from the free world of
KDE

Reactive programming and Qt

Qt World Summit, Berlin 2015

Ivan Čukić

ivan.cukic@kde.org
http://cukic.co

About me

- KDE development
- Talks and teaching
- Functional programming enthusiast, but not a purist

Disclaimer

> Make your code readable. Pretend the next person who looks at your code is a psychopath and they know where you live.
>
> — Philip Wadler

Disclaimer

The code snippets are optimized for presentation, it is not production-ready code.

`std` namespace is omitted, value arguments used instead of const-refs or forwarding refs, etc.

REACTIVE

What is reactive?

> We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems. Systems built as Reactive Systems are more flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving users effective interactive feedback.
>
> Reactive Manifesto 2.0


BUZZWORDS
SPACE RANGER
LIGHTYEAR
BUZZWORDS EVERYWHERE

What is reactive?

Showing a response to a stimulus

Oxford Dictionary

What is reactive?

- C: event call-backs
- Java: event listeners
- C++/Qt: signals and slots
- even IO streams?

What is reactive?

- No shared state
- Separate isolated components
- Communication only through message passing

Reactive programming

```
connect(mouse,  SIGNAL(mouseMoved(int, int)),
        widget,   SLOT(resize(int, int)));
```

Reactive programming

```
connect(mouse,  SIGNAL(mouseMoved(int, int)),
        widget,   SLOT(setWidth(int)));
```

Reactive programming

```
Rectangle {
    width:  mouse.x
    height: mouse.y
}

MouseArea {
    id: mouse
    ...
}
```

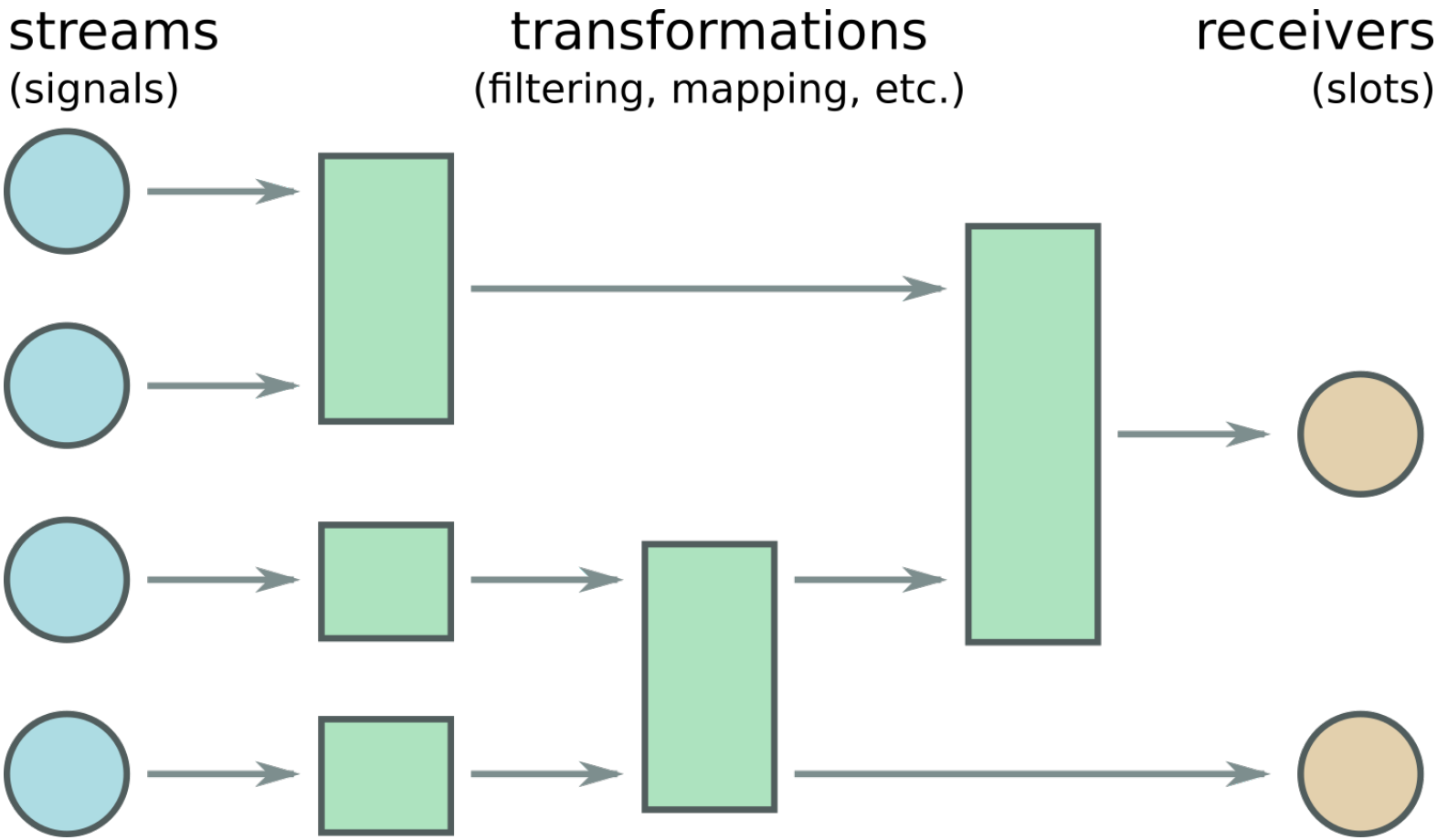
Functional reactive programming

```
Rectangle {
    width:  Math.sin(mouse.x - mouse.width / 2)
    height: Math.cos(mouse.y - mouse.height / 2)
}

MouseArea {
    id: mouse
    ...
}

Connections {
    ...
}
```

