Disruptor.Net

comit

Rasmus Nygaard Andersen

What is difficult in concurrency?

Sharing data between threads

Many solutions: Locks, Semaphores, CAS, Actors...

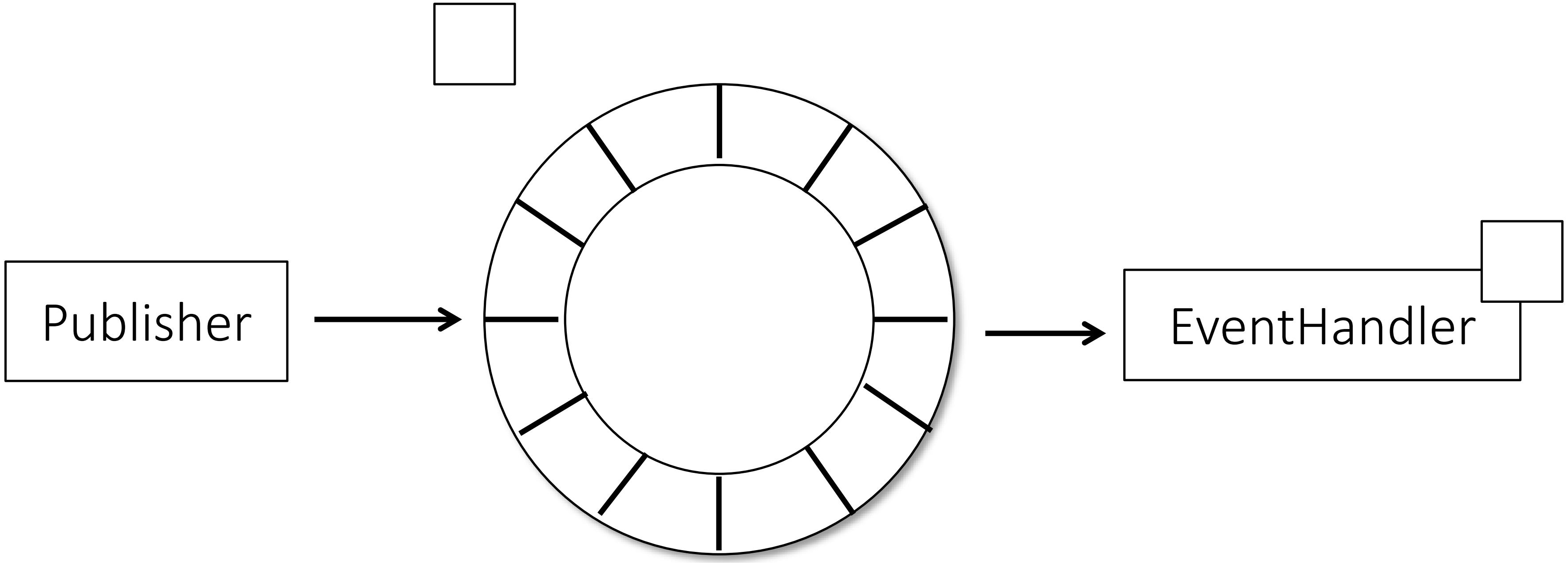
What is Disruptor?

Framework for handling concurrency in an easy and performance efficient way

More specifically it helps with passing data between threads

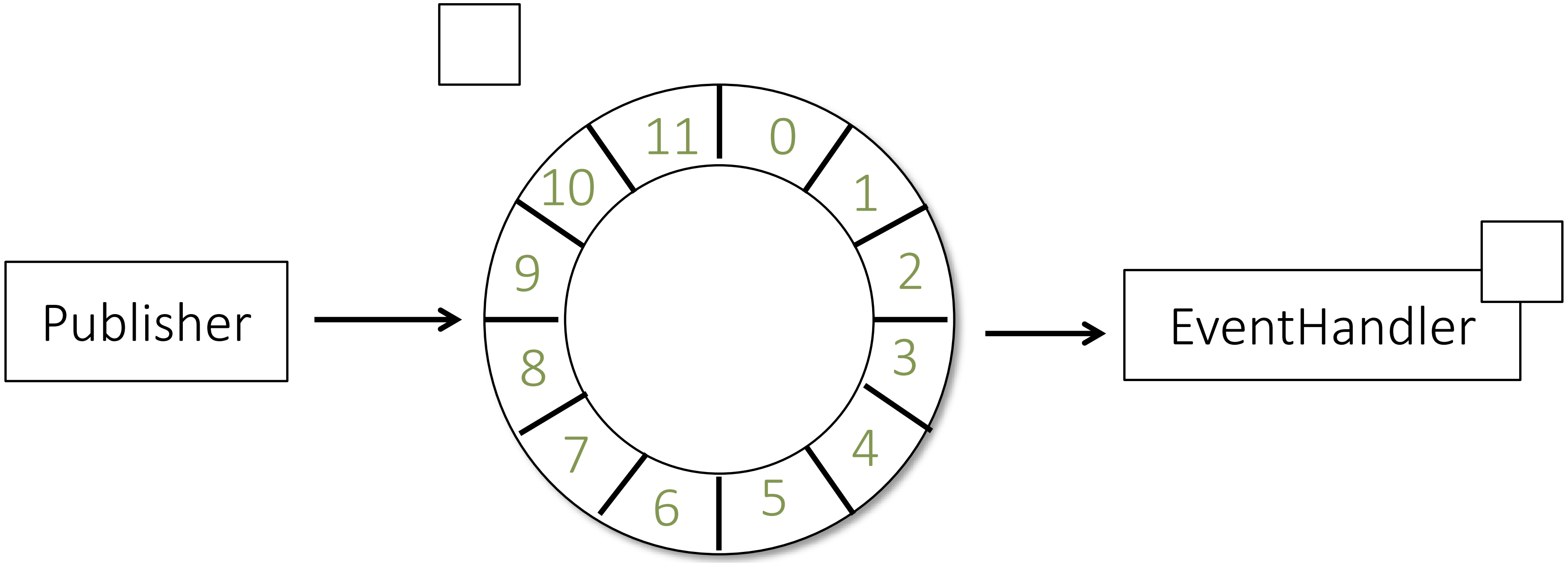
Part 1:

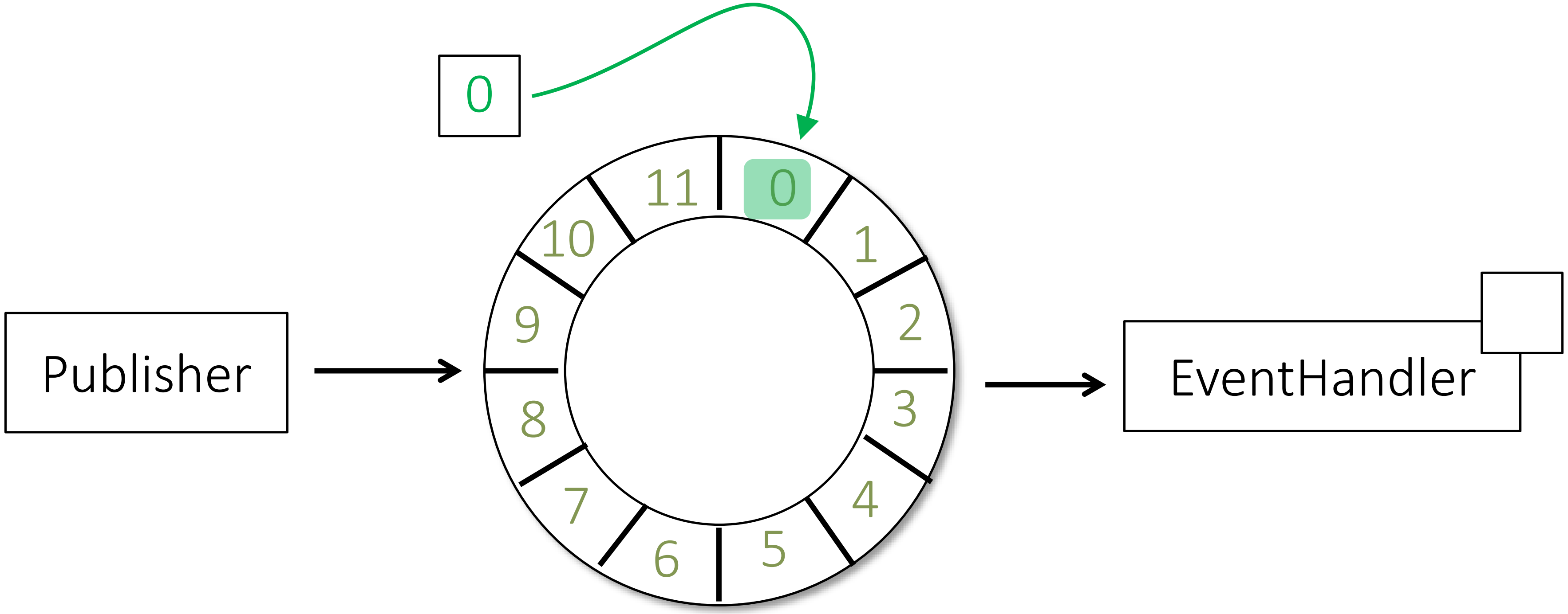
How to use Disruptor

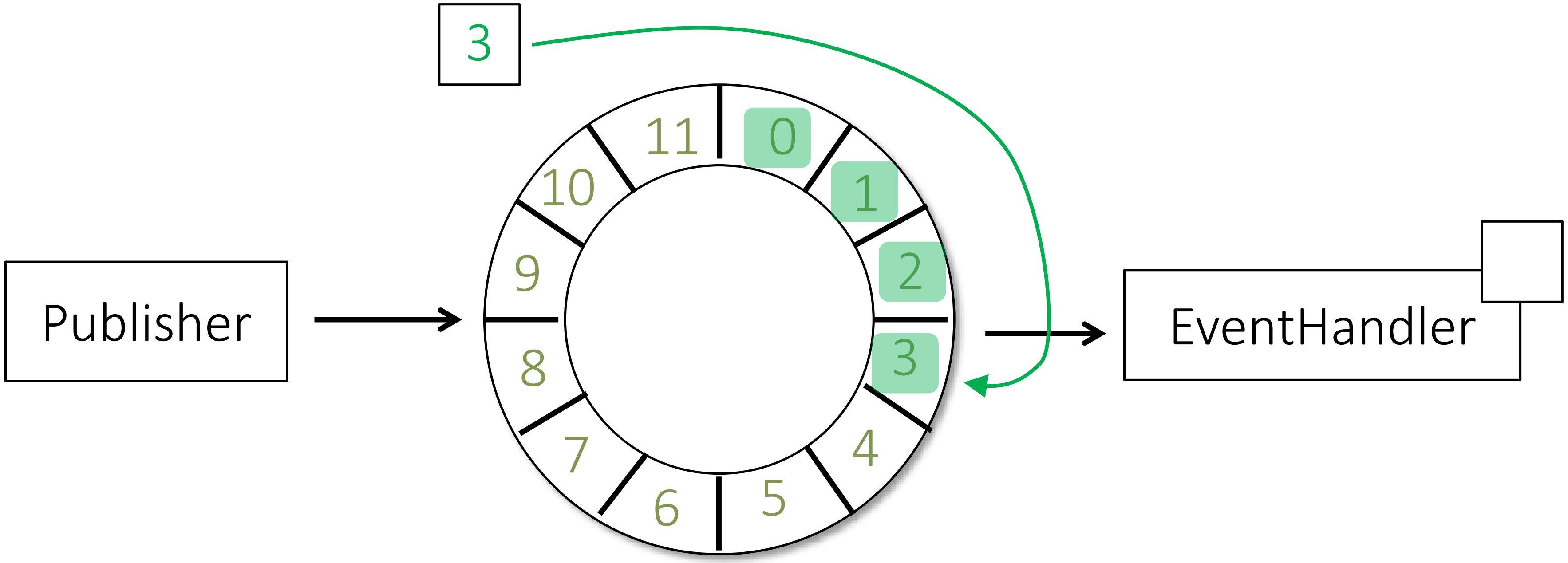


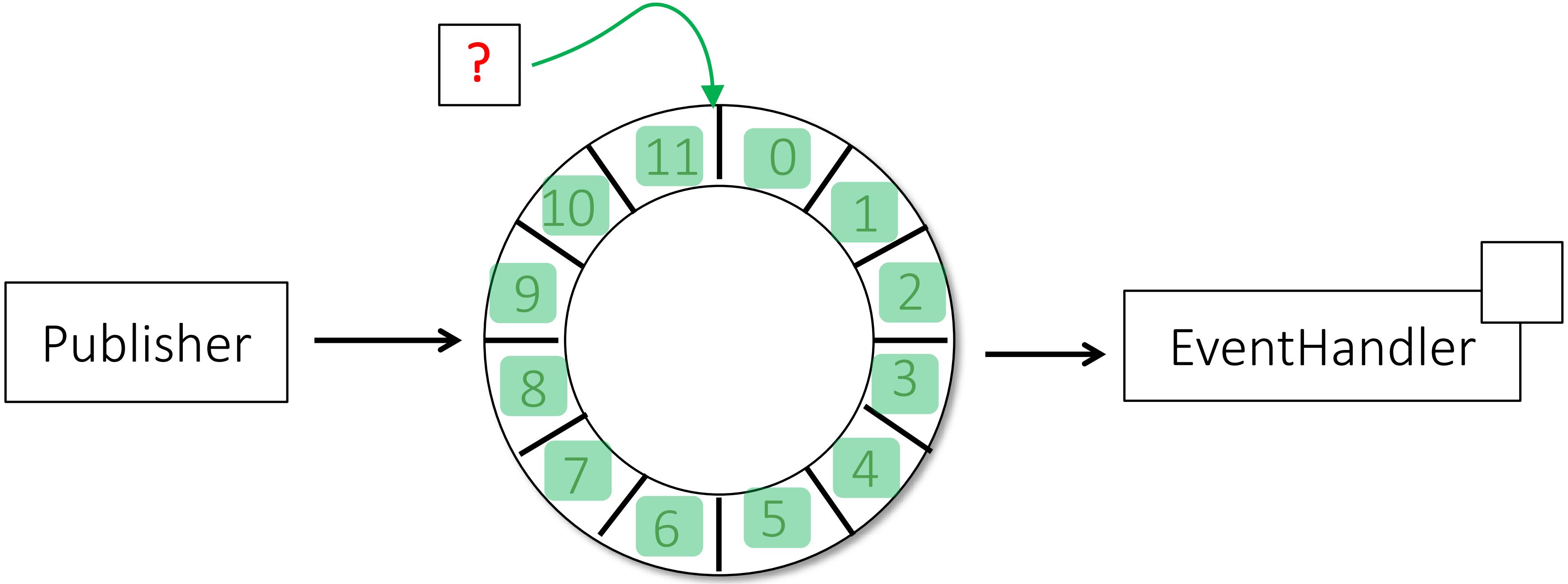
Publisher

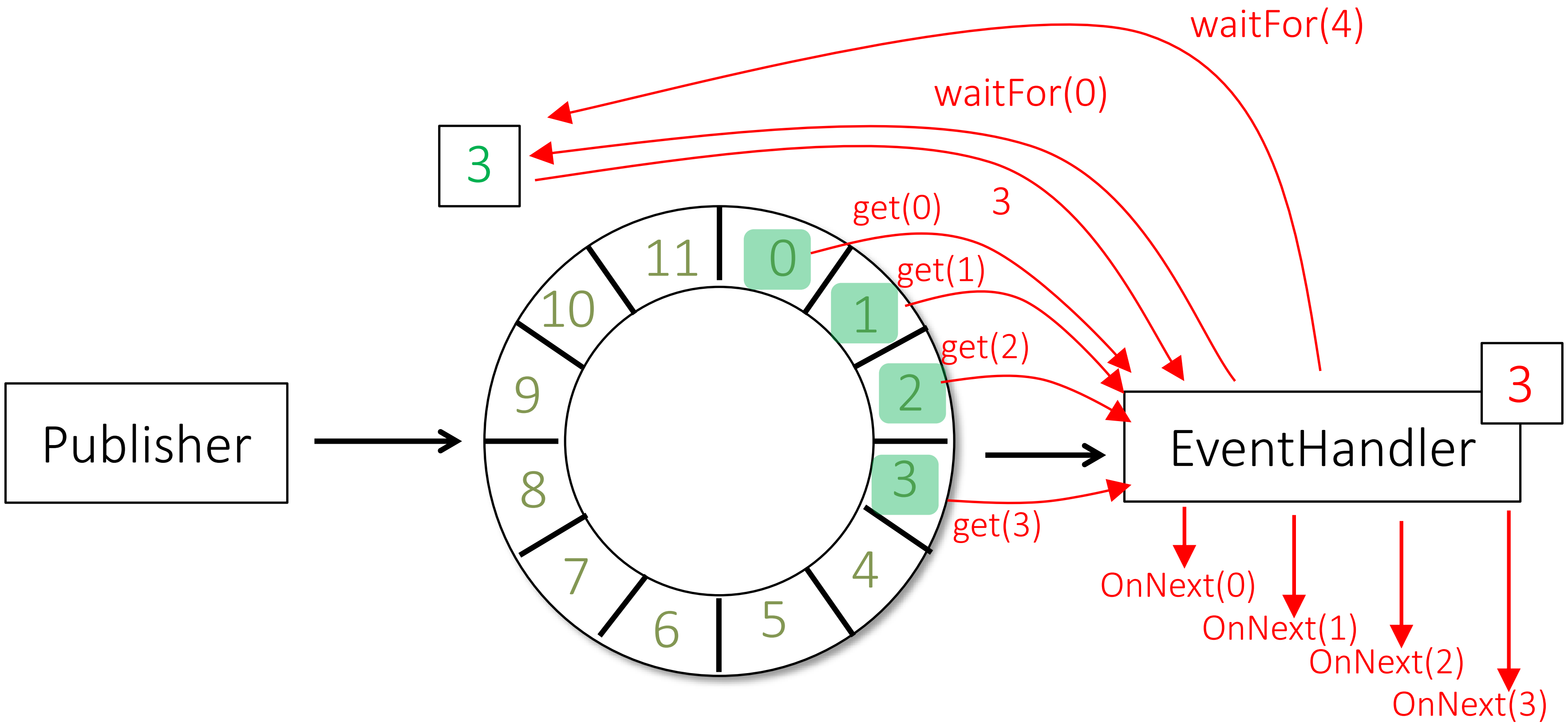
EventHandler


Publisher
11
0
1
2
3
4
5
6
7
8
9
10
EventHandler


Publisher
0
11
0
1
2
3
4
5
6
7
8
9
10
EventHandler


3
Publisher
11
0
1
2
3
4
5
6
7
8
9
10
EventHandler


?
Publisher
11
0
1
2
3
4
5
6
7
8
9
10
EventHandler


waitFor(4)
waitFor(0)
3
get(0)
3
get(1)
get(2)
get(3)
Publisher
0
1
2
3
4
5
6
7
8
9
10
11
EventHandler
3
OnNext(0)
OnNext(1)
OnNext(2)
OnNext(3)

DEMO

Exceptions

- What happens if an error occurs in an EventHandler?
  - Default is that the Disruptor stops processing more events
  - However you can implement your own event handler and decide your self.

- How you should handle it highly depends on the application
  - Log the exception and keep processing
  - Stop the application because something is terribly wrong
  - ...

Ringbuffer is full

