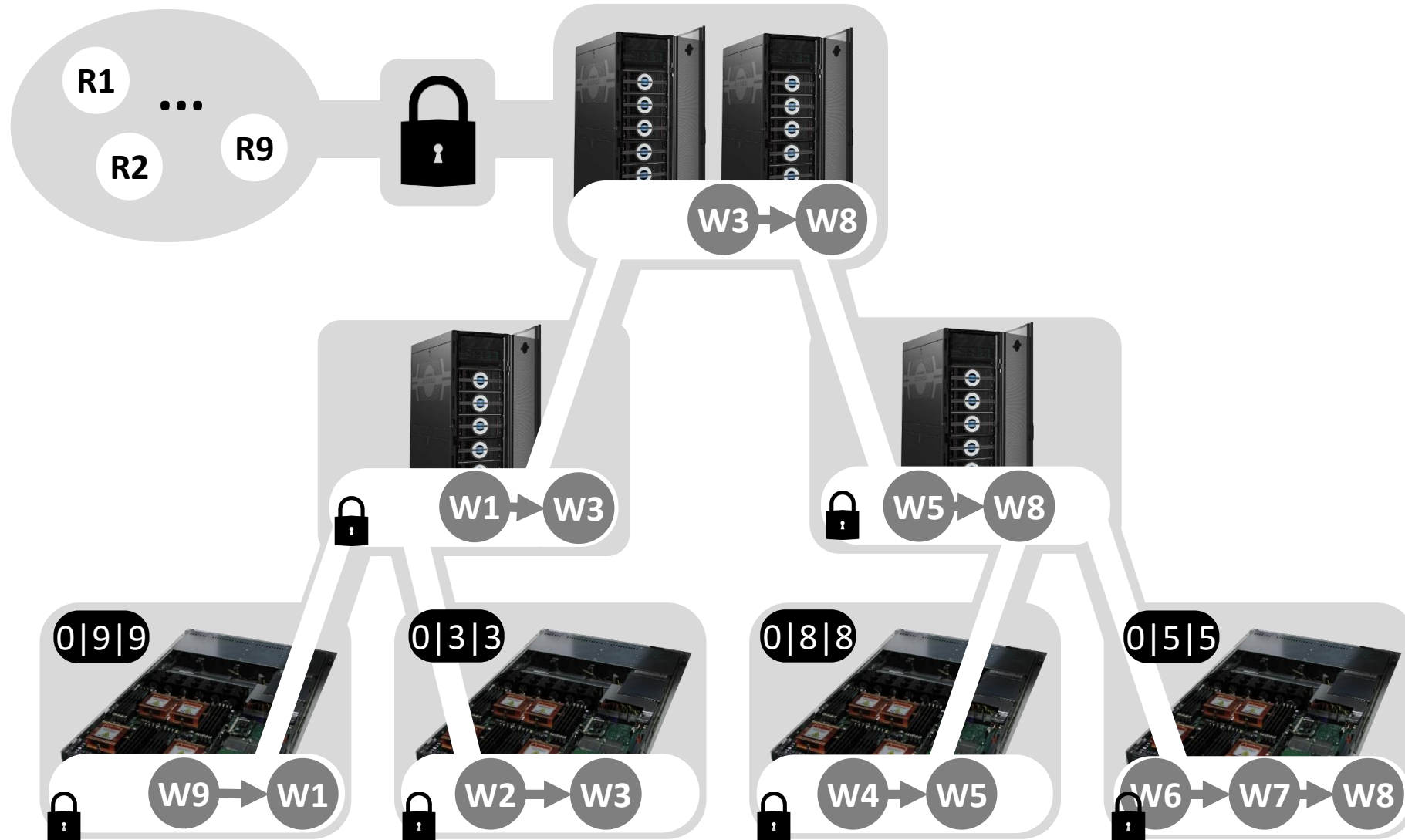
! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