Functional reactive programming

```
Timer {
    id: delayedStatusUpdate
    interval: 1000
    running: true
    onTriggered: {
        if (!hasBlahBlah) {
            item.status = Core.PassiveStatus
            return
        }

        item.status = enabled
                ? Core.PassiveStatus
                : Core.ActiveStatus
    }
}
```

Stream processing



CONTINUATIONS

Concurrency

- Interactive systems
- Threads
- Multiple processes
- Distributed systems

Note: "concurrency" will mean that different tasks are executed at overlapping times.

Plain threads are bad

A large fraction of the flaws in software development are due to programmers not fully understanding all the **possible states** their code may execute in. In a multithreaded environment, the lack of understanding and the resulting problems are **greatly amplified**, almost to the point of panic if you are paying attention.

John Carmack
In-depth: Functional programming in C++

Plain threads are bad

Threads are not composable

Parallelism can't be 'disabled'

Difficult to ensure balanced load manually

Hartmut Kaiser
Plain Threads are the GOTO of Today's Computing
Meeting C++ 2014

Plain synchronization primitives are bad

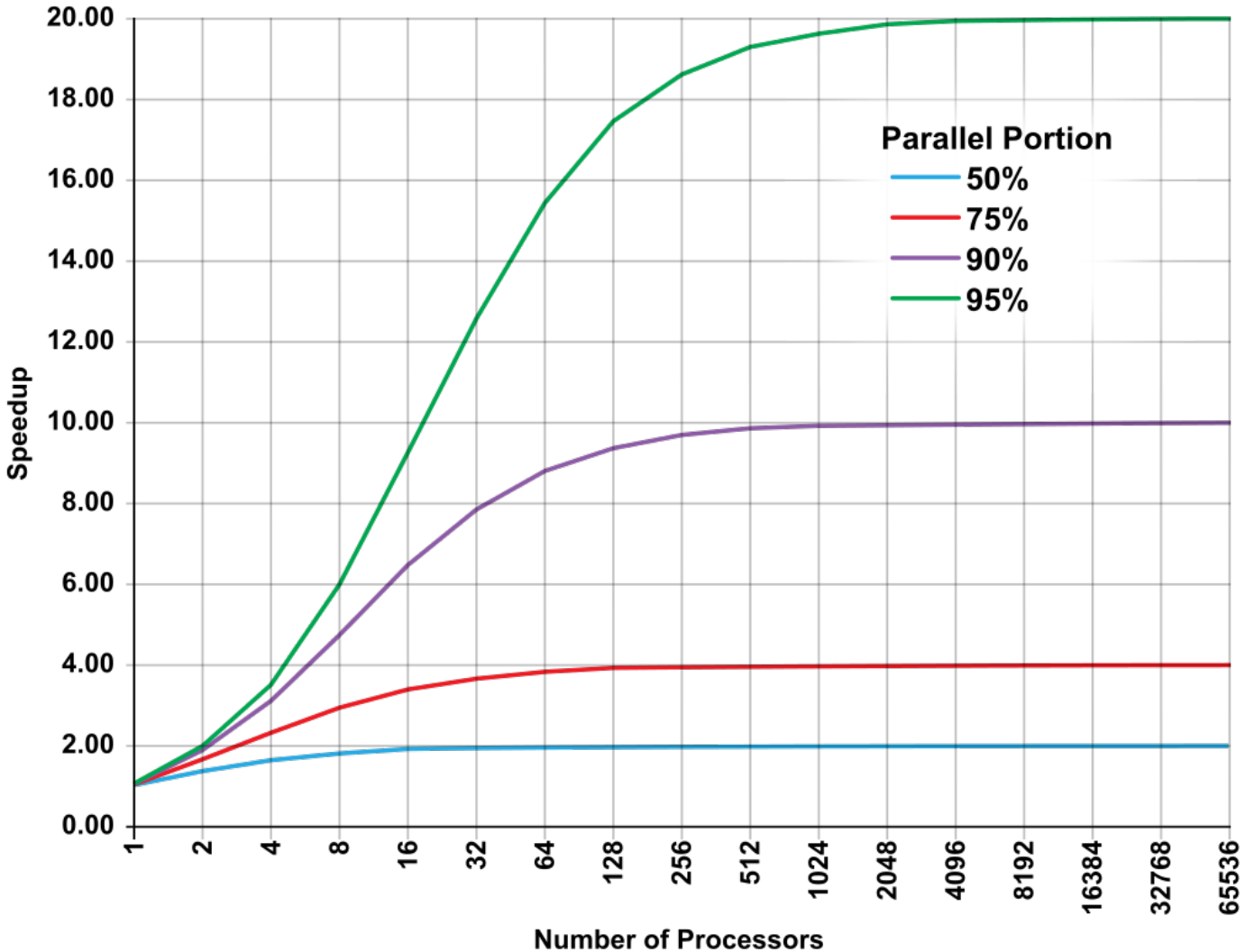
You will likely get it wrong

S.L.O.W. (starvation, latency, overhead, wait)

