



# EFFECTIVE REPLACEMENT OF DYNAMIC POLYMORPHISM WITH `std::variant`

Mateusz Pusz  
September 27, 2018



**WHY?**



# WHY?

---

## LATENCY

- *Time required to perform some action* or to produce some result
- Measured in units of time like hours, minutes, seconds, nanoseconds or clock periods



# WHY?

## LATENCY

- *Time required to perform some action* or to produce some result
- Measured in units of time like hours, minutes, seconds, nanoseconds or clock periods

## LOW LATENCY

- In capital markets, the use of algorithmic trading to *react to market events faster than the competition* to increase profitability of trades
- Many use cases where *predictability of latency in message delivery is just as important*, if not more important than *achieving a low average latency*

# HOW **NOT** TO DEVELOP SOFTWARE THAT HAVE PREDICTABLE PERFORMANCE?

- In Low Latency system we care a lot about **WCET** (**W**orst **C**ase **E**xecution **T**ime)
- In order to limit **WCET** we should *limit the usage of specific C++ language features*



# THINGS TO AVOID ON THE FAST PATH

---

# THINGS TO AVOID ON THE FAST PATH

---

- C++ tools that trade performance for usability (e.g. `std::shared_ptr`, `std::function`)

# THINGS TO AVOID ON THE FAST PATH

---

- C++ tools that trade performance for usability (e.g. `std::shared_ptr`, `std::function`)
- Throwing exceptions on likely code path



# THINGS TO AVOID ON THE FAST PATH

---

- C++ tools that trade performance for usability (e.g. `std::shared_ptr`, `std::function`)
- Throwing exceptions on likely code path
- Dynamic polymorphism

# THINGS TO AVOID ON THE FAST PATH

---

- C++ tools that trade performance for usability (e.g. `std::shared_ptr`, `std::function`)
- Throwing exceptions on likely code path
- Dynamic polymorphism
- Multiple inheritance

# THINGS TO AVOID ON THE FAST PATH

---

- C++ tools that trade performance for usability (e.g. `std::shared_ptr`, `std::function`)
- Throwing exceptions on likely code path
- Dynamic polymorphism
- Multiple inheritance
- RTTI

# THINGS TO AVOID ON THE FAST PATH

---

- C++ tools that trade performance for usability (e.g. `std::shared_ptr`, `std::function`)
- Throwing exceptions on likely code path
- Dynamic polymorphism
- Multiple inheritance
- RTTI
- Dynamic memory allocations

# THINGS TO AVOID ON THE FAST PATH

---

- C++ tools that trade performance for usability (e.g. `std::shared_ptr`, `std::function`)
- Throwing exceptions on likely code path
- Dynamic polymorphism
- Multiple inheritance
- RTTI
- Dynamic memory allocations



**HOW?**



# HOW?

## DYNAMIC POLYMORPHISM

```
class base : noncopyable {  
public:  
    virtual ~base() = default;  
    virtual void foo() = 0;  
};
```

```
class x : public base {  
public:  
    void foo() override;  
};
```

```
class y : public base {  
public:  
    void foo() override;  
};
```

```
std::unique_ptr<base> b = std::make_unique<x>();  
b->foo();
```

# HOW?

## DYNAMIC POLYMORPHISM

```
class base : noncopyable {  
public:  
    virtual ~base() = default;  
    virtual void foo() = 0;  
};
```

```
class x : public base {  
public:  
    void foo() override;  
};
```

```
class y : public base {  
public:  
    void foo() override;  
};
```

```
std::unique_ptr<base> b = std::make_unique<x>();  
b->foo();
```

# HOW?

## DYNAMIC POLYMORPHISM

```
class base : noncopyable {  
public:  
    virtual ~base() = default;  
    virtual void foo() = 0;  
};
```

```
class x : public base {  
public:  
    void foo() override;  
};
```

```
class y : public base {  
public:  
    void foo() override;  
};
```

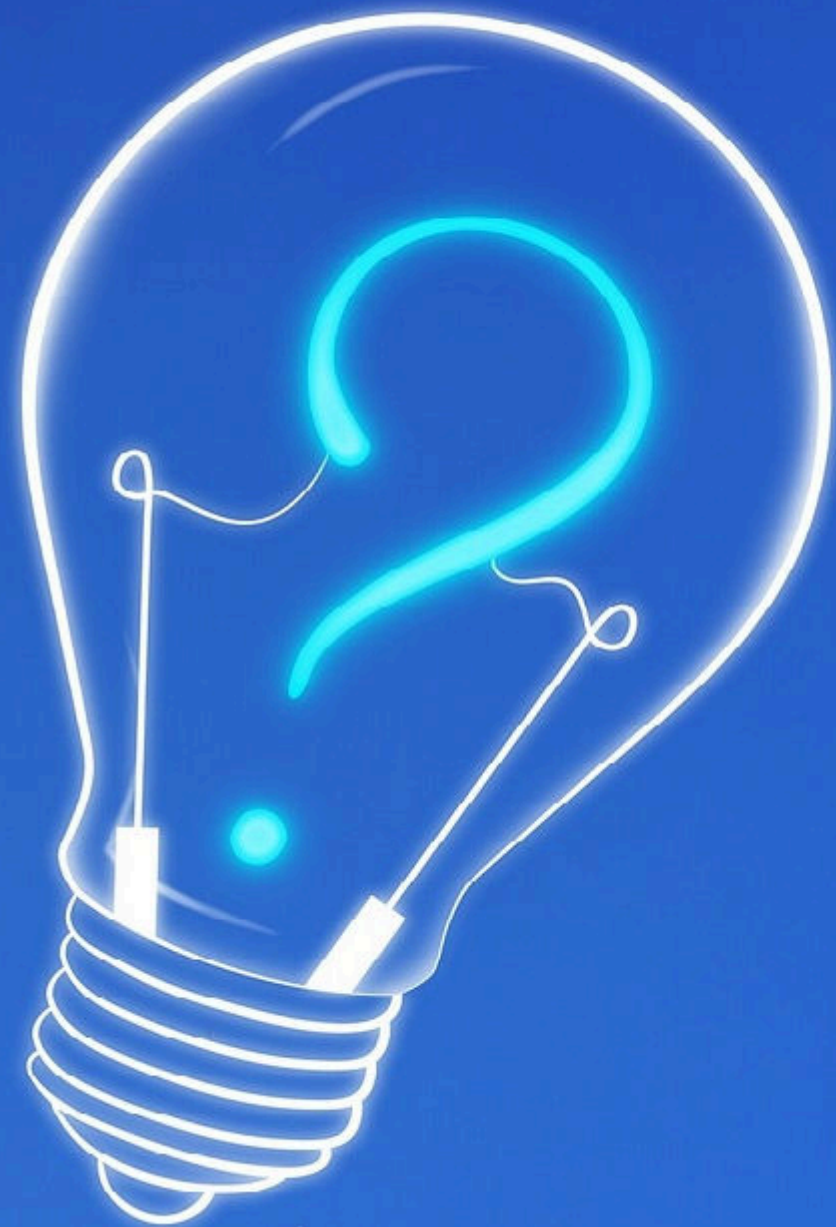
```
std::unique_ptr<base> b = std::make_unique<x>();  
b->foo();
```

## VARIANT

```
struct x { void foo(); };  
struct y { void foo(); };
```

```
std::variant<x, y> b;  
std::visit([](auto&& v){ v.foo(); }, b);
```

- Shorter
- Faster
- Value semantics
- Works on unrelated classes
- More flexible thanks to duck typing





**BONUS SLIDES**

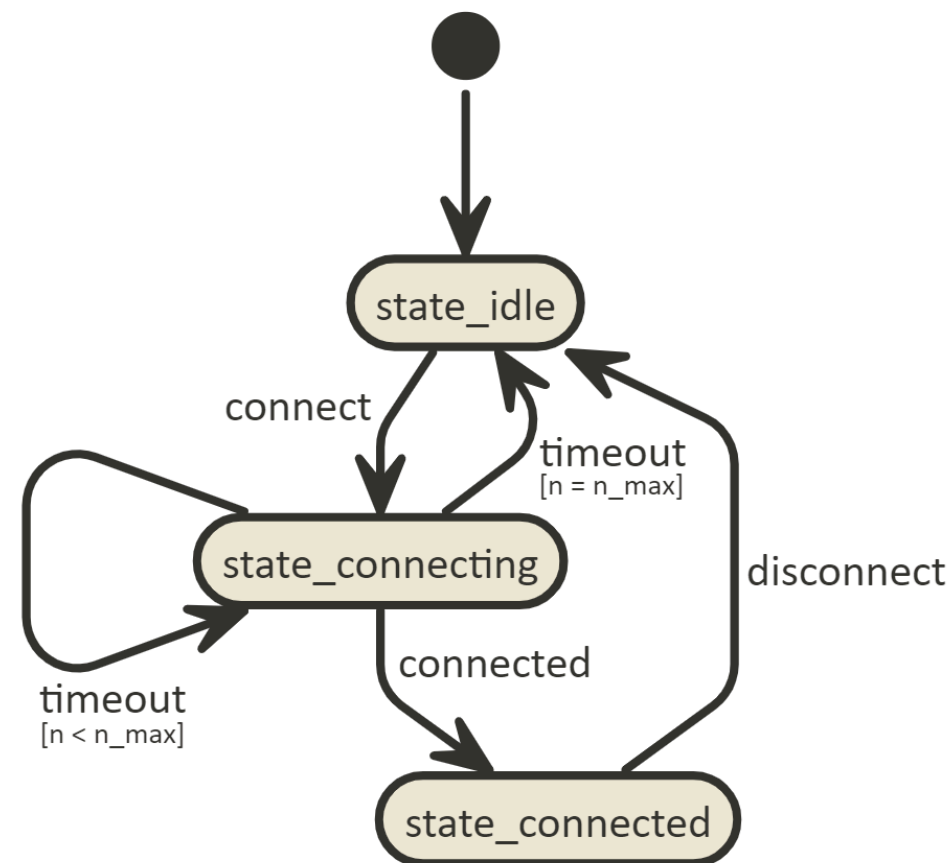


# FINITE STATE MACHINE

Abstract machine that can be in exactly one of a finite number of states at any given time.

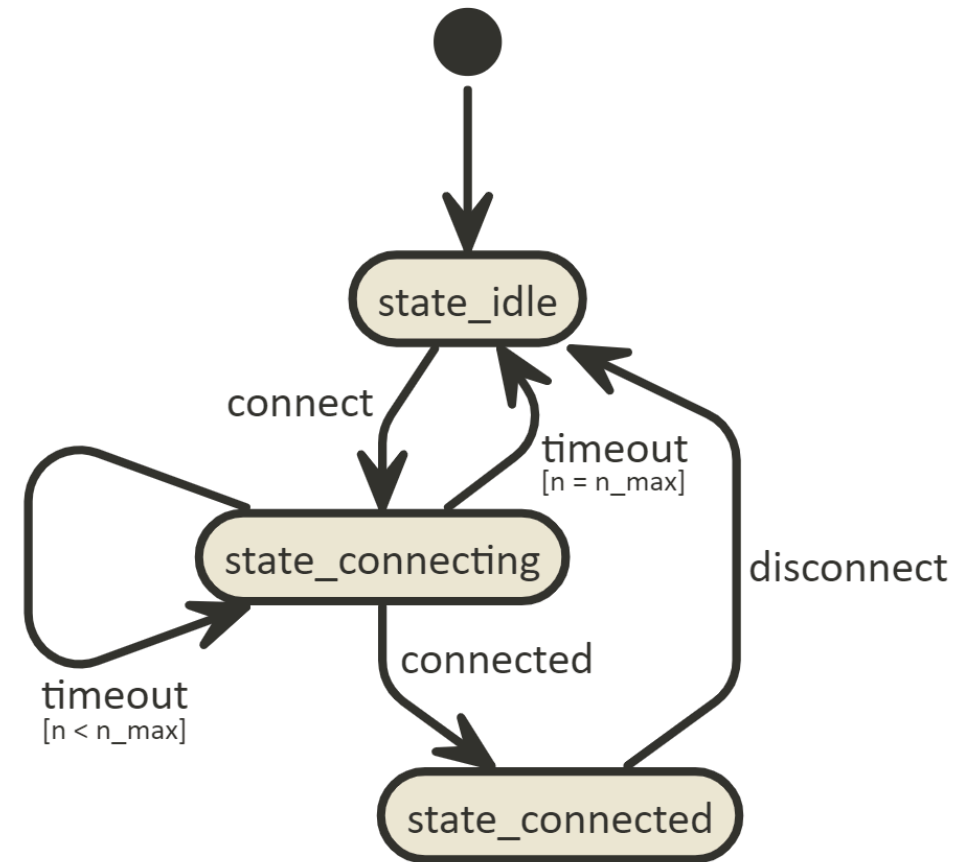
-- *Wikipedia*

# FINITE STATE MACHINE



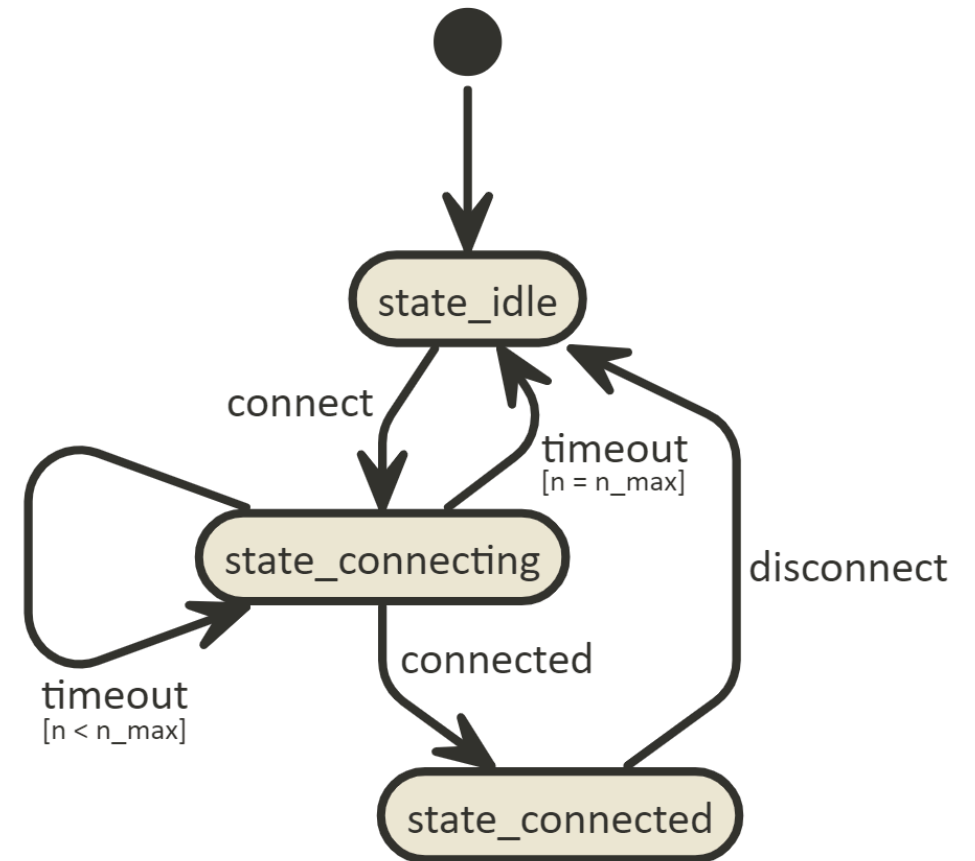
# FINITE STATE MACHINE

- **State** changes to another in a response to some external inputs called **events**