- What happens when the ringbuffer is full?
- The publisher blocks and waits for a slot to become available
- This is a good thing because you don't want the publisher to fill up the memory and crash the application
  - You can decide on exactly how much the publisher can fill memory by the size of the ring-buffer
- Instead you create back-pressure through the system
- It is also possible to monitor how full the ring-buffer so you can get warned if it is too full

Multiple publishers

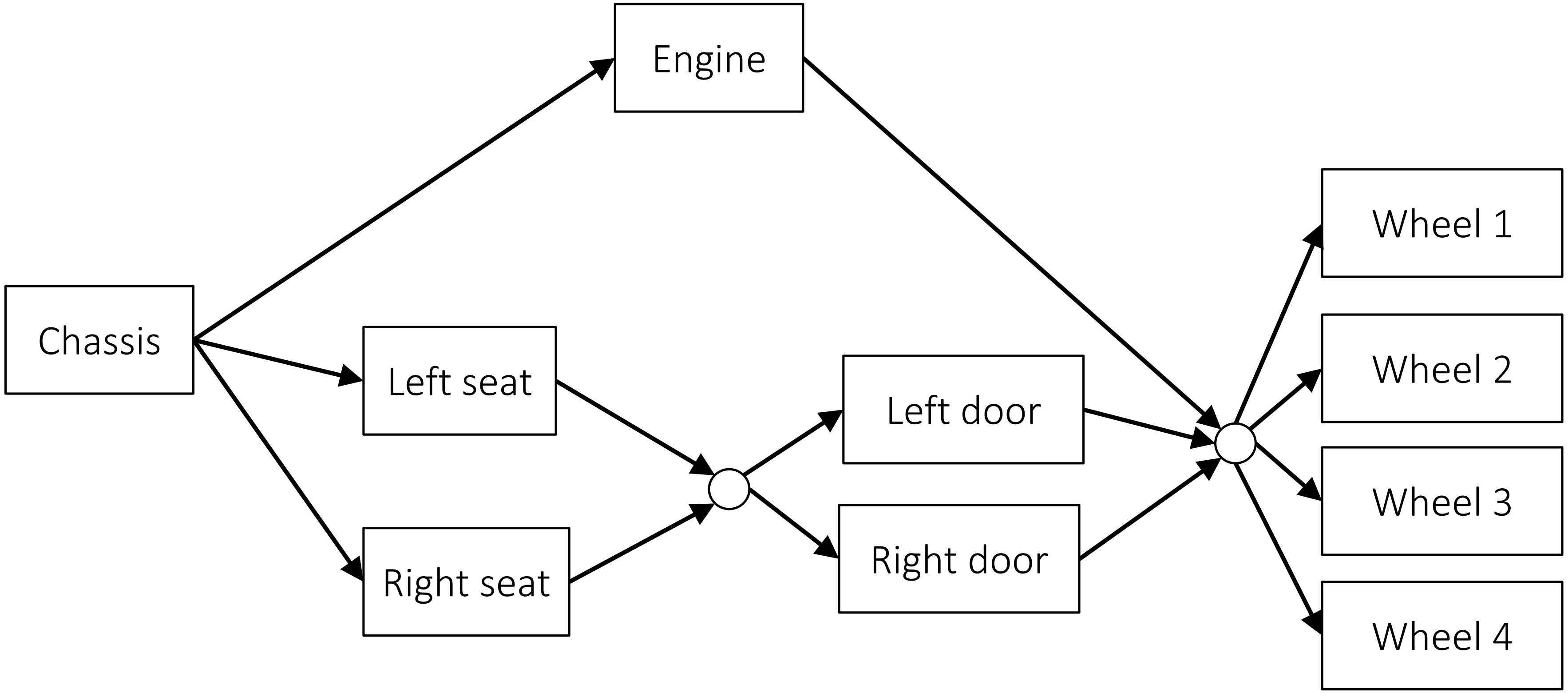
- It is possible to have multiple publishers publishing to the same ringbuffer
- However you need to configure the Disruptor to handle this
  - Change Claim Strategy to Multi Threaded

Workflows

- Multiple handlers is also possible
- And these handlers can be dependend on each other
  - So you can define handlers only to run after others have finished
- This means that you can built more complex workflows than just thread-to-thread communication