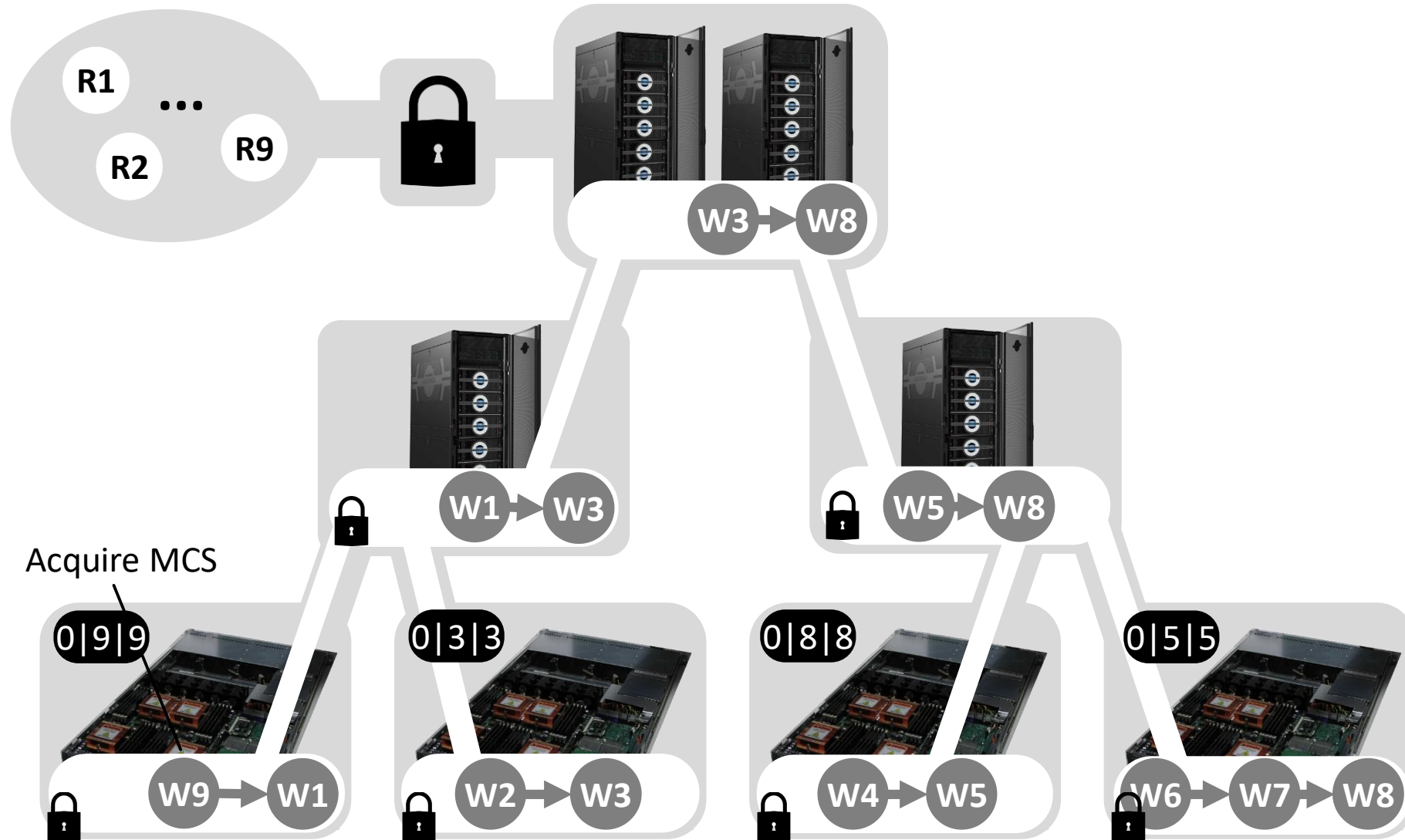
Lock Acquire by Writers



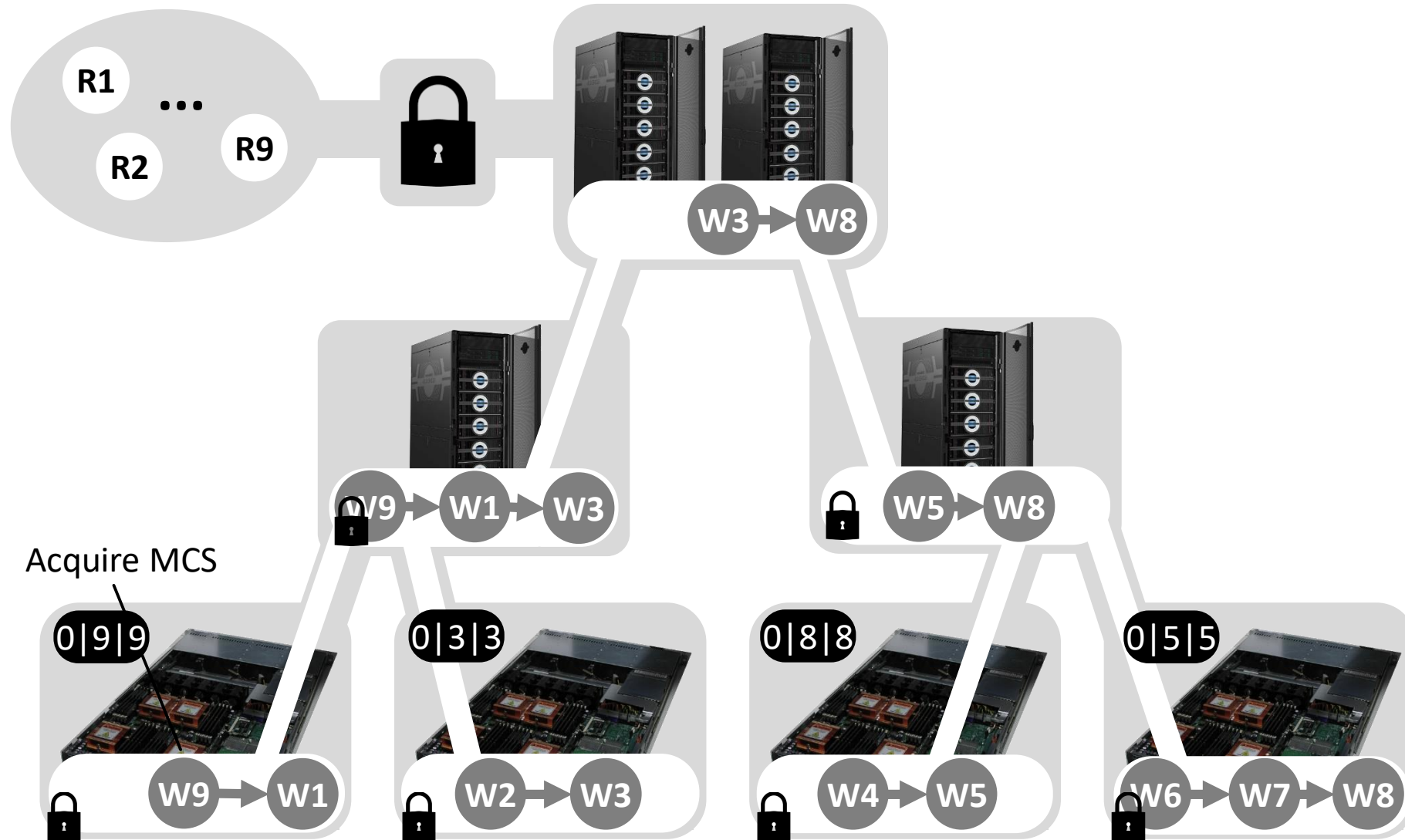
Lock Acquire by Writers



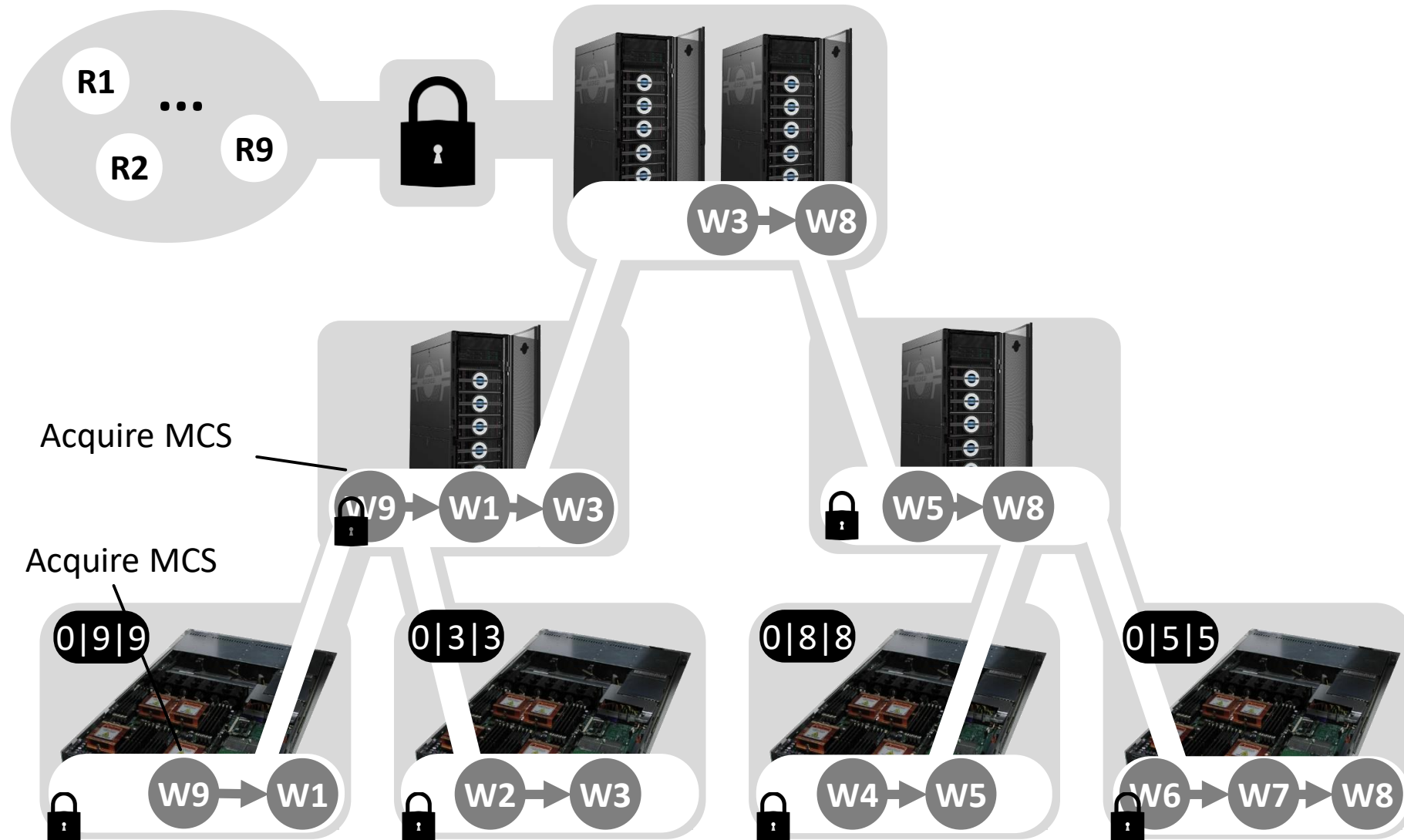
Lock Acquire by Writers



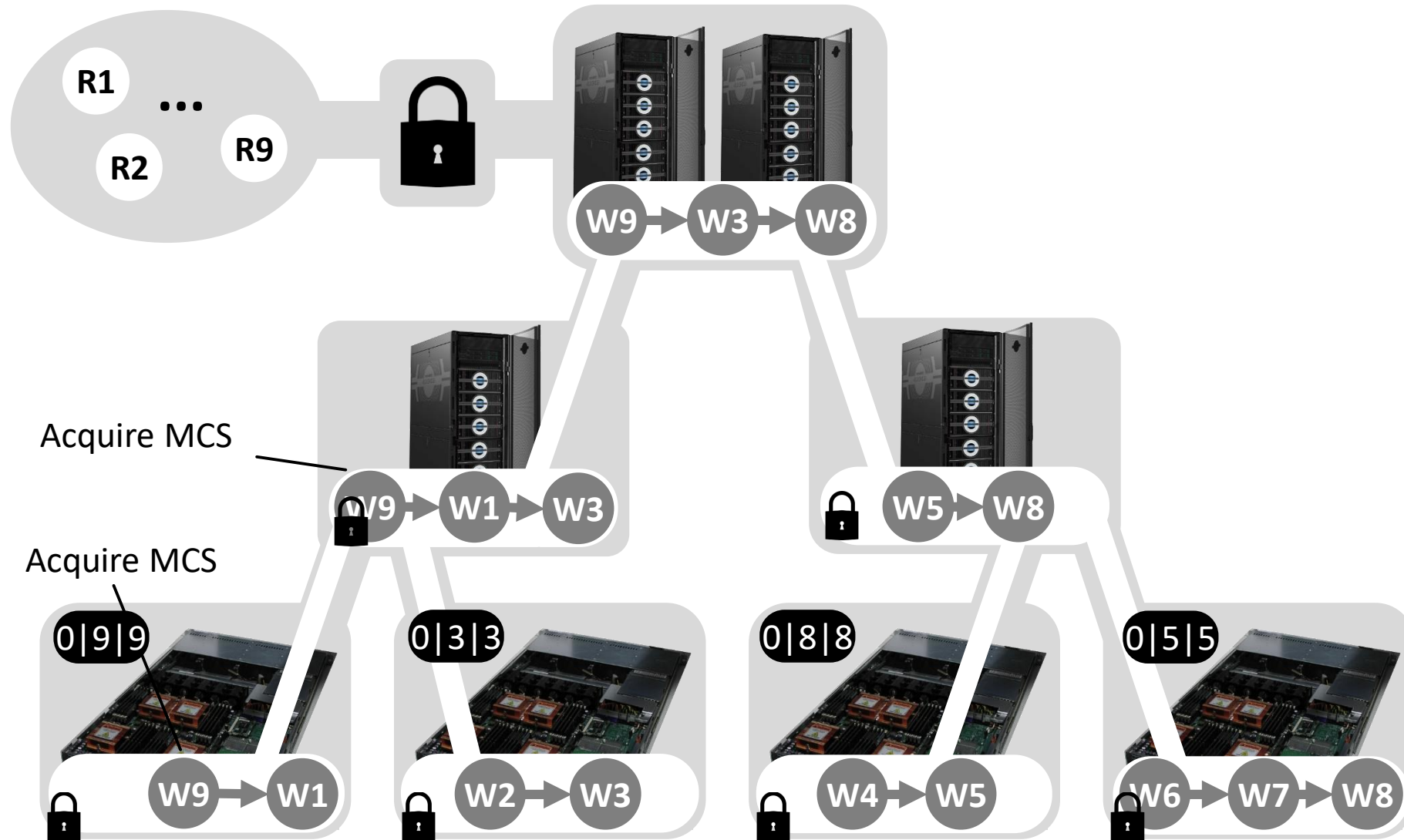