Sean Parent
Better Code: Concurrency
C++ Russia, 2015

Amdahl's Law

$$\frac{1}{(1-P)+\frac{P}{N}}$$



Locks are the main problem

The biggest of all the big problems with recursive mutexes is that they encourage you to completely lose track of your locking scheme and scope. This is deadly. Evil. It's the "thread eater". You hold locks for the absolutely shortest possible time. Period. Always. If you're calling something with a lock held simply because you don't know it's held, or because you don't know whether the callee needs the mutex, then you're holding it too long. You're aiming a shotgun at your application and pulling the trigger. You presumably started using threads to get concurrency; but you've just PREVENTED concurrency.

I've often joked that instead of picking up Djikstra's cute acronym we **should have called the basic synchronization object "the bottleneck"**. Bottlenecks are useful at times, sometimes indispensible – but they're never GOOD.

David Butenhof
Re: recursive mutexes

Futures

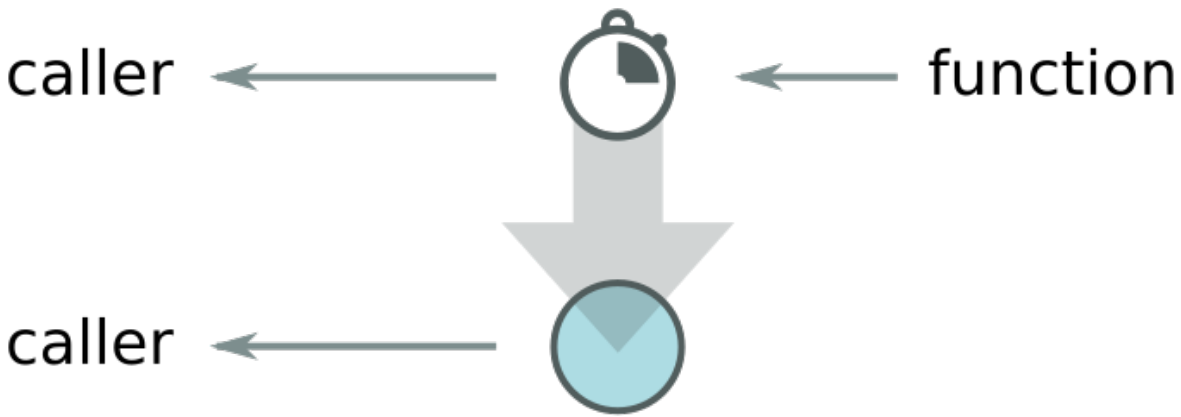
Futures should be the lowest level concurrency abstractions.

```
std::future
boost::future
QFuture
Folly Future

any continuation - *.then([] ...)
```

Future

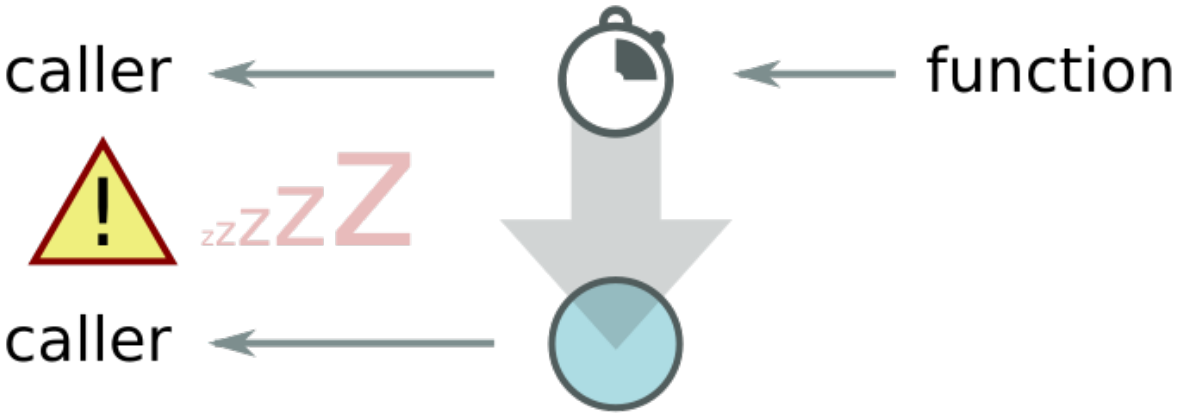
```
T value = function();
```


```
future<T> value = function(); ...; value.get();)
```

