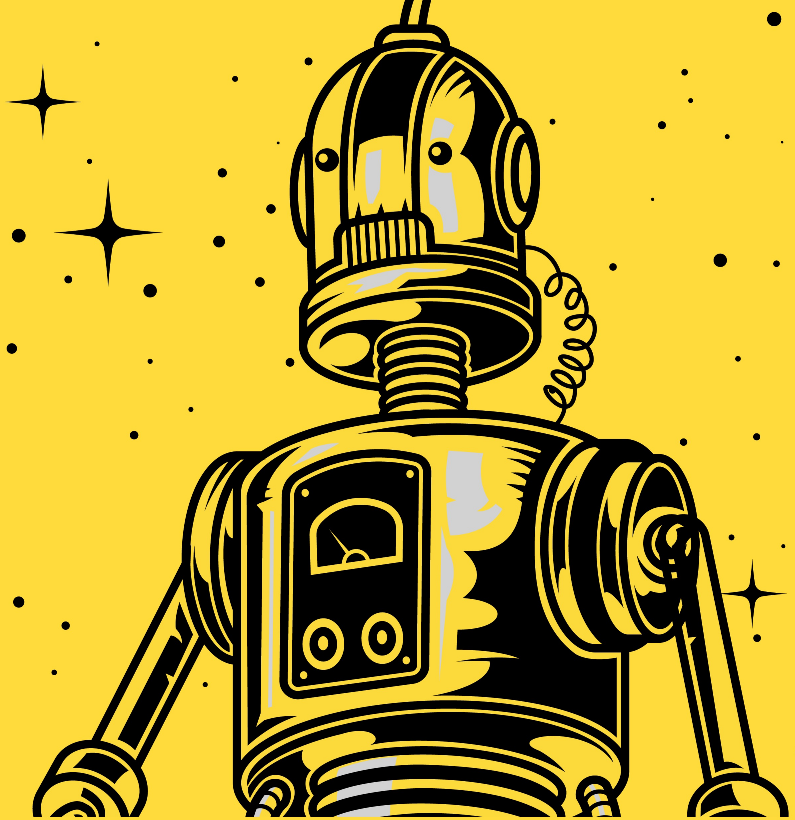
iTdays

#NewPerspectives



New Perspectives on Solving Concurrency

Lucian Radu Teodorescu

Staff Software Engineer @ Garmin

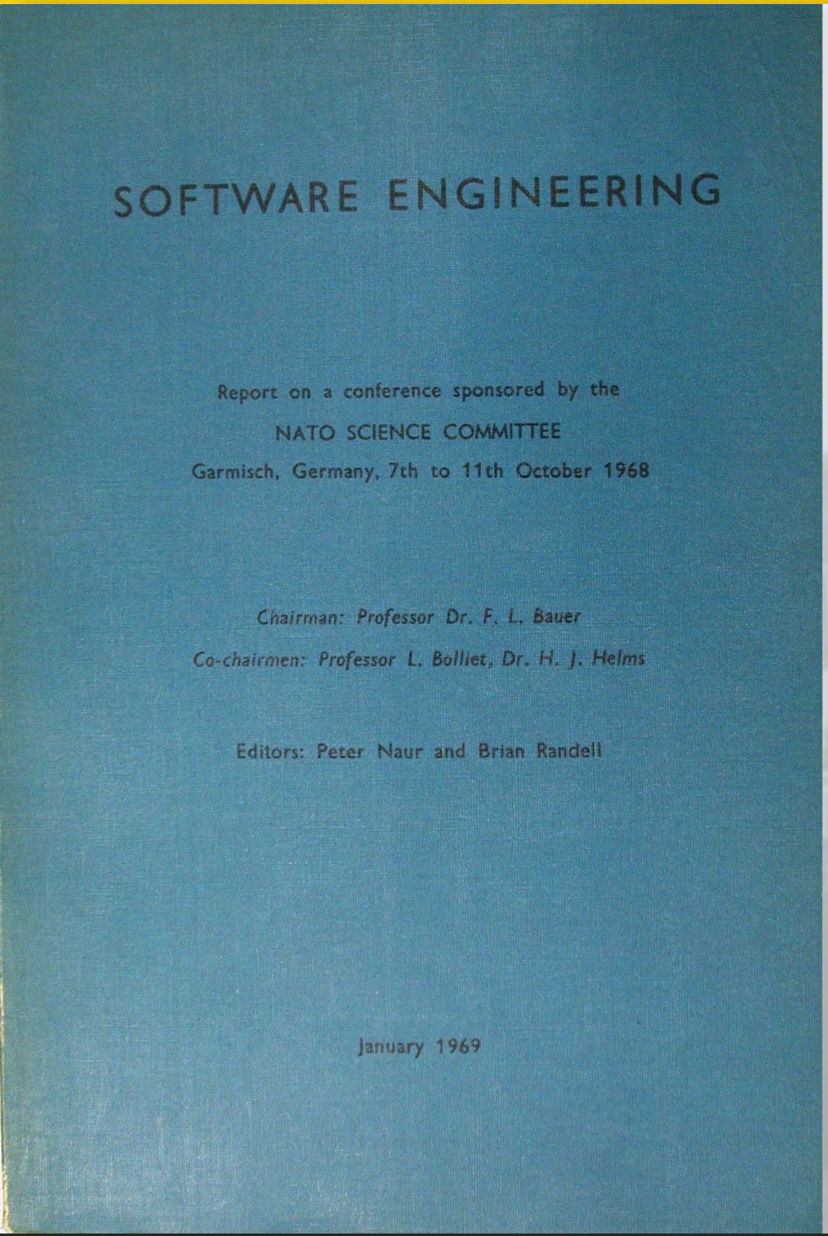
@LucT3o

New Perspectives on Solving Concurrency

lucteo.ro/pres/2021-itdays/

New Perspectives on Solving Concurrency

my perspectives


SOFTWARE ENGINEERING

Report on a conference sponsored by the
NATO SCIENCE COMMITTEE
Garmisch, Germany, 7th to 11th October 1968

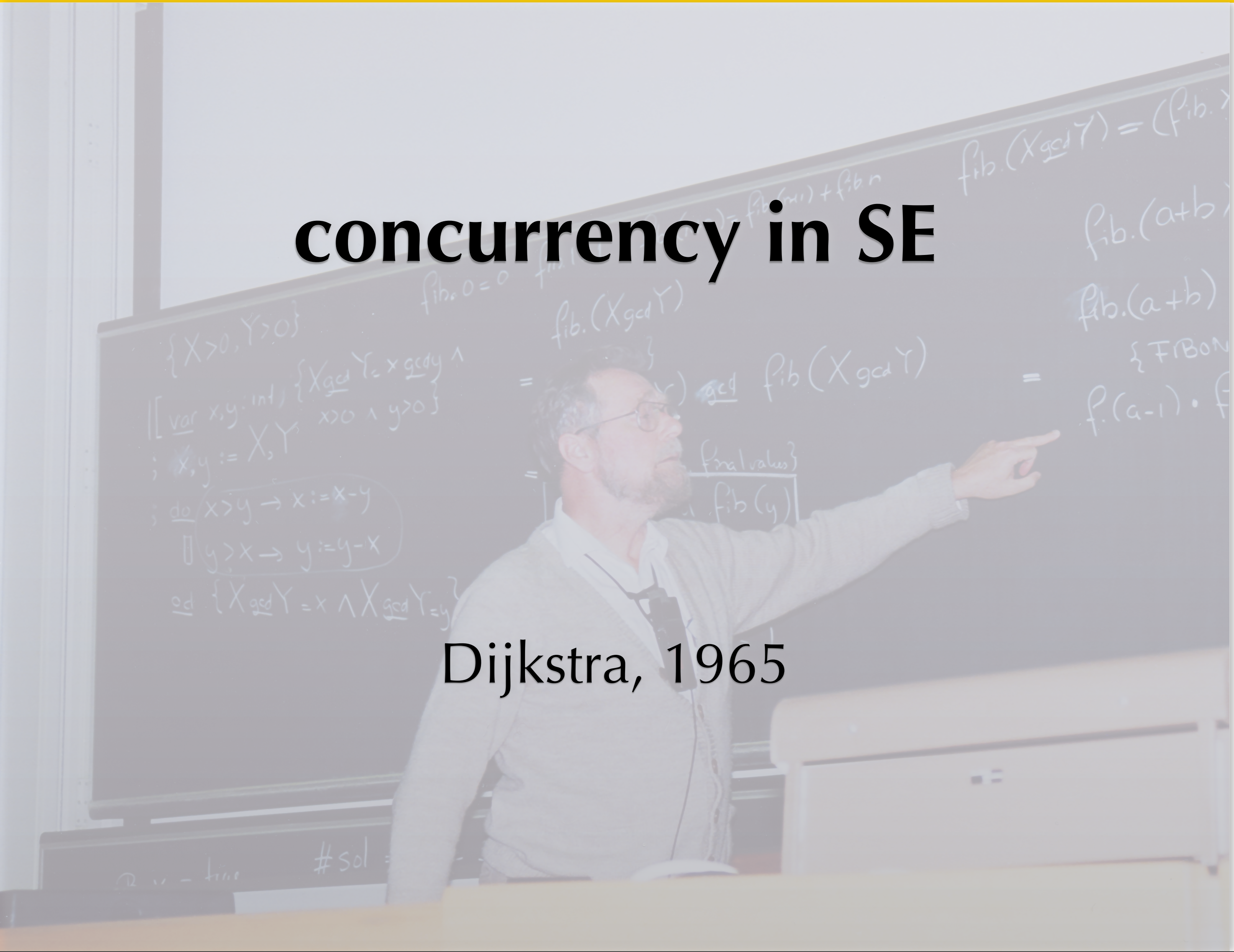
Chairman: Professor Dr. F. L. Bauer
Co-chairmen: Professor L. Bolliet, Dr. H. J. Helms

