

Higher-Order Linearisability

Andrzej Murawski
University of Warwick

Nikos Tzevelekos
Queen Mary U. of London

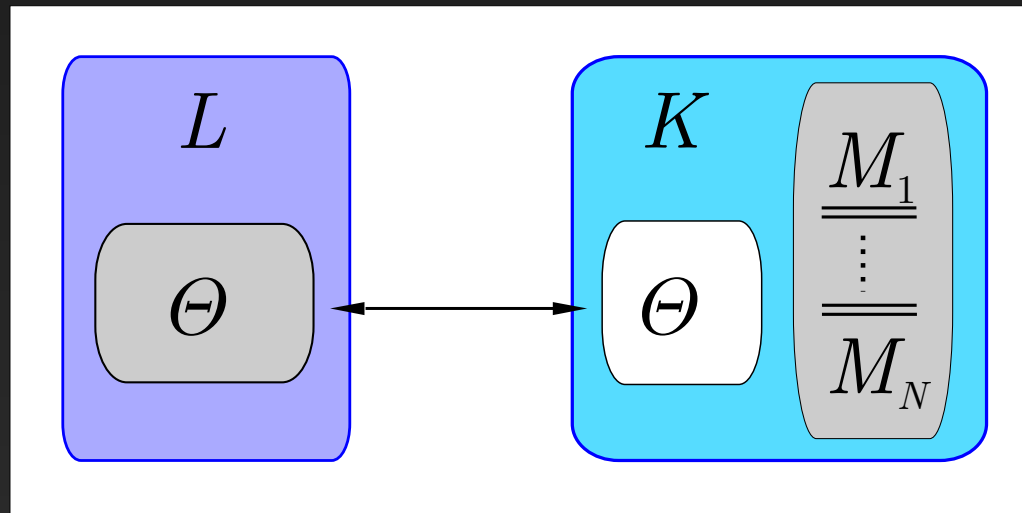
Highlights of Logic, Games and Automata, September 2017

Concurrent libraries

Concurrent *queue* library L , with methods:

enqueue: $\text{int} \rightarrow \text{void}$

dequeue: $\text{void} \rightarrow \text{int}$



Library $L : \Theta$, client K (multi-threaded)

$\Theta = \{ \mathbf{nq}: \text{int} \rightarrow \text{void}, \mathbf{dq}: \text{void} \rightarrow \text{int} \}$