# FINITE STATE MACHINE

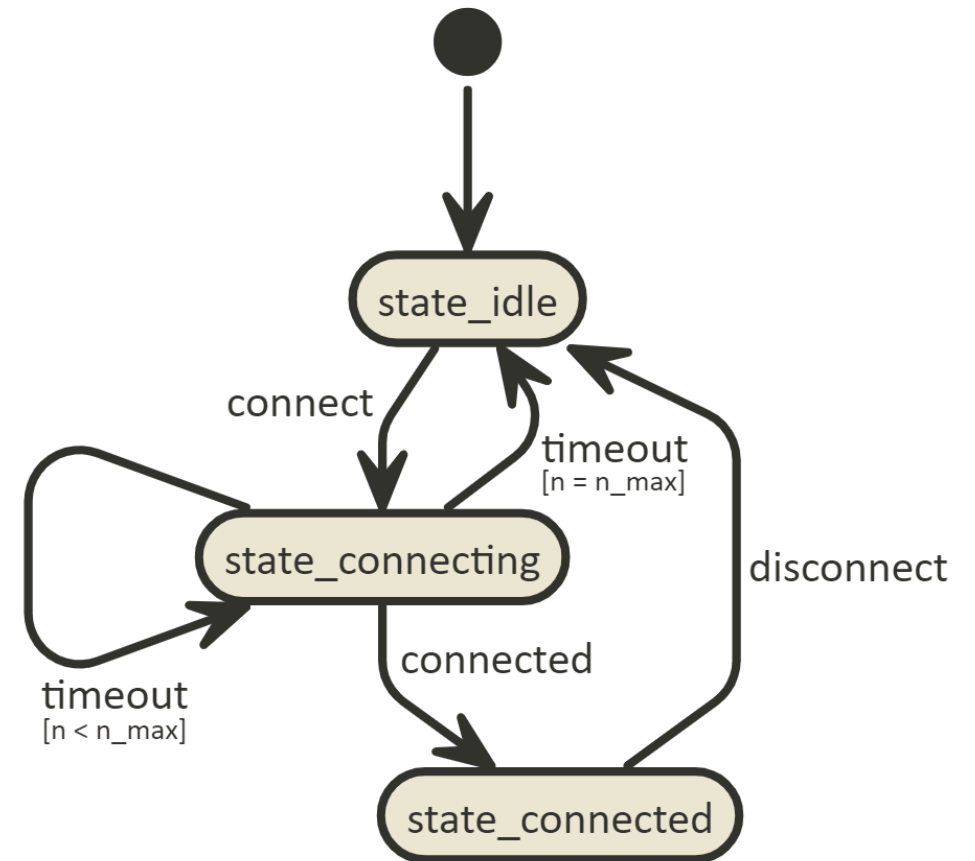
- **State** changes to another in a response to some external inputs called **events**
- The change from one state to another is called a **transition**





# FINITE STATE MACHINE

- **State** changes to another in a response to some external inputs called **events**
- The change from one state to another is called a **transition**
- A FSM is **defined by**
  - a *list of states*
  - its *initial state*
  - the *conditions for each transition*

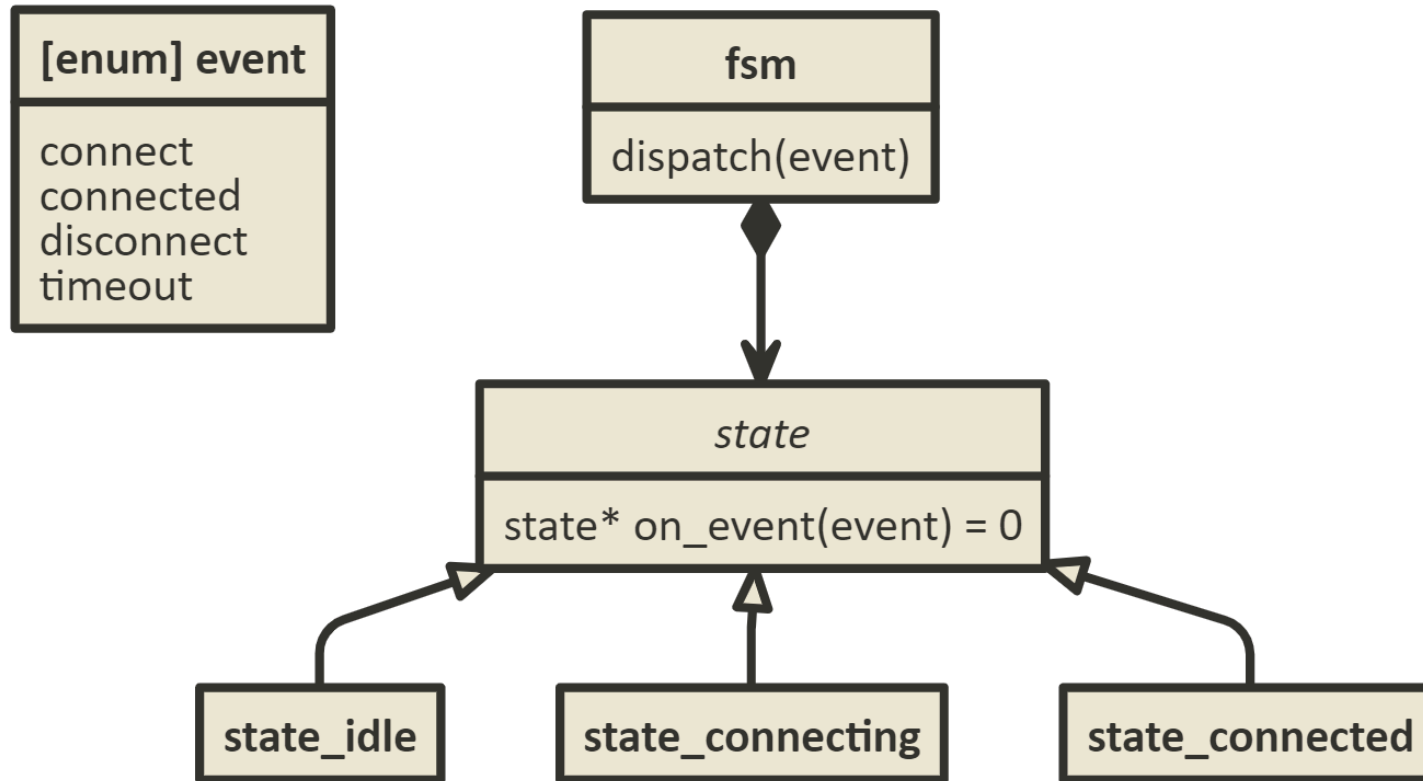




**CASE #1**

# **SINGLE DYNAMIC DISPATCH**

# CLASS DIAGRAM



# SINGLE DYNAMIC DISPATCH

```
template<typename Event>
class state : noncopyable {
public:
    virtual ~state() = default;
    virtual std::unique_ptr<state> on_event(Event) = 0;
};
```

# SINGLE DYNAMIC DISPATCH

```
template<typename Event>
class state : noncopyable {
public:
    virtual ~state() = default;
    virtual std::unique_ptr<state> on_event(Event) = 0;
};
```

```
template<typename Event>
class fsm {
    std::unique_ptr<state<Event>> state_;
public:
    explicit fsm(std::unique_ptr<state<Event>> state) : state_(std::move(state)) {}
    void dispatch(Event e)
    {
        auto new_state = state_->on_event(e);
        if (new_state)
            state_ = std::move(new_state);
    }
};
```



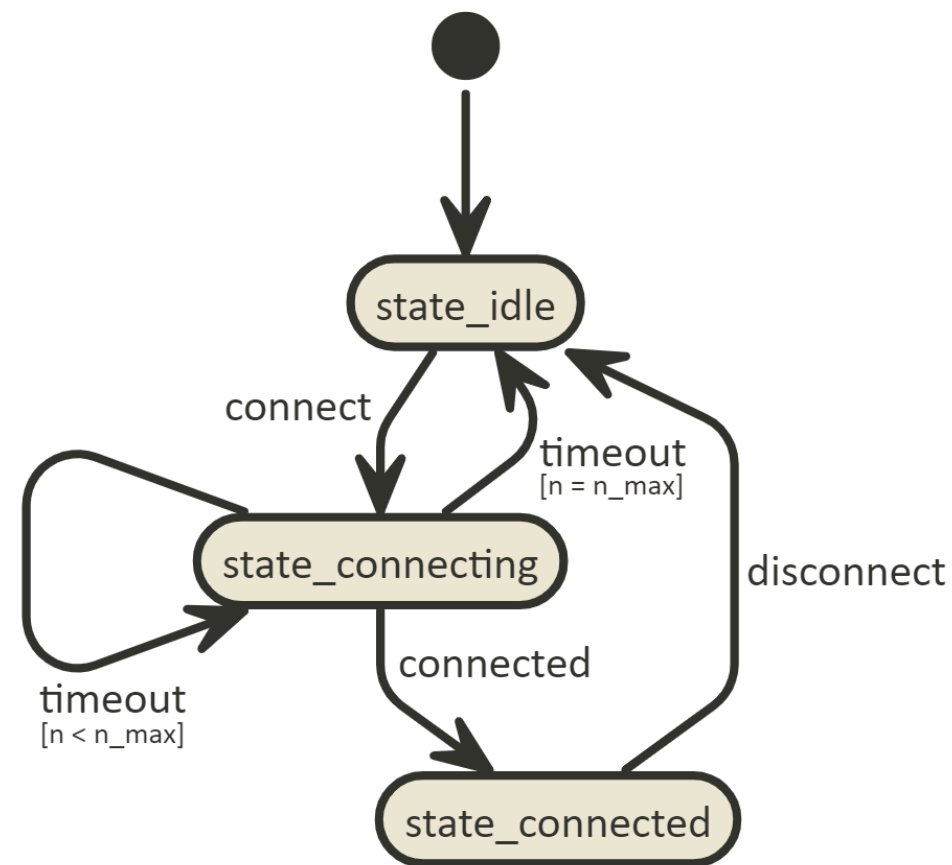
# SINGLE DYNAMIC DISPATCH

```
template<typename Event>
class state : noncopyable {
public:
    virtual ~state() = default;
    virtual std::unique_ptr<state> on_event(Event) = 0;
};
```

```
template<typename Event>
class fsm {
    std::unique_ptr<state<Event>> state_;
public:
    explicit fsm(std::unique_ptr<state<Event>> state) : state_(std::move(state)) {}
    void dispatch(Event e)
    {
        auto new_state = state_->on_event(e);
        if (new_state)
            state_ = std::move(new_state);
    }
};
```

# CONNECTION FSM

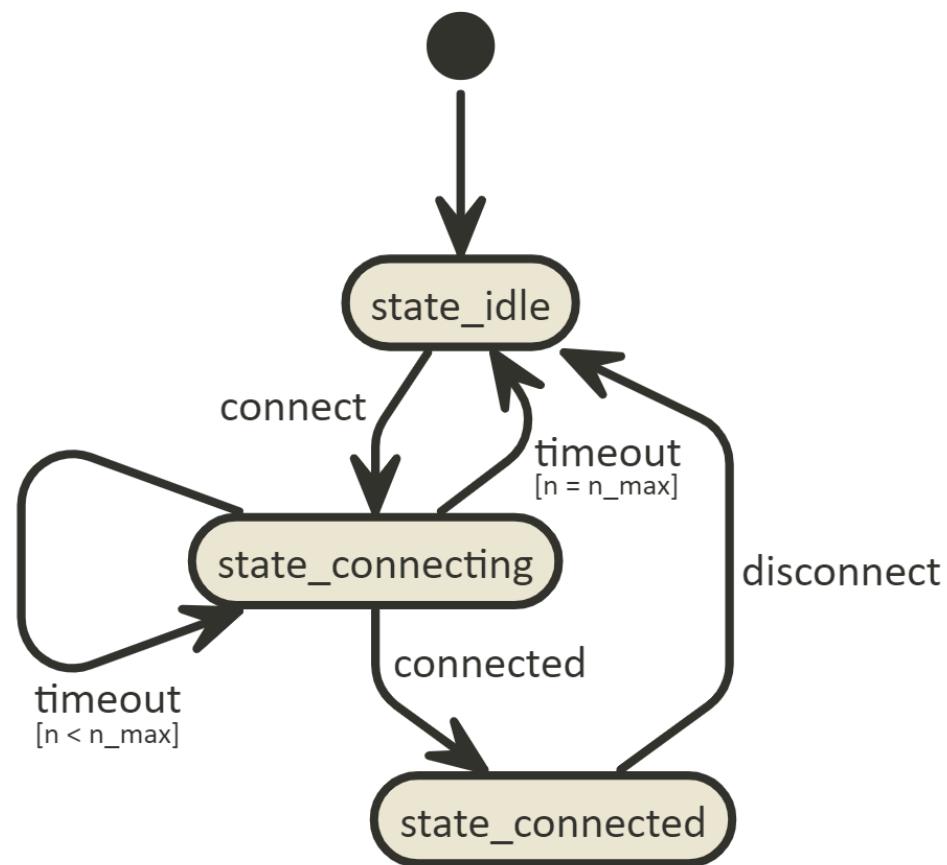
```
class connection_fsm : public fsm<event> {  
public:  
    connection_fsm():  
        fsm<event>(std::make_unique<state_idle>()) {}  
};
```



# CONNECTION FSM

```
class connection_fsm : public fsm<event> {  
public:  
    connection_fsm():  
        fsm<event>(std::make_unique<state_idle>()) {}  
};
```

```
using s = state<event>;
```