Lock Acquire by Writers



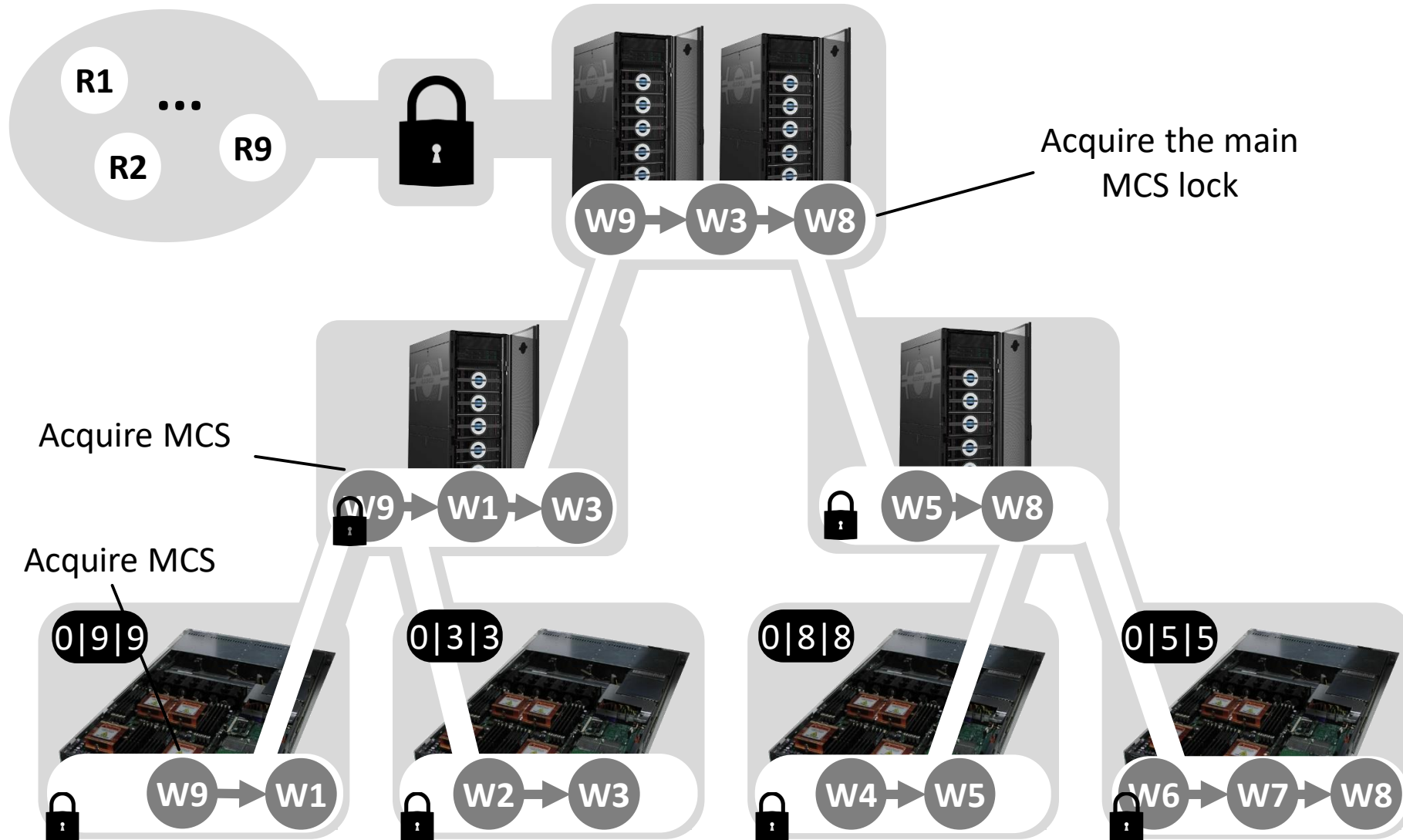
Lock Acquire by Writers



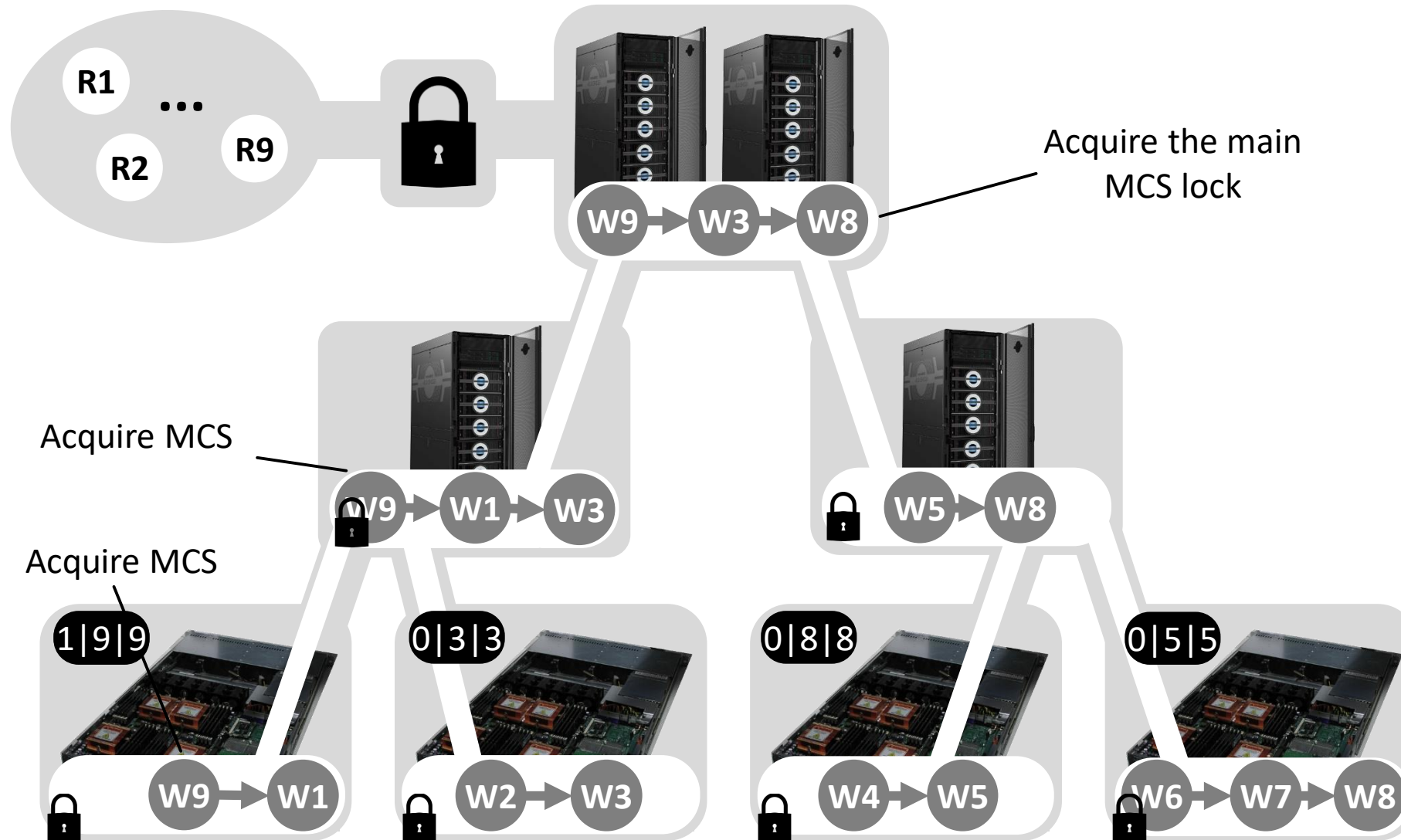
Lock Acquire by Writers



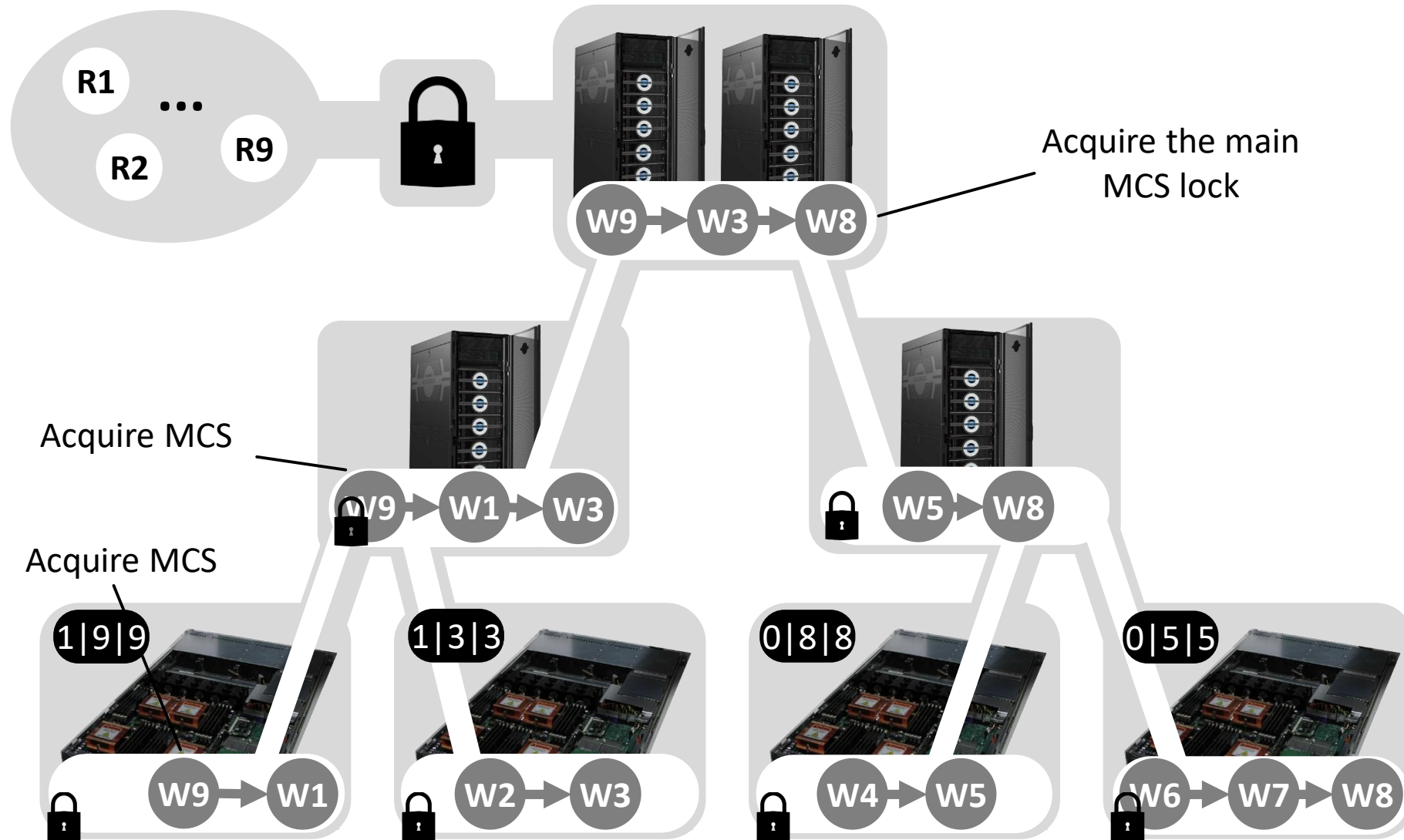
Lock Acquire by Writers