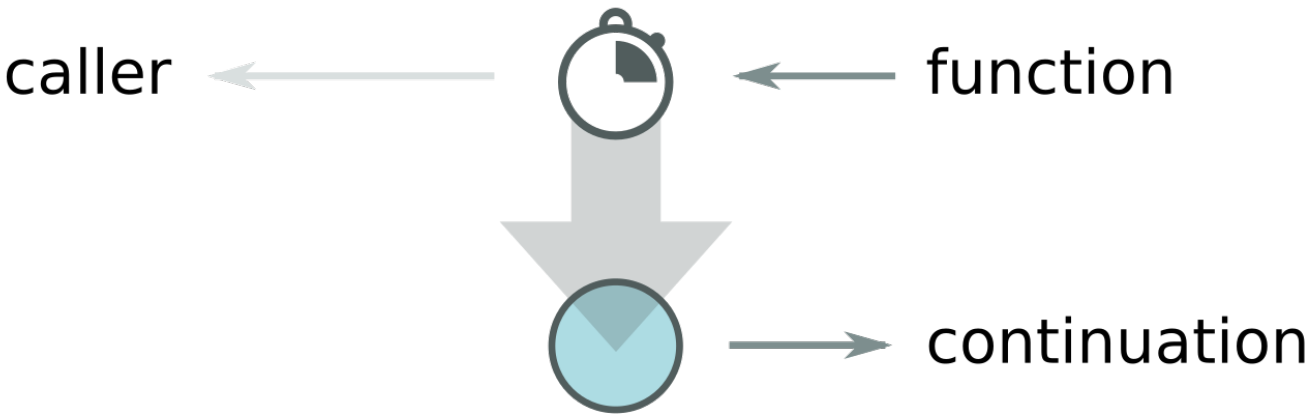


Future

```
future<T> value = function(); ...; value.get();
```




```
future<T2> value = function().then(continuation);
```



Futures

```
page("http://qtworldsummit.com/").get()
.then(
    [] (auto &&result) {
        cout << result.headers();
    }
)
```

Futures

```
page("http://qtworldsummit.com/").get()
.then(
    [] (auto &&result) {
        cout << result.headers();

        for (image: result.image_tags) {
            image.get().then(
                [] (auto &&image_result) {
                    // do something
                    // with image_result
                }
            );
        }
    }
)
```

Imperative chaining of futures

```
result = get("http://qtworldsummit.com/"),
for_(image = result.image_tags) (
    image_result = image.get(),
    // do something with image_result
    ...
)
```

Imperative chaining of futures

```
while_(
    // Wait until we get a connection.
    client = ws::server::accept(server),

    // Start a detached execution path to process the client.
    detach_([] {
        ...

        serial_(
            // WebSocket handshake
            header = ws::client::get_header(),
            server_key = ws::server::create_key(header),
            ws::client::send_header(client, server_key),

            // Sending the initial greeting message
            ws::client::message_write(client, "Hello, I'm Echo"),

            // Connection established
            while_(
                // getting and echoing the message
                message = ws::client::message_read(client),
                ws::client::message_write(client, message)
            )
        )
    })
)
```

Check out "Monads in chains" from Meeting C++ 2014

RANGES

Ranges in C++

```
vector<int> xs;
int sum = 0;

for (x: xs) {
    sum += x;
}

return sum;
```

Ranges in C++

```
return accumulate(
    xs.cbegin(), xs.cend(),
    0
    );
```

Ranges in C++

```
return accumulate(
    xs.cbegin(), xs.cend(),
    1,
    _1 * _2
    );
```

Ranges in C++

How to do an aggregation on a transformed list?

```
vector<int> xs;
int sum = 0;

for (x: xs) {
    sum += x * x;
}

return sum;
```

Ranges in C++

How to do an aggregation on a transformed list?

```
sum $ map (λ x → x * x) xs
```

Ranges in C++

How to do an aggregation on a transformed list?

```
vector<int> temp;

std::transform(
    xs.cbegin(), xs.cend(),
    std::back_inserter(temp),
    _1 * _1
    );

return std::accumulate(
    temp.cbegin(),
    temp.cend()
    );
```

Ranges in C++, boost.range, N4128

How to do an aggregation on a transformed list?

```
return accumulate(xs | transformed(_1 * _1));
```

Example

```
transactions
     | filter(Transactions::price() > 1000)
     | groupBy(Transactions::customerId())
     | sort(
         Transactions::price().desc() |
         Transactions::customerName()
       );
```

Example boilerplate

```
namespace Transactions {
    struct Record {
        int customerId;
        ...
    };

    DECL_COLUMN(customerId)
    ...
}
```

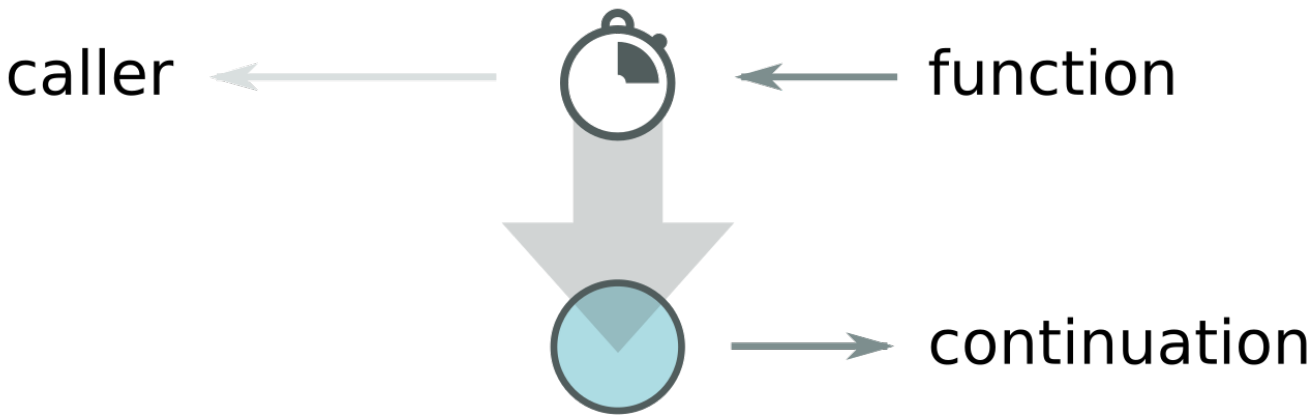
Column meta-type has all operators implemented, asc(), desc(), etc.