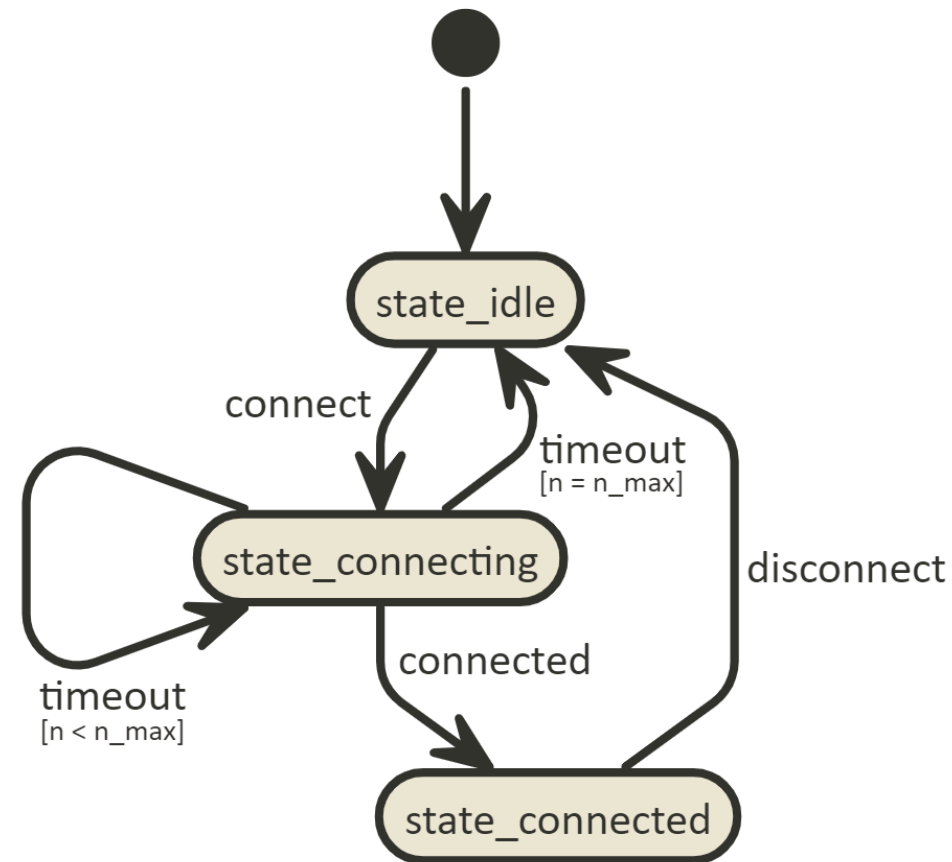


# CONNECTION FSM

```
class connection_fsm : public fsm<event> {  
public:  
    connection_fsm():  
        fsm<event>(std::make_unique<state_idle>()) {}  
};
```

```
using s = state<event>;
```

```
class state_idle final : public s {  
public:  
    std::unique_ptr<s> on_event(event e) override;  
};  
  
class state_connecting final : public s {  
    static constexpr int n_max = 3;  
    int n = 0;  
public:  
    std::unique_ptr<s> on_event(event e) override;  
};  
  
class state_connected final : public s {  
public:  
    std::unique_ptr<s> on_event(event e) override;  
};
```



# TRANSITIONS

```
std::unique_ptr<s> state_idle::on_event(event e)
{

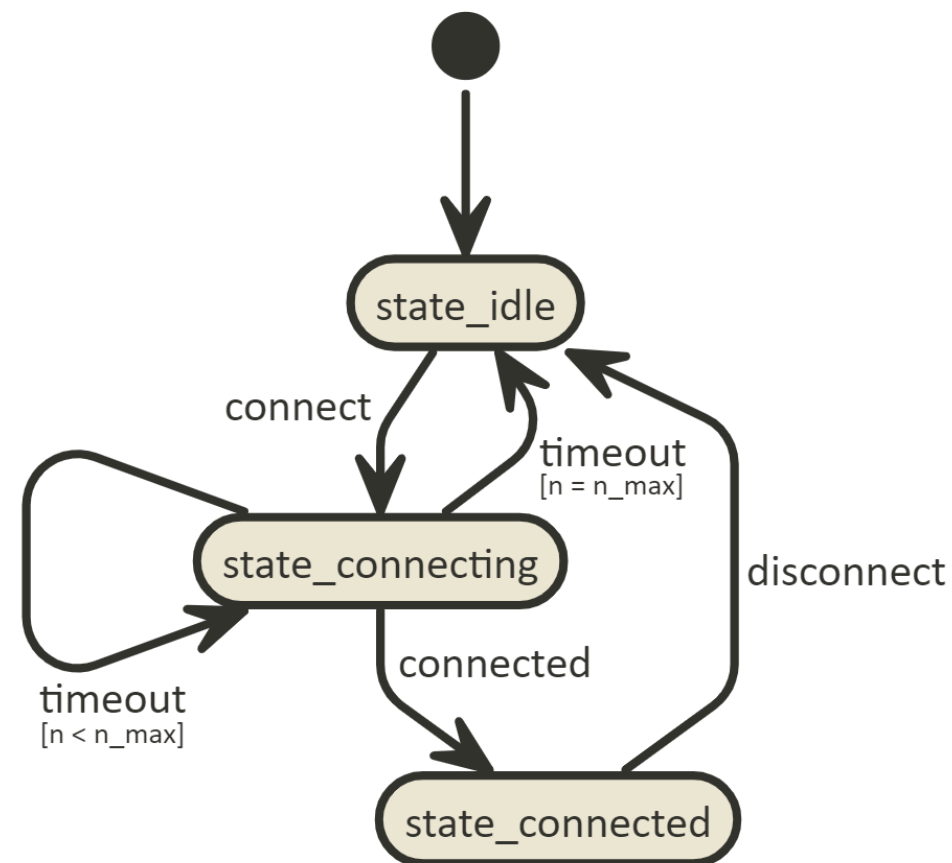
}

std::unique_ptr<s> state_connecting::on_event(event e)
{

}

std::unique_ptr<s> state_connected::on_event(event e)
{

}
```

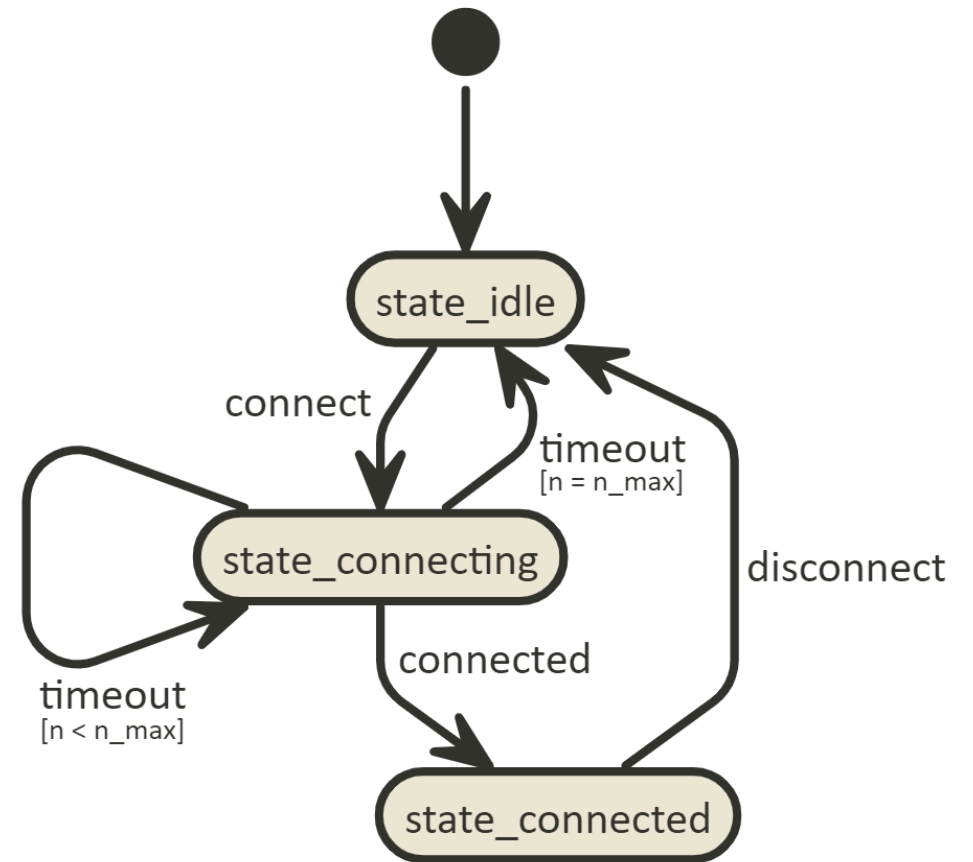


# TRANSITIONS

```
std::unique_ptr<s> state_idle::on_event(event e)
{
    if(e == event::connect)
        return std::make_unique<state_connecting>();
    return nullptr;
}

std::unique_ptr<s> state_connecting::on_event(event e)
{
}

std::unique_ptr<s> state_connected::on_event(event e)
{
}
```

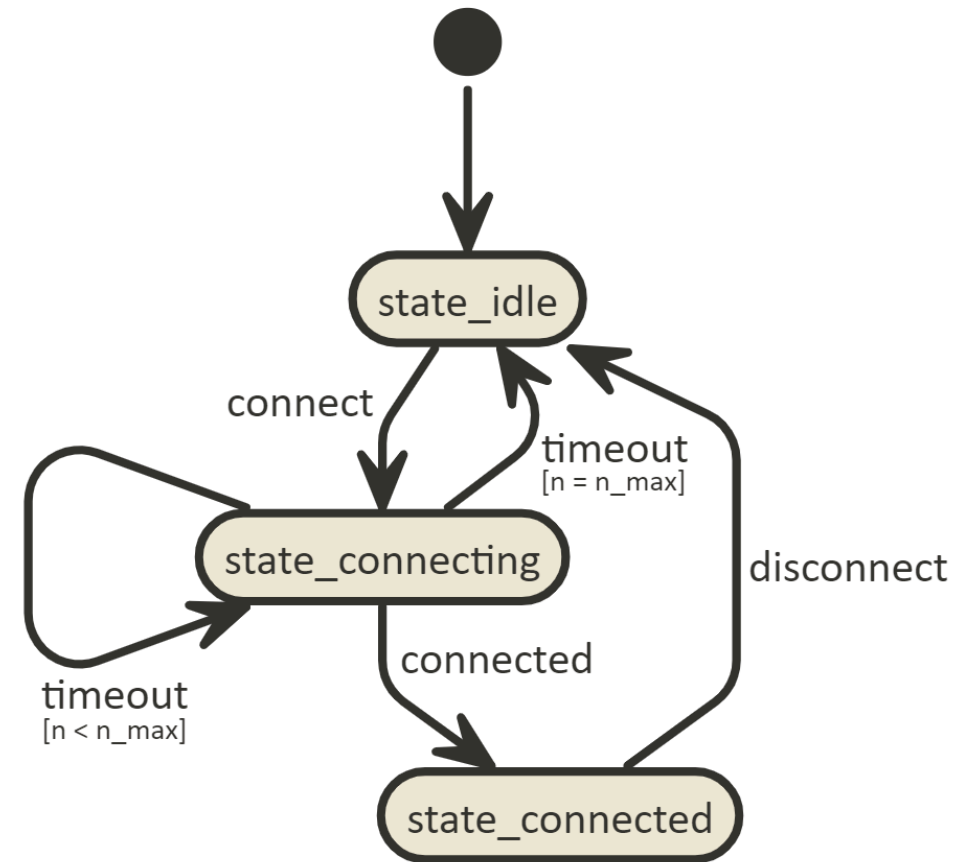


# TRANSITIONS

```
std::unique_ptr<s> state_idle::on_event(event e)
{
    if(e == event::connect)
        return std::make_unique<state_connecting>();
    return nullptr;
}

std::unique_ptr<s> state_connecting::on_event(event e)
{
    switch(e) {
        case event::connected:
            return std::make_unique<state_connected>();
        case event::timeout:
            return ++n < n_max ?
                nullptr : std::make_unique<state_idle>();
        default:
            return nullptr;
    }
}

std::unique_ptr<s> state_connected::on_event(event e)
{
}
```

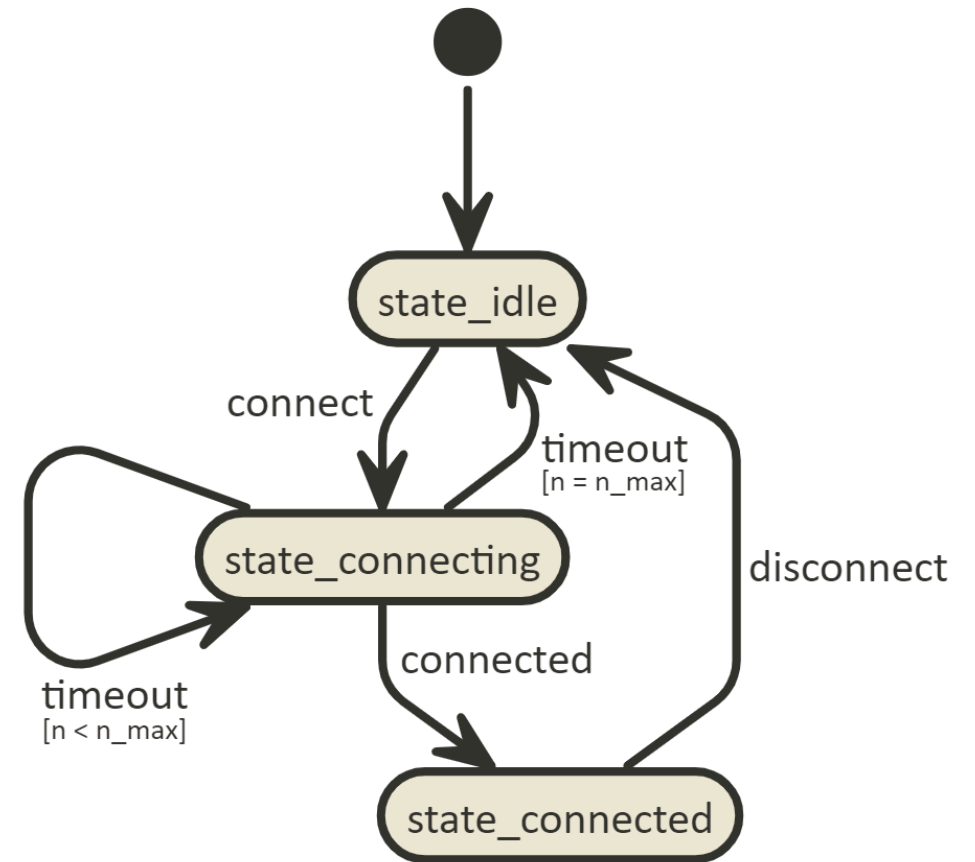


# TRANSITIONS

```
std::unique_ptr<s> state_idle::on_event(event e)
{
    if(e == event::connect)
        return std::make_unique<state_connecting>();
    return nullptr;
}

std::unique_ptr<s> state_connecting::on_event(event e)
{
    switch(e) {
        case event::connected:
            return std::make_unique<state_connected>();
        case event::timeout:
            return ++n < n_max ?
                nullptr : std::make_unique<state_idle>();
        default:
            return nullptr;
    }
}

std::unique_ptr<s> state_connected::on_event(event e)
{
    if(e == event::disconnect)
        return std::make_unique<state_idle>();
    return nullptr;
}
```



