Higher-Order Linearisability

Andrzej Murawski
University of Warwick

Nikos Tzevelekos
Queen Mary U. of London

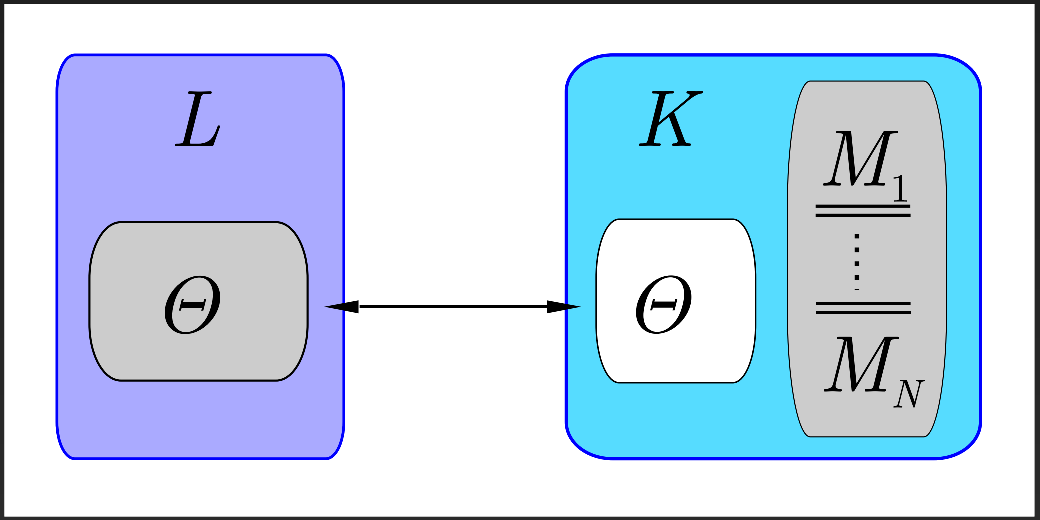
[paper appeared at CONCUR'17]

Highlights of Logic, Games and Automata, September 2017

Concurrent libraries

Concurrent *queue* library L with methods:

enqueue: int $\rightarrow$ void **dequeue**: void $\rightarrow$ int



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

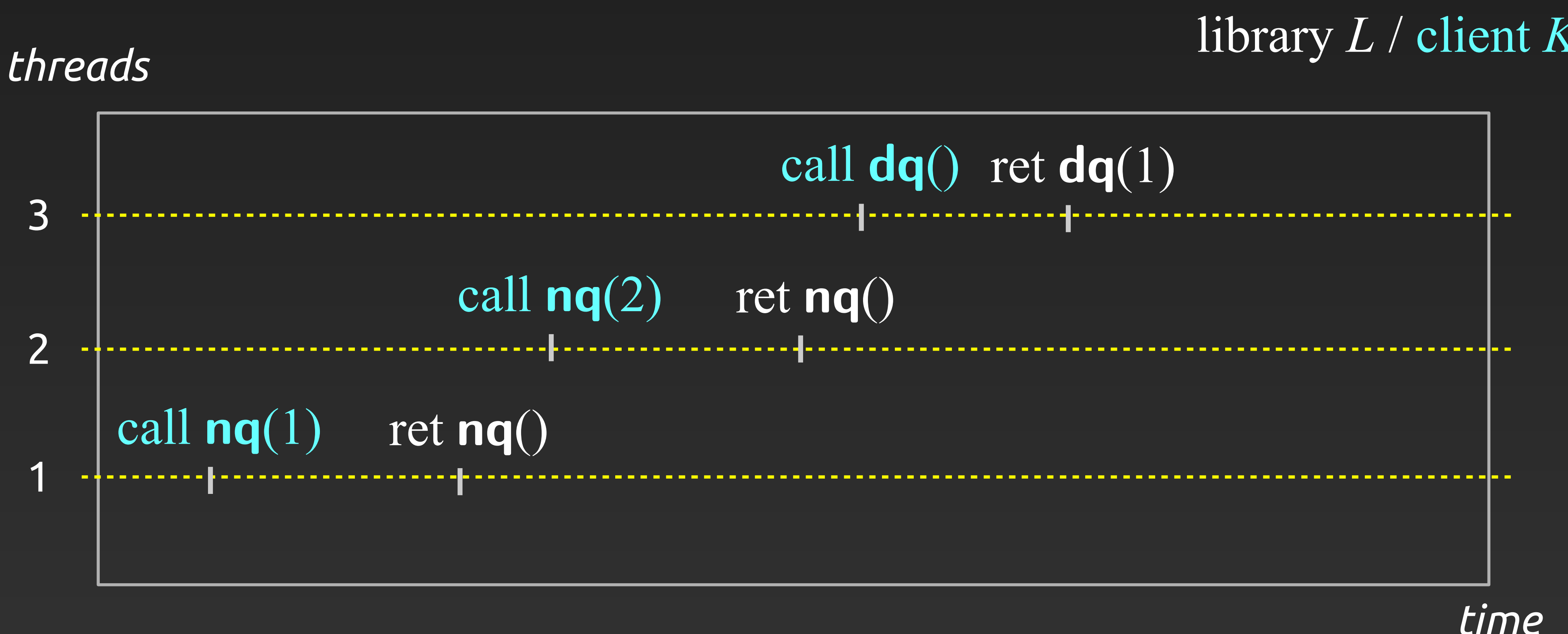
$\Theta = \{$ **nq**: int $\rightarrow$ void, **dq**: void $\rightarrow$ int $\}$