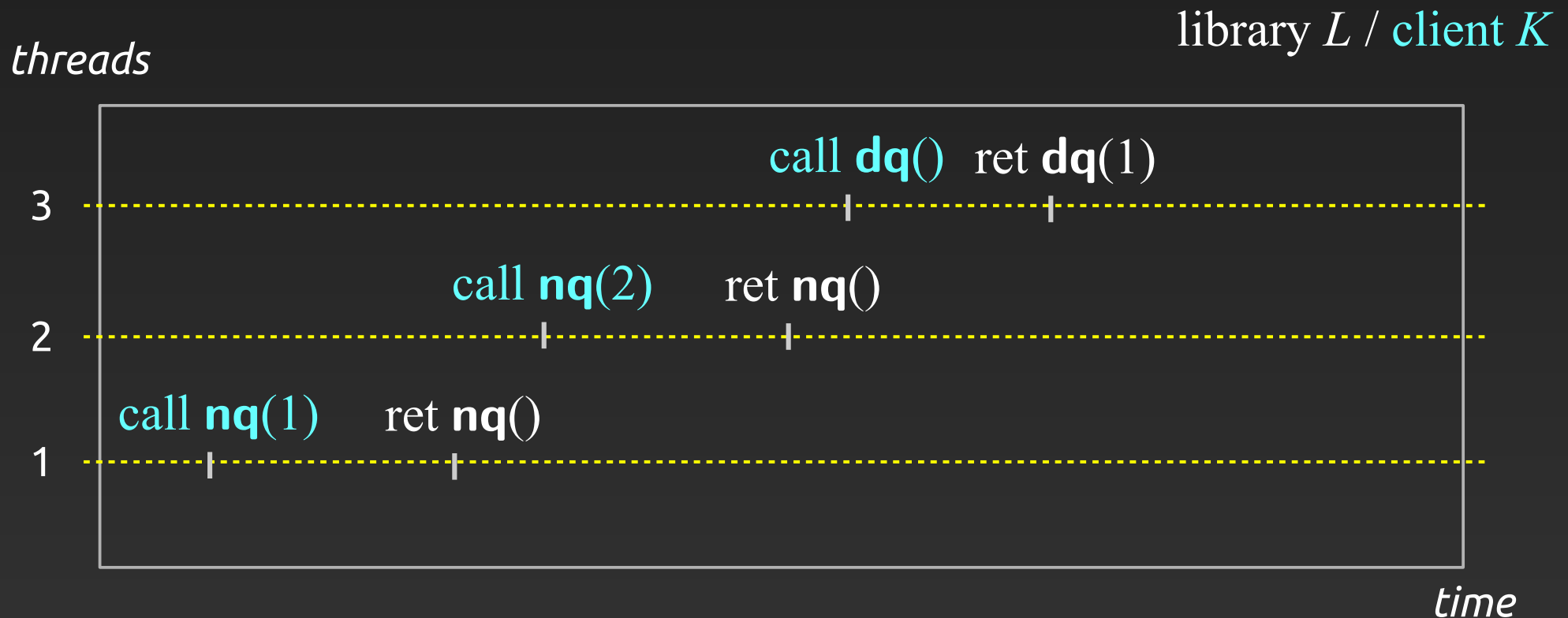
Concurrent queue behaviour

Concurrent *queue* library L , with methods:

enqueue: $\text{int} \rightarrow \text{void}$

dequeue: $\text{void} \rightarrow \text{int}$

A (correct) library behaviour:



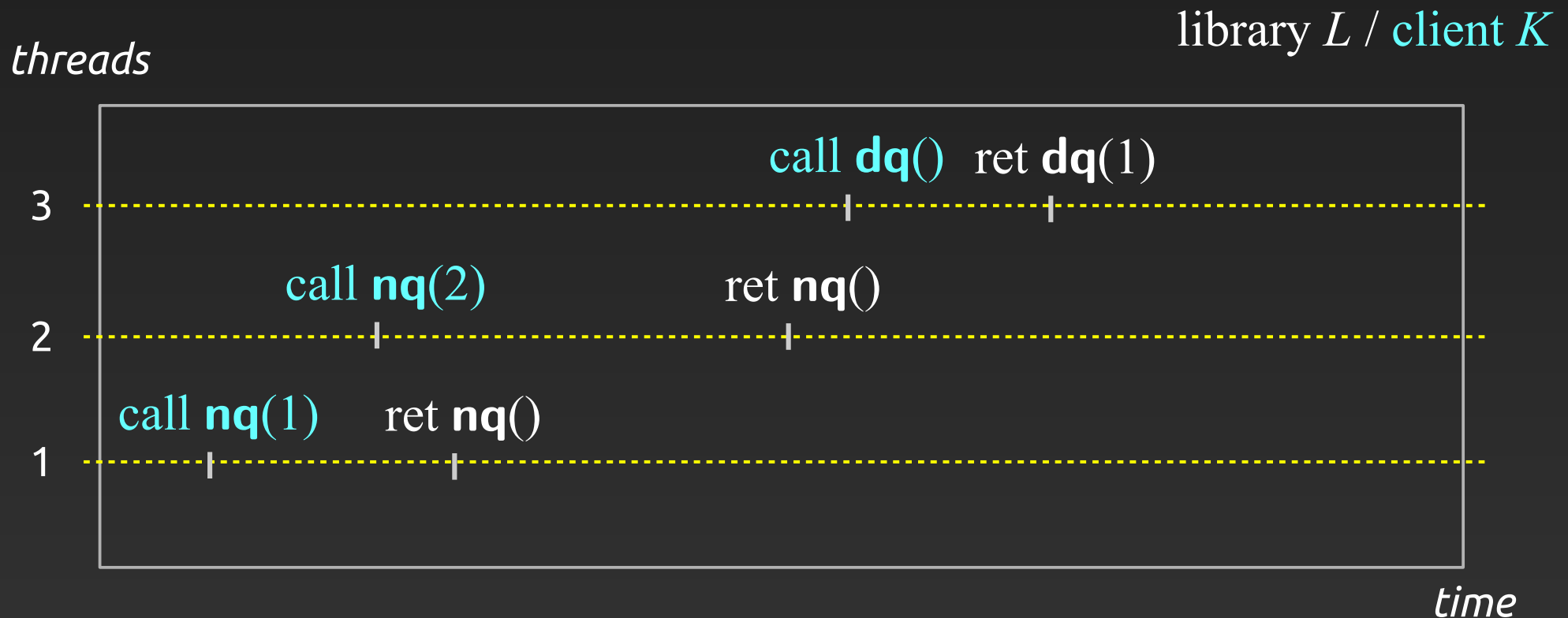
Concurrent queue behaviour

Concurrent *queue* library L , with methods:

enqueue: $\text{int} \rightarrow \text{void}$

dequeue: $\text{void} \rightarrow \text{int}$

A (correct) library behaviour:



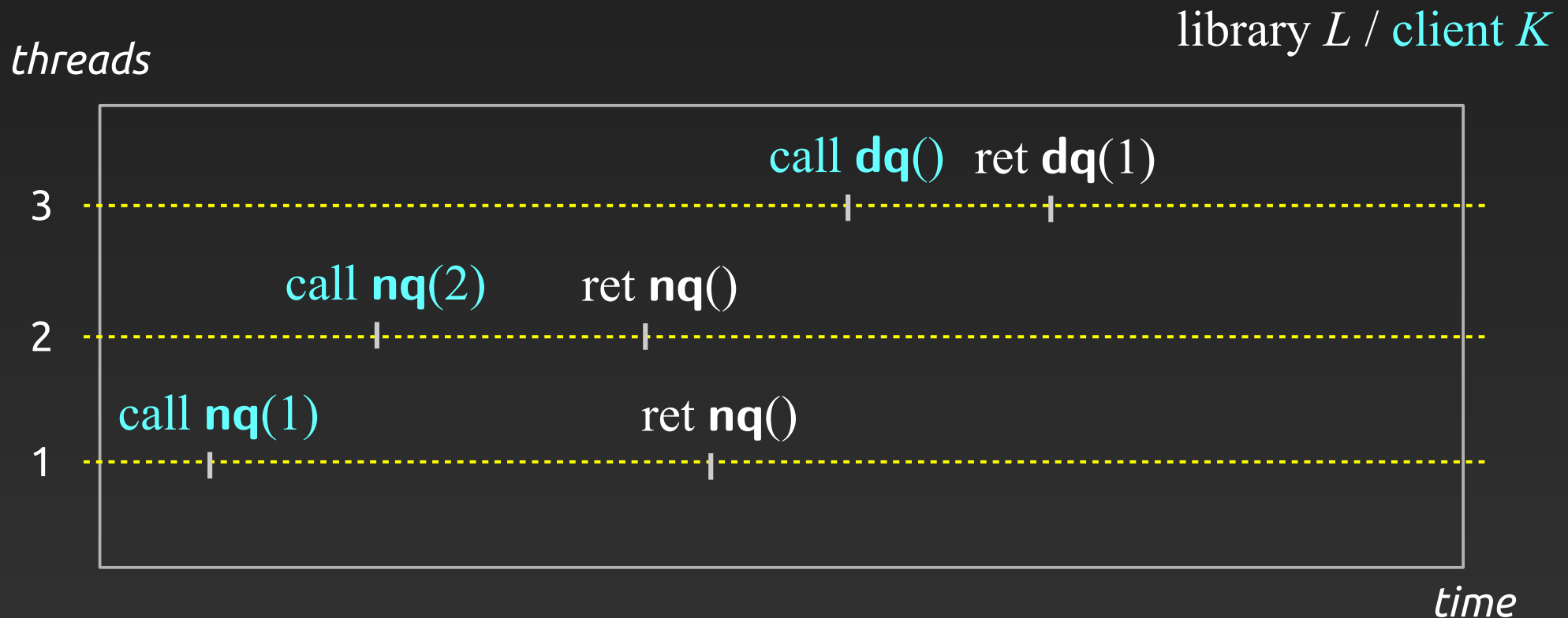
Concurrent queue behaviour

Concurrent *queue* library L , with methods:

enqueue: $\text{int} \rightarrow \text{void}$

dequeue: $\text{void} \rightarrow \text{int}$

A (correct) library behaviour:



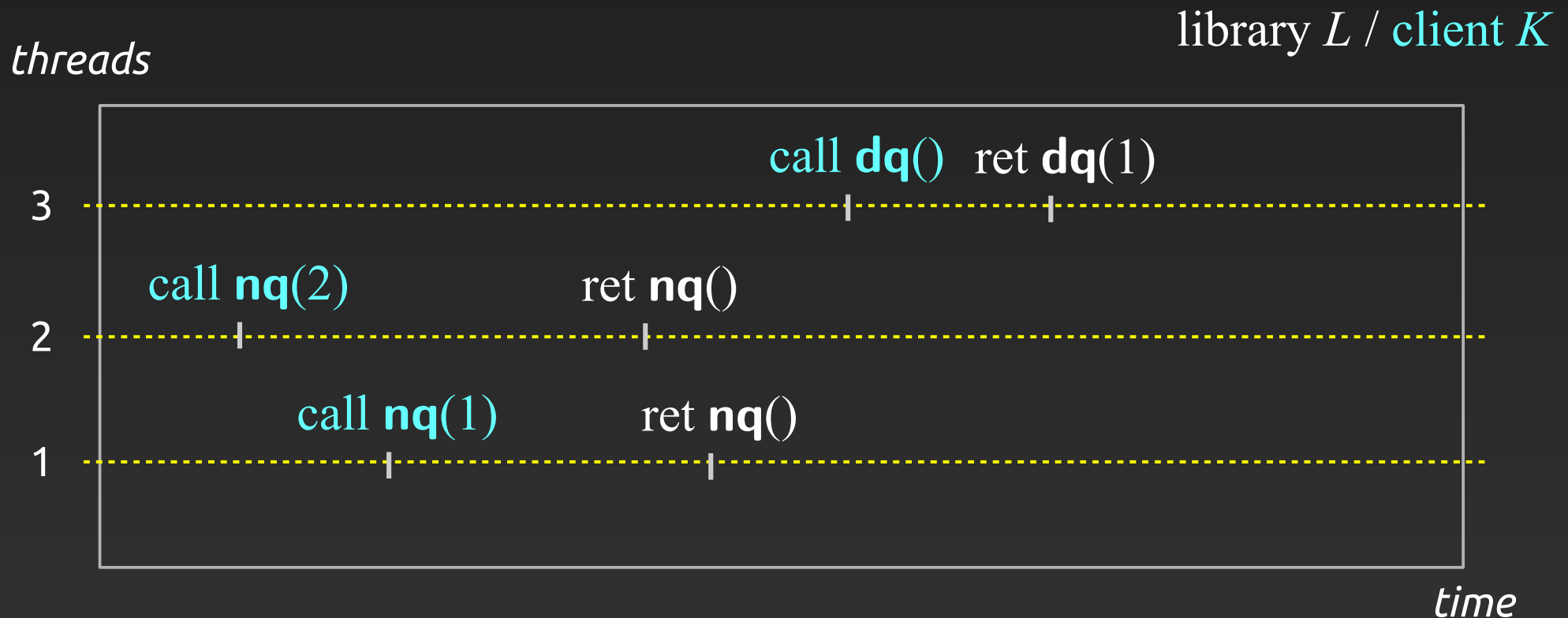
Concurrent queue behaviour

Concurrent *queue* library L , with methods:

enqueue: $\text{int} \rightarrow \text{void}$

dequeue: $\text{void} \rightarrow \text{int}$

A (correct) library behaviour:



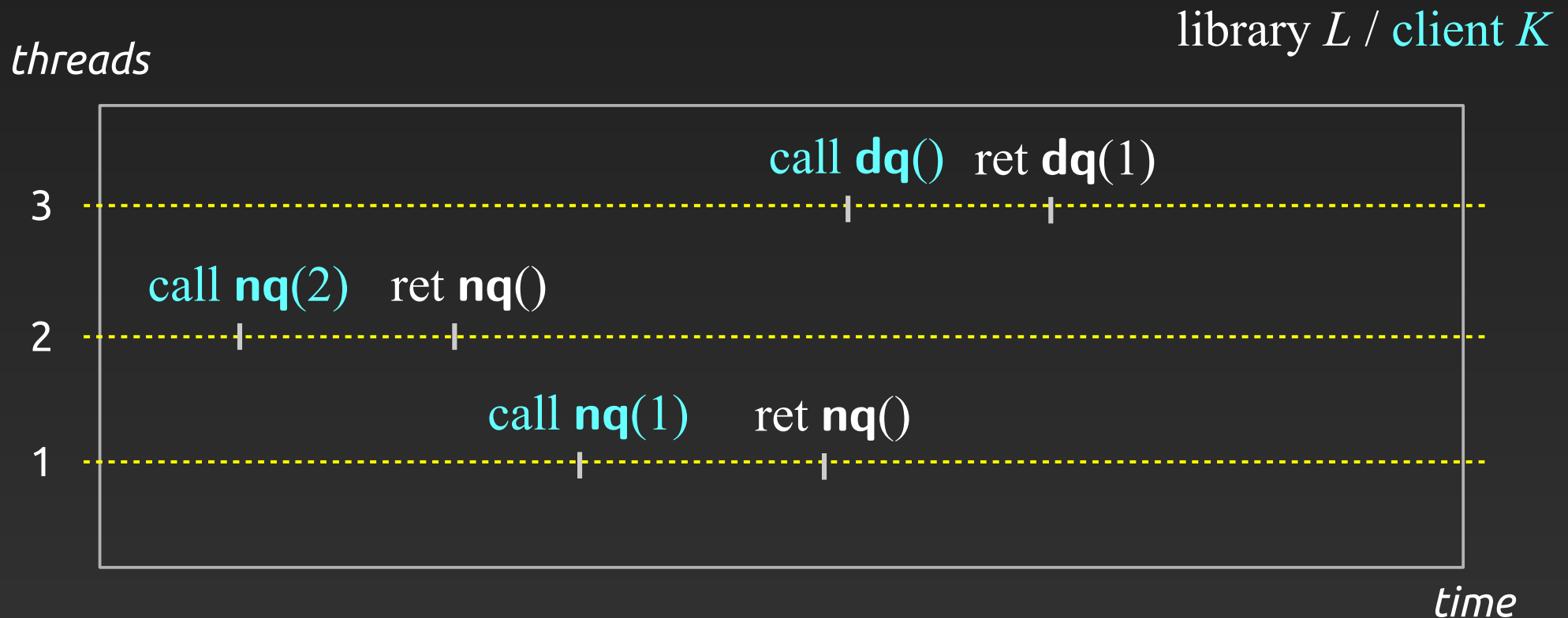
Concurrent queue behaviour

Concurrent *queue* library L , with methods:

enqueue: $\text{int} \rightarrow \text{void}$

dequeue: $\text{void} \rightarrow \text{int}$

A (correct) library behaviour:



Correctness formally

How can we characterise “correct” behaviours?

→ use **linearisability**:

*a behaviour is correct if it follows a given specification,
modulo legal reordering of actions:*

$$\begin{aligned}(t, \text{call}) (t', x) &\rightarrow (t', x) (t, \text{call}) \\ (t', x) (t, \text{ret}) &\rightarrow (t, \text{ret}) (t', x)\end{aligned}$$