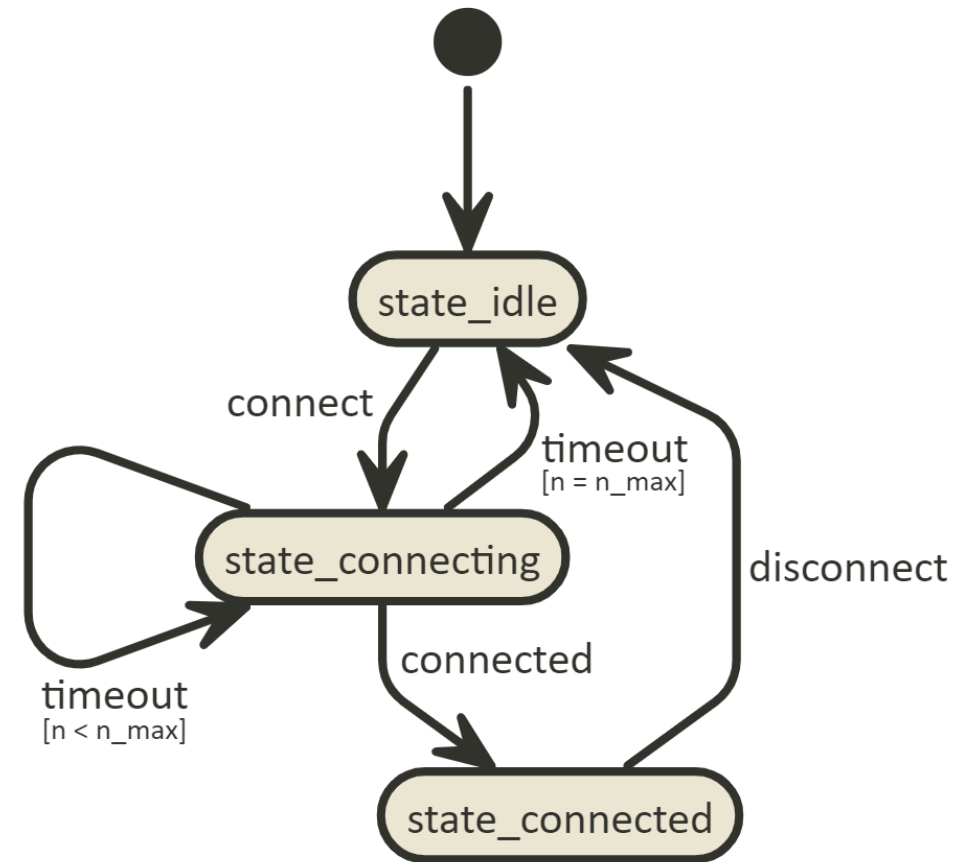


# THE SLOW PART

```
std::unique_ptr<s> state_idle::on_event(event e)
{
    if(e == event::connect)
        return std::make_unique<state_connecting>();
    return nullptr;
}

std::unique_ptr<s> state_connecting::on_event(event e)
{
    switch(e) {
        case event::connected:
            return std::make_unique<state_connected>();
        case event::timeout:
            return ++n < n_max ?
                nullptr : std::make_unique<state_idle>();
        default:
            return nullptr;
    }
}

std::unique_ptr<s> state_connected::on_event(event e)
{
    if(e == event::disconnect)
        return std::make_unique<state_idle>();
    return nullptr;
}
```



# TESTING TRANSITIONS

- Fold expressions come handy ;-)

```
template<typename Fsm, typename... Events>
void dispatch(Fsm& fsm, Events... events)
{
    (fsm.dispatch(events), ...);
}
```

# TESTING TRANSITIONS

- Fold expressions come handy ;-)

```
template<typename Fsm, typename... Events>
void dispatch(Fsm& fsm, Events... events)
{
    (fsm.dispatch(events), ...);
}
```

- Simple message flow

```
connection_fsm fsm;
dispatch(fsm, event::connect, event::timeout, event::connected, event::disconnect);
```

# SINGLE DYNAMIC DISPATCH

---

- Open to new alternatives
  - new derived types may be added by clients at any point of time (long after base class implementation is finished)

# SINGLE DYNAMIC DISPATCH

---

- **Open to new alternatives**
  - new derived types may be added by clients at any point of time (long after base class implementation is finished)
- **Closed to new operations**
  - clients cannot add new operations to dynamic dispatch

# SINGLE DYNAMIC DISPATCH

---

- Open to new alternatives
  - new derived types may be added by clients at any point of time (long after base class implementation is finished)
- Closed to new operations
  - clients cannot add new operations to dynamic dispatch
- Multi-level
  - many levels of inheritance possible

# SINGLE DYNAMIC DISPATCH

---

- **Open to new alternatives**
  - new derived types may be added by clients at any point of time (long after base class implementation is finished)
- **Closed to new operations**
  - clients cannot add new operations to dynamic dispatch
- **Multi-level**
  - many levels of inheritance possible
- **Object Oriented**
  - whole framework is based on objects



**CASE #2**

# **DOUBLE DYNAMIC DISPATCH**



# WHAT IF WE WANT OUR EVENTS TO PASS DATA?

```
class event : noncopyable {
public:
    virtual ~event() = default;
};

class event_connect final : public event {
    std::string_view address_;
public:
    explicit event_connect(std::string_view address): address_(address) {}
    std::string_view address() const { return address_; }
};
```

# WHAT IF WE WANT OUR EVENTS TO PASS DATA?

```
class event : noncopyable {
public:
    virtual ~event() = default;
};

class event_connect final : public event {
    std::string_view address_;
public:
    explicit event_connect(std::string_view address): address_(address) {}
    std::string_view address() const { return address_; }
};
```

```
std::unique_ptr<state> state_idle::on_event(const event& e)
{
    if(auto ptr = dynamic_cast<const event_connect*>(&e)) { /* ... */ }
    else { /* ... */ }
}
```

# WHAT IF WE WANT OUR EVENTS TO PASS DATA?

```
class event : noncopyable {
public:
    virtual ~event() = default;
};

class event_connect final : public event {
    std::string_view address_;
public:
    explicit event_connect(std::string_view address): address_(address) {}
    std::string_view address() const { return address_; }
};
```

```
std::unique_ptr<state> state_idle::on_event(const event& e)
{
    if(auto ptr = dynamic_cast<const event_connect*>(&e)) { /* ... */ }
    else { /* ... */ }
}
```

A really bad idea :-)

# DOUBLE DISPATCH (AKA VISITOR PATTERN)

Special form of multiple dispatch, and a mechanism that dispatches a function call to different concrete functions depending on the runtime types of two objects involved in the call

-- *Wikipedia*

# DOUBLE DISPATCH (AKA VISITOR PATTERN)

```
template<typename State>
struct event : private noncopyable {
    virtual ~event() = default;
    virtual std::unique_ptr<State> dispatch(State& s) const = 0;
};
```

# DOUBLE DISPATCH (AKA VISITOR PATTERN)

```
template<typename State>
struct event : private noncopyable {
    virtual ~event() = default;
    virtual std::unique_ptr<State> dispatch(State& s) const = 0;
};
```

```
template<typename State, typename Event>
class fsm {
    std::unique_ptr<State> state_;
public:
    explicit fsm(std::unique_ptr<State> state) : state_(std::move(state)) {}

    void dispatch(const Event& e)
    {
        auto new_state = e.dispatch(*state_);
        if (new_state)
            state_ = std::move(new_state);
    }
};
```



# DOUBLE DISPATCH (AKA VISITOR PATTERN)

```
template<typename State>
struct event : private noncopyable {
    virtual ~event() = default;
    virtual std::unique_ptr<State> dispatch(State& s) const = 0;
};
```

```
struct event_connect final : public event<state> {
    std::unique_ptr<state> dispatch(state& s) const override { return s.on_event(*this); }
    // ...
};

struct event_connected final : public event<state> {
    std::unique_ptr<state> dispatch(state& s) const override { return s.on_event(*this); }
};

struct event_disconnect final : public event<state> {
    std::unique_ptr<state> dispatch(state& s) const override { return s.on_event(*this); }
};

struct event_timeout final : public event<state> {
    std::unique_ptr<state> dispatch(state& s) const override { return s.on_event(*this); }
};
```

# DOUBLE DISPATCH (AKA VISITOR PATTERN)

```
template<typename State>
struct event : private noncopyable {
    virtual ~event() = default;
    virtual std::unique_ptr<State> dispatch(State& s) const = 0;
};
```

```
class state : noncopyable {
public:
    virtual ~state() = default;
    virtual std::unique_ptr<state> on_event(const event_connect&) { return nullptr; }
    virtual std::unique_ptr<state> on_event(const event_connected&) { return nullptr; }
    virtual std::unique_ptr<state> on_event(const event_disconnect&) { return nullptr; }
    virtual std::unique_ptr<state> on_event(const event_timeout&) { return nullptr; }
};
```

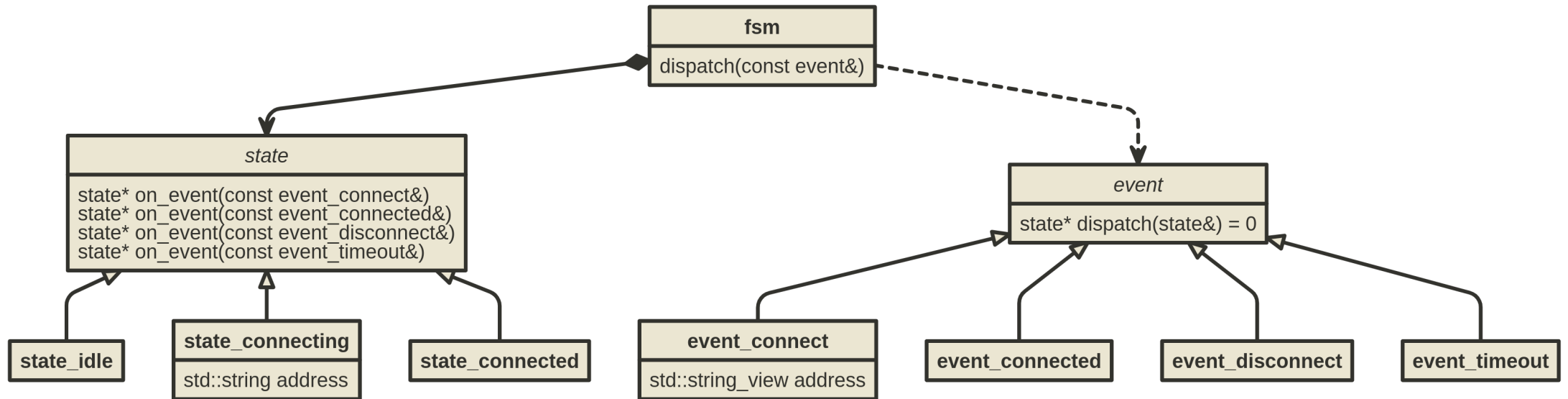
# DOUBLE DISPATCH (AKA VISITOR PATTERN)

```
template<typename State>
struct event : private noncopyable {
    virtual ~event() = default;
    virtual std::unique_ptr<State> dispatch(State& s) const = 0;
};
```

```
class state : noncopyable {
public:
    virtual ~state() = default;
    virtual std::unique_ptr<state> on_event(const event_connect&) { return nullptr; }
    virtual std::unique_ptr<state> on_event(const event_connected&) { return nullptr; }
    virtual std::unique_ptr<state> on_event(const event_disconnect&) { return nullptr; }
    virtual std::unique_ptr<state> on_event(const event_timeout&) { return nullptr; }
};
```

```
class state_idle final : public state {
public:
    using state::on_event;
    std::unique_ptr<state> on_event(const event_connect& e) override
    {
        return std::make_unique<state_connecting>(std::string(e.address()));
    }
};
```

# CLASS DIAGRAM



# CURIOSLY RECURRING TEMPLATE PATTERN (CRTP)

```
template<typename State>
struct event : private noncopyable {
    virtual ~event() = default;
    virtual std::unique_ptr<State> dispatch(State& s) const = 0;
};
```

```
struct event_connect final : public event<state> {
    std::unique_ptr<state> dispatch(state& s) const override { return s.on_event(*this); }
    // ...
};

struct event_connected final : public event<state> {
    std::unique_ptr<state> dispatch(state& s) const override { return s.on_event(*this); }
};

struct event_disconnect final : public event<state> {
    std::unique_ptr<state> dispatch(state& s) const override { return s.on_event(*this); }
};

struct event_timeout final : public event<state> {
    std::unique_ptr<state> dispatch(state& s) const override { return s.on_event(*this); }
};
```

# CURIOSLY RECURRING TEMPLATE PATTERN (CRTP)

```
template<typename State>
struct event : private noncopyable {
    virtual ~event() = default;
    virtual std::unique_ptr<State> dispatch(State& s) const = 0;
};
```

```
template<typename Child>
struct event_crtp : event<state> {
    std::unique_ptr<state> dispatch(state& s) const override { return s.on_event(*static_cast<const Child*>(this)); }
};
```

# CURIOSLY RECURRING TEMPLATE PATTERN (CRTP)

```
template<typename State>
struct event : private noncopyable {
    virtual ~event() = default;
    virtual std::unique_ptr<State> dispatch(State& s) const = 0;
};
```

```
template<typename Child>
struct event_crtp : event<state> {
    std::unique_ptr<state> dispatch(state& s) const override { return s.on_event(*static_cast<const Child*>(this)); }
};
```

```
struct event_connect final : public event_crtp<event_connect> {
    // ...
};

struct event_connected final : public event_crtp<event_connected> {};
struct event_disconnect final : public event_crtp<event_disconnect> {};
struct event_timeout final : public event_crtp<event_timeout> {};
```



# CURIOSLY RECURRING TEMPLATE PATTERN (CRTP)

```
template<typename State>
struct event : private noncopyable {
    virtual ~event() = default;
    virtual std::unique_ptr<State> dispatch(State& s) const = 0;
};
```

```
template<typename Child>
struct event_crtp : event<state> {
    std::unique_ptr<state> dispatch(state& s) const override { return s.on_event(*static_cast<const Child*>(this)); }
};
```

```
struct event_connect final : public event_crtp<event_connect> {
    explicit event_connect(std::string_view address): address_(address) {}
    std::string_view address() const { return address_; }
private:
    std::string_view address_;
};

struct event_connected final : public event_crtp<event_connected> {};
struct event_disconnect final : public event_crtp<event_disconnect> {};
struct event_timeout final : public event_crtp<event_timeout> {};
```

- Hey look ma, now all fits on one slide ;-)

# TRANSITIONS

```
std::unique_ptr<state>
state_idle::on_event(const event_connect& e)
{

}

std::unique_ptr<state>
state_connecting::on_event(const event_connected&)
{

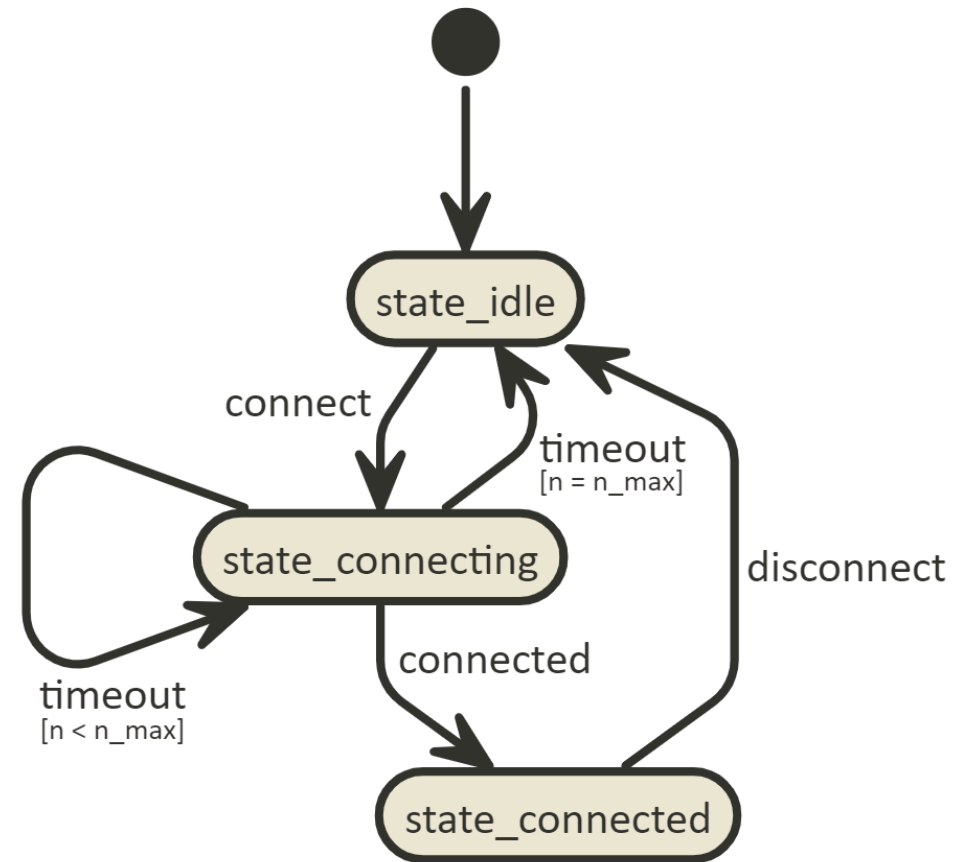
}

std::unique_ptr<state>
state_connecting::on_event(const event_timeout&)
{

}

std::unique_ptr<state>
state_connected::on_event(const event_disconnect&)
{

}
```