STREAMS

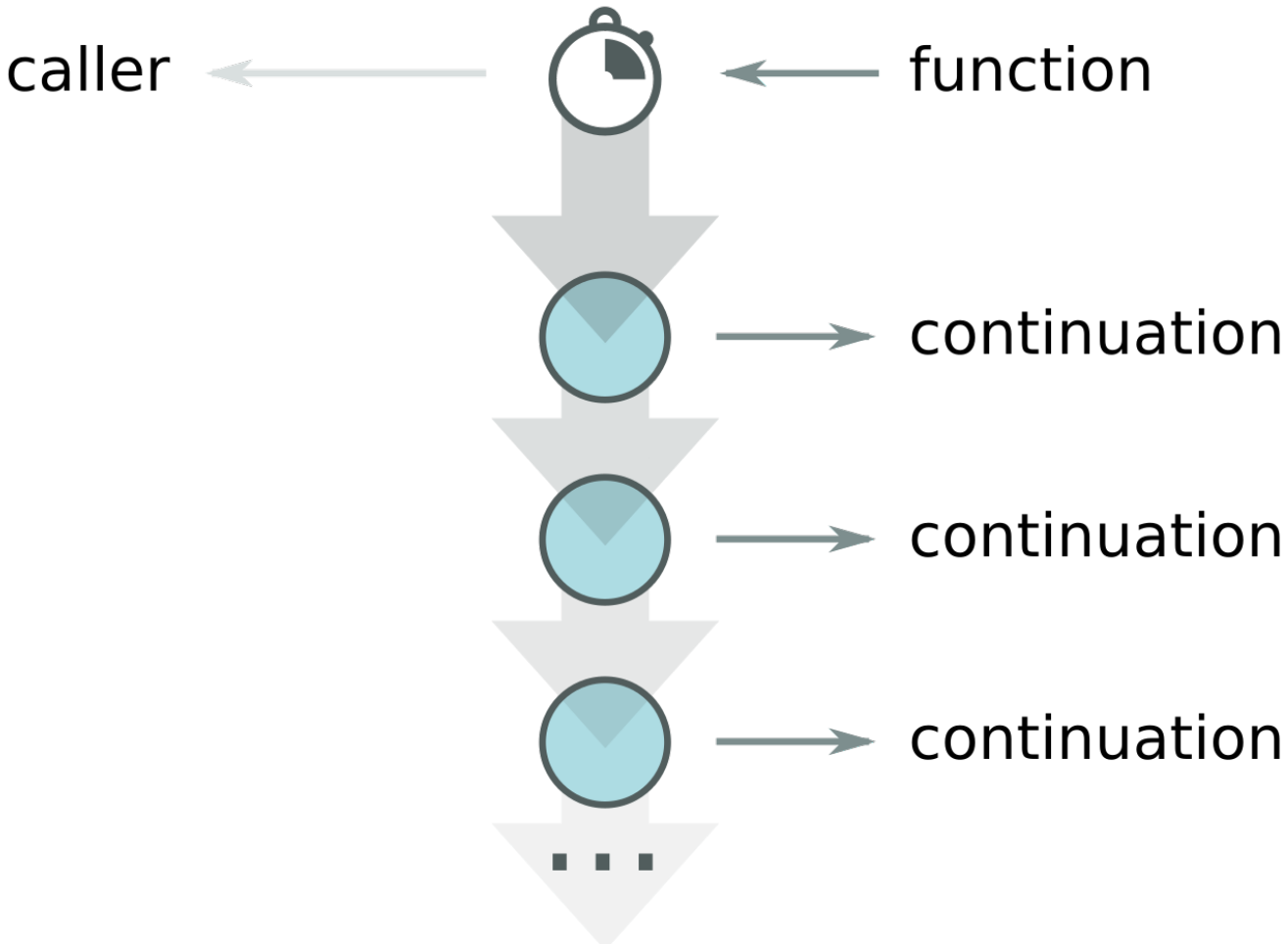
Anything you think that you could ever be

```
for (item: items) {
    // do something
}

for_each(items, [] (item i) {
    // do something
});
```