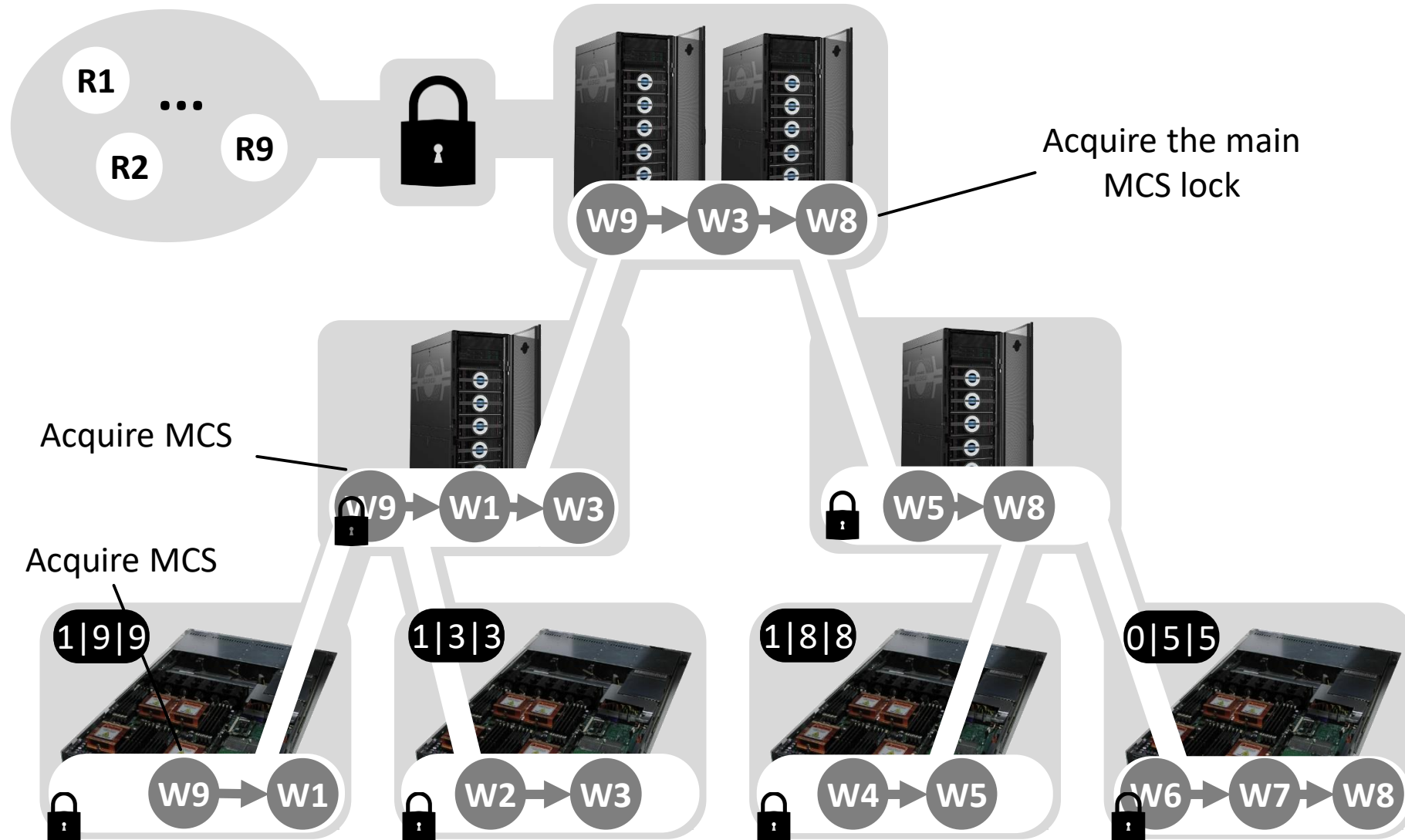
Lock Acquire by Writers



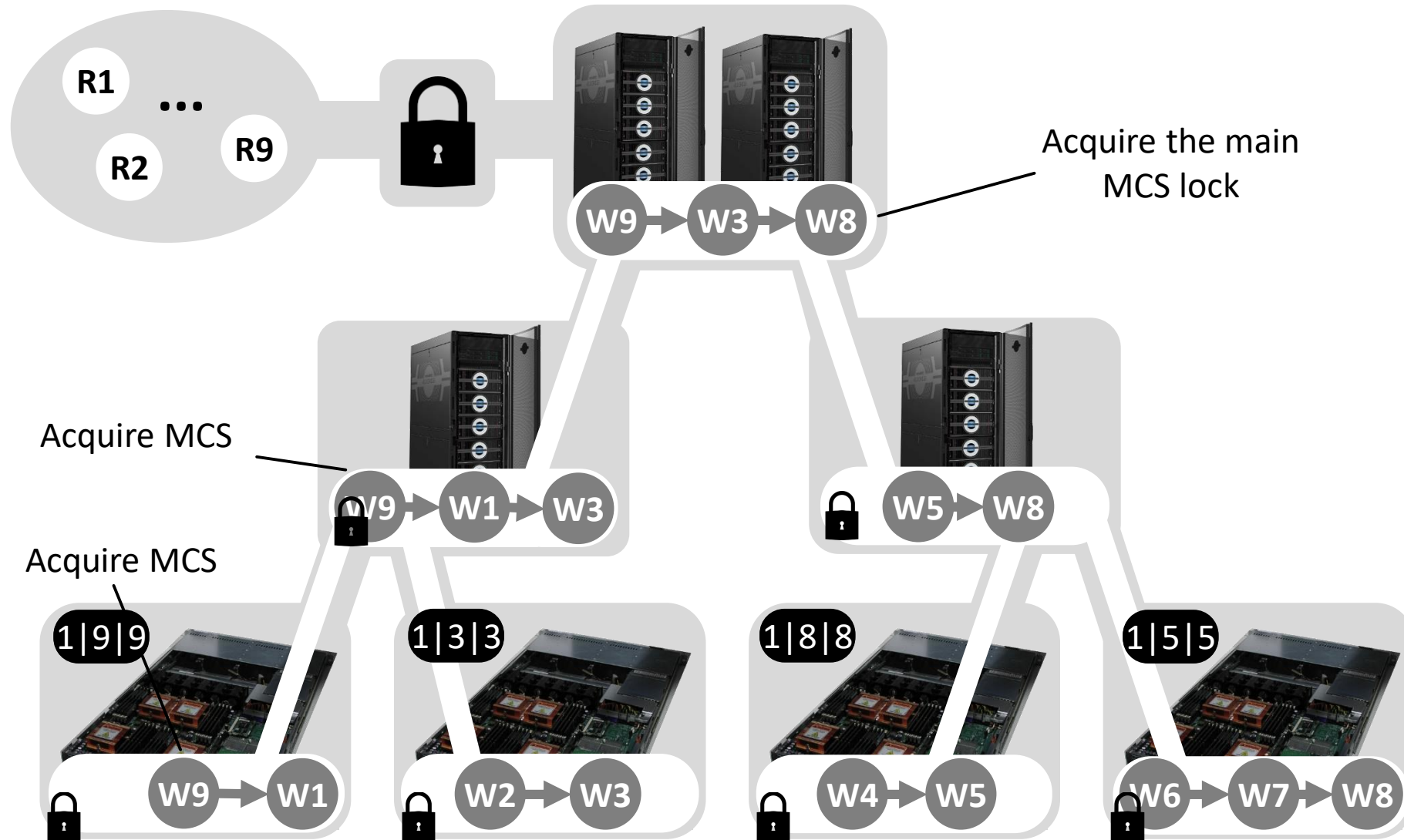
Lock Acquire by Writers



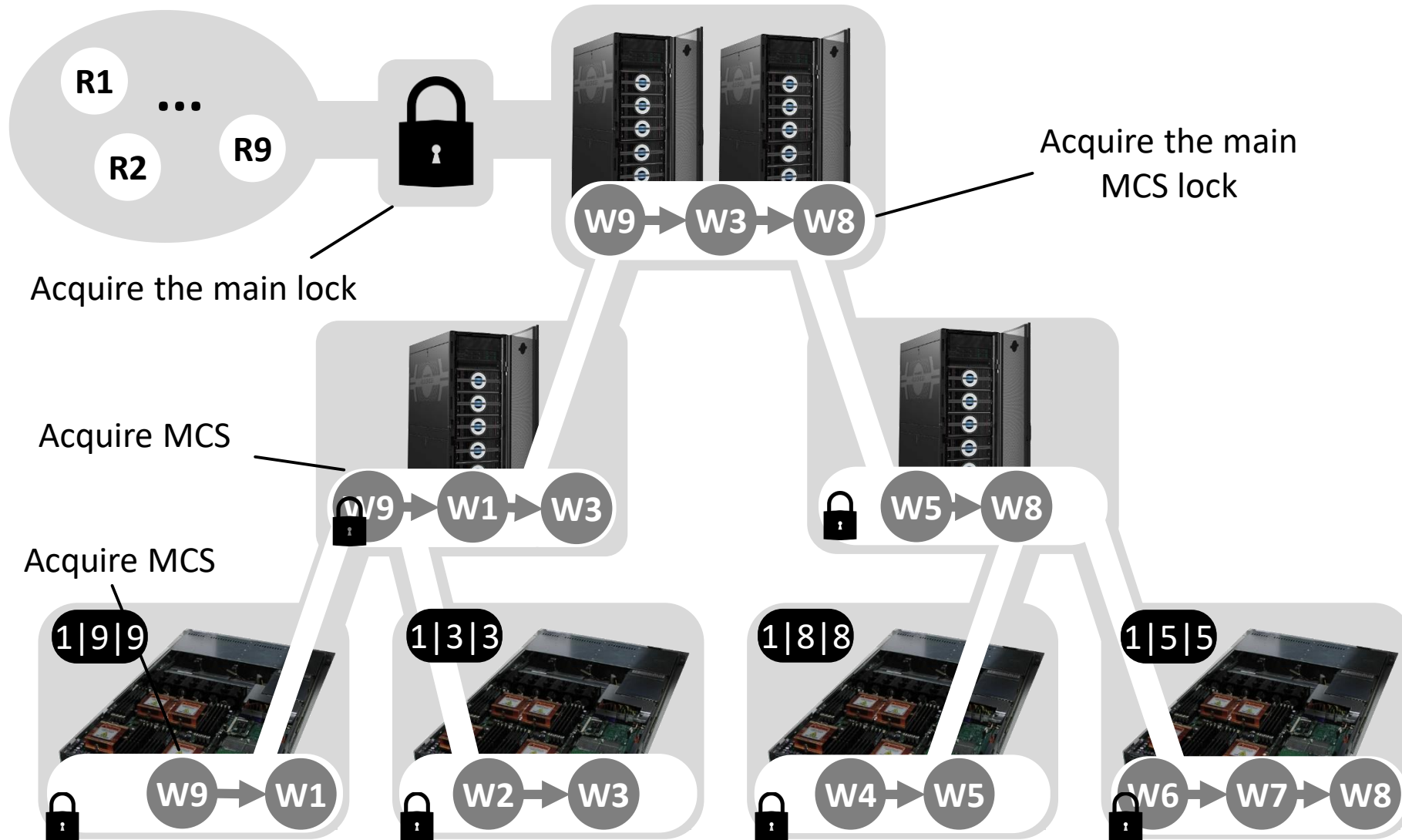
Lock Acquire by Writers



Lock Acquire by Writers