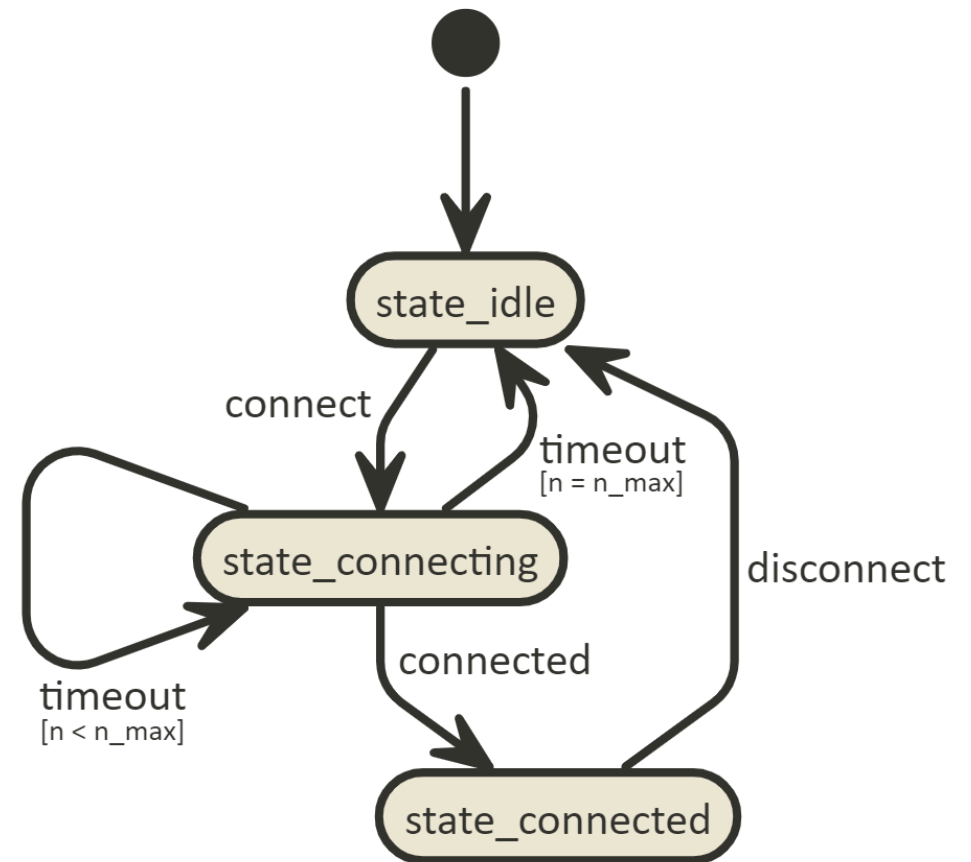
# TRANSITIONS

```
std::unique_ptr<state>
state_idle::on_event(const event_connect& e)
{
    return std::make_unique<state_connecting>(
        std::string{e.address()});
}

std::unique_ptr<state>
state_connecting::on_event(const event_connected&)
{
}

std::unique_ptr<state>
state_connecting::on_event(const event_timeout&)
{
}

std::unique_ptr<state>
state_connected::on_event(const event_disconnect&)
{
}
```



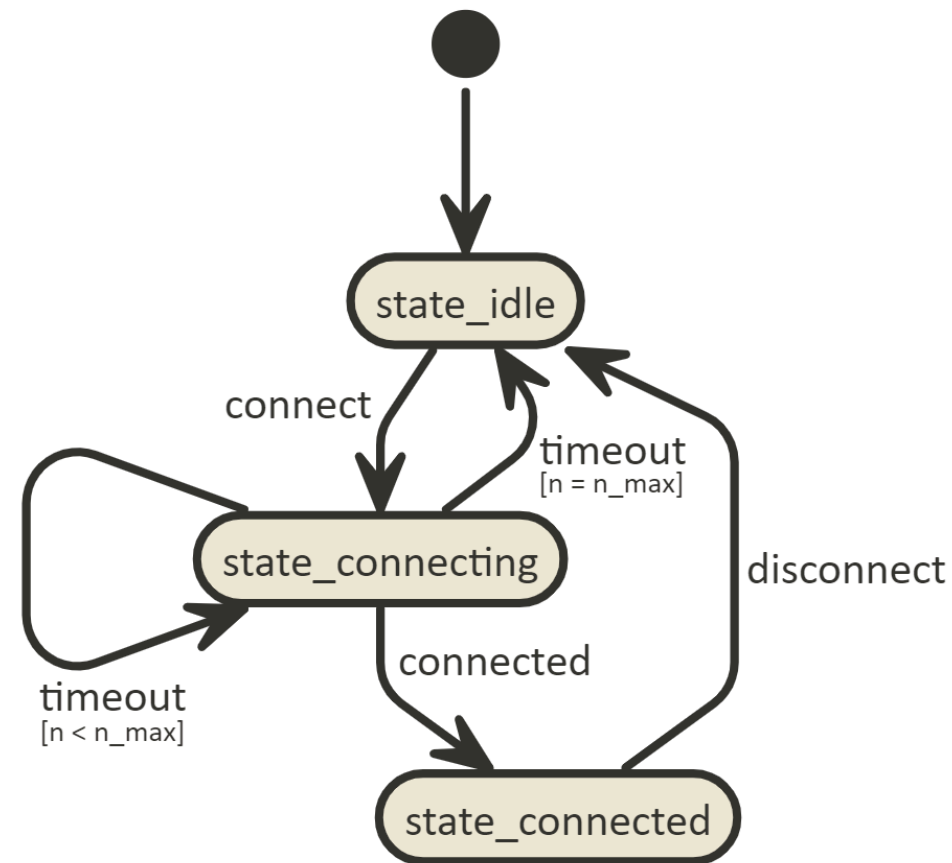
# TRANSITIONS

```
std::unique_ptr<state>
state_idle::on_event(const event_connect& e)
{
    return std::make_unique<state_connecting>(
        std::string{e.address()});
}

std::unique_ptr<state>
state_connecting::on_event(const event_connected&)
{
    return std::make_unique<state_connected>();
}

std::unique_ptr<state>
state_connecting::on_event(const event_timeout&)
{
}

std::unique_ptr<state>
state_connected::on_event(const event_disconnect&)
{
}
```



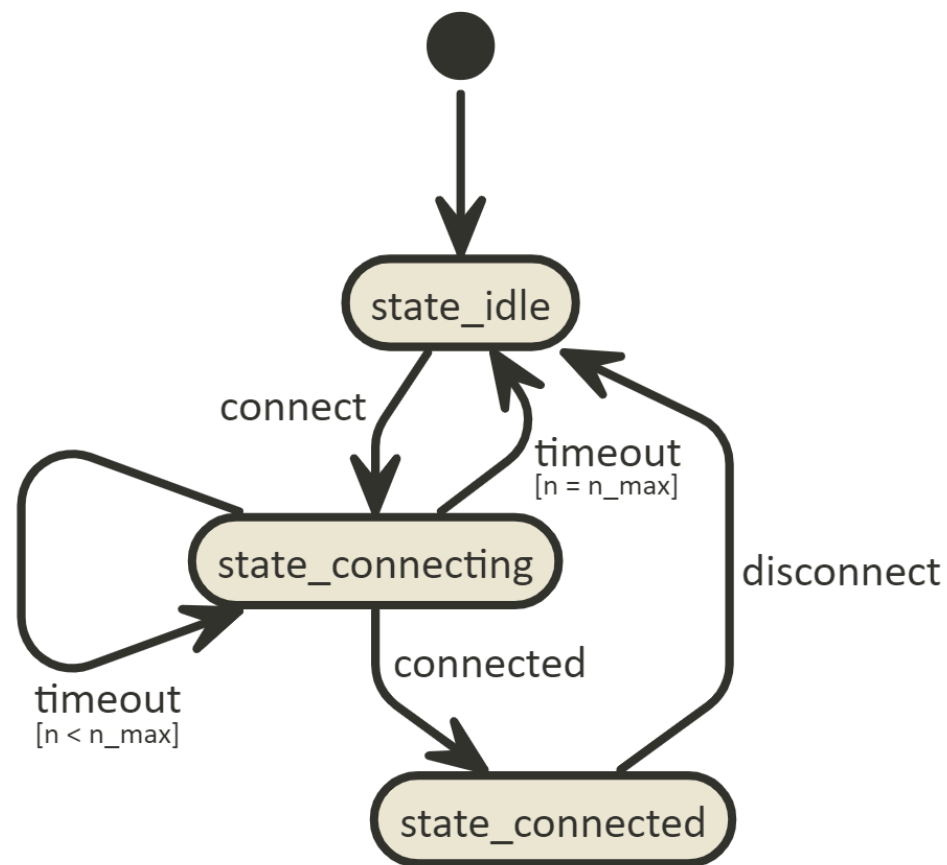
# TRANSITIONS

```
std::unique_ptr<state>
state_idle::on_event(const event_connect& e)
{
    return std::make_unique<state_connecting>(
        std::string{e.address()});
}

std::unique_ptr<state>
state_connecting::on_event(const event_connected&)
{
    return std::make_unique<state_connected>();
}

std::unique_ptr<state>
state_connecting::on_event(const event_timeout&)
{
    return ++n < n_max ?
        nullptr : std::make_unique<state_idle>();
}

std::unique_ptr<state>
state_connected::on_event(const event_disconnect&)
{
}
```



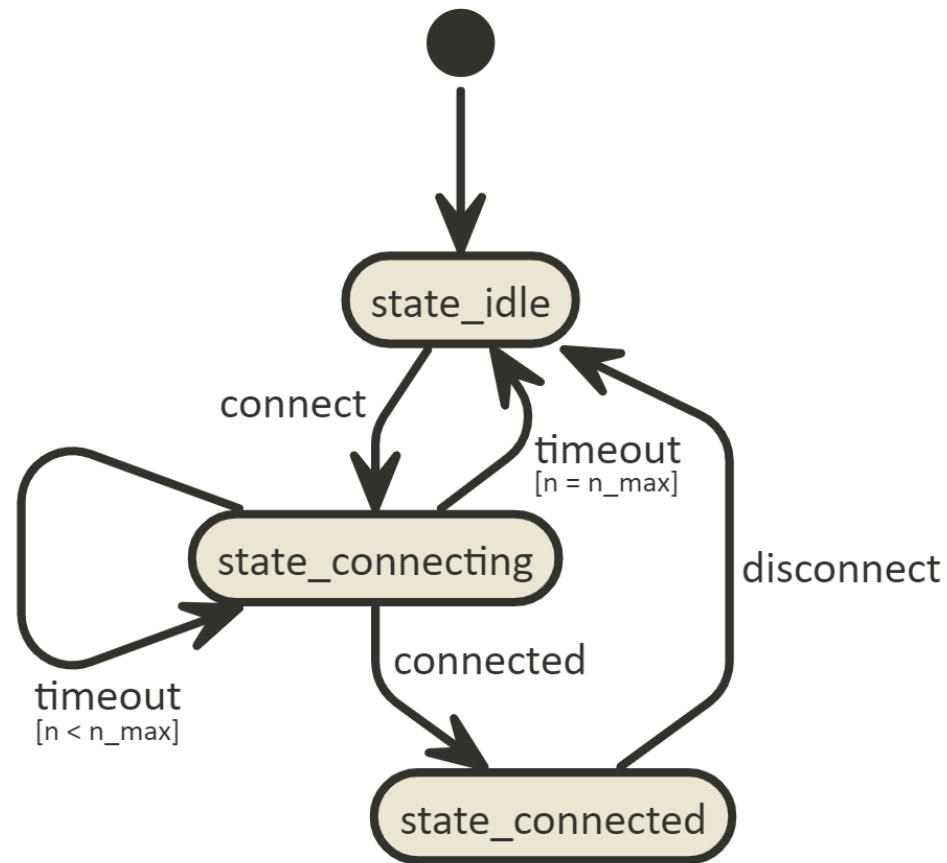
# TRANSITIONS

```
std::unique_ptr<state>
state_idle::on_event(const event_connect& e)
{
    return std::make_unique<state_connecting>(
        std::string{e.address()});
}

std::unique_ptr<state>
state_connecting::on_event(const event_connected&)
{
    return std::make_unique<state_connected>();
}

std::unique_ptr<state>
state_connecting::on_event(const event_timeout&)
{
    return ++n < n_max ?
        nullptr : std::make_unique<state_idle>();
}

std::unique_ptr<state>
state_connected::on_event(const event_disconnect&)
{
    return std::make_unique<state_idle>();
}
```



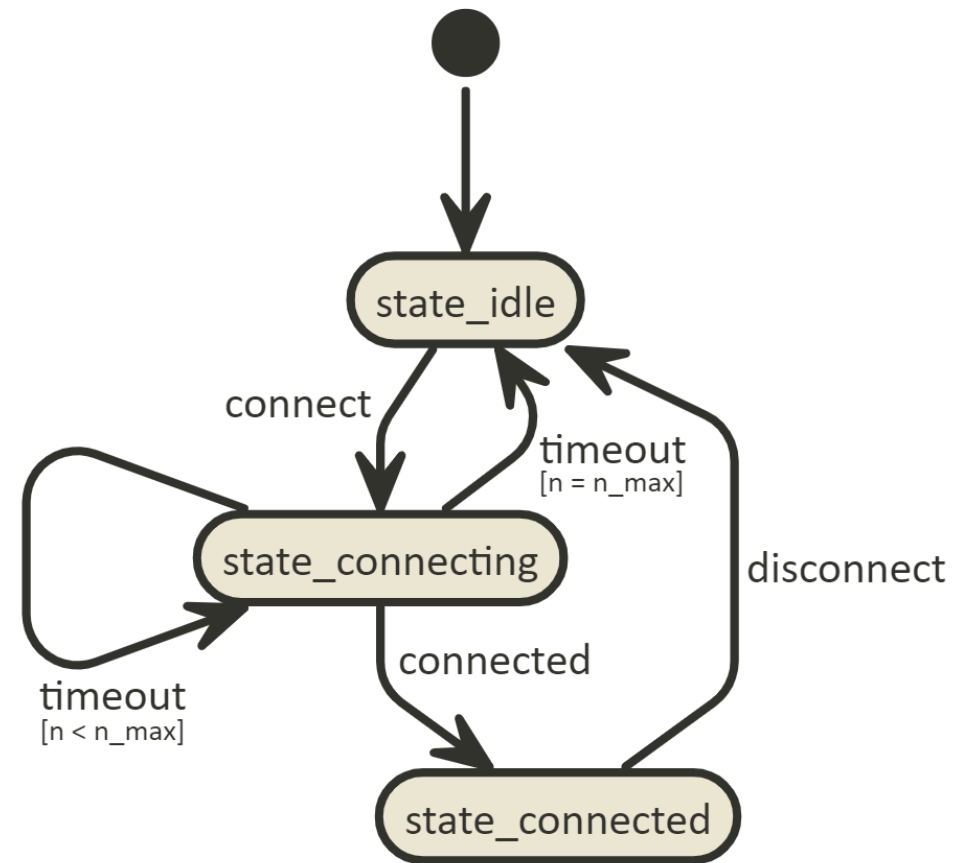
# THE SLOW PART

```
std::unique_ptr<state>
state_idle::on_event(const event_connect& e)
{
    return std::make_unique<state_connecting>(
        std::string{e.address()});
}

std::unique_ptr<state>
state_connecting::on_event(const event_connected&)
{
    return std::make_unique<state_connected>();
}

std::unique_ptr<state>
state_connecting::on_event(const event_timeout&)
{
    return ++n < n_max ?
        nullptr : std::make_unique<state_idle>();
}

std::unique_ptr<state>
state_connected::on_event(const event_disconnect&)
{
    return std::make_unique<state_idle>();
}
```