Editors: Peter Naur and Brian Randell

January 1969

software engineering

1968 NATO conference



concurrency in SE

Dijkstra, 1965

Solution of a Problem in Concurrent Programming Control

E. W. Dijkstra
Technological University, Eindhoven, The Netherlands

A number of mainly independent sequential-cyclic processes with restricted means of communication with each other can be made in such a way that at any moment one and only one of them is engaged in the "critical section" of its cycle.

Introduction

Given in this paper is a solution to a problem for which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. The paper consists of three parts: the problem, the solution, and the proof. Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved.

The Problem

To begin, consider N computers, each engaged in a process which, for our aims, can be regarded as cyclic. In each of the cycles a so-called "critical section" occurs and the computers have to be programmed in such a way that at any moment only one of these N cyclic processes is in its critical section. In order to effectuate this mutual exclusion of critical-section execution the computers can communicate with each other via a common store. Writing a word into or nondestructively reading a word from this store are undividable operations; i.e., when two or more computers try to communicate (either for reading or for writing) simultaneously with the same common location, these communications will take place one after the other, but in an unknown order.

The solution must satisfy the following requirements.

(a) The solution must be symmetrical between the N computers; as a result we are not allowed to introduce a static priority.

(b) Nothing may be assumed about the relative speeds of the N computers; we may not even assume their speeds to be constant in time.

(c) If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others.

(d) If more than one computer is about to enter its critical section, it must be impossible to devise for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until eternity. In other words, constructions in which "After you"-"After you"-blocking is still possible, although improbable, are not to be regarded as valid solutions.

We beg the challenged reader to stop here for a while and have a try himself, for this seems the only way to get a feeling for the tricky consequences of the fact that each computer can only request one one-way message at a time. And only this will make the reader realize to what extent this problem is far from trivial.

The Solution

The common store consists of:

"**Boolean array** b, $c[1{:}N]$; **integer** k"

The integer k will satisfy $1 \leq k \leq N$, $b[i]$ and $c[i]$ will only be set by the ith computer; they will be inspected by the others. It is assumed that all computers are started well outside their critical sections with all Boolean arrays mentioned set to **true**; the starting value of k is immaterial.

The program for the ith computer $(1 \leq i \leq N)$ is:

```
"integer j;
Li0:  b[i] := false;
Li1:  if k ≠ i then
Li2:  begin c[i] := true;
Li3:  if b[k] then k := i;
      go to Li1
      end
        else
Li4:  begin c[i] := false;
        for j := 1 step 1 until N do
          if j ≠ i and not c[j] then go to Li1
      end;
      critical section;
      c[i] := true;  b[i] := true;
      remainder of the cycle in which stopping is allowed;
      go to Li0"
```

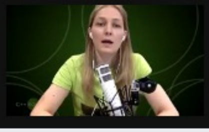
The Proof

We start by observing that the solution is safe in the sense that no two computers can be in their critical section simultaneously. For the only way to enter its critical section is the performance of the compound statement $Li4$ without jumping back to $Li1$, i.e., finding all other c's **true** after having set its own c to **false**.

The second part of the proof must show that no infinite "After you"-"After you"-blocking can occur; i.e., when none of the computers is in its critical section, of the computers looping (i.e., jumping back to $Li1$) at least one—and therefore exactly one—will be allowed to enter its critical section in due time.

If the kth computer is not among the looping ones, $b[k]$ will be **true** and the looping ones will all find $k \neq i$. As a result one or more of them will find in $Li3$ the Boolean $b[k]$ **true** and therefore one or more will decide to assign "$k := i$". After the first assignment "$k := i$", $b[k]$ becomes **false** and no new computers can decide again to assign a new value to k. When all decided assignments to k have been performed, k will point to one of the looping computers and will not change its value for the time being, i.e., until $b[k]$ becomes **true**, viz., until the kth computer has completed its critical section. As soon as the value of k does not change any more, the kth computer will wait (via the compound statement $Li4$) until all other c's are **true**, but this situation will certainly arise, if not already present, because all other looping ones are forced to set their c **true**, as they will find $k \neq i$. And this, the author believes, completes the proof.

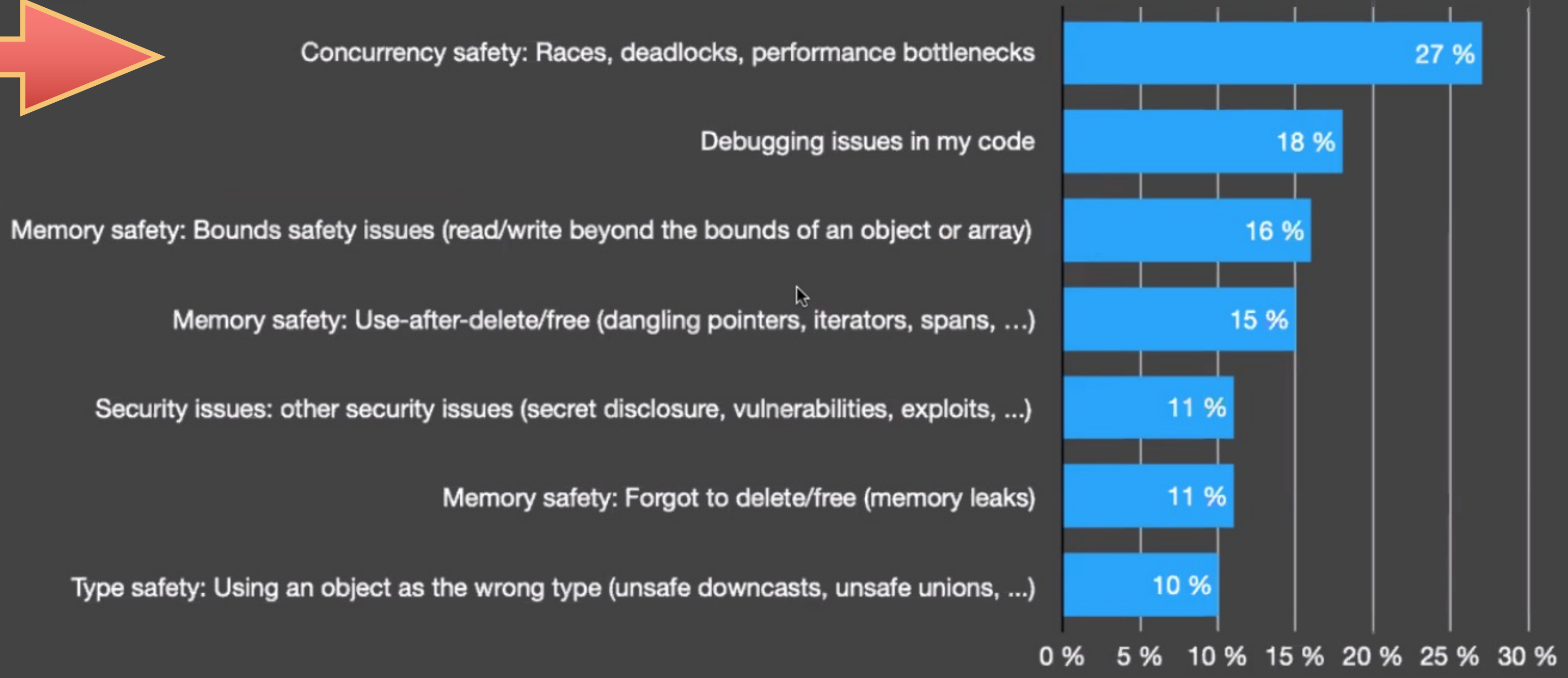
is the problem **solved**?