Lock Acquire by Writers



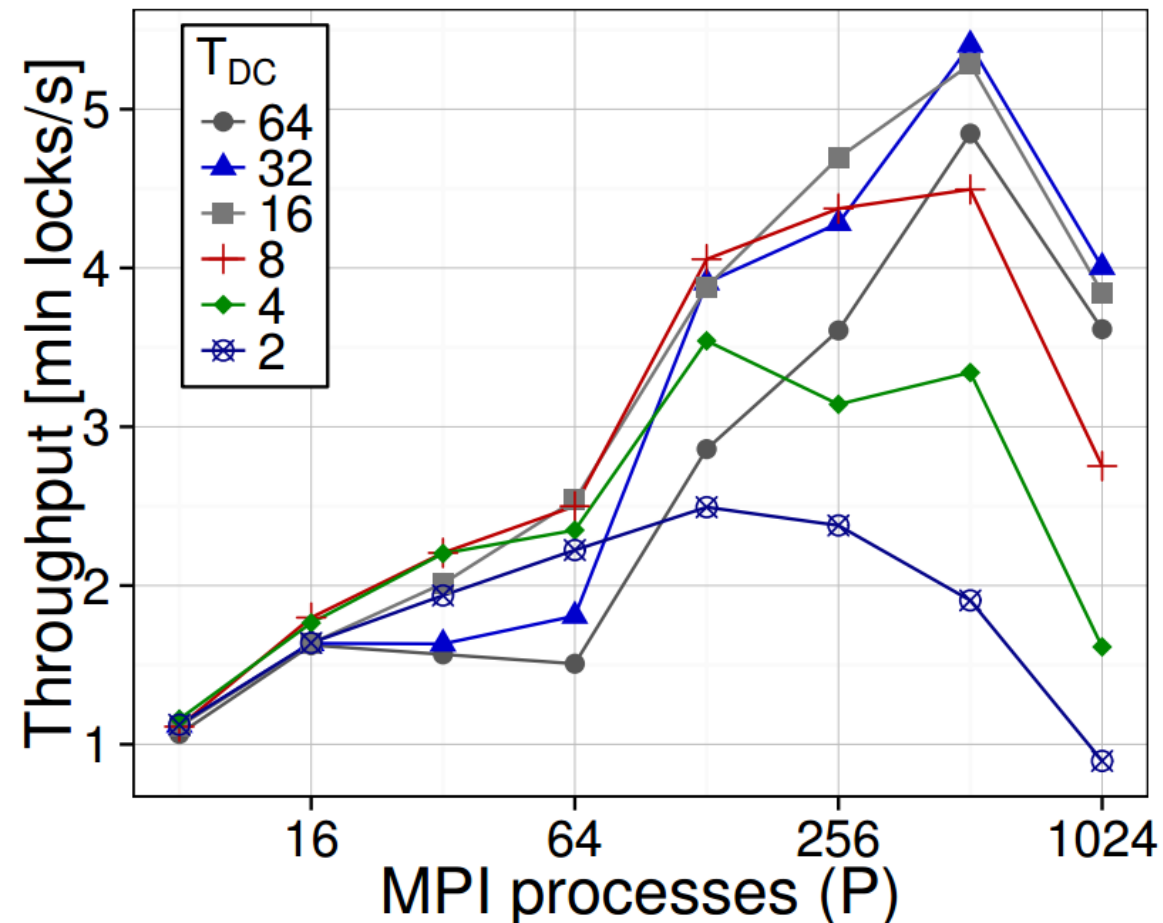
EVALUATION

- CSCS Piz Daint (Cray XC30)
- 5272 compute nodes
- 8 cores per node
- 169TB memory
- Microbenchmarks: acquire/release: latency, throughput
- Distributed hashtable