Car assembly workflow

Complex workflow



Engine

Chassis

Left seat

Right seat

Left door

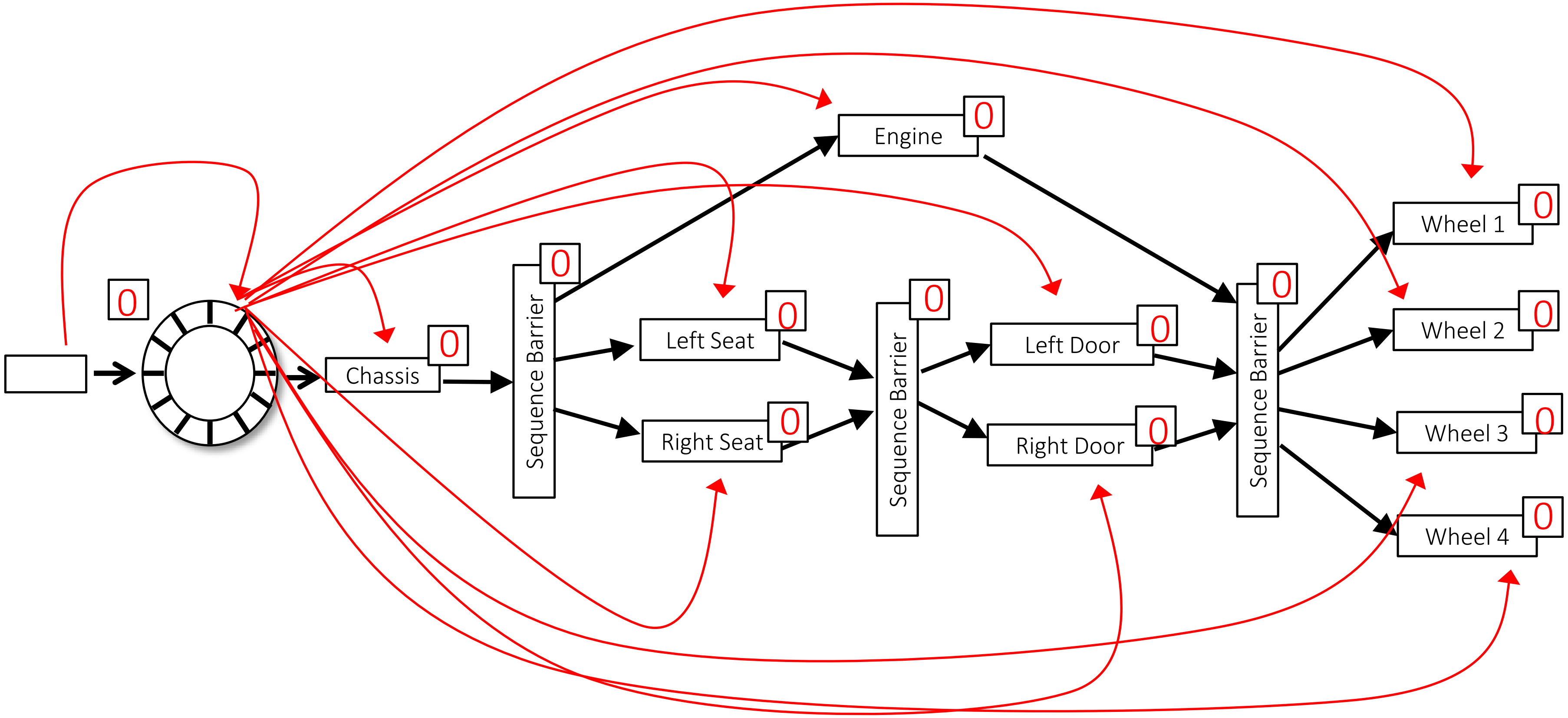
Right door

Wheel 1

Wheel 2

Wheel 3

Wheel 4


0
Chassis
0
Sequence Barrier
0
Engine
0
Left Seat
0
Right Seat
0
Sequence Barrier
0
Left Door
0
Right Door
0
Sequence Barrier
0
Wheel 1
0
Wheel 2
0
Wheel 3
0
Wheel 4
0

comrit

Part 2:

Why is Disruptor so fast?

Why is Disruptor so fast?

- Mechanical Sympathy
  - You need to understand your hardware to go really fast
  - Mechanical sympathy was a term coined by Jackie Stewart during his time as a Formula 1 driver. It was his opinion that the best drivers were those who also best understood the machines they drove.
  - *"The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry."*

Mechanical Sympathy

- No locks
- Use memory barriers
- Avoid write contention
- Avoid false sharing
- Warm caches
- Pre-allocating objects => avoiding garbage collection
- Batching

Avoid locks

- Disruptor avoids locks

- Locks can be slow
  - Especially when multiple threads are trying to acquire the same lock at the same time

Locks can be slow

- Pseudo-code: Acquiring a lock
  - Spin-wait loop for a while trying to obtain the lock (user-mode)
  - If no success thread is switched to the *kernel-mode* and waits for signal from kernel to receive the lock
  - When lock is released by a thread the kernel then decides which thread to run next
- Overhead of acquiring the lock
  - Pollution of caches because switching to kernel
  - Context switch from user- to kernel-space (~1000 CPU cycles)
  - Overhead of managing queue of threads

Example of lock contention

Increment a counter 500.000.000 times

One Thread:	300 ms	
One Thread (volatile)	4.700 ms	(15x)
One Thread (Atomic)	5.700 ms	(19x)
One Thread (Lock)	10.000 ms	(33x)
Two Threads (Atomic)	30.000 ms	(100x)
Two Threads (Lock)	224.000 ms	(746x)

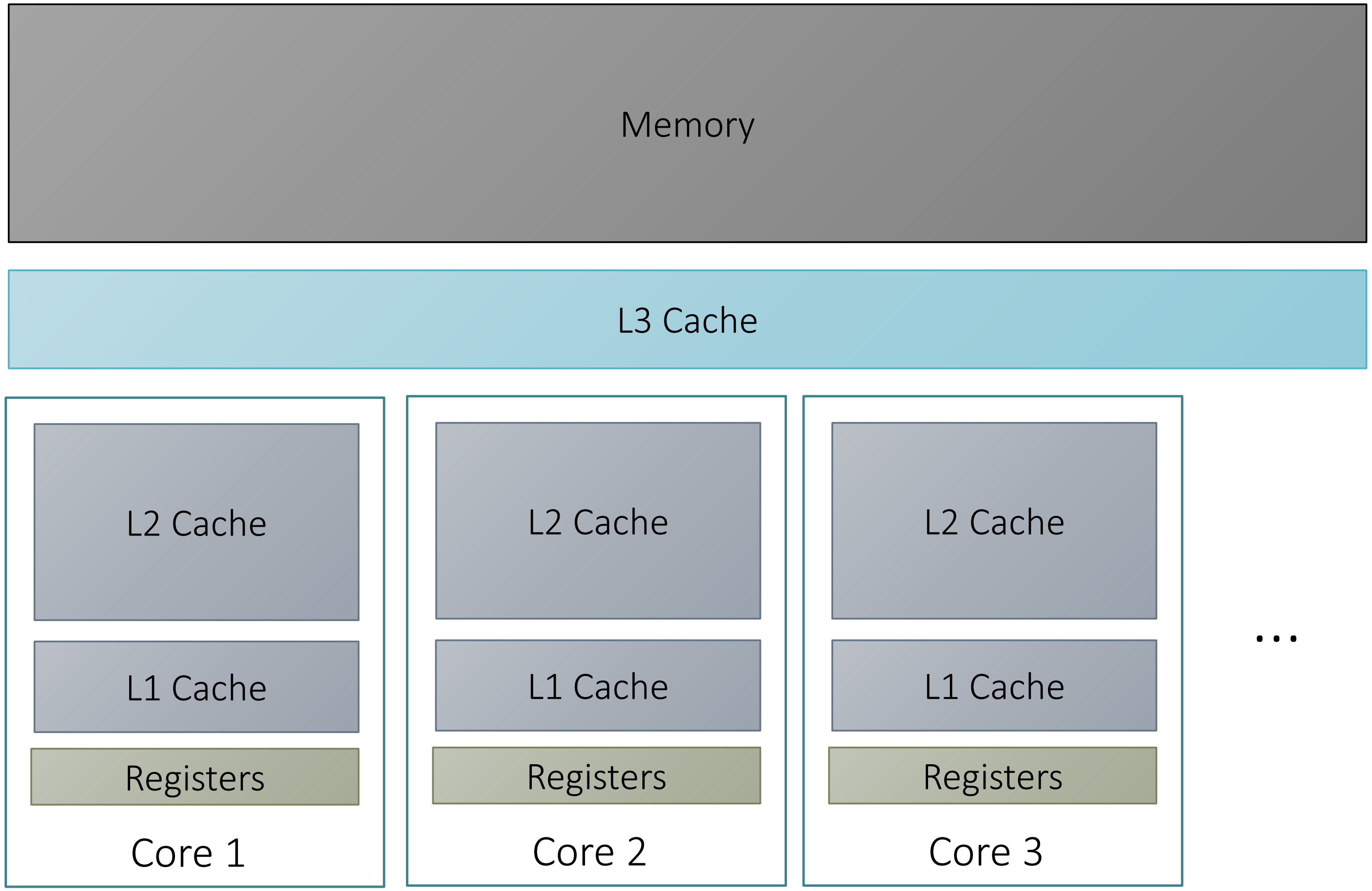
Instead use memory barriers

- Instead of locks then Disruptor uses memory barriers to coordinate sharing data between threads

Use memory barriers

- Make memory visible from a processor core to other processor cores
  - Store barrier
  - Load barrier
- CPU instruction
  - Ensure the order in which certain operations are executed
  - Influence visibility of some data
- Complex subject and different across CPU architectures

http://mechanical-sympathy.blogspot.dk/2011/07/memory-barriersfences.html



Memory

L3 Cache

L2 Cache

L1 Cache

Registers

Core 1

L2 Cache

L1 Cache

Registers

Core 2

L2 Cache

L1 Cache

Registers

Core 3

…

Reorder CPU instructions

Thread 1:

```
a = 1;
b = 1;
b = readFromHd();
while(a < 500){
    a++;
}
```

Thread 2:

```
while(true){
    Console.WriteLine(“a: ” + a + “b: ” + b);
}
```

Reorder CPU instructions

Thread 1:

```
a = 1;
b = 1;
start readFromHd();
while(a < 200){
    a++;
}
b = receive readFromHd();
while(a < 500){
    a++;
}
```