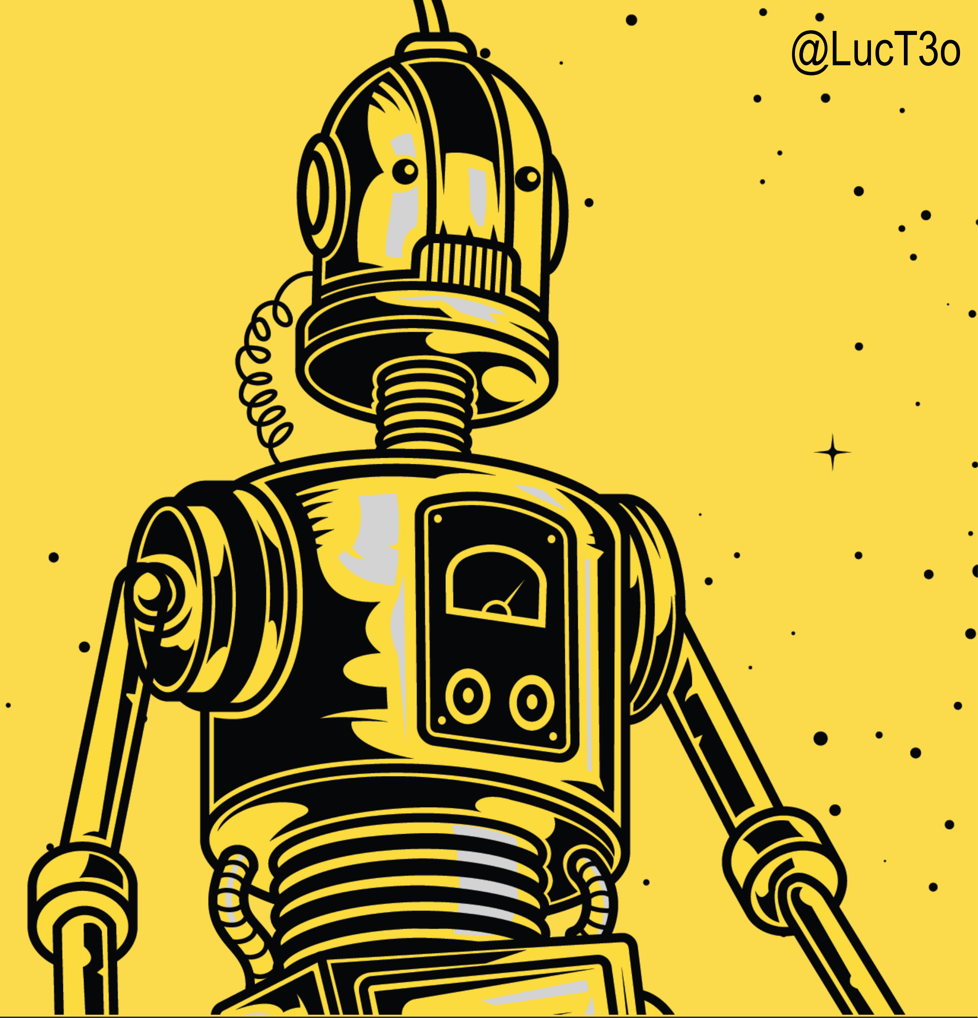


Code analysis++ EN - Google

docs.google.com/presentation/d/1rfdksmy28o0Oc4EA1MnP2dQE-vyf7G4MDbZKPj0AAmU/edit#slide=id.p