Evaluation - Distributed Counter Analysis

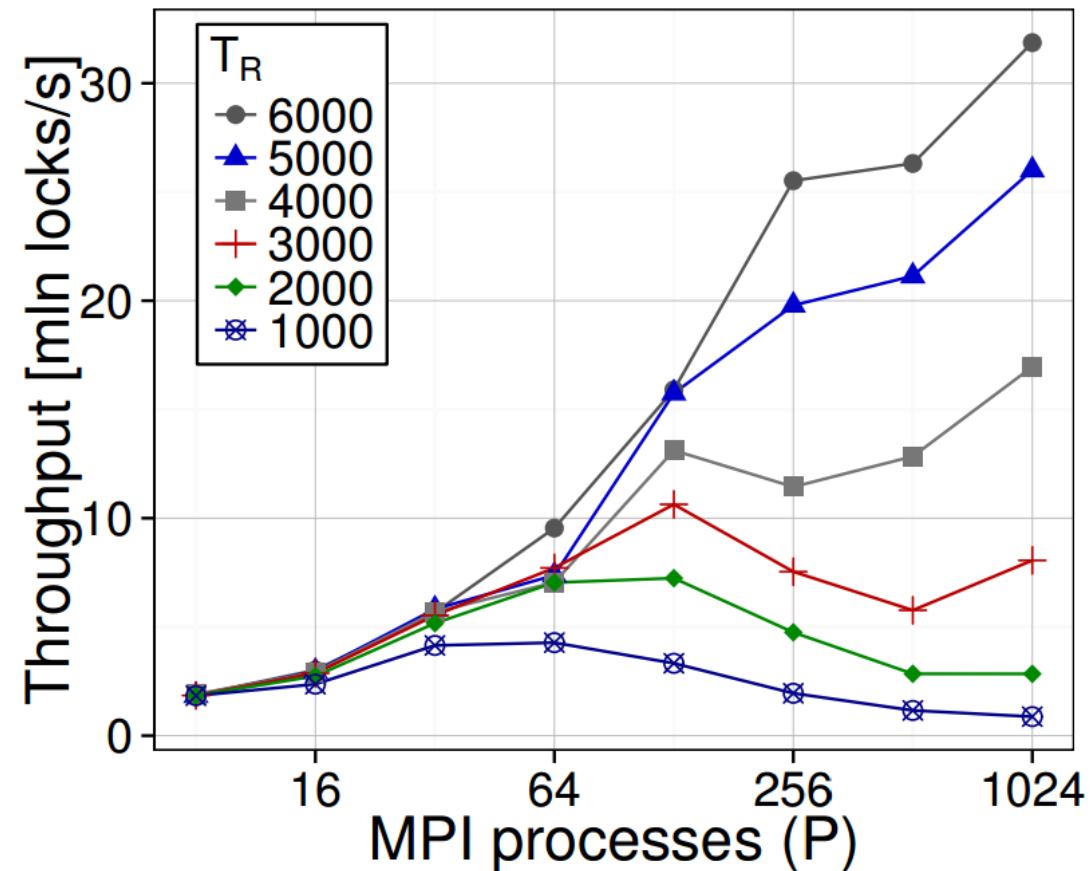
0|12|8

Throughput, 2% writers
 Single-operation benchmark

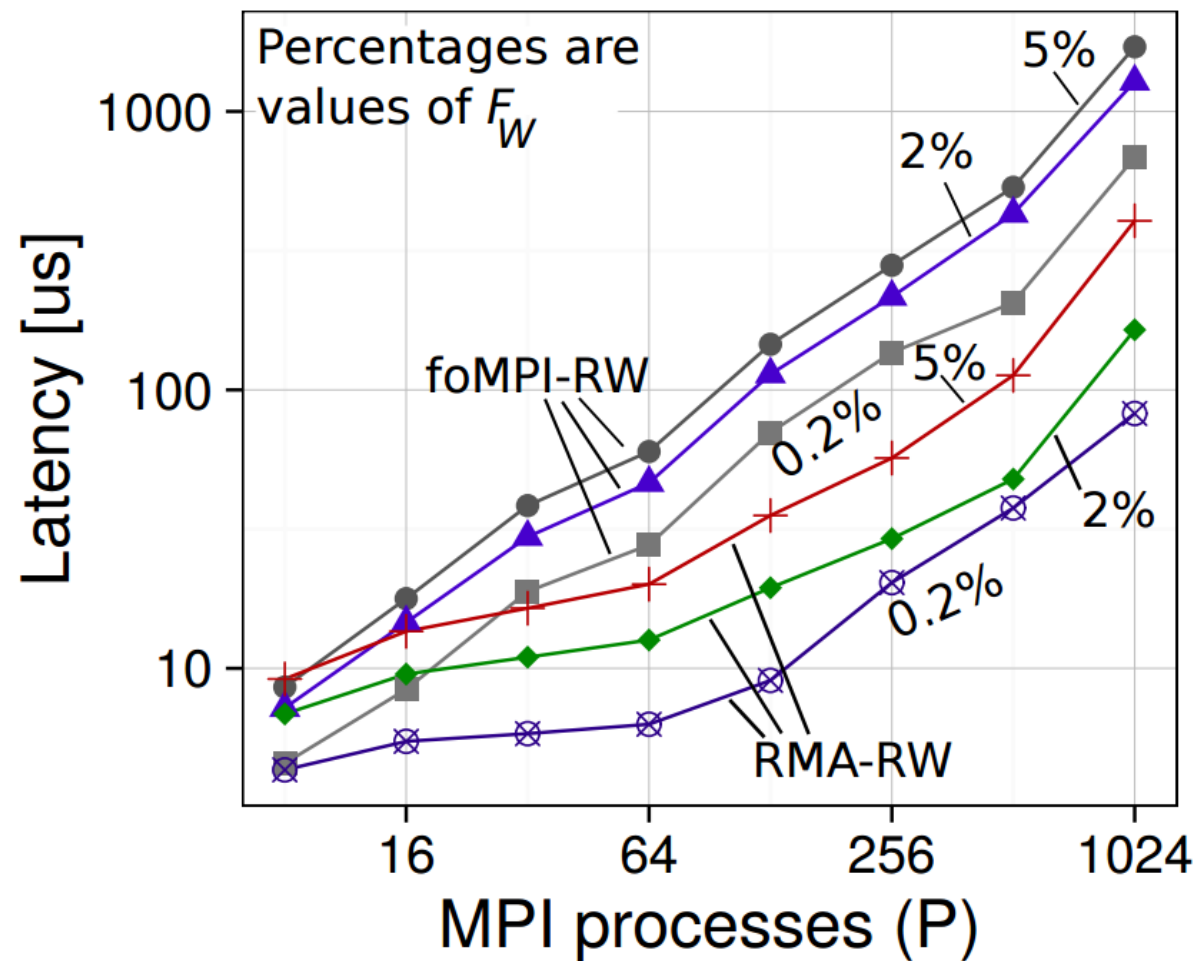


Evaluation - Reader Threshold Analysis

Throughput, 0.2% writers,
Empty-critical-section benchmark

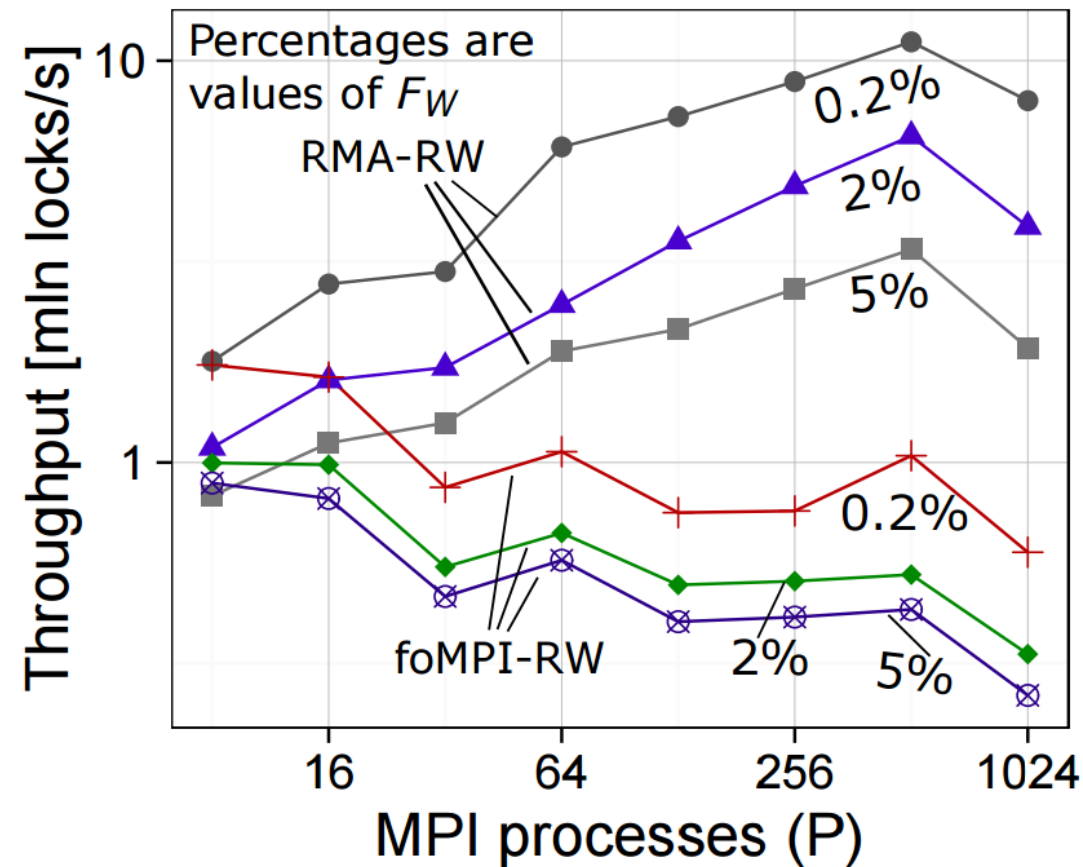