Concurrent queue behaviour

Concurrent *queue* library L with methods:

enqueue: int $\rightarrow$ void **dequeue**: void $\rightarrow$ int

A (correct) library behaviour:

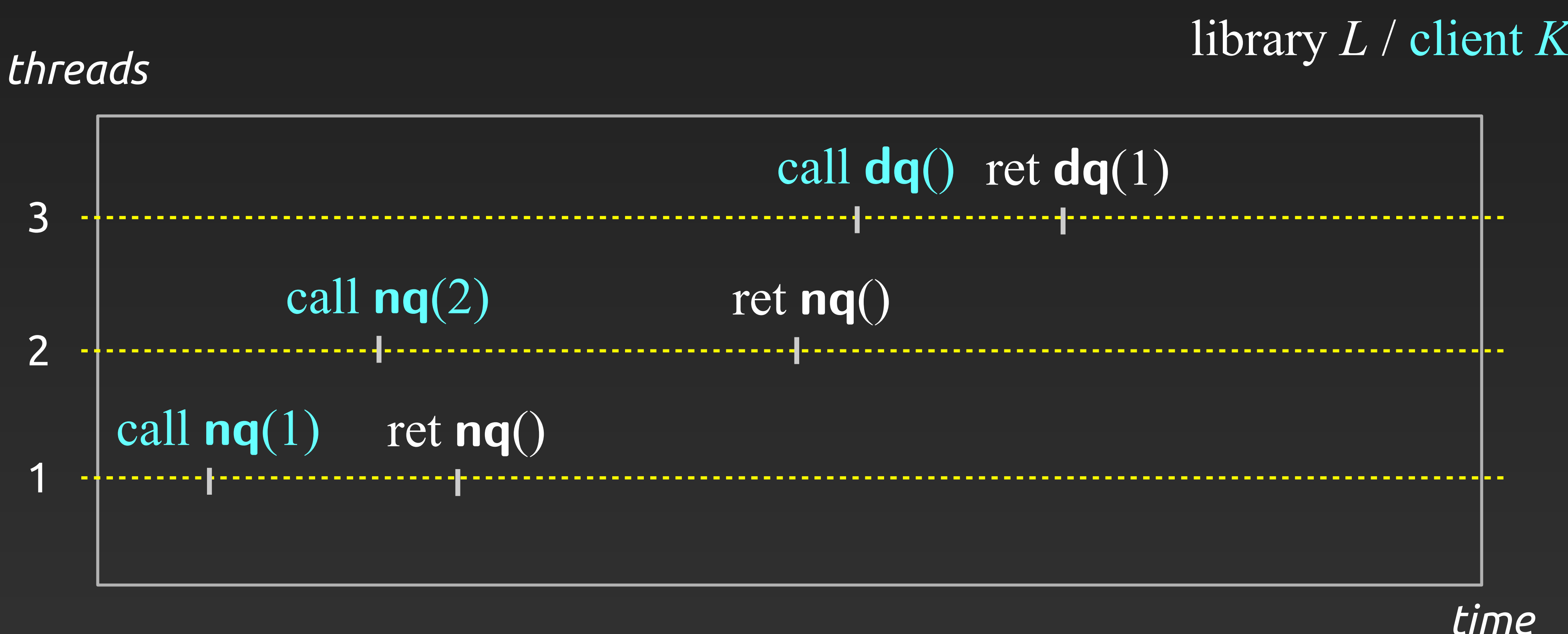


Concurrent queue behaviour

Concurrent *queue* library L with methods:

enqueue: int $\rightarrow$ void   **dequeue**: void $\rightarrow$ int

A (correct) library behaviour:

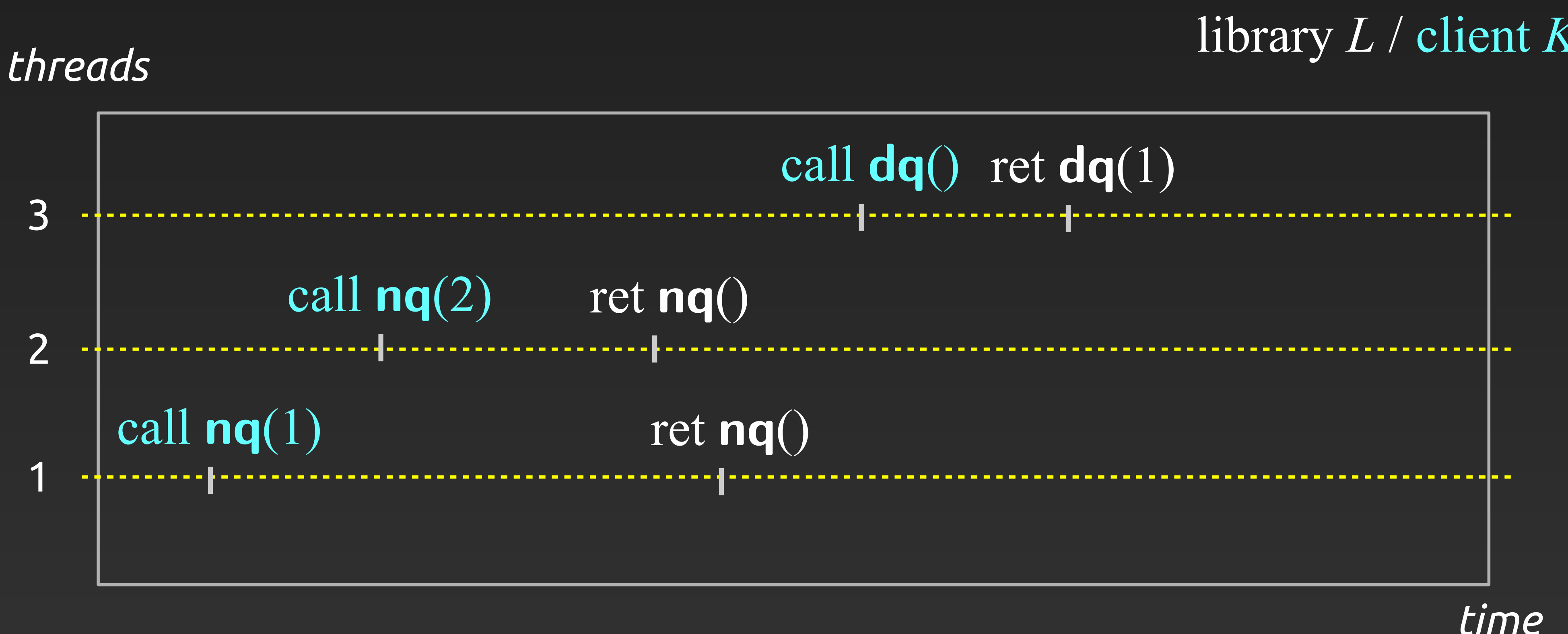


Concurrent queue behaviour

Concurrent *queue* library L with methods:

enqueue: int $\to$ void   **dequeue**: void $\to$ int

A (correct) library behaviour:

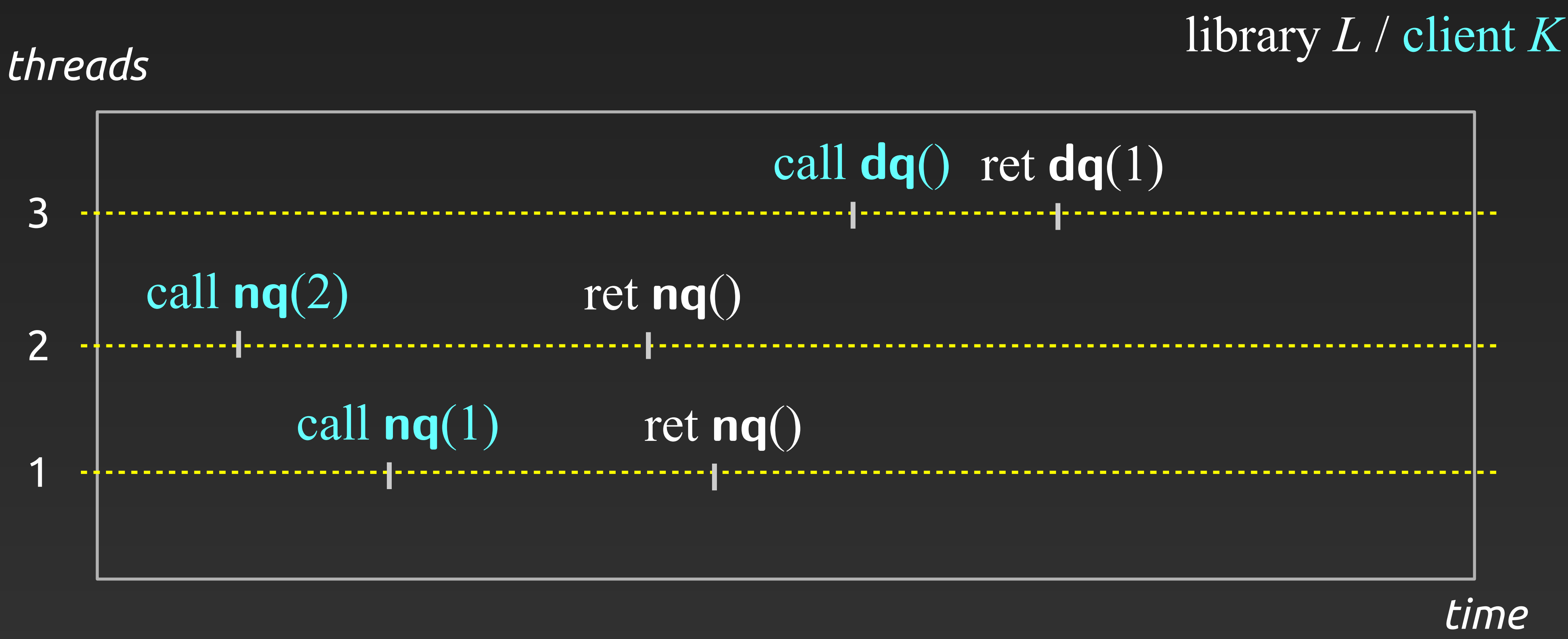


Concurrent queue behaviour

Concurrent *queue* library L with methods:

enqueue: int $\to$ void　　　　**dequeue**: void $\to$ int

A (correct) library behaviour:

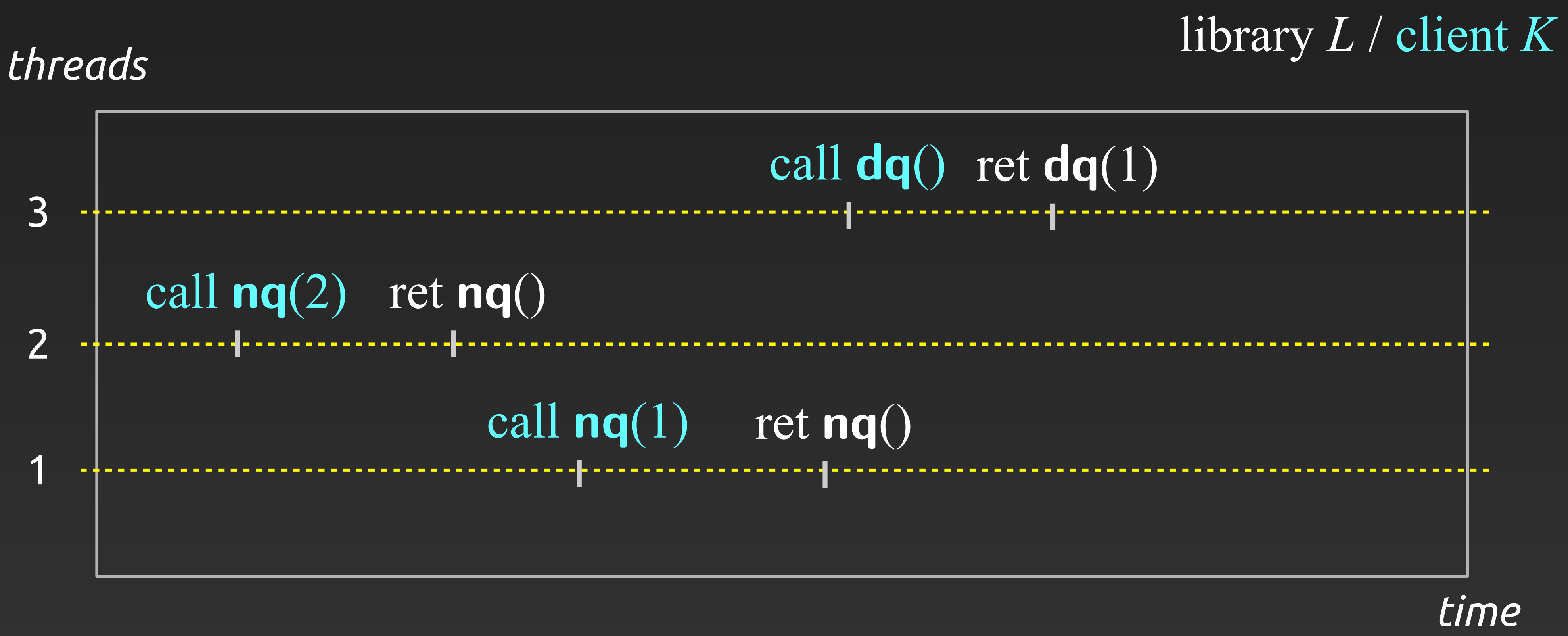


Concurrent queue behaviour

Concurrent *queue* library L with methods:

enqueue: int $\rightarrow$ void **dequeue**: void $\rightarrow$ 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:

$$
\boxed{
\begin{array}{ccc}
(t, \mathrm{call})\,(t', x) & \longrightarrow & (t', x)\,(t, \mathrm{call}) \\
(t', x)\,(t, \mathrm{ret}) & \longrightarrow & (t, \mathrm{ret})\,(t', x)
\end{array}
}
$$

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:

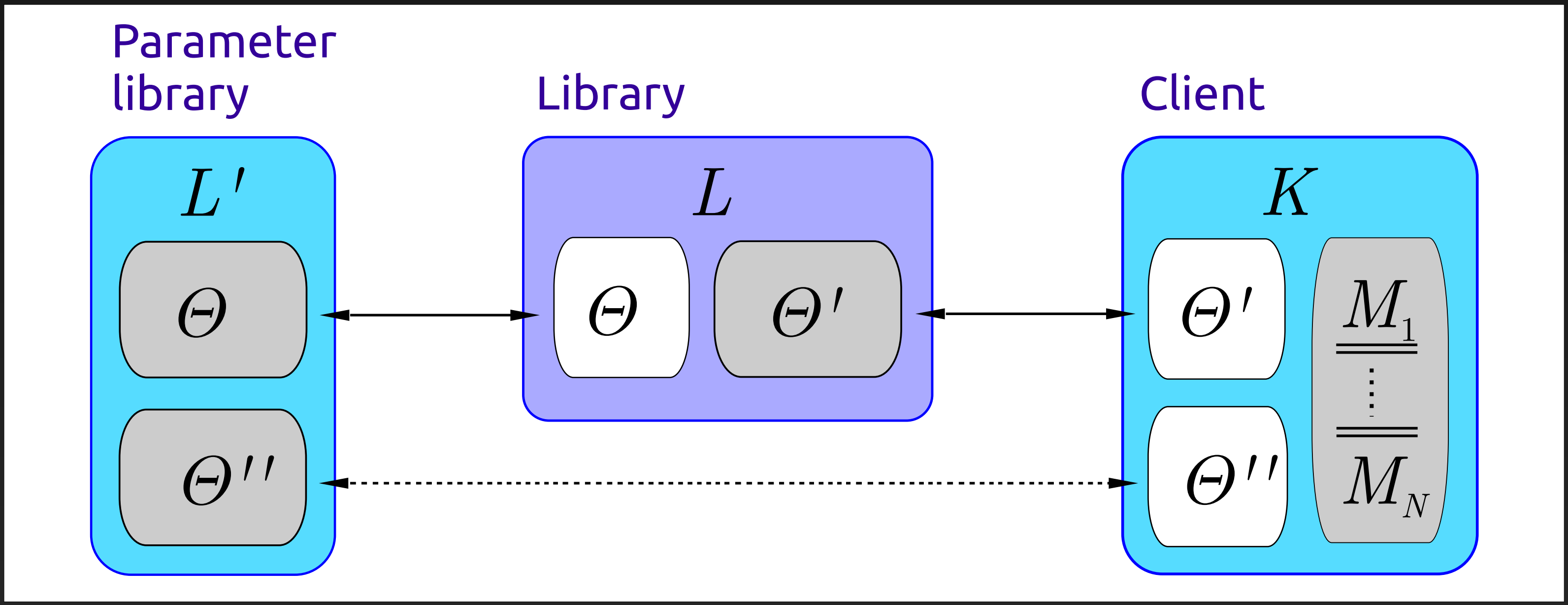
$$\boxed{\begin{array}{ccc} (t,\ \mathrm{call})\ (t',\ x) & \longrightarrow & (t',\ x)\ (t,\ \mathrm{call}) \\ (t',\ x)\ (t,\ \mathrm{ret}) & \longrightarrow & (t,\ \mathrm{ret})\ (t',\ x) \end{array}}$$

→ 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. int → int → int)

write $L : \Theta \to \Theta'$

Previously only 1st order methods examined

[Jagadeesan, Petri, Pitcher, Riely '13] [Cerone, Gotsman & Yang '14]

```
public count, update;
Lock lock;
F := λx.0;

count = λi. (!F)i
update = λ(i, g). aux(i, g, count i)

aux = λ(i, g, j).
  let y = |g j| in
    lock.acquire();
    let f = !F in
      if (j == (f i)) then {
        F := λx. if (x == i) then y
                 else (f x) ;
        lock.release();
        y }
      else {
        lock.release();
        aux(i, g, f i) }
```

E.g. multiset library:

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

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

Higher-order difficulties

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

$$(1,\text{call } \mathbf{m}_1(\widetilde{\mathbf{m}}_1,\widetilde{\mathbf{m}}_2))\ (2,\text{call } \mathbf{m}_2(\ldots))\ (2,\text{call } \widetilde{\mathbf{m}}_1(\ldots))\ (2,\text{ret } \widetilde{\mathbf{m}}_1(\widetilde{\mathbf{m}}_3))\ \ldots$$

- 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!