C++ development frustration: safety & security





Old perspective

building blocks

independent threads
synchronization primitives

roads analogy

thread → road
sync primitive → intersection

works well

for long roads
and few intersections

different reality

synchronization issues

deadlock
livelock
starvation
priority inversion
busy waiting

performance

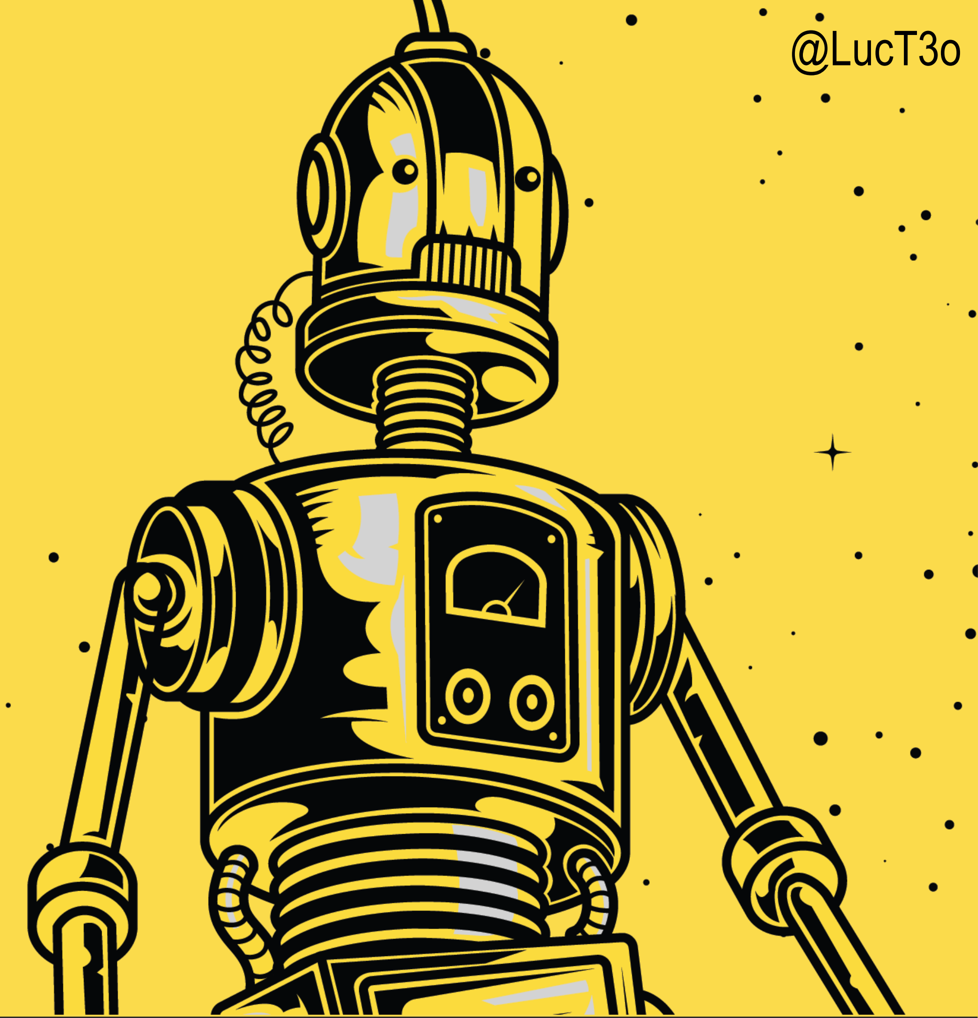
far from expected

threads

hard to think of them as independent

primitives are **not OK**

threads
synchronization



Concurrency with tasks

primitive

task = independent unit of work

primitive

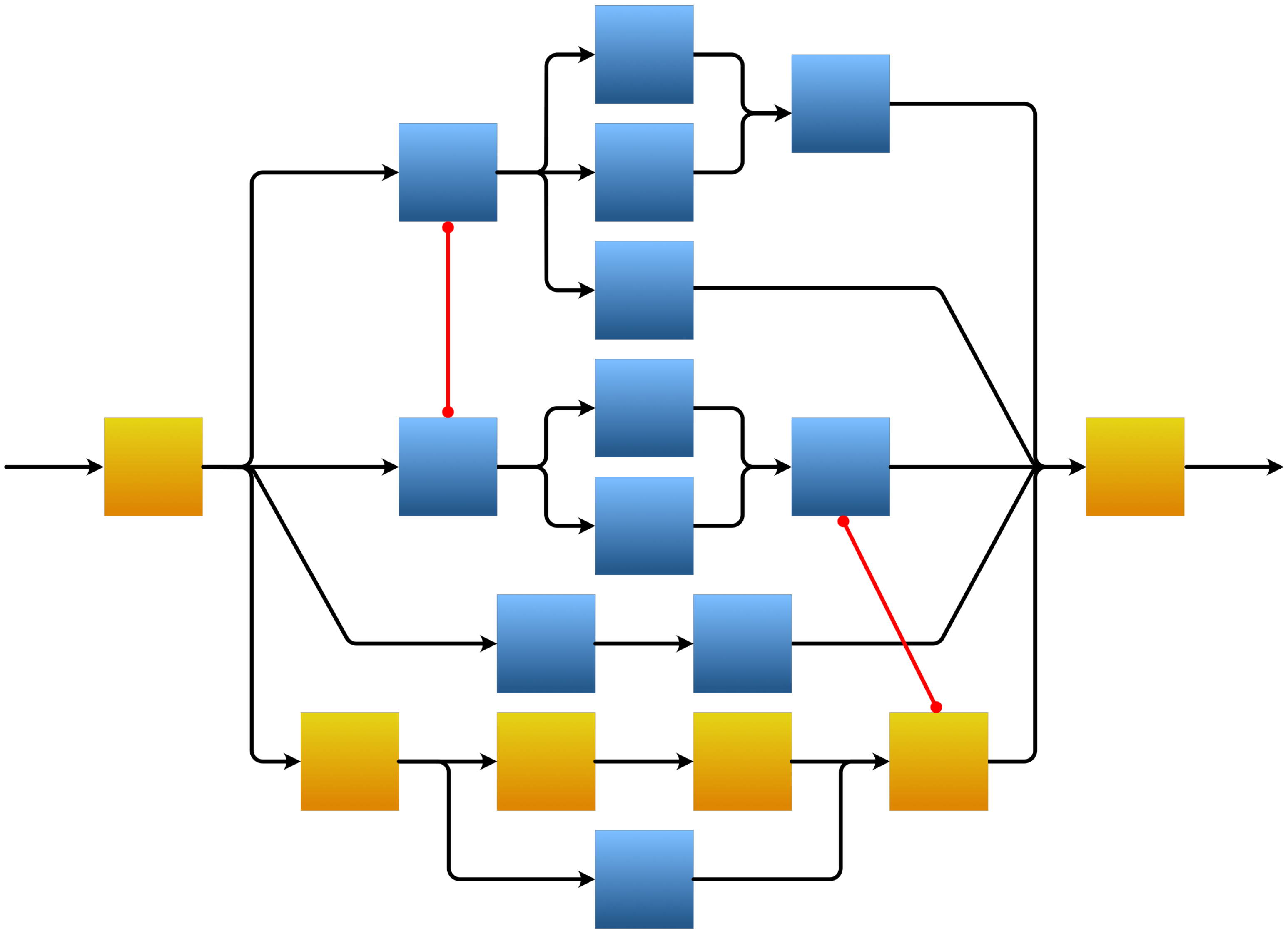
task = **<u>independent</u>** unit of work

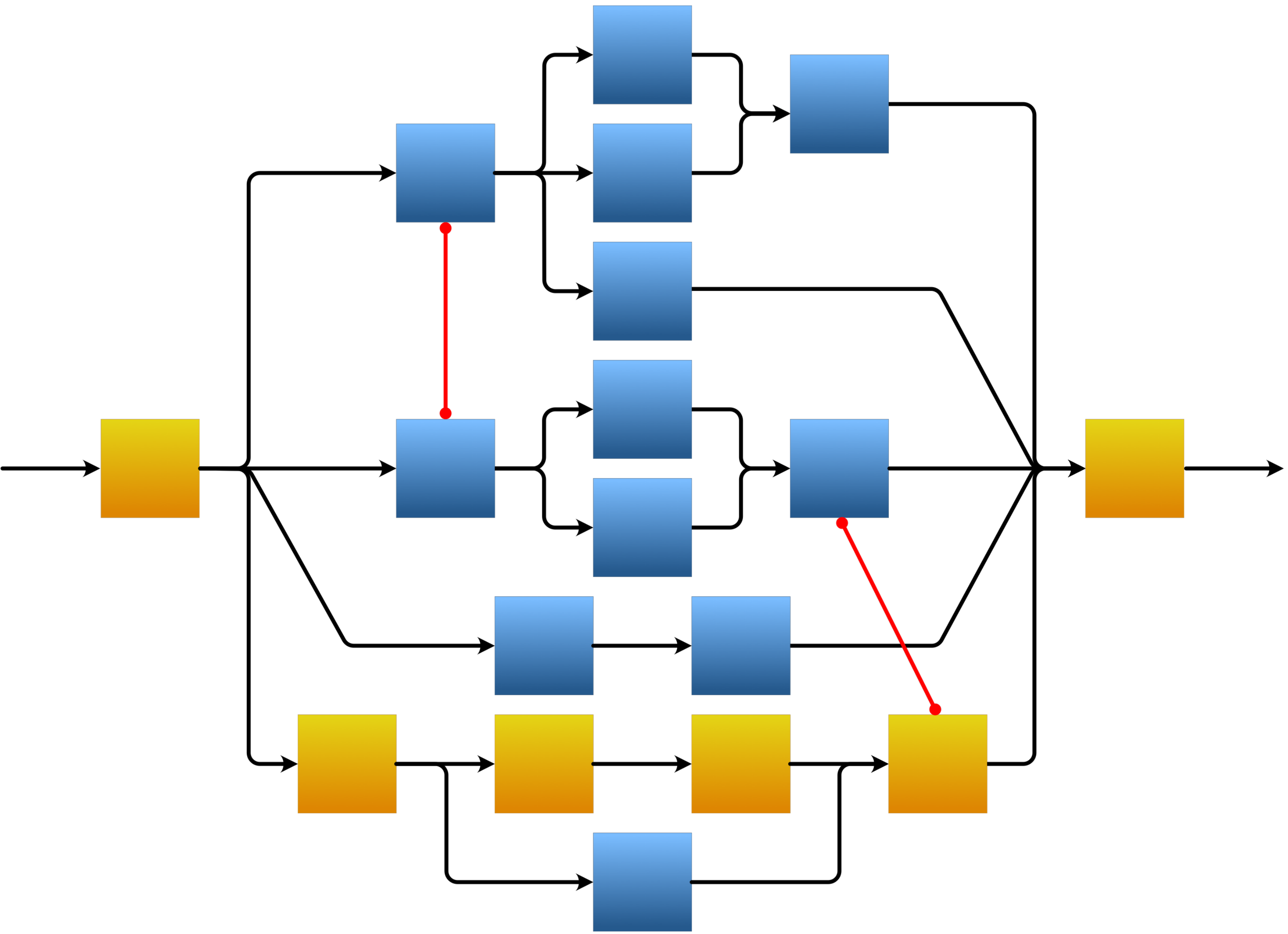
<u>independent</u> := does not depend on anything but its inputs