# TESTING TRANSITIONS

```
template<typename Fsm, typename... Events>
void dispatch(Fsm& fsm, const Events&... events)
{
    (fsm.dispatch(*events), ...);
}
```



# TESTING TRANSITIONS

```
template<typename Fsm, typename... Events>
void dispatch(Fsm& fsm, const Events&... events)
{
    (fsm.dispatch(*events), ...);
}
```

```
dispatch(fsm,
    std::make_unique<event_connect>("train-it.eu"),
    std::make_unique<event_timeout>(),
    std::make_unique<event_connected>(),
    std::make_unique<event_disconnect>());
```

# THE SLOW PART

```
template<typename Fsm, typename... Events>
void dispatch(Fsm& fsm, const Events&... events)
{
    (fsm.dispatch(*events), ...);
}
```

```
dispatch(fsm,
    std::make_unique<event_connect>("train-it.eu"),
    std::make_unique<event_timeout>(),
    std::make_unique<event_connected>(),
    std::make_unique<event_disconnect>());
```

# DOUBLE DYNAMIC DISPATCH (AKA VISITOR PATTERN)

- ~~Open to new alternatives~~
- Closed to new alternatives
  - *one of class hierarchies fixed at design time* and cannot be extended by clients
- *Closed to new operations*
  - clients cannot add new operations to dynamic dispatch
- *Multi-level*
  - many levels of inheritance possible
- *Object Oriented*
  - whole framework is based on objects



std:variant IN 3 MINUTES

# std::variant<Types...>

```
template<class... Types>  
class variant;
```

- Represents a *type-safe union*
- At any given point in time either
  - *holds a value* of one of its alternative types
  - is in special **valueless\_by\_exception** state (reached if an *exception is thrown* during *contained value initialization or assignment*)

# std::variant<Types...> INTERFACE (SIMPLIFICATION)

```
// not a valid C++
template<typename Type1, typename Type2, typename Type3...>
class variant {
    union options {
        Type1 t1;
        Type2 t2;
        Type3 t3;
        ...
    };
    int index_;
public:
    variant();
    template<class T> variant(T&& t);
    template<class T, class... Args> variant(std::in_place_type_t<T>, Args&&... args);

    std::size_t index() const;

    //...
};
```

# std::variant<Types...>

- *Not allowed to allocate dynamic memory*
- *Not permitted* to hold references, arrays, or **void**
- *Empty variants are ill-formed* (**std::variant<std::monostate>** can be used instead)
- Permitted to *hold the same type more than once*, and to hold differently cv-qualified versions of the same type
- *Default-initialized variant* holds a value of its first alternative unless that alternative is not default-constructible

# std::variant<Types...>

```
void print(const std::variant<int, double, X>& v)
{
    switch (v.index()) {
        case 0: std::cout << "int: " << std::get<0>(v) << '\n'; break;
        case 1: std::cout << "double: " << std::get<1>(v) << '\n'; break;
        case 2: std::cout << "X: " << std::get<2>(v).x << ", " << std::get<2>(v).y << '\n'; break;
    }
}
```



# std::variant<Types...>

```
void print(const std::variant<int, double, X>& v)
{
    switch (v.index()) {
        case 0: std::cout << "int: " << std::get<0>(v) << '\n'; break;
        case 1: std::cout << "double: " << std::get<1>(v) << '\n'; break;
        case 2: std::cout << "X: " << std::get<2>(v).x << ", " << std::get<2>(v).y << '\n'; break;
    }
}
```

```
struct visitor {
    void operator()(int v) const { std::cout << "int: " << v << '\n'; }
    void operator()(double v) const { std::cout << "double: " << v << '\n'; }
    void operator()(const X& v) const { std::cout << "X: " << v.x << ", " << v.y << '\n'; }
};

void print(const std::variant<int, double, X>& v) { std::visit(visitor{}, v); }
```



**CASE #3**

# **VARIANT + EXTERNAL TRANSITIONS**

# EVENTS AND STATES

## EVENTS

```
struct event_connect { std::string_view address; };  
struct event_connected {};  
struct event_disconnect {};  
struct event_timeout {};  
using event = std::variant<event_connect, event_connected, event_disconnect, event_timeout>;
```

# EVENTS AND STATES

## EVENTS

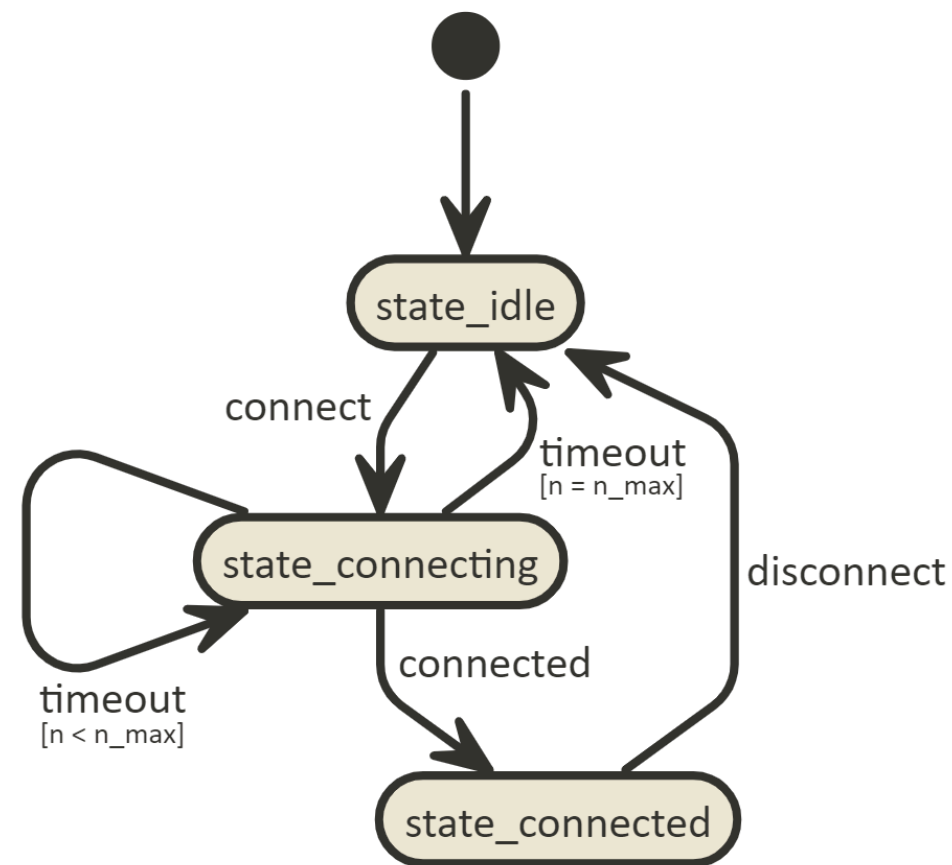
```
struct event_connect { std::string_view address; };
struct event_connected {};
struct event_disconnect {};
struct event_timeout {};
using event = std::variant<event_connect, event_connected, event_disconnect, event_timeout>;
```

## STATES

```
struct state_idle {};
struct state_connecting {
    static constexpr int n_max = 3;
    int n = 0;
    std::string address;
};
struct state_connected {};
using state = std::variant<state_idle, state_connecting, state_connected>;
```

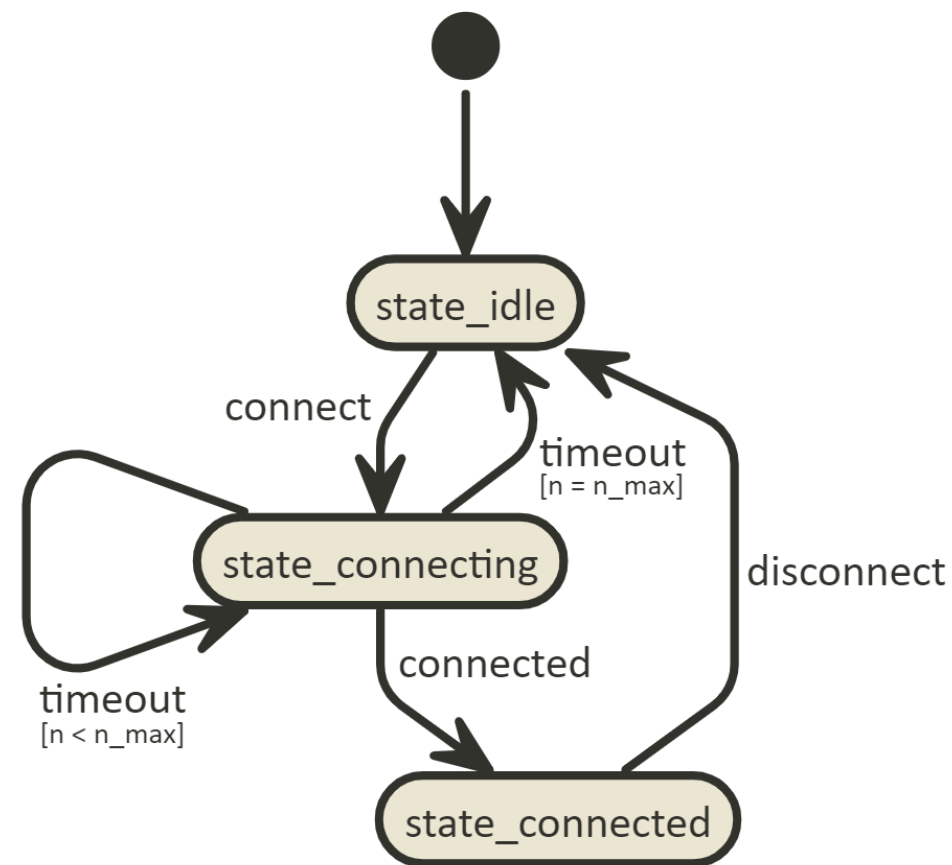
# TRANSITIONS

```
struct transitions {  
    optional<state> operator()(state_idle&,  
                               const event_connect& e)  
    {  
  
    optional<state> operator()(state_connecting&,  
                               const event_connected&)  
    {  
  
    optional<state> operator()(state_connecting& s,  
                               const event_timeout&)  
    {  
  
    }  
  
    optional<state> operator()(state_connected&,  
                               const event_disconnect&)  
    {  
  
    }  
  
};
```



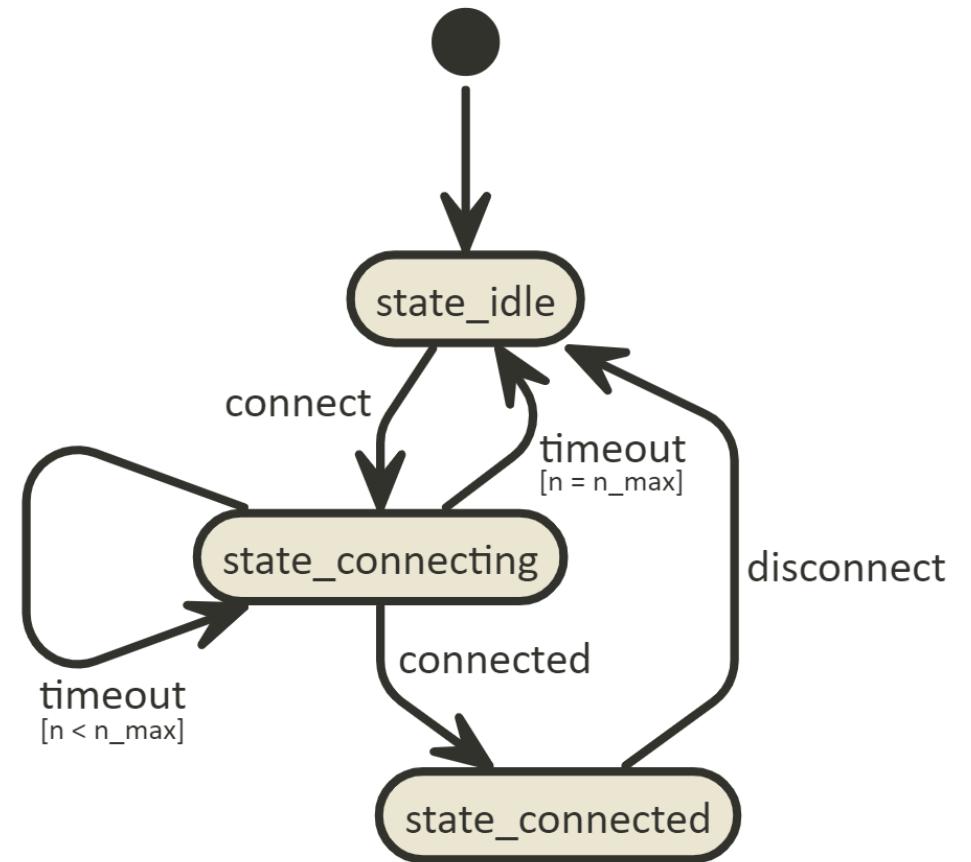
# TRANSITIONS

```
struct transitions {  
    optional<state> operator()(state_idle&,  
                               const event_connect& e)  
    { return state_connecting{std::string(e.address)}; }  
  
    optional<state> operator()(state_connecting&,  
                               const event_connected&)  
    {  
    }  
  
    optional<state> operator()(state_connecting& s,  
                               const event_timeout&)  
    {  
    }  
  
    optional<state> operator()(state_connected&,  
                               const event_disconnect&)  
    {  
    }  
};
```