Evaluation - Comparison to the State-of-the-Art



Evaluation - Comparison to the State-of-the-Art

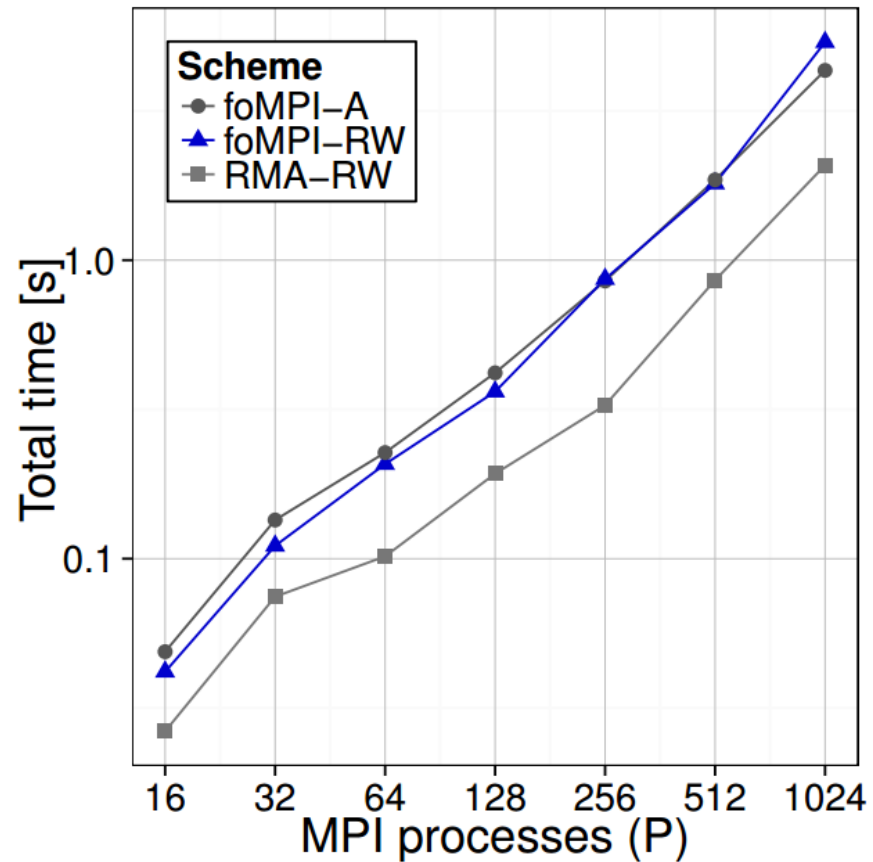
Throughput, single-operation benchmark



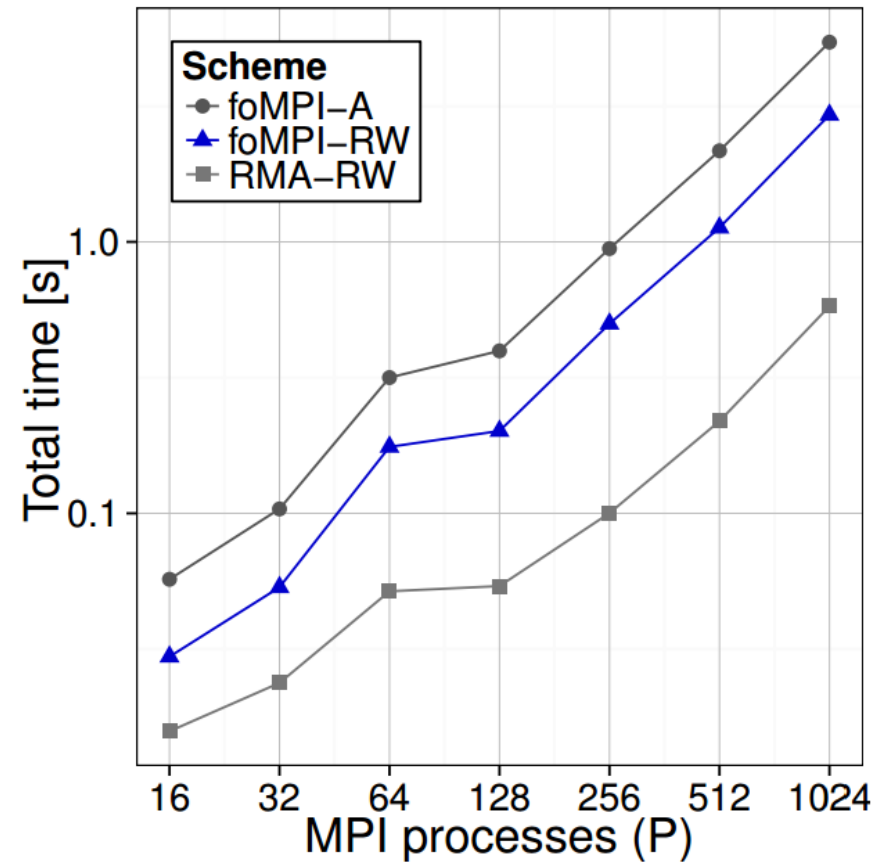
[1] R. Gerstenberger et al. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. ACM/IEEE Supercomputing 2013.