primitive

task = independent **unit of work**

unit := doesn't make sense to divide it





constraints
instead of locks

new problem

encoding concurrency with tasks

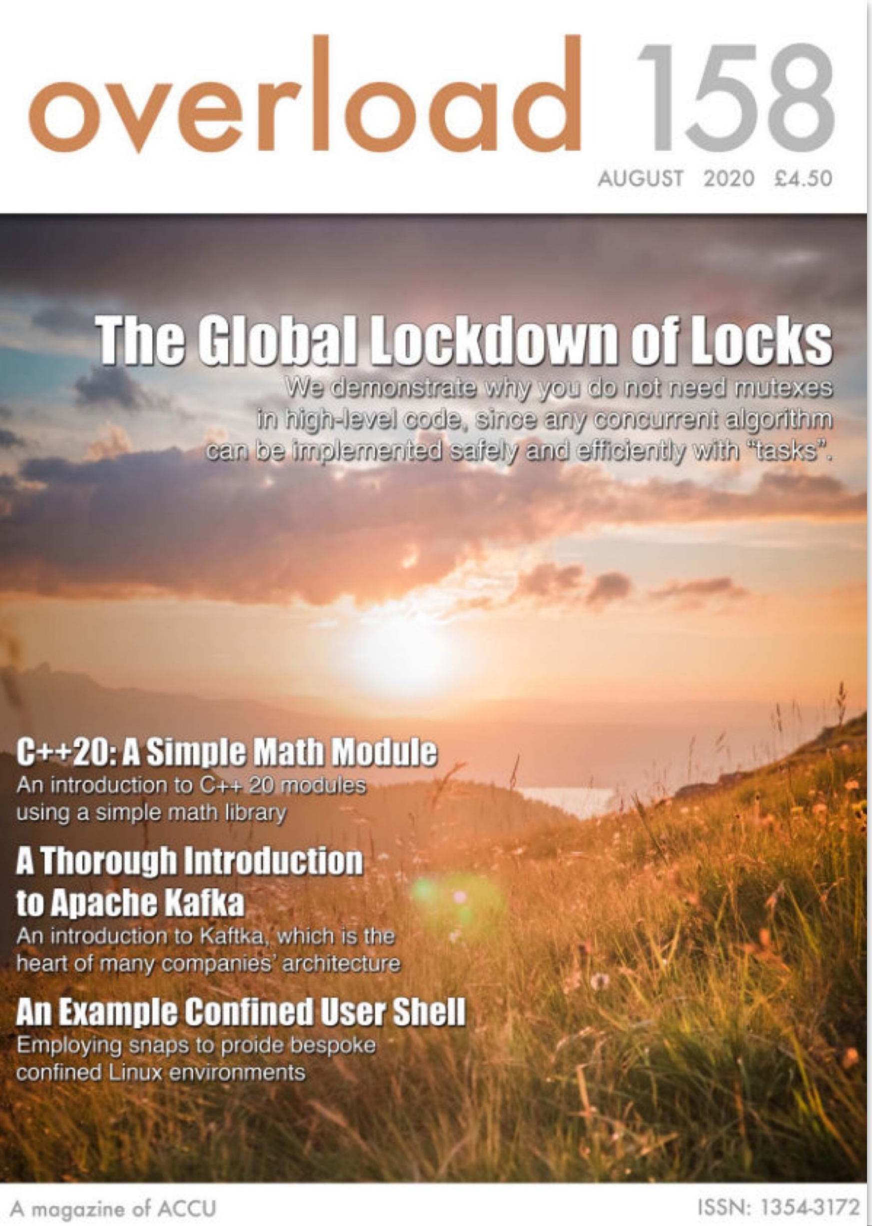
Refocusing Amdahl's Law

high efficiency for Greedy algo
high speedups

$$S_p \geq \frac{N}{K + \frac{N-K}{P}}$$

$$S_{1000} = 500.25 \qquad S_{10} = 9.91$$

$$N = 1000 \qquad K = 1$$

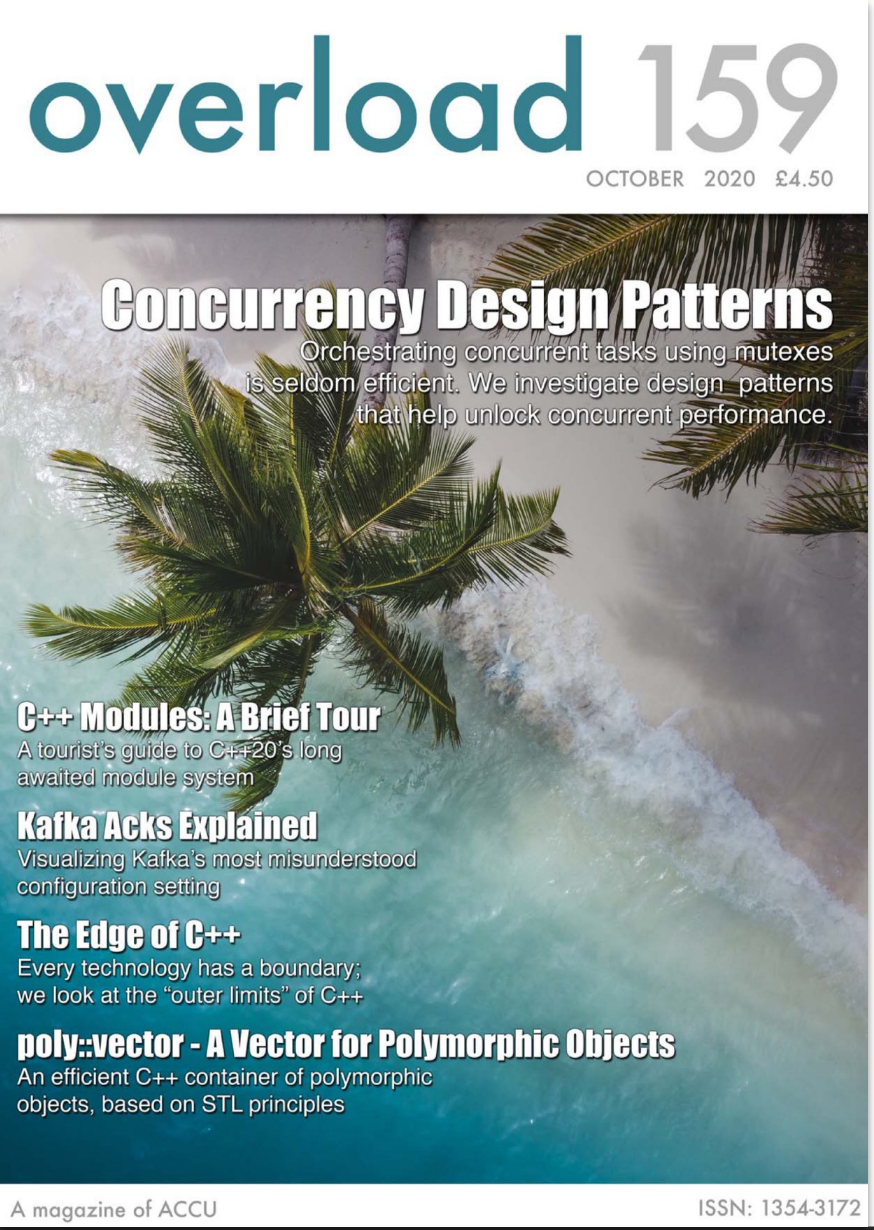


The Global Lockdown of Locks

global solution

safety ensured

no need for locks



Concurrency Design Patterns

building blocks for concurrent applications

https://youtu.be/_T1XjxXNSCs

Designing Concurrent C++ Applications