What game semantics

Computation modelled as a 2-player game between

- *Opponent* (the environment), aka O
- *Proponent* (the program), aka P

Moves are method calls and returns

Programs = *strategies* for P

$$\text{int m (f: int} \rightarrow \text{int, x:int) = return f(x)+1}$$

$$\mapsto \quad \text{call m(f, 5)}_O \ \text{call f(5)}_P \ \text{ret f(41)}_O \ \text{ret m(42)}_P$$

(formulated operationally as “HO open trace semantics”)

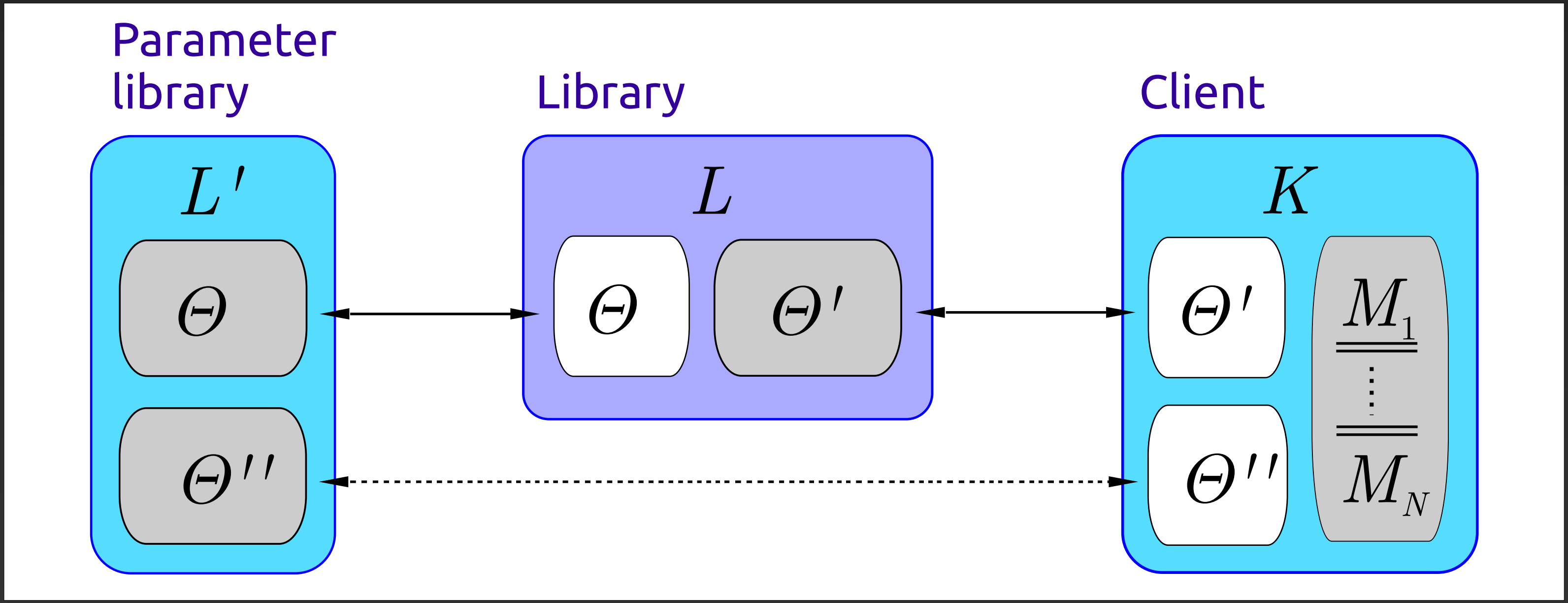
Higher-order linearisability

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:

$$\boxed{\begin{array}{c} (t, a_O)\,(t', a') \ \lhd_{PO}\ (t', a')\,(t, a_O) \\ (t', a')\,(t, a_P) \ \lhd_{PO}\ (t, a_P)\,(t', a') \end{array}}$$

Def: History h_1 linearises to h_2 if we can get h_2 from h_1 by a series of $\lhd_{PO}$-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:

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

Results

Soundness

$$L_1 \text{ linearises into } L_2 \implies L_1 \text{ obs. approximates } L_2$$

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

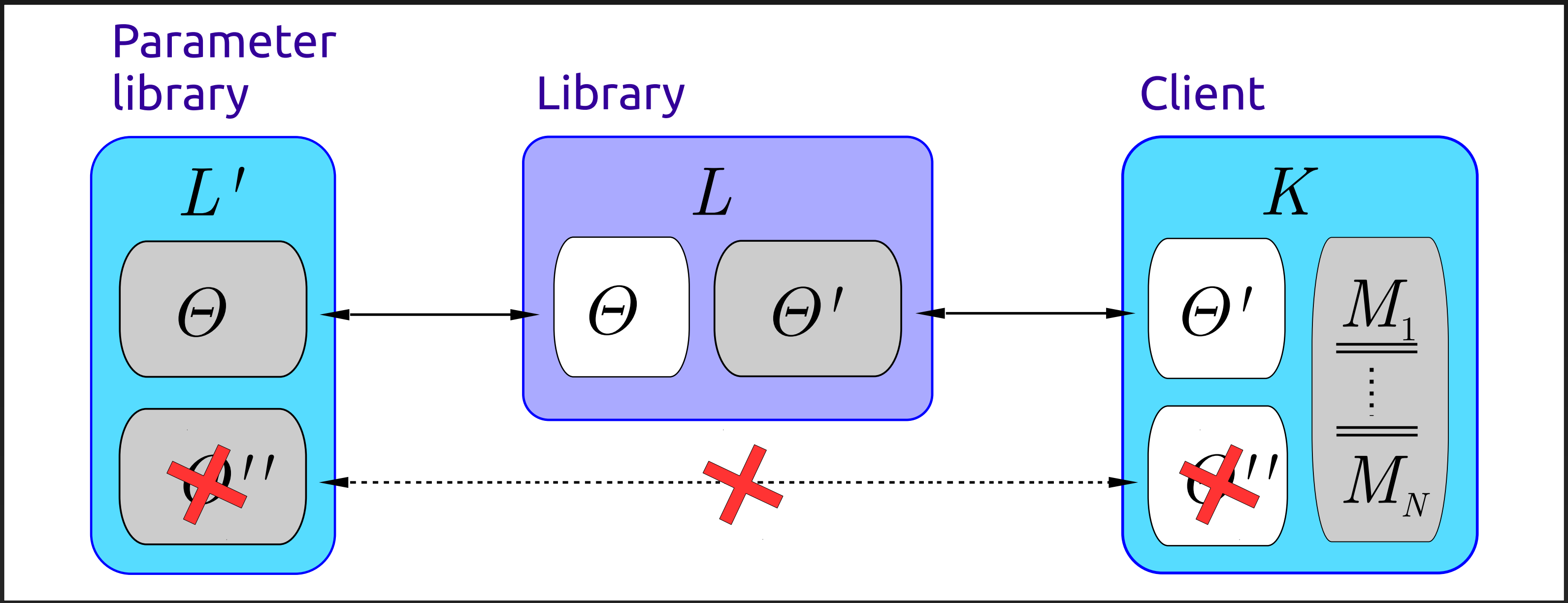
(link L'; L_1 in K) terminates $\implies$ (link L'; L_2 in K) terminates

Compositionality

$$L_1 \text{ linearises into } L_2 \implies L_1 + L \text{ linearises into } L_2 + L$$

Encapsulation

[Cerone, Gotsman, Yang '14]



Environment more restricted

$\implies$ more reorderings allowed:

$$(t, a)_{\mathcal{K}}\,(t', a)_{\mathcal{L}} \quad \diamond \quad (t', a)_{\mathcal{L}}\,(t, a)_{\mathcal{K}}$$

$$(t, a)_{\mathcal{L}}\,(t', a)_{\mathcal{K}} \quad \diamond \quad (t', a)_{\mathcal{K}}\,(t, a)_{\mathcal{L}}$$

Summary

Introduce extension of linearisability for higher-order libraries

Prove it is sound and compositional

Encapsulation, relational linearisability

Examples

Further on:

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

Flat combining (relational linearisability)

$run : ((\mathsf{int} \to \mathsf{int}) * \mathsf{int}) \to \mathsf{int}$

```
public run; ... ;
Lock lock;
struct {fun, arg, wait, retv} requests[N];

run = λ(f,x).
   requests[t_id].fun := f;
   requests[t_id].arg := x;
   requests[t_id].wait := 1;
  while (requests[t_id].wait)
     if (lock.tryacquire()) {
        for (t=0; t<N; t++)
           if (requests[t].wait) {
               requests[t].retv :=
                      requests[t].fun (requests[t].arg);
               requests[t].wait := 0;
      }; lock.release() };
   requests[t_id].retv;
```