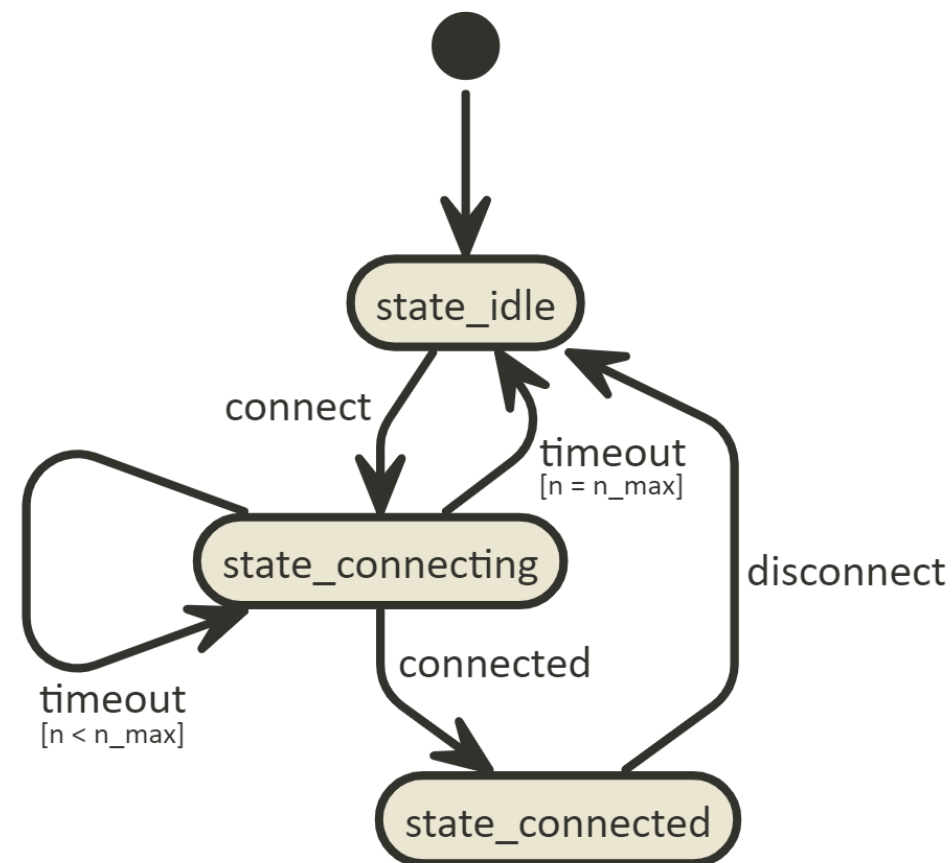
# TRANSITIONS

```
struct transitions {  
    optional<state> operator()(state_idle&,  
                               const event_connect& e)  
    { return state_connecting{std::string(e.address)}; }  
  
    optional<state> operator()(state_connecting&,  
                               const event_connected&)  
    { return state_connected{}; }  
  
    optional<state> operator()(state_connecting& s,  
                               const event_timeout&)  
    {  
  
    }  
  
    optional<state> operator()(state_connected&,  
                               const event_disconnect&)  
    {  
  
    }  
  
};
```



# TRANSITIONS

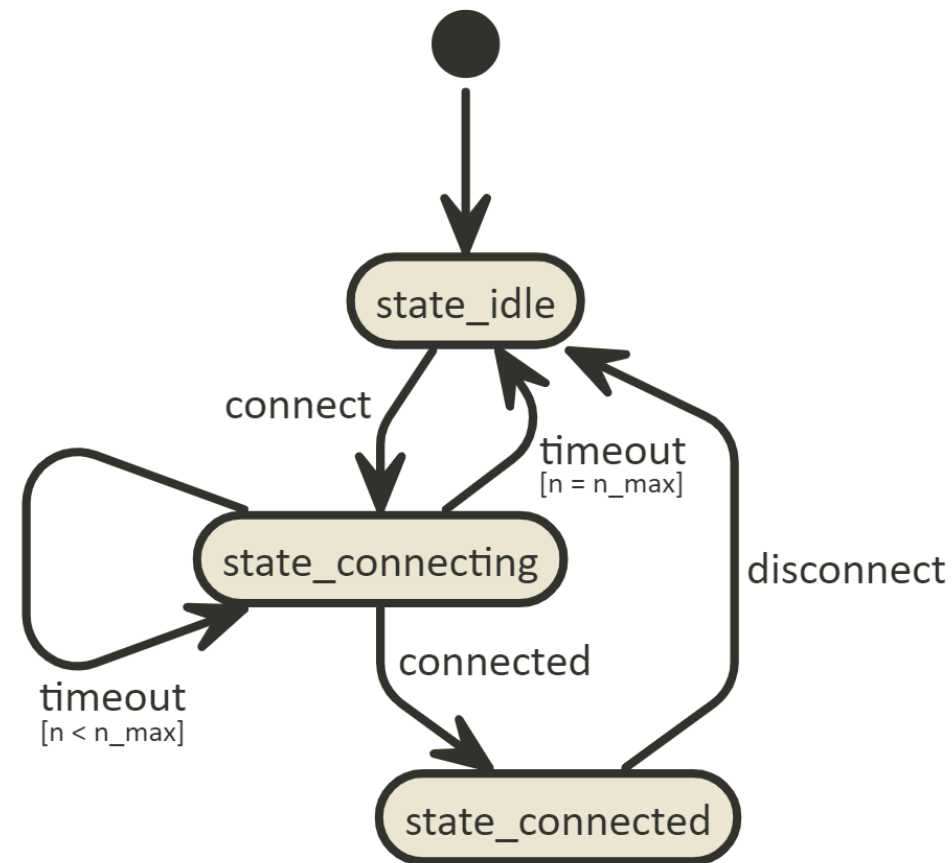
```
struct transitions {  
    optional<state> operator()(state_idle&,  
                               const event_connect& e)  
    { return state_connecting{std::string(e.address)}; }  
  
    optional<state> operator()(state_connecting&,  
                               const event_connected&)  
    { return state_connected{}; }  
  
    optional<state> operator()(state_connecting& s,  
                               const event_timeout&)  
    {  
        return ++s.n < state_connecting::n_max ?  
            std::nullopt : optional<state>(state_idle{});  
    }  
  
    optional<state> operator()(state_connected&,  
                               const event_disconnect&)  
    {  
    }  
};
```





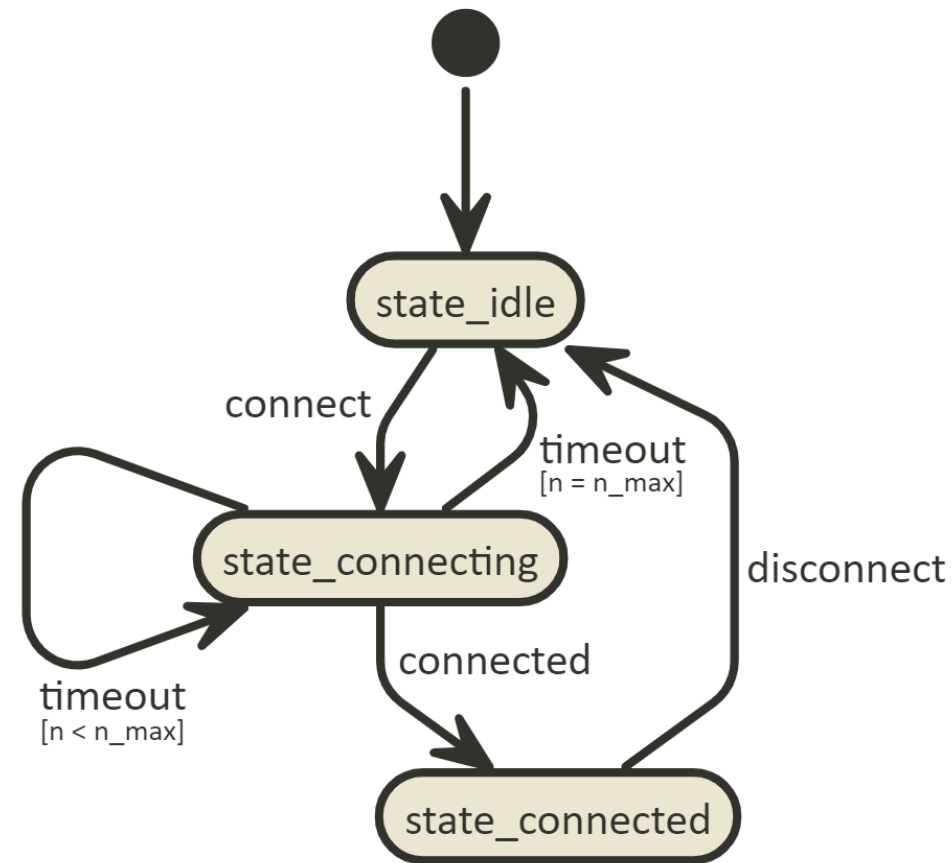
# TRANSITIONS

```
struct transitions {  
    optional<state> operator()(state_idle&,  
                               const event_connect& e)  
    { return state_connecting{std::string(e.address)}; }  
  
    optional<state> operator()(state_connecting&,  
                               const event_connected&)  
    { return state_connected{}; }  
  
    optional<state> operator()(state_connecting& s,  
                               const event_timeout&)  
    {  
        return ++s.n < state_connecting::n_max ?  
            std::nullopt : optional<state>(state_idle{});  
    }  
  
    optional<state> operator()(state_connected&,  
                               const event_disconnect&)  
    { return state_idle{}; }  
  
};
```



# TRANSITIONS

```
struct transitions {  
    optional<state> operator()(state_idle&,  
                               const event_connect& e)  
    { return state_connecting{std::string(e.address)}; }  
  
    optional<state> operator()(state_connecting&,  
                               const event_connected&)  
    { return state_connected{}; }  
  
    optional<state> operator()(state_connecting& s,  
                               const event_timeout&)  
    {  
        return ++s.n < state_connecting::n_max ?  
            std::nullopt : optional<state>(state_idle{});  
    }  
  
    optional<state> operator()(state_connected&,  
                               const event_disconnect&)  
    { return state_idle{}; }  
  
    template<typename State, typename Event>  
    optional<state> operator()(State&,  
                               const Event&) const  
    { return std::nullopt; }  
};
```



# FSM ENGINE

```
template<typename StateVariant, typename EventVariant, typename Transitions>
class fsm {
    StateVariant state_;
public:
    void dispatch(const EventVariant& event)
    {
        std::optional<StateVariant> new_state = std::visit(Transitions{}, state_, event);
        if(new_state)
            state_ = *std::move(new_state);
    }
};
```

```
using connection_fsm = fsm<state, event, transitions>;
```

# FSM ENGINE

```
template<typename StateVariant, typename EventVariant, typename Transitions>
class fsm {
    StateVariant state_;
public:
    void dispatch(const EventVariant& event)
    {
        std::optional<StateVariant> new_state = std::visit(Transitions{}, state_, event);
        if(new_state)
            state_ = *std::move(new_state);
    }
};
```

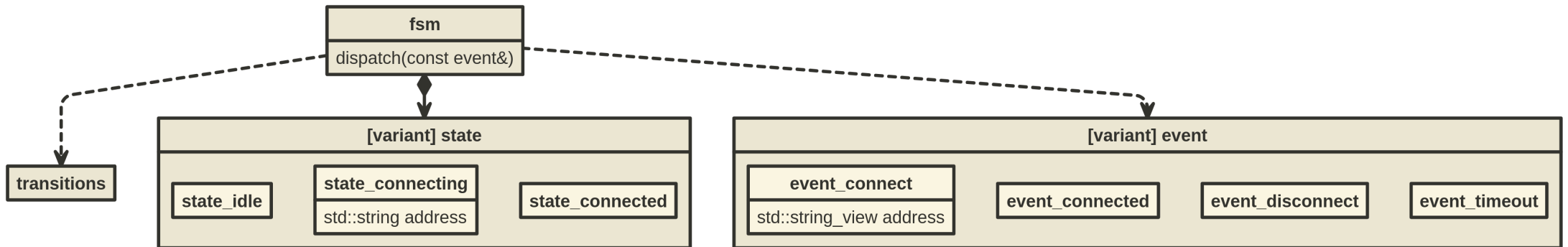
```
using connection_fsm = fsm<state, event, transitions>;
```

# FSM ENGINE

```
template<typename StateVariant, typename EventVariant, typename Transitions>
class fsm {
    StateVariant state_;
public:
    void dispatch(const EventVariant& event)
    {
        std::optional<StateVariant> new_state = std::visit(Transitions{}, state_, event);
        if(new_state)
            state_ = *std::move(new_state);
    }
};
```

```
using connection_fsm = fsm<state, event, transitions>;
```

# CLASS DIAGRAM



# TESTING TRANSITIONS

```
template<typename Fsm, typename... Events>
void dispatch(Fsm& fsm, Events&&... events)
{
    (fsm.dispatch(std::forward<Events>(events)), ...);
}
```

```
dispatch(fsm,
    event_connect{"train-it.eu"},
    event_timeout{},
    event_connected{},
    event_disconnect{});
```



**CASE #4**

# **VARIANT + TRANSITIONS IN FSM**



# FSM ENGINE

```
template<typename Derived, typename StateVariant, typename EventVariant>
class fsm {
    StateVariant state_;
public:
    void dispatch(const EventVariant& event)
    {
        auto new_state = std::visit(
            state_, event);
        if(new_state)
            state_ = *std::move(new_state);
    }
};
```

# FSM ENGINE

```
template<typename Derived, typename StateVariant, typename EventVariant>
class fsm {
    StateVariant state_;
public:
    void dispatch(const EventVariant& event)
    {
        Derived& child = static_cast<Derived&>(*this);
        auto new_state = std::visit(
            [&](auto& s, const auto& e) -> std::optional<StateVariant>
            { return child.on_event(s, e); },
            state_, event);
        if(new_state)
            state_ = *std::move(new_state);
    }
};
```

- CRTP again ;-)

# FSM ENGINE

```
template<typename Derived, typename StateVariant>
class fsm {
    StateVariant state_;
public:
    template<typename Event>
    void dispatch(Event&& event)
    {
        Derived& child = static_cast<Derived&>(*this);
        auto new_state = std::visit(
            [&](auto& s) -> std::optional<StateVariant>
            { return child.on_event(s, std::forward<Event>(event)); },
            state_);
        if(new_state)
            state_ = *std::move(new_state);
    }
};
```