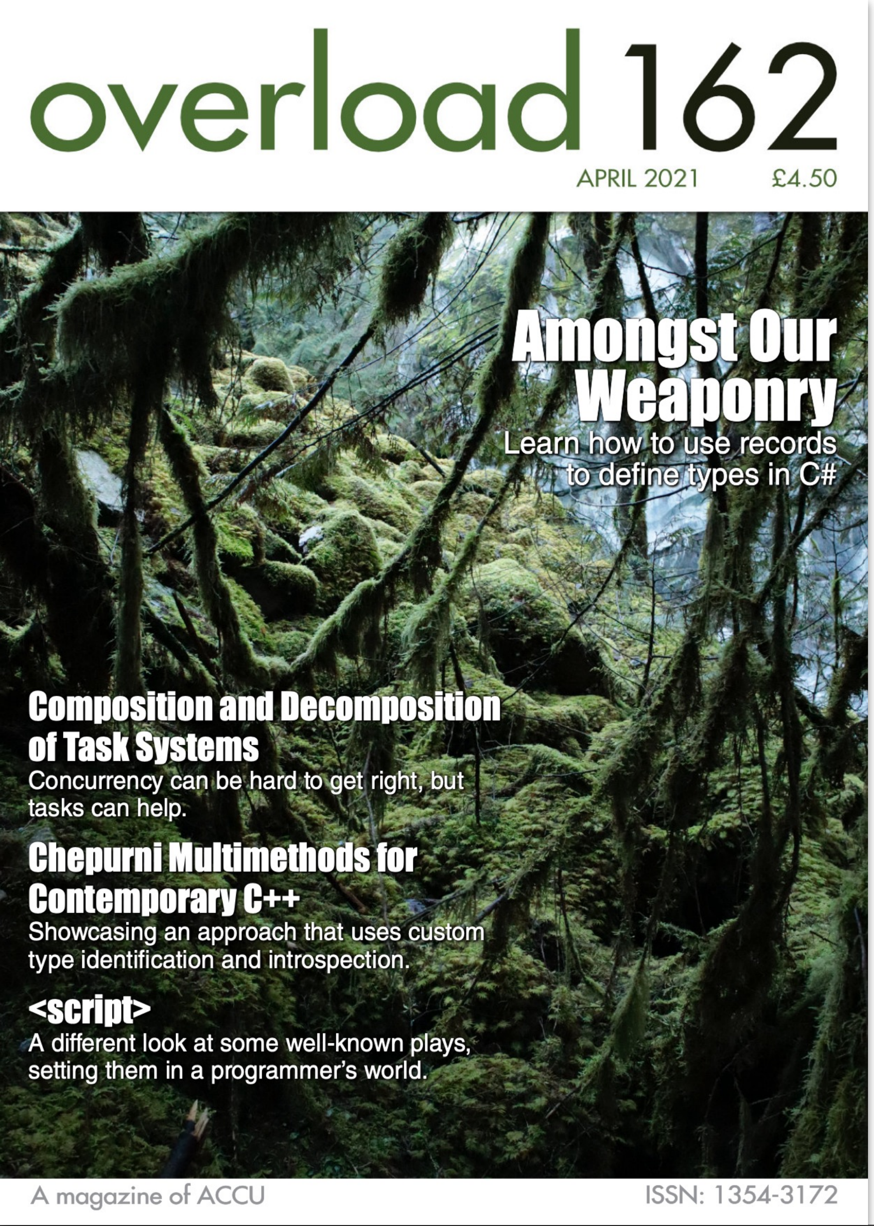
Lucian Radu Teodorescu

C++ now

https://bit.ly/2YnVG5U

tasks

a new **solution** for concurrency

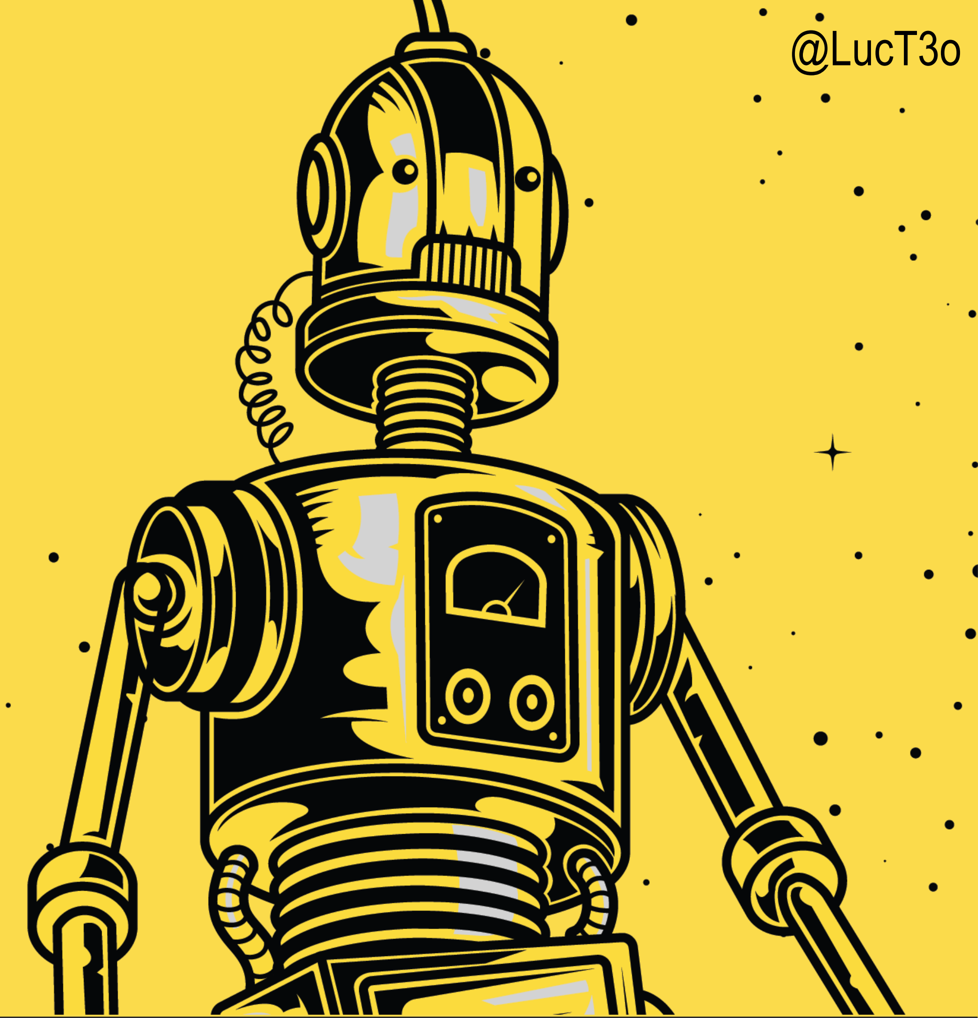


Composition and Decomposition of Task Systems

top-down and bottom-up design

composition of tasks

not the best solution



Async computations

C++ executors

P0443: A Unified Executors Proposal for C++

https://wg21.link/p0443r14

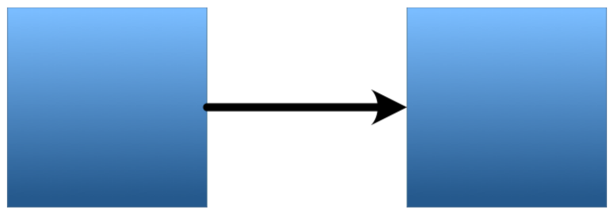
P2300: std::execution

https://wg21.link/p2300r2

Executors: a Change of Perspective

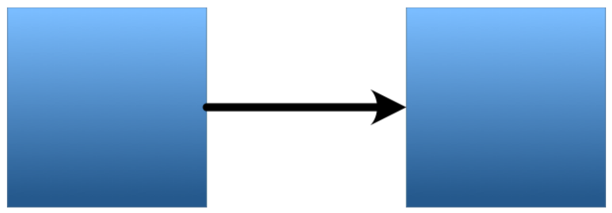
senders/receivers
are a **better**
concurrency abstraction