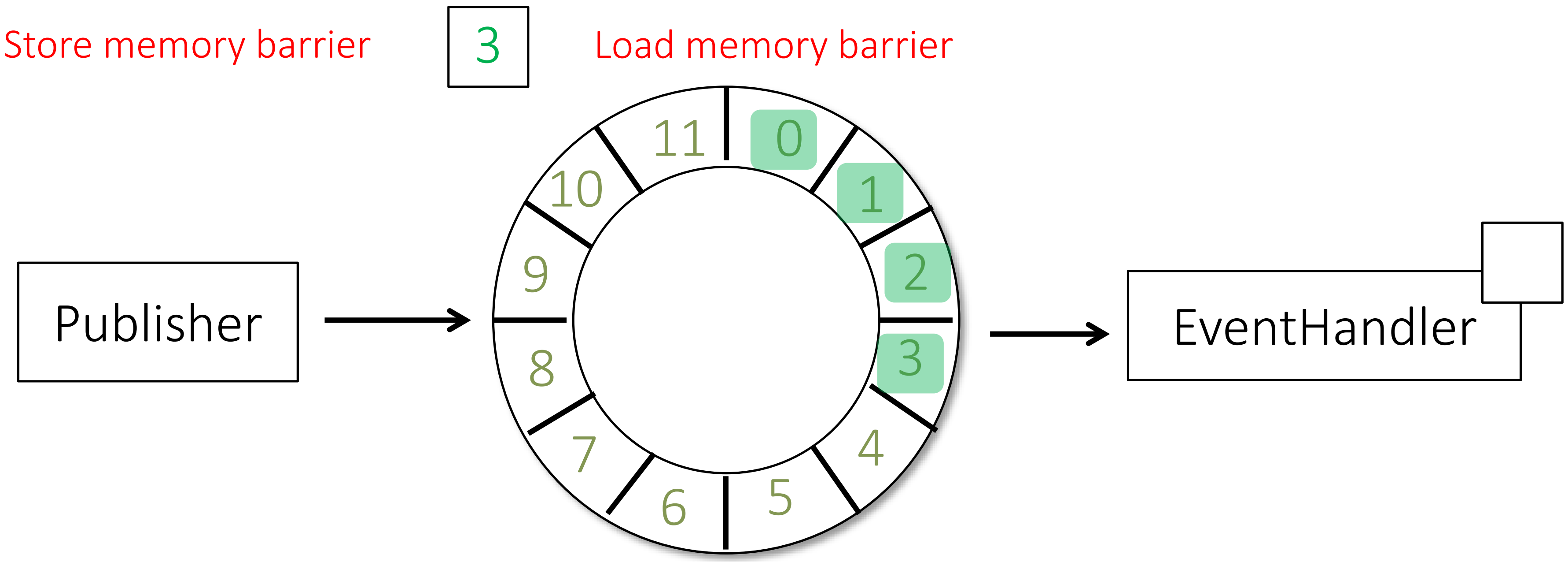
Thread 2:

```
while(true){
    Console.WriteLine(“a: ” + a + “b: ” + b);
}
```

Memory barriers in .Net

- Volatile keyword
  - Do a load memory barrier before reading a volatile variable
  - Do a store memory barrier after writing a volatile variable
- Memory Barrier API
  - Explicitly do memory barriers

Memory barrier in Disruptor



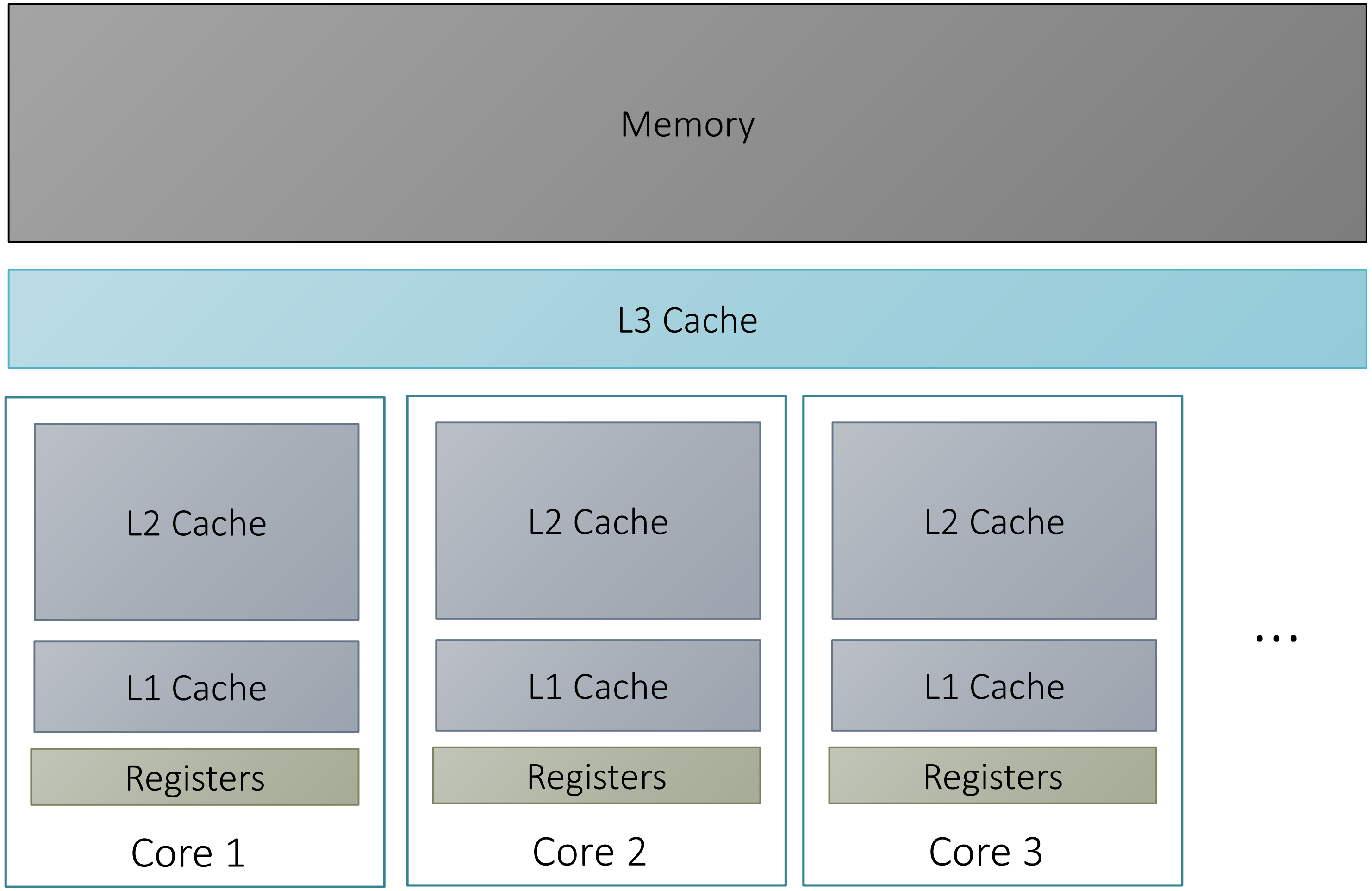
Memory barriers != locks

- Memory barriers are not substitution for locks
  - (Locks are many times implemented using memory barriers)