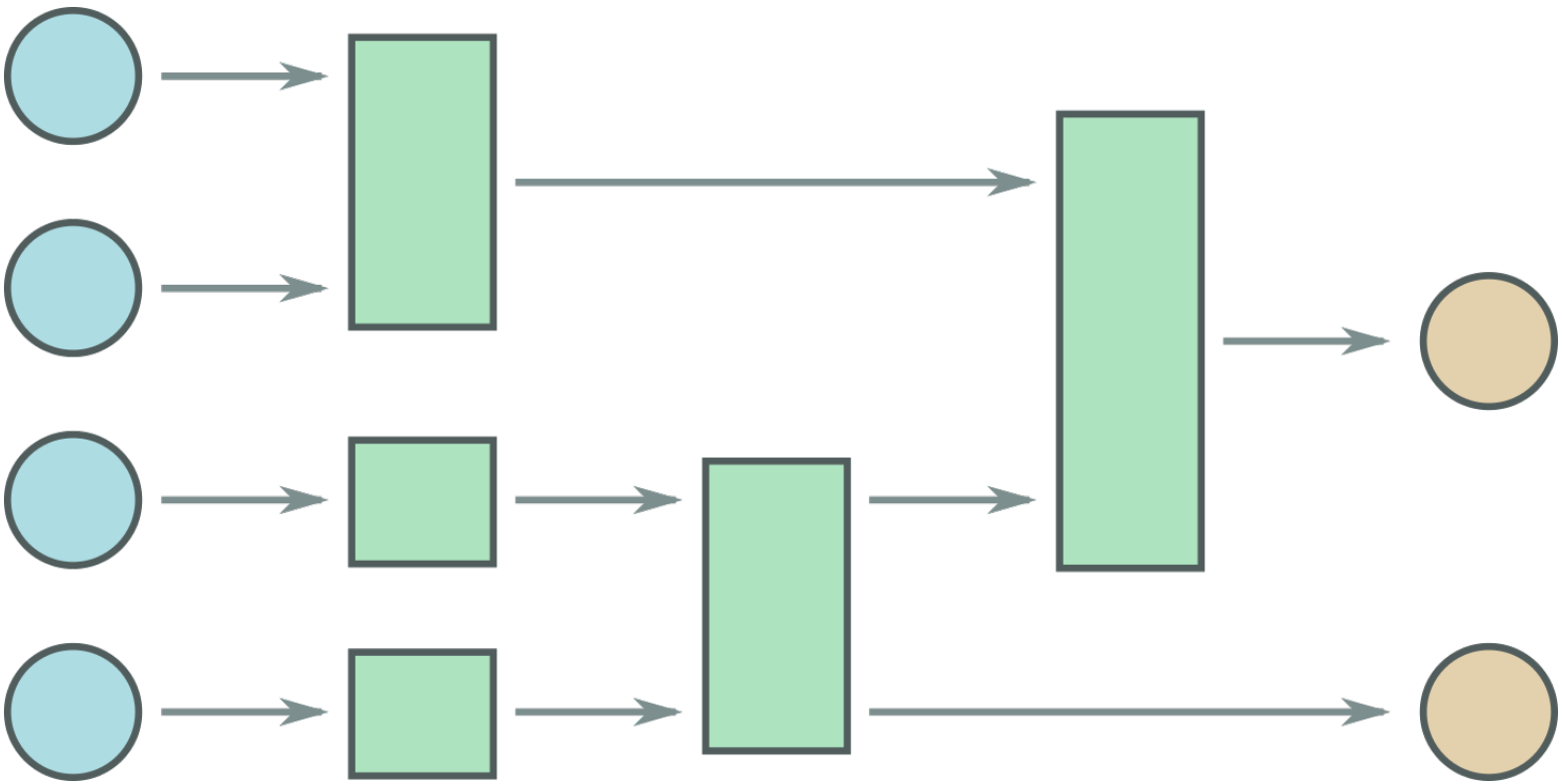
Just passing our time



Oh we'll keep on trying



Flow of information



Through the eons, and on and on

- Web server client connection requests
- User interface events
- Database access
- I/O
- Anything and everything