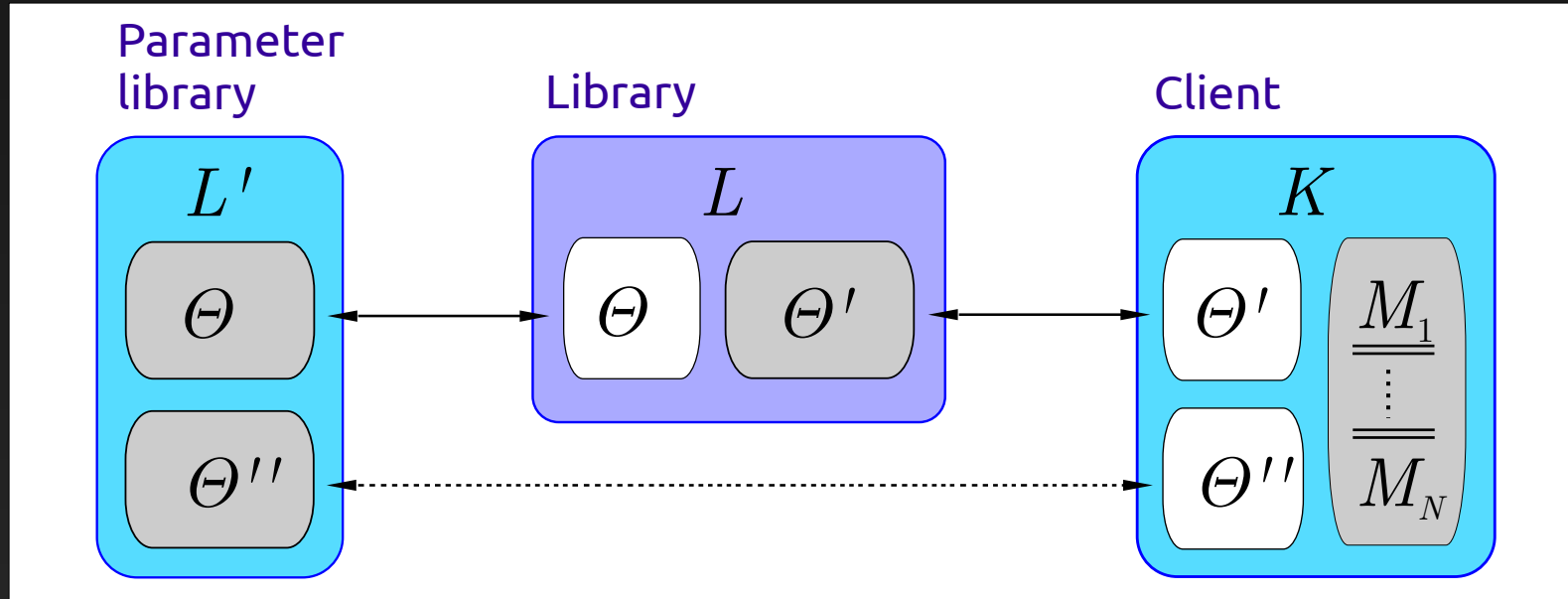
→ introduced in the 90's, by now standard

[Herlihy & Wing '90]

→ **soundness**: L linearises into $L' \implies L$ obs. approximates L'

[Filipovic, O'Hearn, Rinetzky, Yang '10]

Higher-order libraries



→ Library L is open (depends on abstract methods Θ)

→ Methods are higher order (e.g. $\text{int} \rightarrow \text{int} \rightarrow \text{int}$)

write $L : \Theta \rightarrow \Theta'$

Previously only 1st order methods examined

Higher-order difficulties

The HO situation is much richer as traces are higher-order:

$(1, \text{call } m_1(\widetilde{m}_1, \widetilde{m}_2)) (2, \text{call } m_2(\dots)) (2, \text{call } \widetilde{m}_1(\dots)) (2, \text{ret } \widetilde{m}_1(\widetilde{m}_3)) \dots$

- a method call need not be followed by its return
- method calls/returns can be issued by *everyone* (the client, the library, the parameter library)
- new methods can appear on-the-fly
- sequential histories?

The setting is great for game semantics!