Avoid write contention

- Write contention
  - Multiple threads writing to the same data
    - *Need to coordinate who has right to write*
    - *Need ping-pong data between threads*
- Single writer principle
  - Ensure only one thread is writing to data, hence different threads are not battling for the same resource

http://mechanical-sympathy.blogspot.dk/2011/09/single-writer-principle.html



Memory

L3 Cache

L2 Cache

L1 Cache

Registers

Core 1

L2 Cache

L1 Cache

Registers

Core 2

L2 Cache

L1 Cache

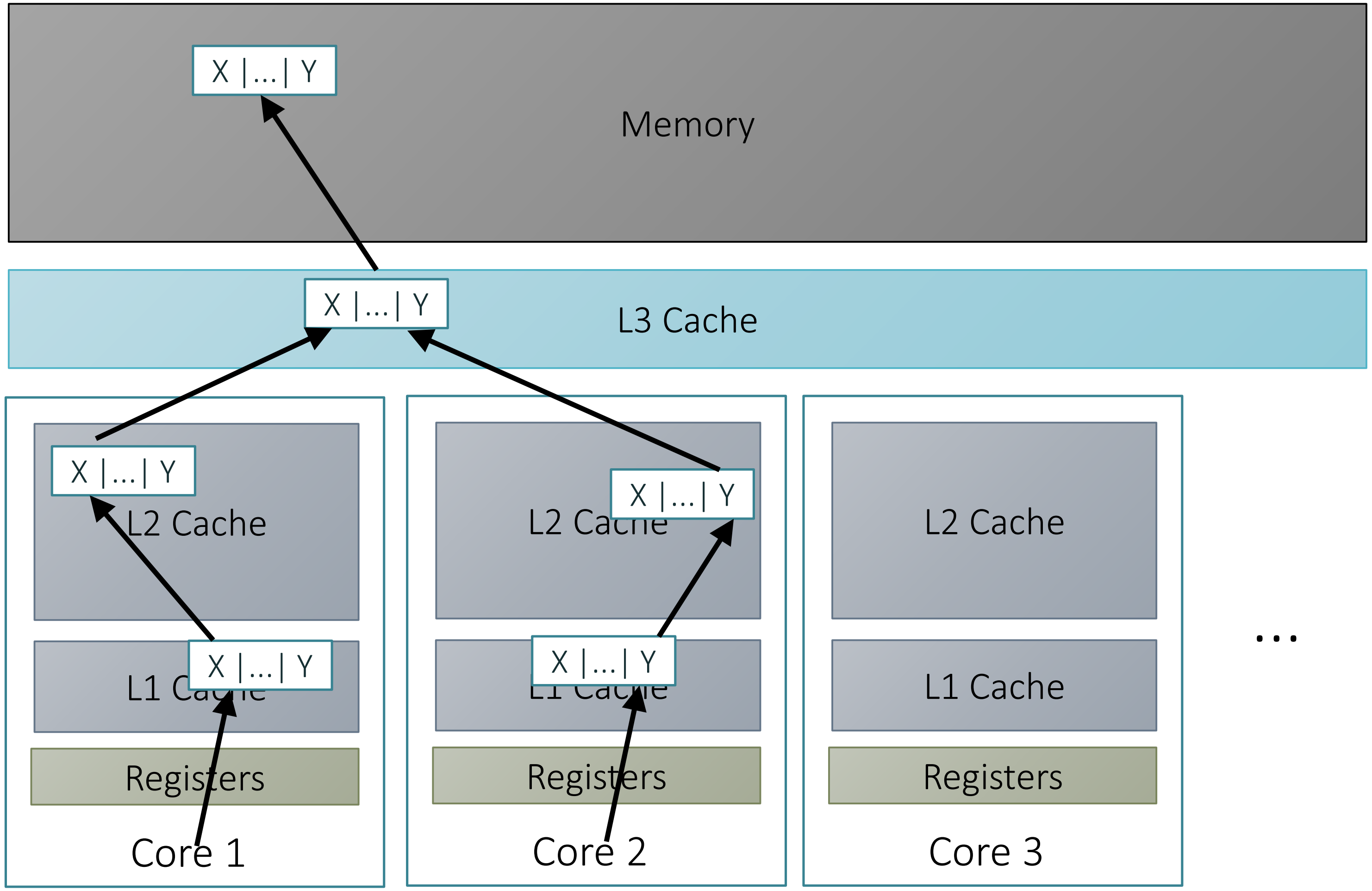
Registers

Core 3

…

False sharing

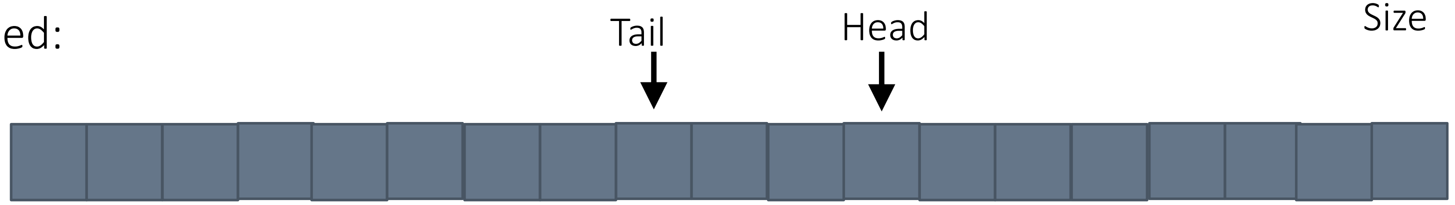
- Special type of write contention
- Memory is stored within the cache system in units know as cache lines (most common cache line size is 64 bytes)
- Happens when threads unwittingly impact the performance of each other while modifying independent variables sharing the same cache line causing a ping-pong of sending data back and forward between cores.
- Disruptor ensure that different threads are not battling for the same cache lines
  - You can see padding in the Disruptor source code.


X |…| Y
Memory
X |…| Y
L3 Cache
X |…| Y
L2 Cache
X |…| Y
L1 Cache
Registers
Core 1
X |…| Y
L2 Cache
X |…| Y
L1 Cache
Registers
Core 2
L2 Cache
L1 Cache
Registers
Core 3
…

What is wrong with ”normal” queues?

- They tend to use locks (or similar) to support multiple producer and consumers
- They tend to have write contention
  - Typically use either linked-lists or arrays for the underlying storage of elements.
  - This means that they tend to have write contention on the head, tail or/and size variables

Array-based:



Batching

- In Disruptor the consumers (EventHandlers) can receive multiple events in a batch.
- This can help consumers catch up with producers if they have come behind.
- For example
  - A consumer are serializing data and sending it out on the network.
  - Then a sudden burst of data comes from the producer.
  - The consumer is able to read all the data from the ringbuffer in a batch and serialize it into the network buffer and send it in one go.

http://mechanical-sympathy.blogspot.dk/2011/10/smart-batching.html

Pre-allocating objects (avoiding garbage collection)

- The objects in the ring-buffer are pre-allocated
  - These objects can be reused
- This results in less garbage
- If you receive 1 million messages per second and you don’t reuse objects then the garbage collector has alot to collect.
- No stop-the-world garbage collection => More predictable latency

Warm cache

- CPU cores are incredible fast
- What many times are slowing them down is I/O
  - L1 < L2 < L3 < RAM < Hard-drive
- Pollute your cache as little as possible and coordinate the I/O so your program can run as much from cache as possible

So how fast?

Disruptor.Net comes with a performance test that test Disruptor against BlockingCollection

- Throughput / Latency

```
Model: Lenovo Z510

Operating System: Microsoft Windows 8.1
- Version: 6.3.9600
 - ServicePack: 0

Number of Processors: 1
 - Name: Intel(R) Core(TM) i7-4702MQ CPU @ 2.20GHz
 - Description: Intel64 Family 6 Model 60 Stepping 3
 - ClockSpeed: 2201 Mhz
 - Number of cores: 4
 - Number of logical processors: 8
 - Hyperthreading: ON

Memory: 8104 MBytes
 - L1Cache: 64 KBytes
 - L2Cache: 256 KBytes
 - L3Cache: 6144 KBytes
```

Throughput

- Uni cast: 1 Producer 1 Consumer
  - Disruptor: 28.368.794 ops
  - BlockingCollection: 1.824.817 ops