Evaluation - Distributed Hashtable

20% writers

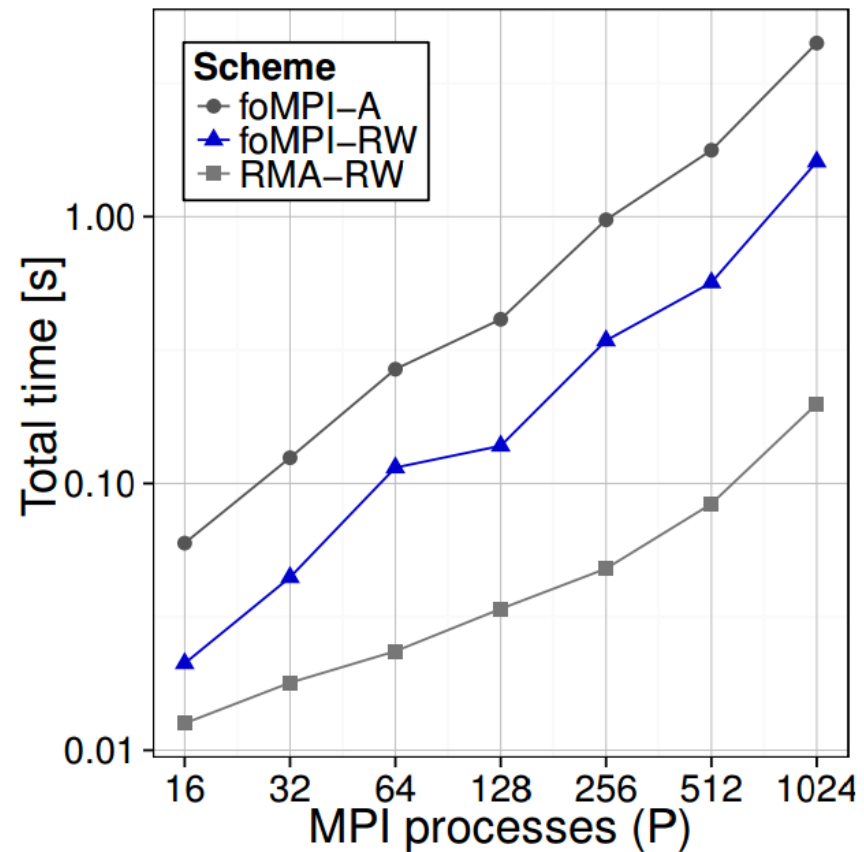


10% writers

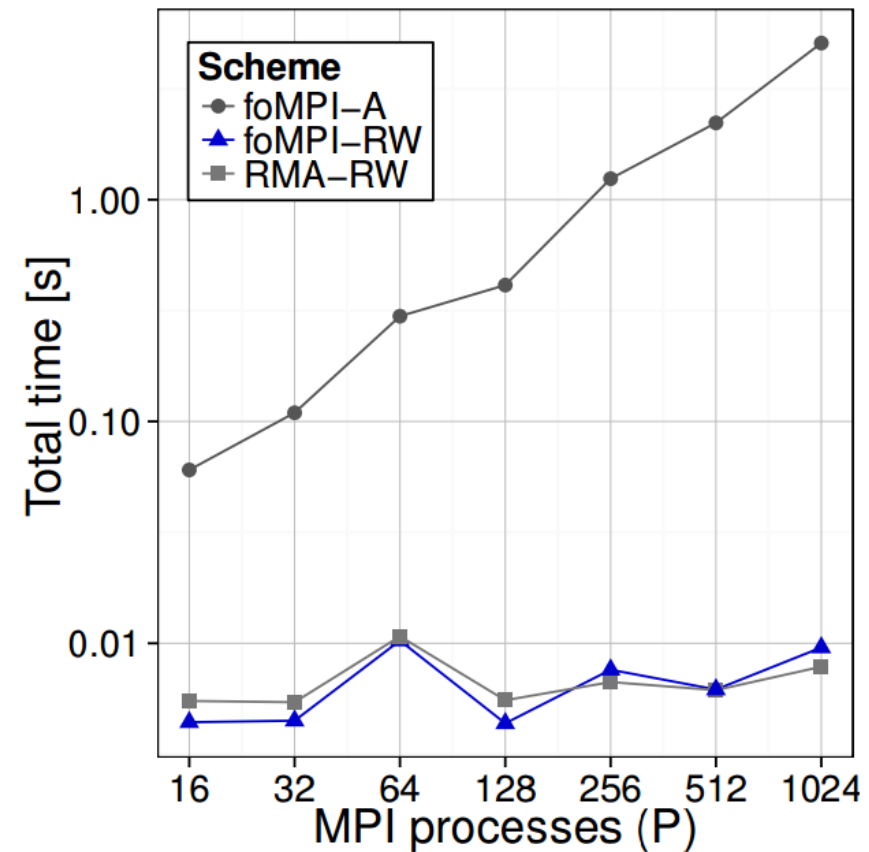


Evaluation - Distributed Hashtable

2% of writers



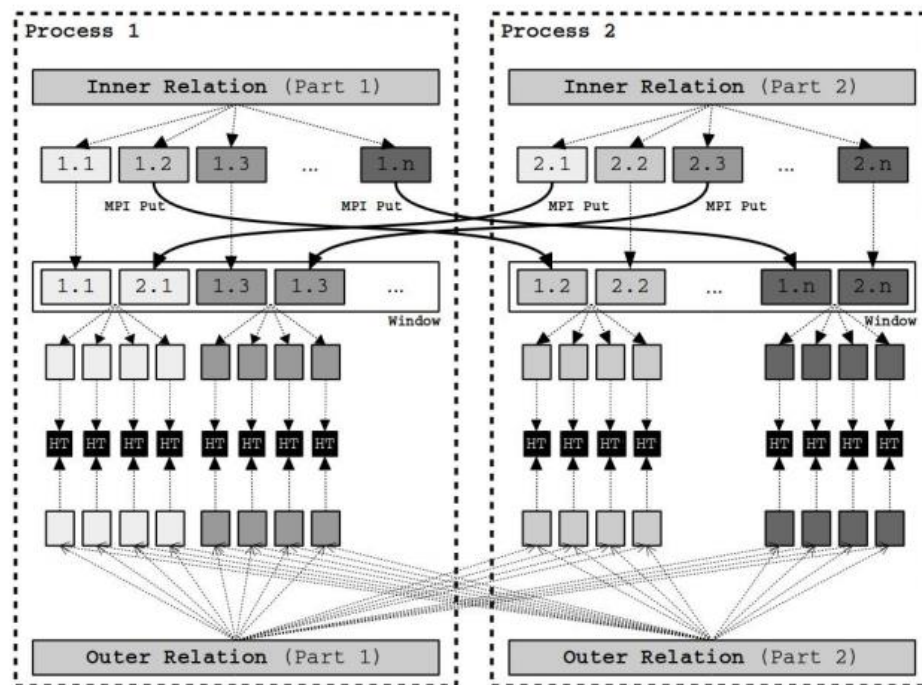
0% of writers



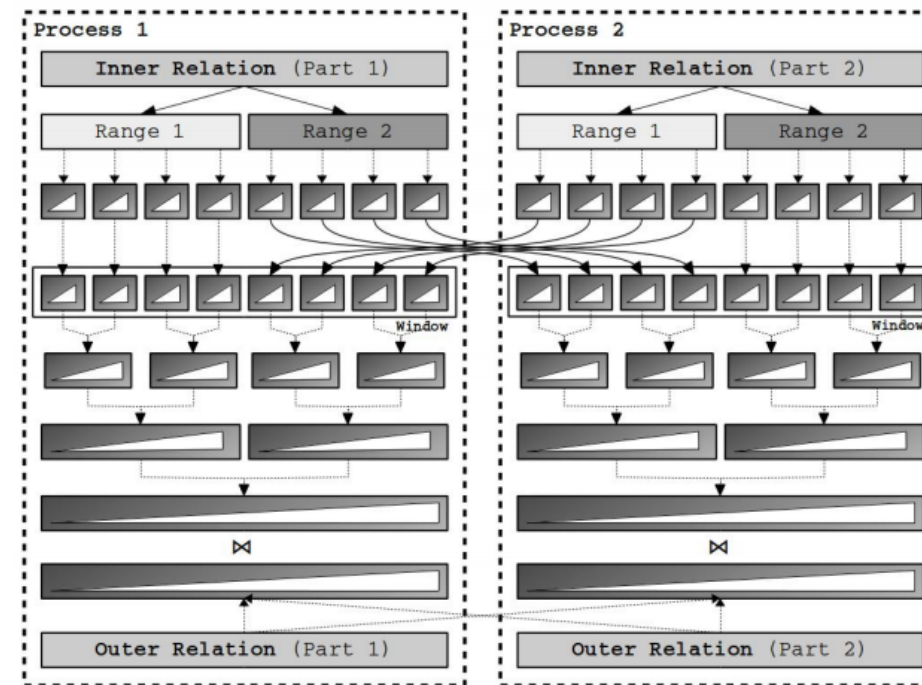
Another application area - Databases

- MPI-RMA for distributed databases?

Hash-Join

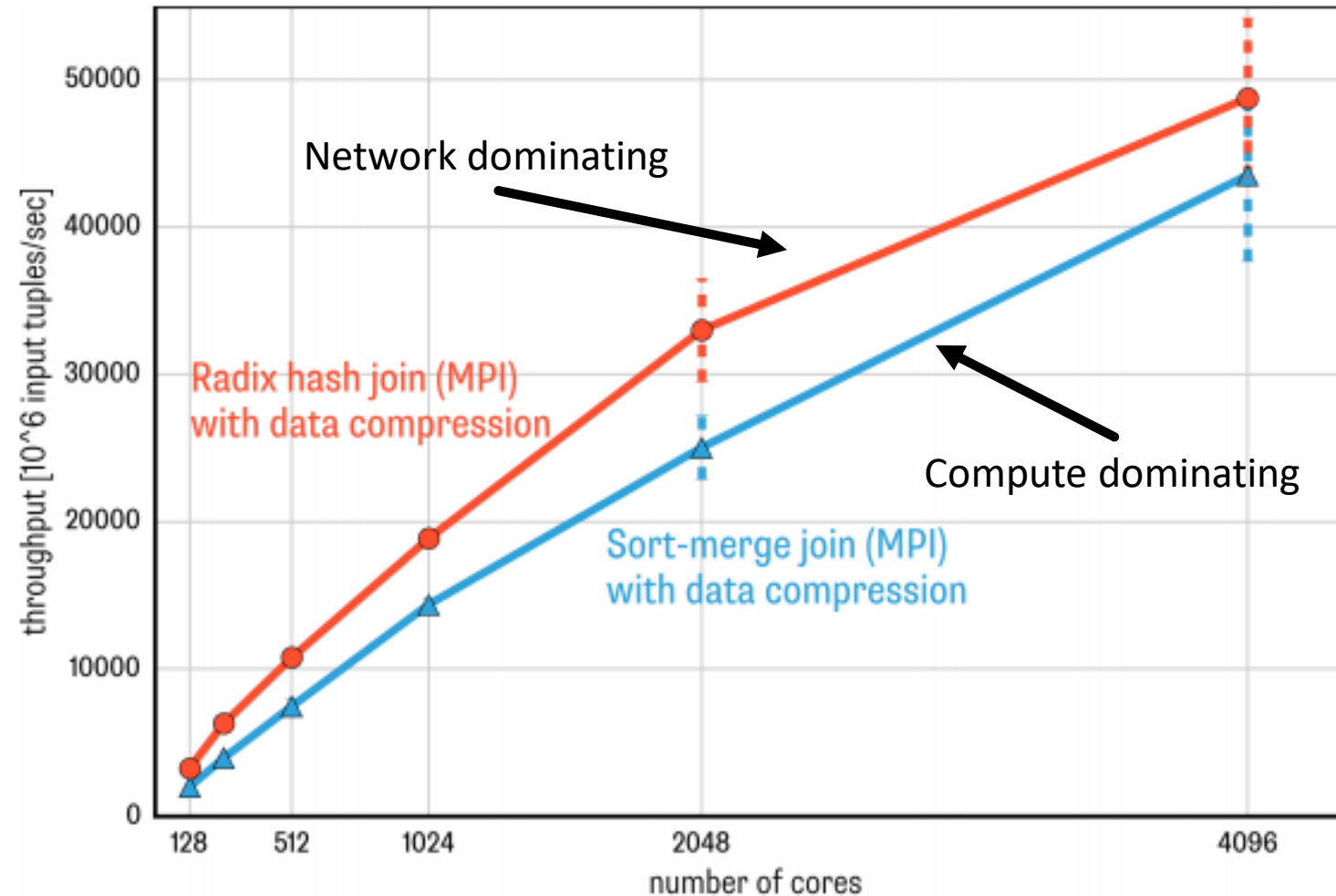


Sort-Join



Another application area - Databases

- MPI-RMA for distributed databases on Piz Daint



Another application area - Databases

- MPI-RMA for distributed databases on Piz Daint