Till the end of time

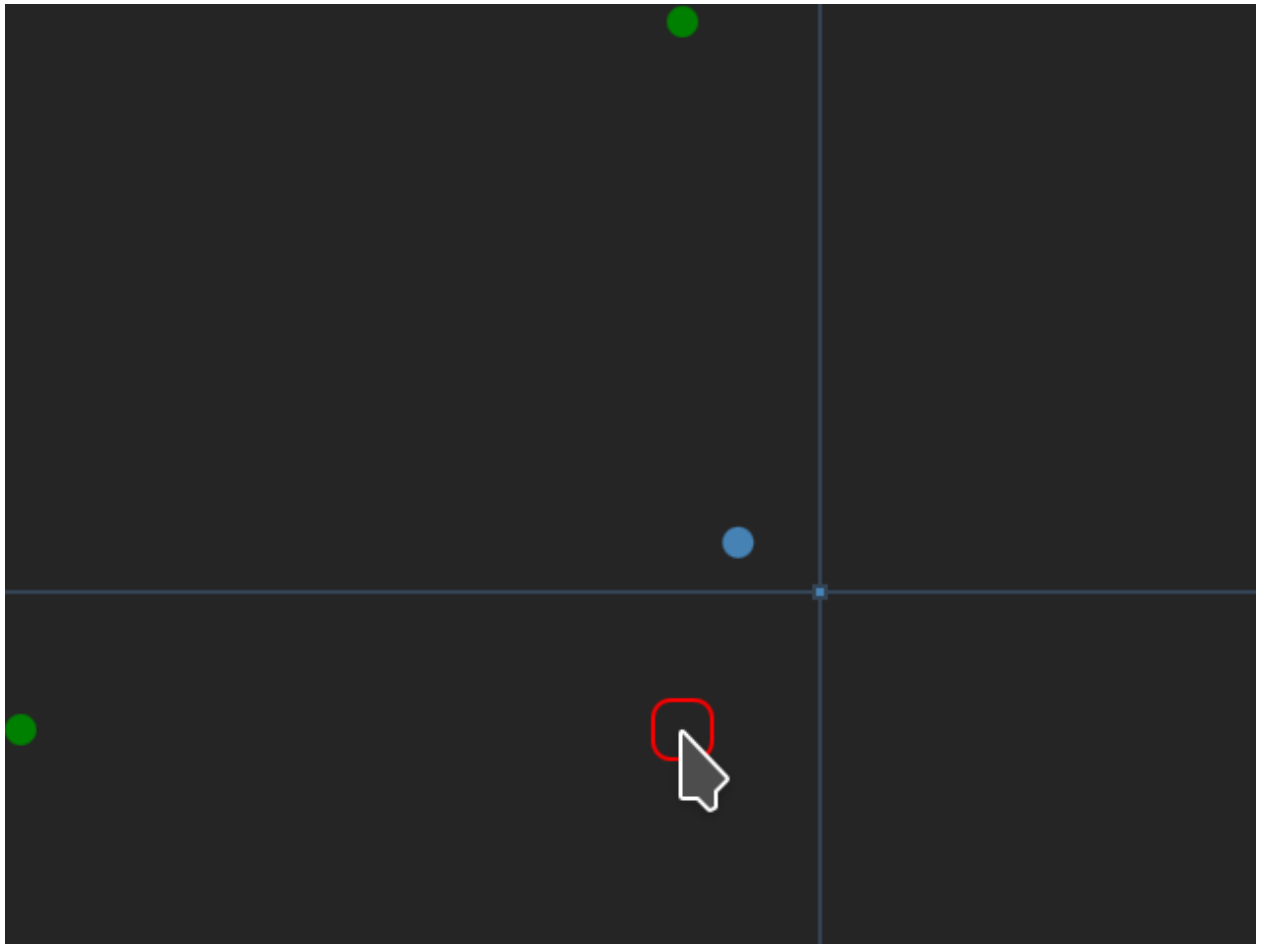
- Message passing:
  `continuation!newClientMessage`
- Call-callback:
  `onNewMessage(continuation)`
- Signals-slots:
```
connect(socket,   &Socket::newConnection,
        receiver, &Receiver::continuation)
```
- Any data collection:
  `for_each(xs, continuation)`

Stream transformation

Streams can only be transformed with algorithms that accept input ranges, since we don't have all the items. We don't even know when (if) they will end.

`map`, `bind`, `filter`, `take`, `drop`, etc.

Stream transformation



Map / Transform

We have a stream of 2D coordinates (mouse coordinates).

```
// Projecting on the x-axis
mouse_position >>=
    map(λ point → (point.x, 0))

// Projecting on the y-axis
mouse_position >>=
    map(λ point → (0, point.y))
```

Implementation detail

```
namespace stream {
    template <typename Stream, typename Cont>
    auto operator >>= (Stream &&stream,
                       Cont &&cont)
    {
        return stream.then(cont);
    }

    template <typename Under>
    auto make_stream(Under &&under);
}
```

Map

```
template <typename Func, typename Cont>
struct map_cont {
    map_cont(Func f, Cont c) : f(f), c(c) { }

    template <typename InType>
    void operator () (const InType &in) {
        c(f(in));
    }

    Func f;
    Cont c;
};
```

Fork (or parallel), tee

```
tee(print) >>=
fork(
    receiver1,
    receiver2
)
```