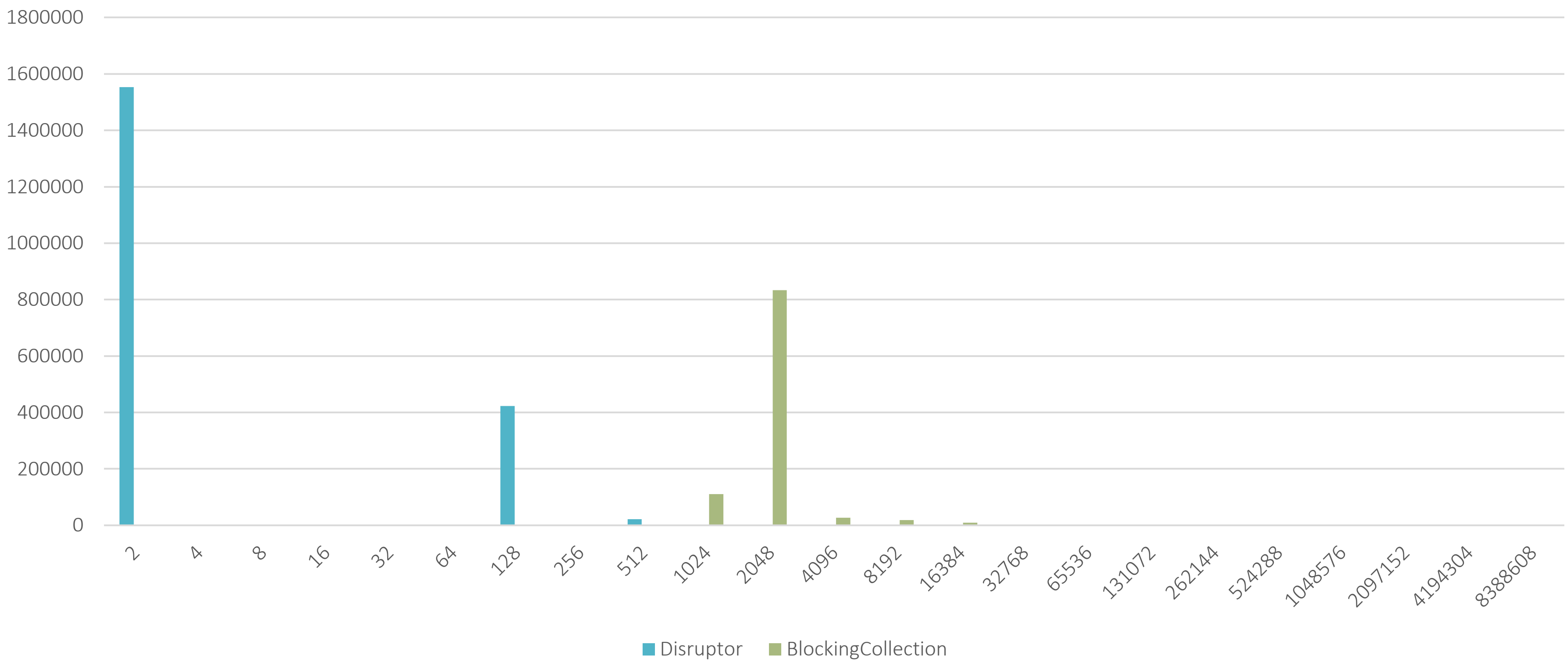
- Multi cast: 1 Producer 3 Consumers
  - Disruptor: 24.832.381 ops
  - BlockingCollection: 685.400 ops

- Pipeline 3 Step
  - Disruptor: 24.160.425 ops
  - BlockingCollection: 791.765 ops

- Sequencer: 3 Producer 1 Consumer
  - Disruptor: 11.529.592 ops
  - BlockingCollection: 3.344.481 ops

Latency

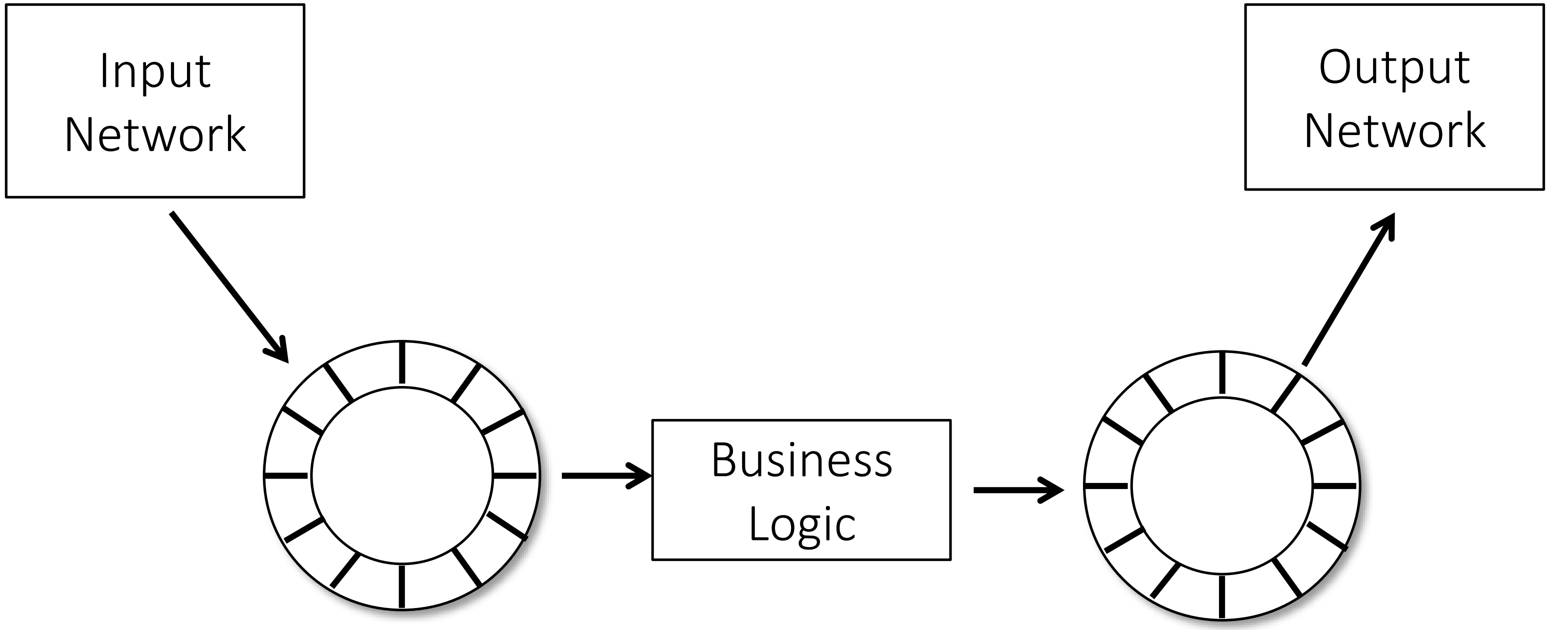
- Pipeline: 3 Step
  - Disruptor: mean=19.87, 99%=512, 99.99%=16384
  - BlockingCollection: mean=1806.51, 99%=16384, 99.99%=65536



Part 3:

How is Disruptor typically used

Typical Disruptor application



Typical Disruptor application

- Business Logic Processor
  - Takes input messages sequentially
  - Runs business logic on it
  - Emits output events

Typical Disruptor application

- Typically operates entirely in-memory
- No ORMs or other mapping to and from storage

Typical Disruptor application

- All business logic runs in a single thread

- There is no transactional behaviour since all processing is done sequentially
    - No need for locking, transactions, semaphores or other mechanisms

Error handling

- Traditional model with database and transactions
    - Anything go wrong, it's easy to throw the data away again

- Errors are harder to handle with an in-memory data structure since you cannot really throw it away.
    - So if there is an error it's important not to leave that memory in an inconsistent state.

Slow ”external” systems

- Do not block the thread!
  - No database calls from the main thread
  - No web-service calls from the main thread
  - Generally no blocking calls to slow systems

- Instead use async programming model
  - Start interaction with external system on separate thread
  - Resume processing of other events
  - When interaction with external system is done then reinsert event in disruptor
  - Process event again

Crashes

- What happens when if the computer crashes?
  - Do you lose your data?