tasks



cannot directly pass values between tasks

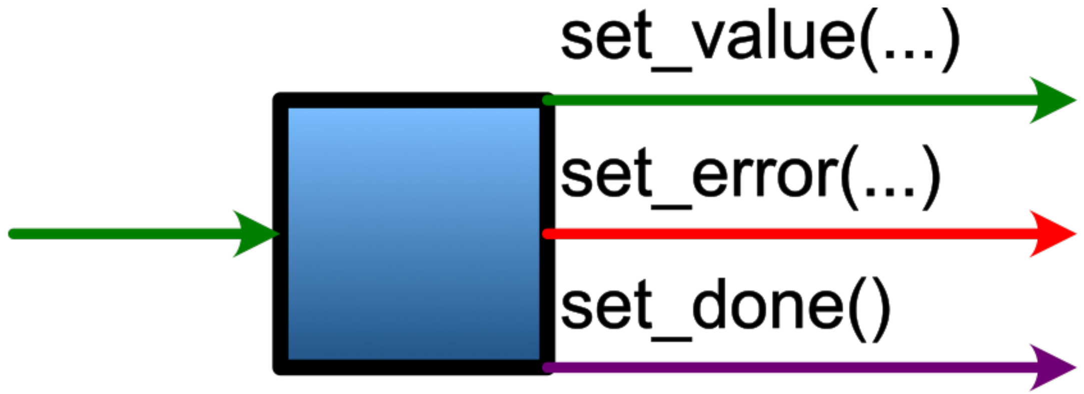
tasks



task body contains the call to the next task

what happens in case of error?

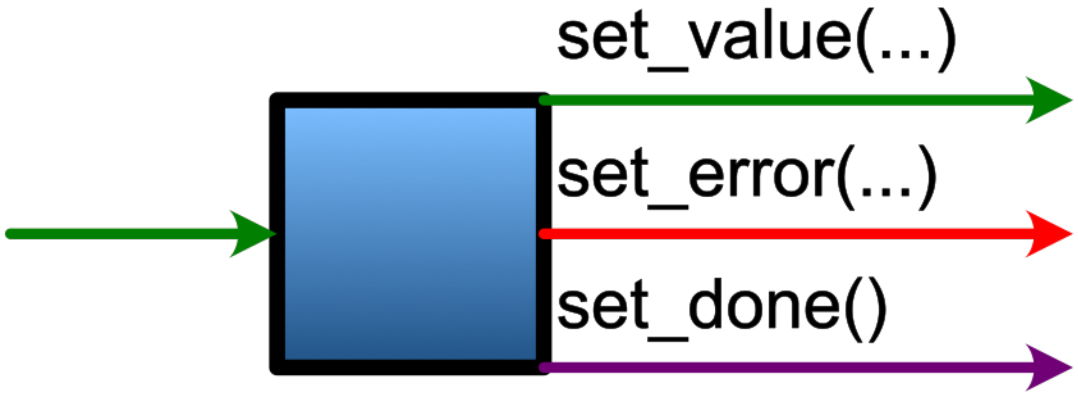
senders



3 notification channels

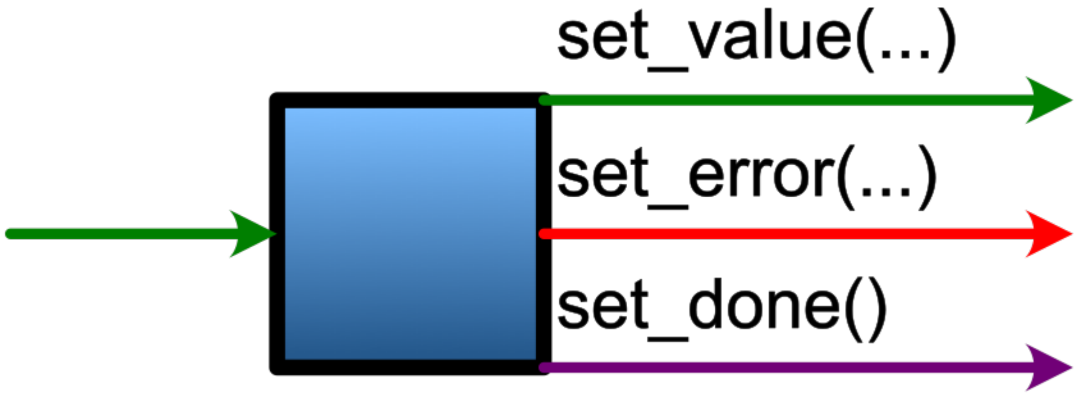
wiring done by the framework

senders



no performance penalty

asynchronous computations



rename **senders** into “**async computations**”

what can be a **computation**?

a small chunk of work
a task
a group of tasks
a group of task groups
the entire application