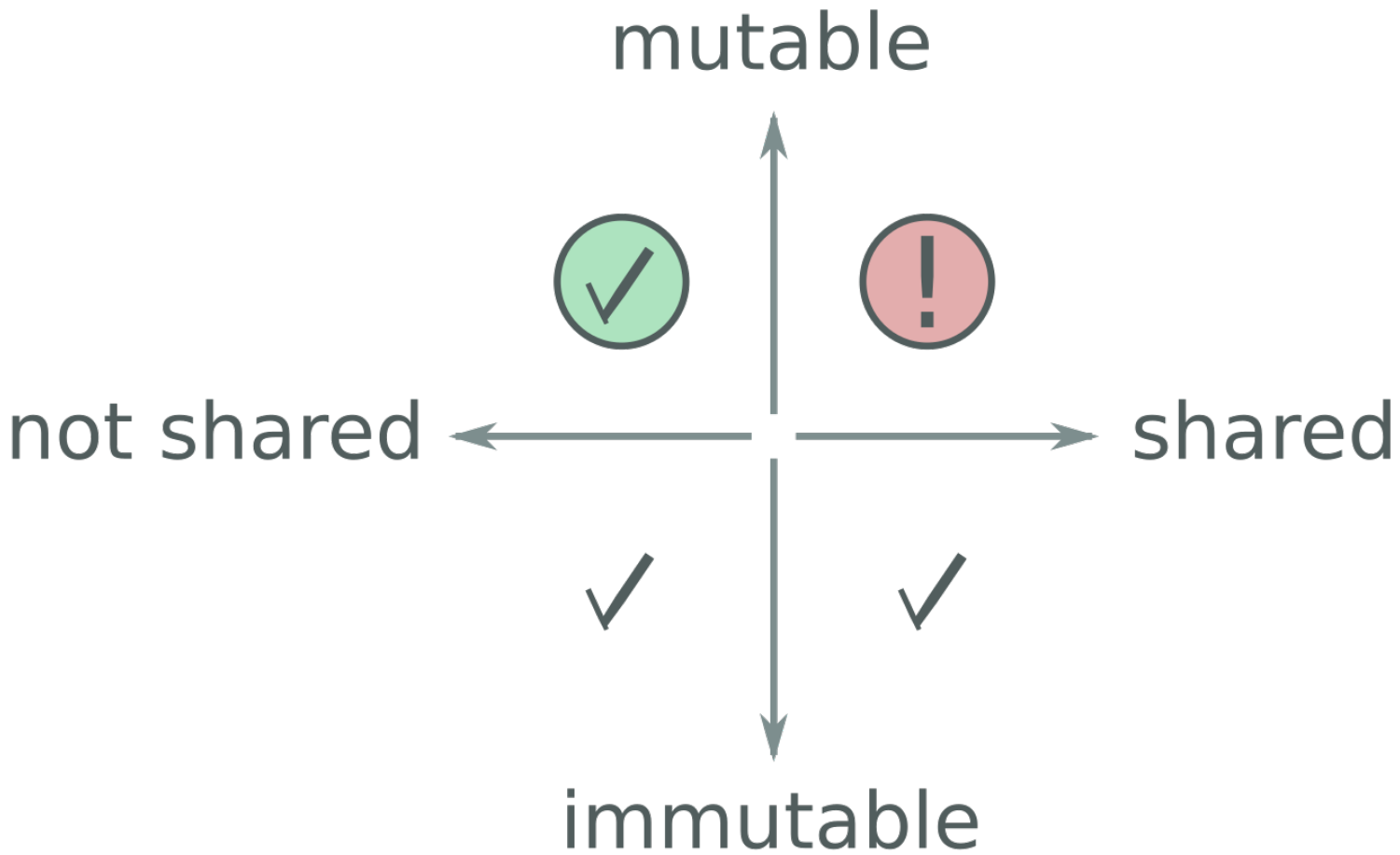
Fork (or parallel), tee

```
template <typename ... Conts>
struct fork_impl;

template <typename Cont, typename ... Conts>
struct fork_impl<Cont, Conts...>: fork_impl<Conts...>
{
    using parent_type = fork_impl<Conts...>;

    fork_impl(Cont c, Conts... cs)
        : parent_type(cs...), c(c)
    { }

    template <typename InType>
    void operator() (const InType &in) {
        c(in);
        parent_type::operator()(in);
    }

    Cont c;
};
```

Stateful function objects

```
class gravity_object {
public:
    gravity_object() { }

    template <typename Cont>
    void then(Cont &&c) { _f = std::forward<Cont>(c); }

    QPointF operator() (const QPointF &new_point) {
        m_point.setX(m_point.x() * .99 + new_point.x() * .01);
        m_point.setY(m_point.y() * .99 + new_point.y() * .01);
        return m_point;
    }

private:
    std::function<void(QPointF)> _f;
    QPointF m_point;
};
```

Stateful function objects



Filter

```
bool pointFilter(const QPointF &point) {
    return int(point.y()) % 100 == 0;
}

events >>=
    filter(predicate) >>=
        ...
```

Flat map

```
template <typename Func, typename Cont>
struct flatmap_cont {
    flatmap_cont(Func f, Cont c)
        : f(f)
        , c(c)
    {
    }

    template <typename InType>
    void operator () (const InType &in) {
        boost::for_each(f(in), c);
    }

    Func f;
    Cont c;
};
```

Flat map

```
class more_precision {
public:
    more_precision() { }

    template <typename Cont>
    void then(Cont &&c) { _f = std::forward<Cont>(c); }

    std::vector<QPointF> operator() (const QPointF &new_point) {
        std::vector<QPointF> result;

        int stepX = (m_previous_point.x() < new_point.x()) ? 1 : -1;
        for (int i = (int)m_previous_point.x(); i != (int)new_point.x(); i += stepX) {
            result.emplace_back(i, m_previous_point.y());
        }

        int stepY = (m_previous_point.y() < new_point.y()) ? 1 : -1;
        for (int i = (int)m_previous_point.y(); i != (int)new_point.y(); i += stepY) {
            result.emplace_back(new_point.x(), i);
        }

        m_previous_point = new_point;
        return result;
    }

private:
    std::function<void(QPointF)> _f;
    QPointF m_previous_point;

};
```

Answers? Questions! Questions? Answers!

Kudos:

Friends at KDE, Dr Saša Malkov

Worth reading and watching:

- Iterators Must Go, Andrei Alexandrescu
- Value Semantics and Range Algorithms, Chandler Carruth
- Systematic Error Handling in C++, Andrei Alexandrescu
- Ranges proposal, Eric Niebler
- Reactive manifesto, Books on Erlang or Scala/Akka