- Events sourcing
  - All input events are stored
  - The current state of the is entirely derivable by processing the input events => Replay
  - Use snapshots to start-up faster

Fail-over

- Have multiple servers run the same processor
- Replicate the events to all the processors
- All but one processor has its output ignored

Part 4

History of Disruptor

LMAX

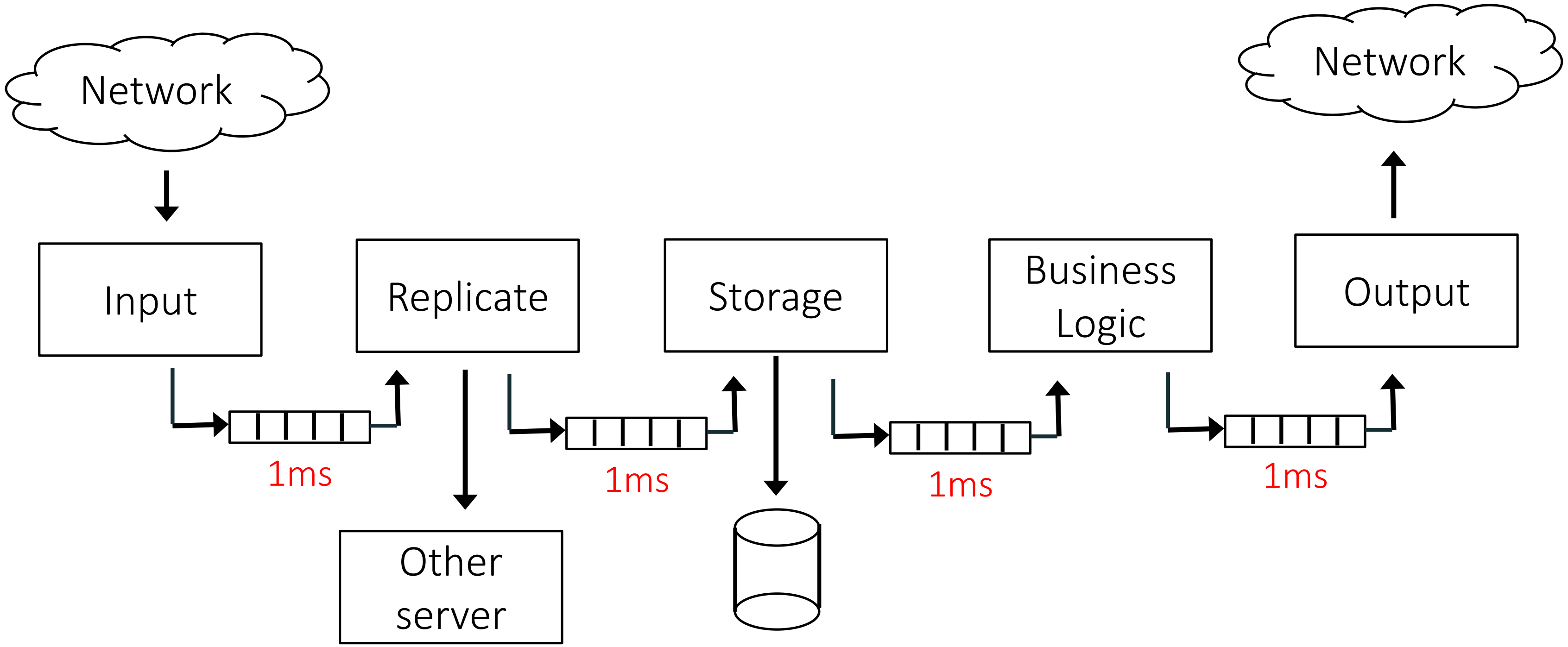
- London Multi-Asset Exchange
- Spin-off from Betfair
- Aimed to build a high performance financial exchange

LMAX tried

- RDBMS (Betfair)
- Java EE
- Actor
- SEDA (Staged event-driven architecture)
- ...

- But none of them worked very well performance-wise

So what was wrong?



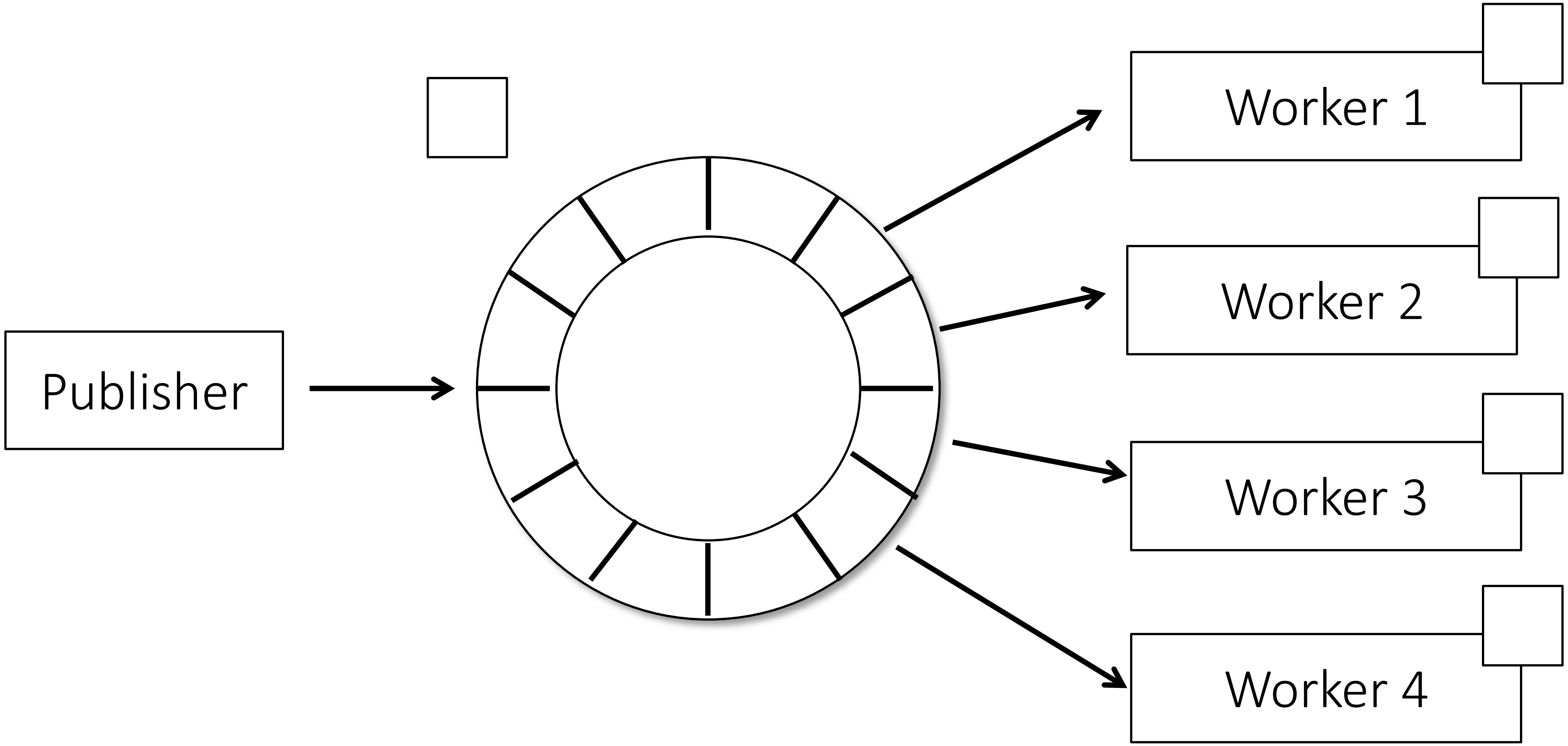
LMAX Performance

- 6.000.000 business transactions per second with latency less than 1 ms
- On commodity hardware

Part 5

Extras

Worker Pool



Publisher

Worker 1

Worker 2

Worker 3

Worker 4

Wait Strategies

- Event Handlers can wait for new events using different strategies
  - Balance between latency(/throughput) and CPU usage

- BusySpinWaitStrategy
  - Busy spin loop
  - Low latency, high CPU usage

- BlockingWaitStrategy
  - Lock with conditional variable
  - High latency, low CPU usage

- SleepingWaitStrategy
  - Uses SpinWait (Initially spins, then Thread.Yield/Thread.Sleep(0)/Thread.Sleep(1))
  - Compromise between latency and CPU usage

- YieldingWaitStrategy
  - Initially spinning followed by Thread.Sleep(0)
  - Compromise between latency and CPU usage

Claim Strategies

- To handle more producers you need to change claim strategy

- SingleThreadedClaimStrategy
  - Uses Memory Barriers
  - Only one producers (single writer principle)
  - ”Higher” performance

- MultiThreadedClaimStrategy
  - Uses CAS operations
  - Multiple producers
  - ”Lower” performance

- MultiThreadedLowContentionClaimStrategy
  - Uses CAS operations
  - Multiple producers
  - ”Lower” performance

Size of ringbuffer needs to be power of 2

- Optimization

- Use a special modulo function when pointer in ring-buffer needs to wrap around

- Modulo with power of 2 can be done using a simple bit mask

Other resources

- Disruptor (Java):
  http://lmax-exchange.github.io/disruptor/
- Disruptor.Net
  https://github.com/disruptor-net/Disruptor-net
- Trisha Gee: Concurrent Programming Using The Disruptor
  (http://www.infoq.com/presentations/Disruptor)
- Martin Thompson: Mechanical sympathy
  (http://mechanical-sympathy.blogspot.dk/)
- Martin Fowler: The LMAX Architecture
  (http://martinfowler.com/articles/lmax.html)

Contact

- rasmus@comiit.com
- https://www.linkedin.com/in/rasmusnygaardandersen