Higher-order linearisability

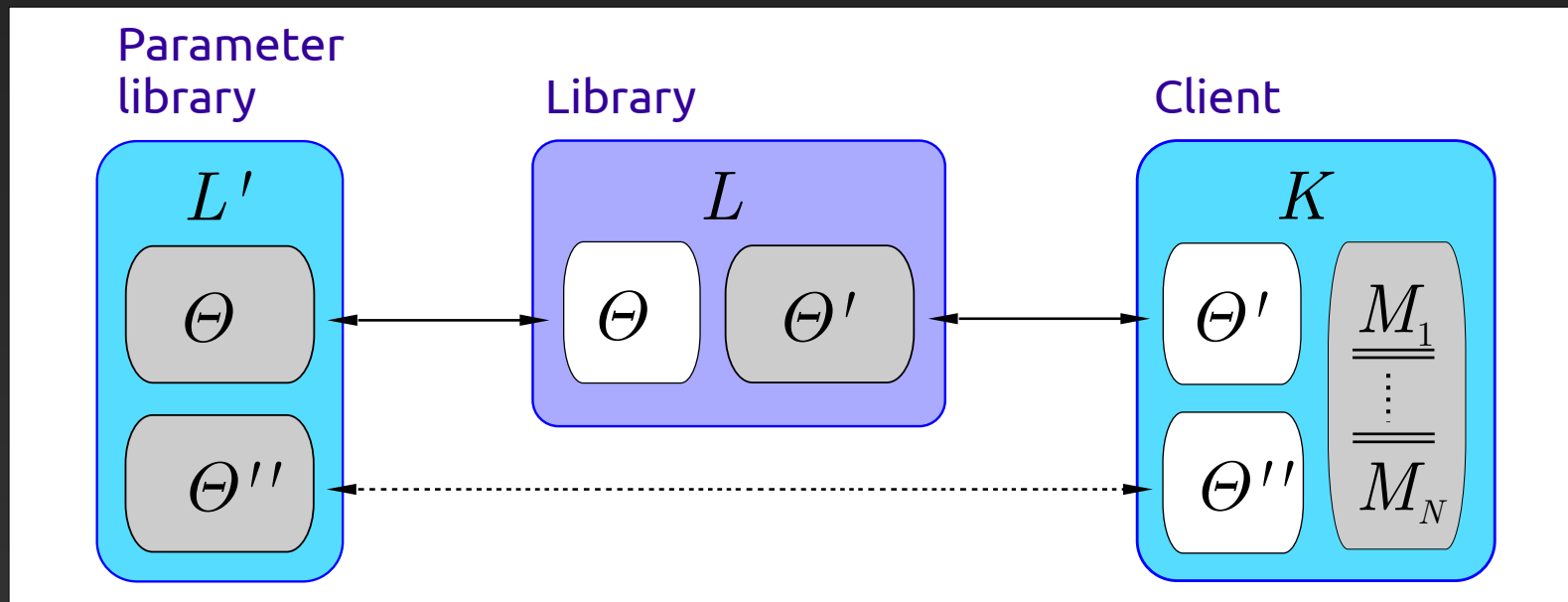
Legal reorderings now defined by:

$$(t, a_O) (t', a') \triangleleft_{PO} (t', a') (t, a_O)$$

$$(t', a') (t, a_P) \triangleleft_{PO} (t, a_P) (t', a')$$

P (Proponent: L)

O (Opponent: K & L')



Higher-order linearisability

Legal reorderings now defined by:

$$\begin{aligned}(t, a_O) (t', a') &\triangleleft_{PO} (t', a') (t, a_O) \\(t', a') (t, a_P) &\triangleleft_{PO} (t, a_P) (t', a')\end{aligned}$$

Def: History h_1 **linearises** to h_2 if we can get h_2 from h_1 by a series of \triangleleft_{PO} -reorderings.

A library L linearises to some **specification** A of sequential histories if every history h_1 of L linearises into some h_2 in A .

A history h is **sequential** if it is of the form:

$$h = (t_1, a_{1(O)}) (t_1, a_{2(P)}) (t_2, a_{3(O)}) (t_2, a_{4(P)}) \dots$$

Soundness

L_1 linearises into $L_2 \implies L_1$ obs. approximates L_2

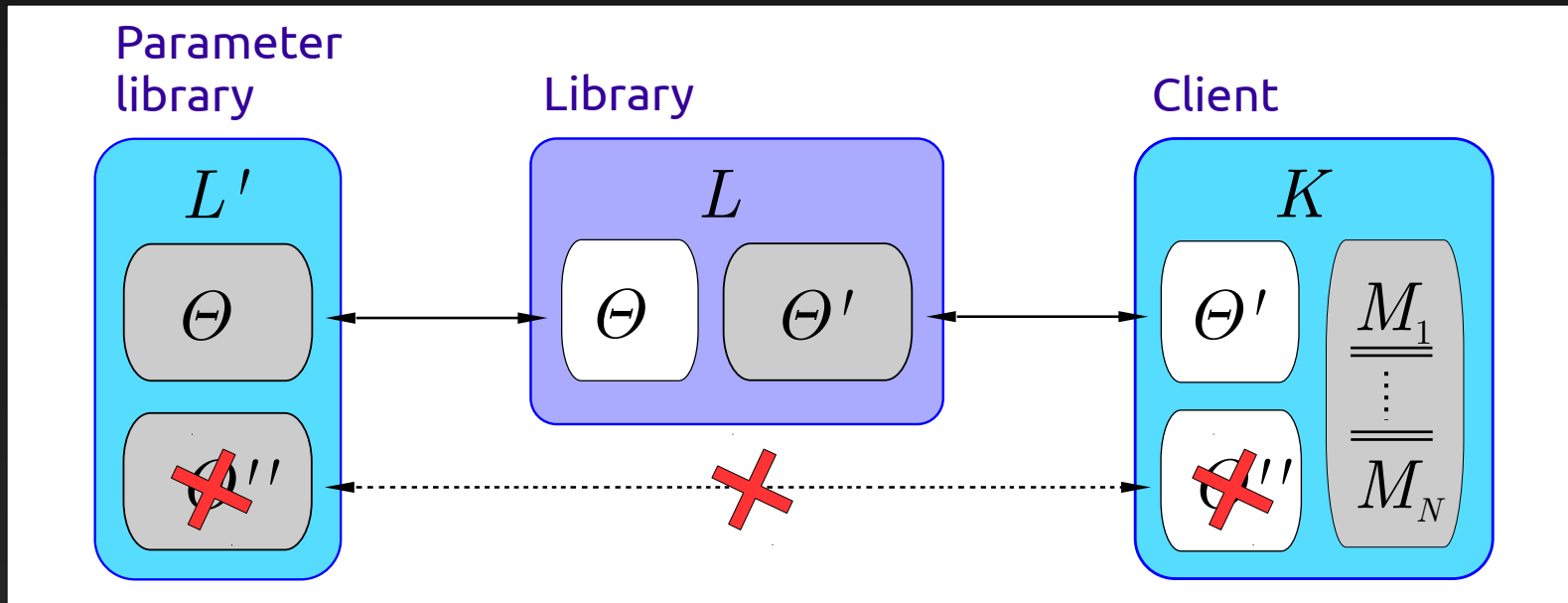
i.e. for all (L', K) :

(link $L'; L_1$ in K) terminates \implies (link $L'; L_2$ in K) terminates

Moreover, linearisability is compositional (wrt code composition)

Encapsulation

[Cerone, Gotsman, Yang '14]



Environment more restricted

\implies more reorderings allowed:

$$\begin{aligned}
 (t, a)_{\mathcal{X}} (t', a)_{\mathcal{L}} &\diamond (t', a)_{\mathcal{L}} (t, a)_{\mathcal{X}} \\
 (t, a)_{\mathcal{L}} (t', a)_{\mathcal{X}} &\diamond (t', a)_{\mathcal{X}} (t, a)_{\mathcal{L}}
 \end{aligned}$$

Flat combining

```
1 public run; ...;
2 Lock lock;
3 struct {fun, arg, wait, retv} requests [N];
4
5 run = λ (f,x).
6   requests [tid].fun := f;
7   requests [tid].arg := x;
8   requests [tid].wait := 1;
9   while ( requests [tid].wait)
10     if ( lock . tryacquire () ) {
11       for ( t=0; t<N; t++)
12         if ( requests [ t ]. wait ) {
13           requests [ t ]. retv :=
14             requests [ t ]. fun ( requests [ t ]. arg );
15           requests [ t ]. wait := 0;
16         }; lock . release () };
17   requests [tid].retv;
```

$run : ((int \rightarrow int) * int) \rightarrow int$

Summary

Introduce extension of linearisability for higher-order libraries

Prove it is sound and compositional

Encapsulation, relational linearisability

Examples

[paper at CONCUR'17]

Further on:

- weaker notions, allowing for context-specific linearisations (cf. encapsulation)
- completeness?
- proof methods for linearisability


```

1  public count, update;
2  Lock lock;
3  F :=  $\lambda x.0$ ;
4
5  count =  $\lambda i. (!F)i$ 
6  update =  $\lambda(i, g). \text{aux}(i, g, \text{count } i)$ 
7
8  aux =  $\lambda(i, g, j).$ 
9      let y = |g j| in
10     lock.acquire ();
11     let f = !F in
12         if (j == (f i)) then {
13             F :=  $\lambda x. \text{if } (x == i) \text{ then } y$ 
14                 else (f x) ;
15             lock.release ();
16             y }
17     else {
18         lock.release ();
19         aux(i, g, f i) }

```

Multiset library

count: $\text{int} \rightarrow \text{int}$

update:
 $(\text{int} * (\text{int} \rightarrow \text{int})) \rightarrow \text{int}$