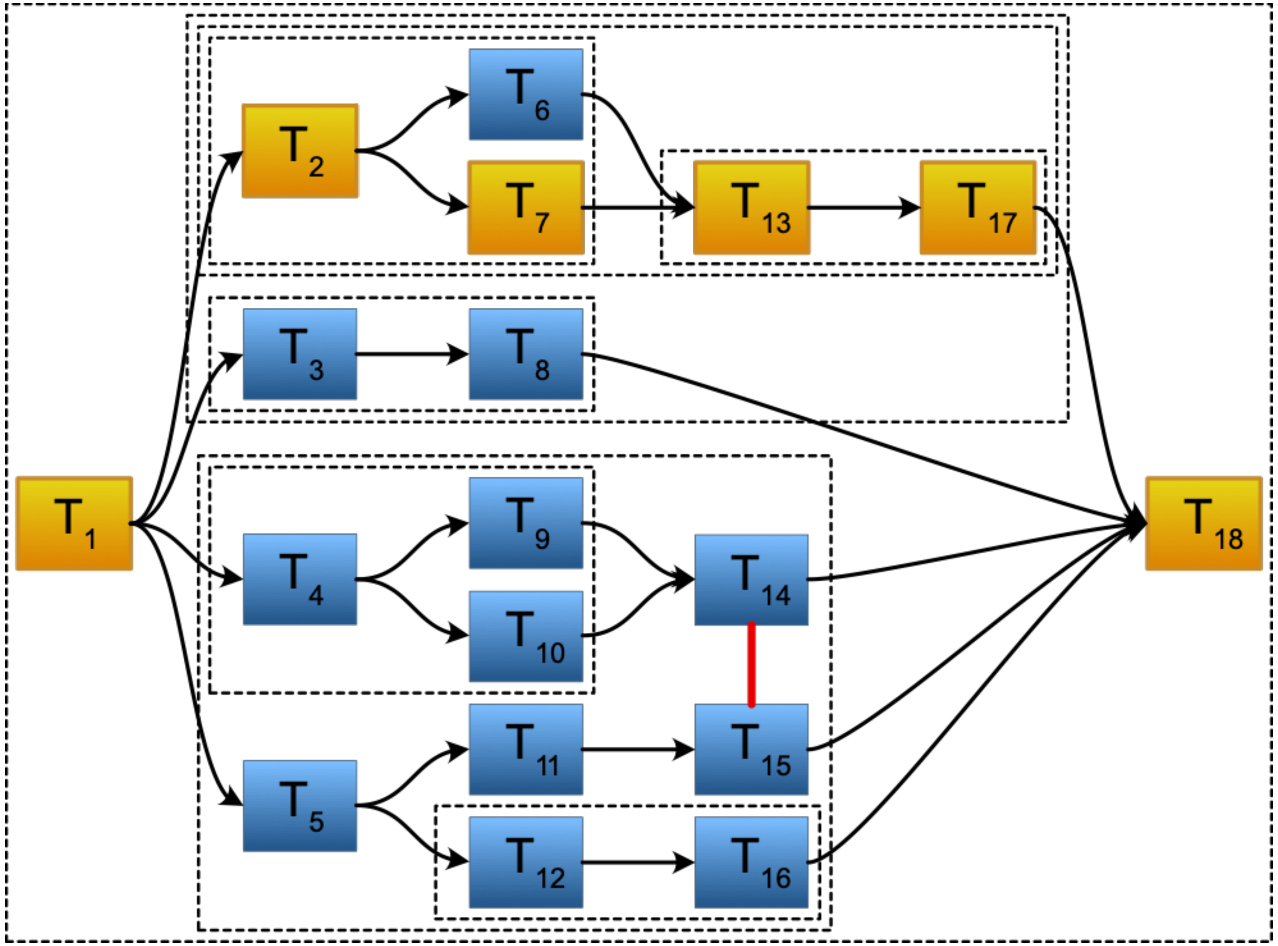
computations

general solution to concurrency

computations

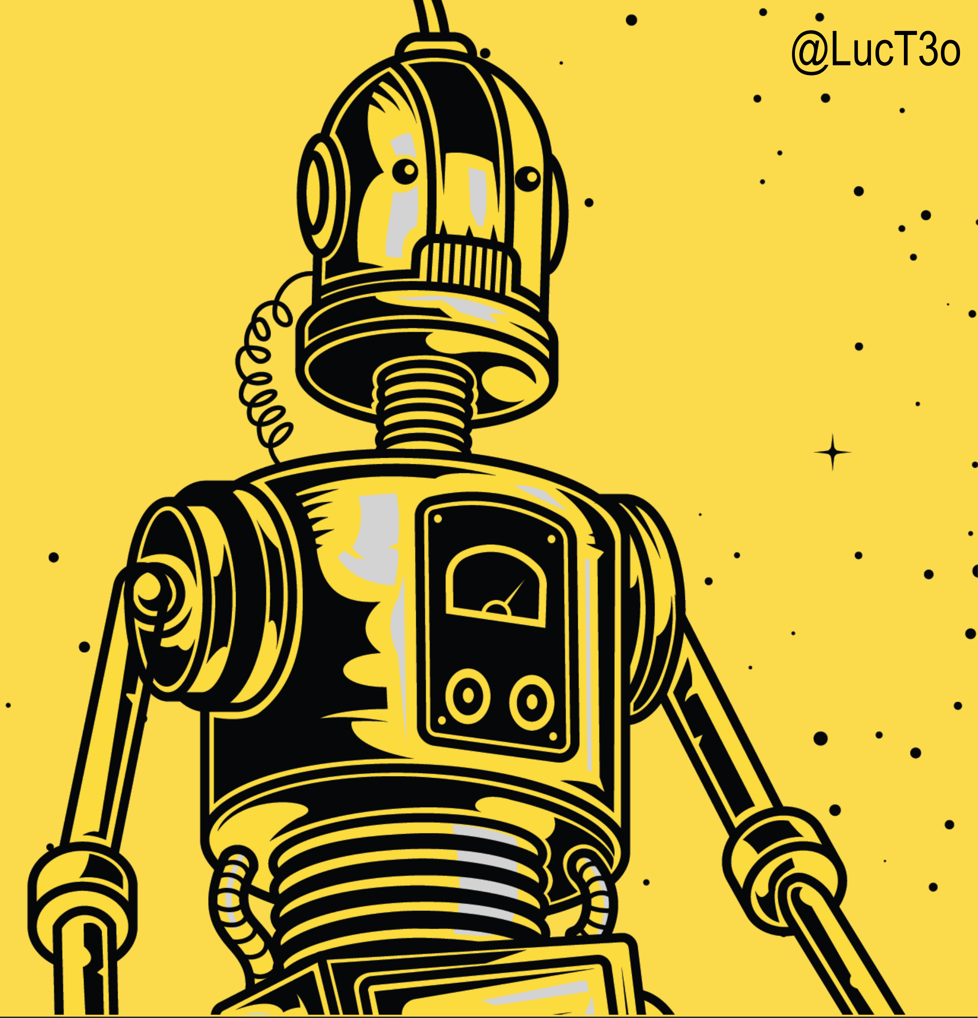
compose better than tasks

computations hierarchy



computation is an **abstraction**

allows us to incorporate concurrency in design



The future

goals

no more thread safety issues
clean design for concurrency

change of primitives

threads & locks

⇩

computations

change of approach

synchronization

⇩

constraints between computations

patterns & examples

make it teachable

widespread use

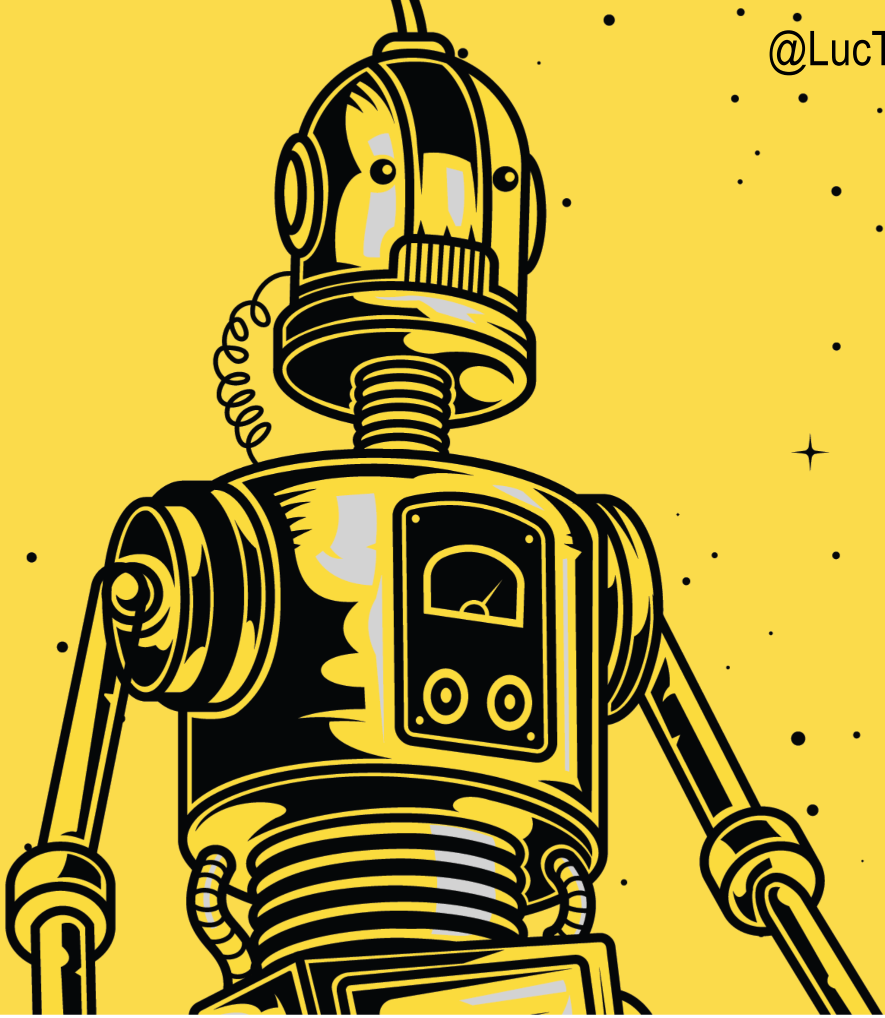
concurrency ≠ frustration

@LucT3o

iT days

#NewPerspectives

Q & A



accenture · BOSCH · betfair ROMANIA DEVELOPMENT · BT Code Crafters shape your future

RWS · /thoughtworks · SIEMENS Ingenuity for life · METRO digital · colors in projects

Connatix · ING · GARMIN. · Grab · accesa RaRo Part of the Ratiodata Group · TELENAV · FLOW TRADERS · msg · fme

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA · Organizer TSM SOFTWARE MAGAZINE · UNIVERSITATEA BABEȘ-BOLYAI · CLUJ IT · UNIVERSITATEA DE ARTĂ ȘI DESIGN CLUJ-NAPOCA