- Variant of events not needed anymore

# TRANSITIONS DEFINED BY THE FSM ITSELF

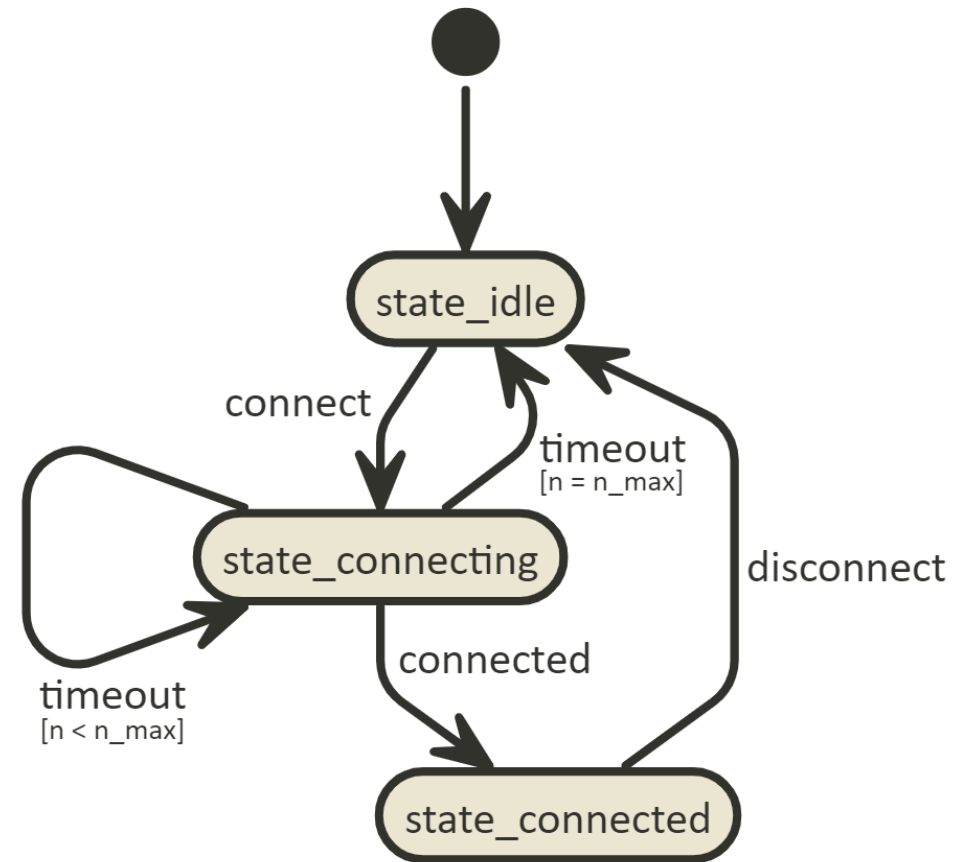
```
class connection_fsm
: public fsm<connection_fsm, state> {
public:
    auto on_event(state_idle&,
                  const event_connect& e)
    {
    }

    auto on_event(state_connecting&,
                  const event_connected&)
    {
    }

    auto on_event(state_connecting& s,
                  const event_timeout&)
    {
    }

    auto on_event(state_connected&,
                  const event_disconnect&)
    {
    }

    template<typename State, typename Event>
    auto on_event(State&, const Event&)
    {
    }
};
```



# TRANSITIONS DEFINED BY THE FSM ITSELF

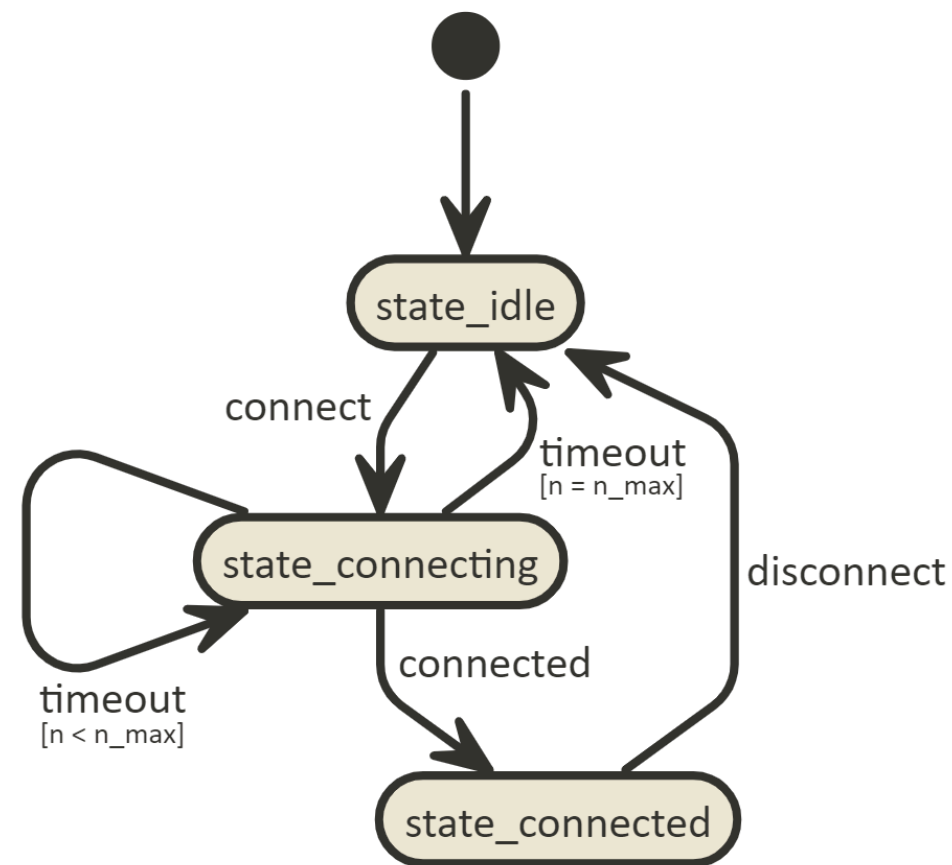
```
class connection_fsm
: public fsm<connection_fsm, state> {
public:
    auto on_event(state_idle&,
                  const event_connect& e)
    { return state_connecting{std::string(e.address)}; }

    auto on_event(state_connecting&,
                  const event_connected&)
    { return state_connected{}; }

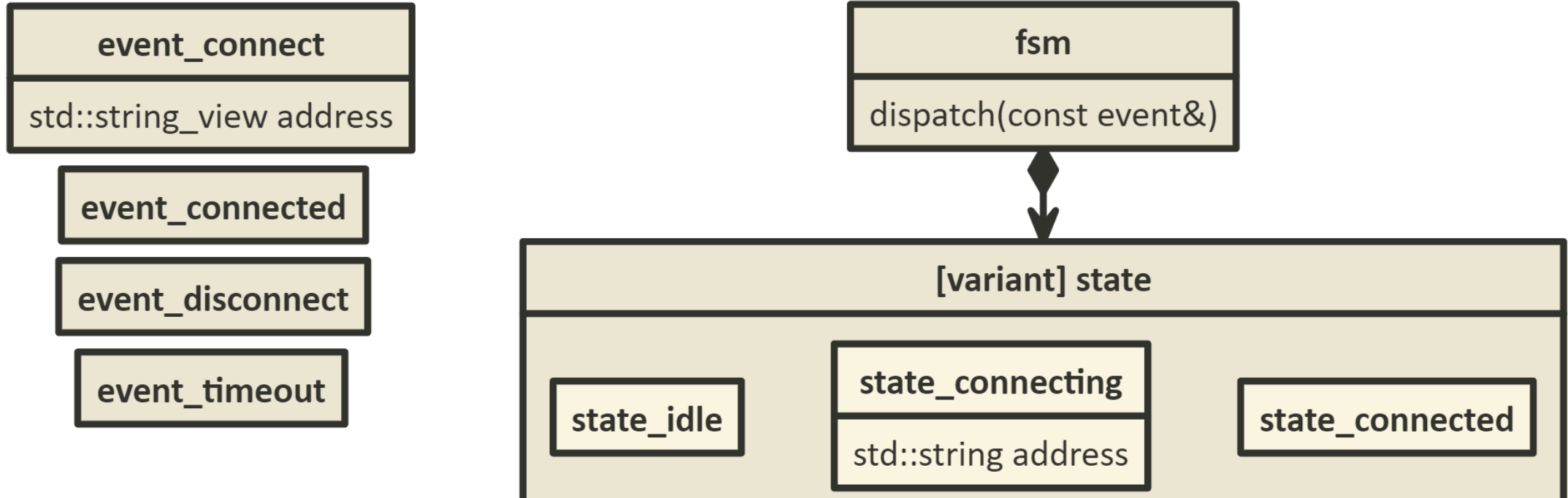
    auto on_event(state_connecting& s,
                  const event_timeout&)
    {
        return ++s.n < state_connecting::n_max ?
            std::nullopt : std::optional<state>(state_idle{});
    }

    auto on_event(state_connected&,
                  const event_disconnect&)
    { return state_idle{}; }

    template<typename State, typename Event>
    auto on_event(State&, const Event&)
    { return std::nullopt; }
};
```



# CLASS DIAGRAM





**CASE #5**

# **VARIANT + TRANSITIONS IN STATES**

# FSM ENGINE

```
template<typename StateVariant, typename OnInvalidTransition>
class fsm {
    StateVariant state_;
public:
    template<typename Event>
    void dispatch(Event&& event)
    {
        auto new_state = std::visit(overloaded{

            }, state_);
        if(new_state)
            state_ = *std::move(new_state);
    }
};
```

- Beware, lambdas are coming :-)



# FSM ENGINE

```
template<typename StateVariant, typename OnInvalidTransition>
class fsm {
    StateVariant state_;
public:
    template<typename Event>
    void dispatch(Event&& event)
    {
        auto new_state = std::visit(overloaded{
            [&](auto& s, decltype(s.on_event(event))* = nullptr) -> std::optional<StateVariant>
            { return s.on_event(std::forward<Event>(event)); },

            }, state_);
        if(new_state)
            state_ = *std::move(new_state);
    }
};
```

- Thanks to SFINAE *matches defined transitions only*

# FSM ENGINE

```
template<typename StateVariant, typename OnInvalidTransition>
class fsm {
    StateVariant state_;
public:
    template<typename Event>
    void dispatch(Event&& event)
    {
        auto new_state = std::visit(overloaded{
            [&](auto& s, decltype(s.on_event(event))* = nullptr) -> std::optional<StateVariant>
            { return s.on_event(std::forward<Event>(event)); },

            [&](auto&... s) -> std::optional<StateVariant>
            { return OnInvalidTransition()(s..., std::forward<Event>(event)); }
        }, state_);
        if(new_state)
            state_ = *std::move(new_state);
    }
};
```

- *Worse overload* thanks to a template parameter pack usage

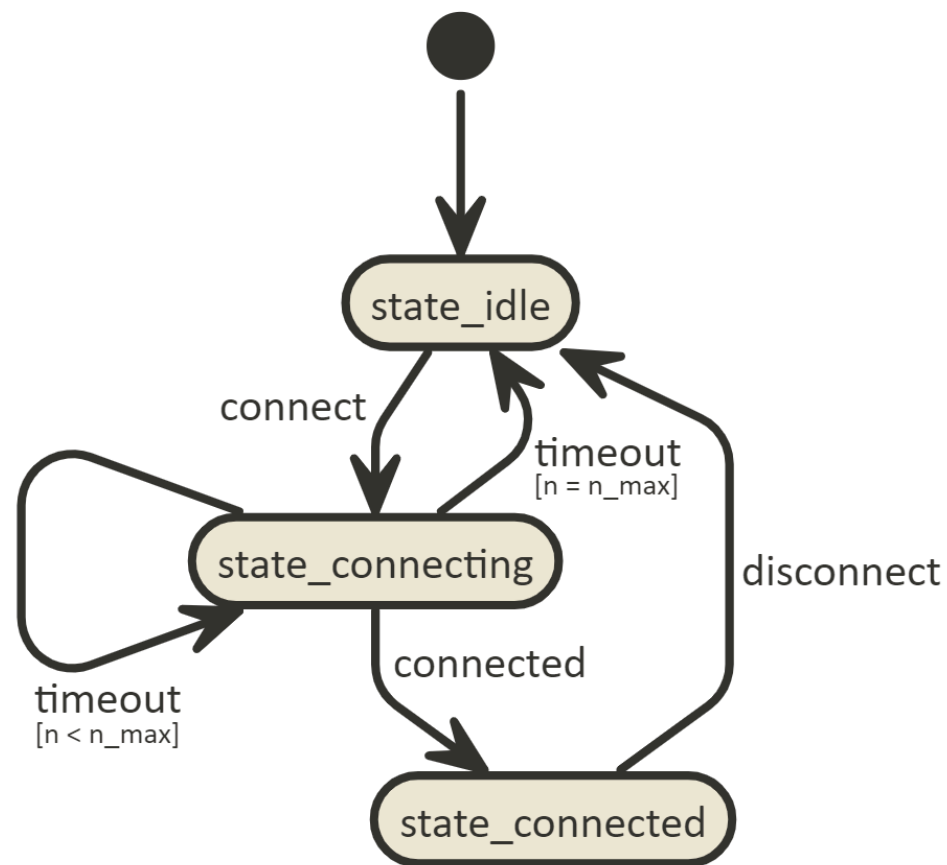
# TRANSITIONS DEFINED BY STATES

```
auto state_idle::on_event(const event_connect& e)
{
    return state_connecting{std::string(e.address)};
}

auto state_connecting::on_event(const event_connected&)
{
    return state_connected{};
}

auto state_connecting::on_event(const event_timeout&)
{
    return ++n_ < n_max ? std::nullopt
        : std::optional<state>(state_idle{});
}

auto state_connected::on_event(const event_disconnect&)
{
    return state_idle{};
}
```



# TRANSITIONS DEFINED BY STATES

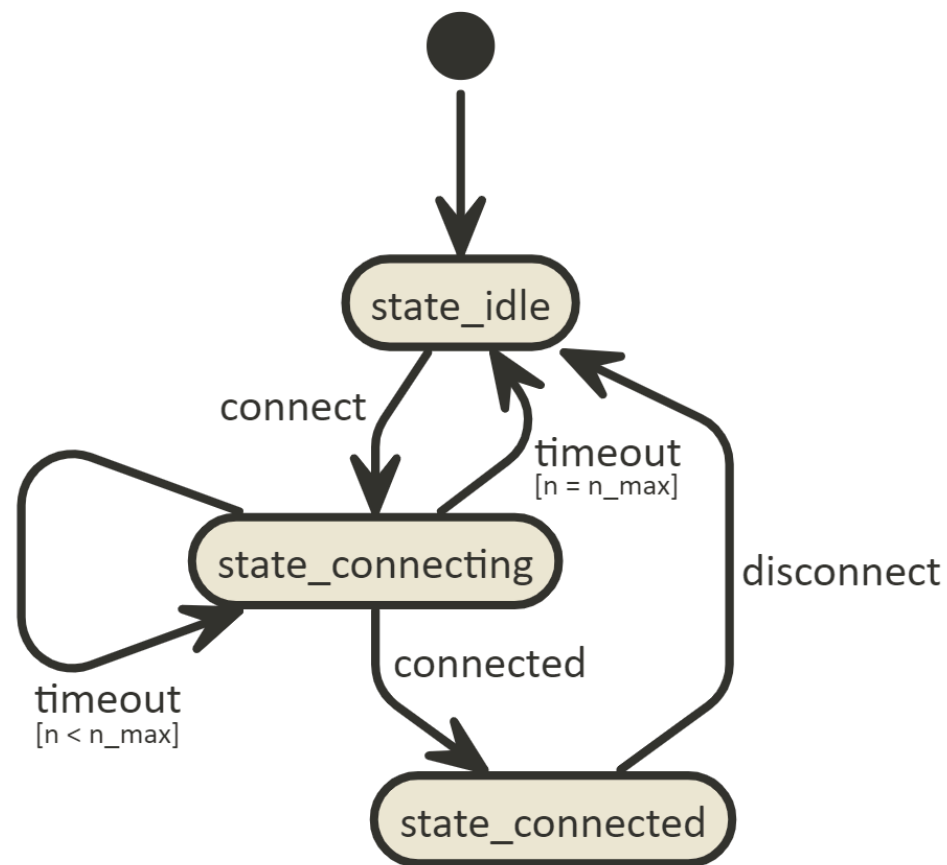
```
auto state_idle::on_event(const event_connect& e)
{
    return state_connecting{std::string(e.address)};
}

auto state_connecting::on_event(const event_connected&)
{
    return state_connected{};
}

auto state_connecting::on_event(const event_timeout&)
{
    return ++n_ < n_max ? std::nullopt
        : std::optional<state>(state_idle{});
}

auto state_connected::on_event(const event_disconnect&)
{
    return state_idle{};
}
```

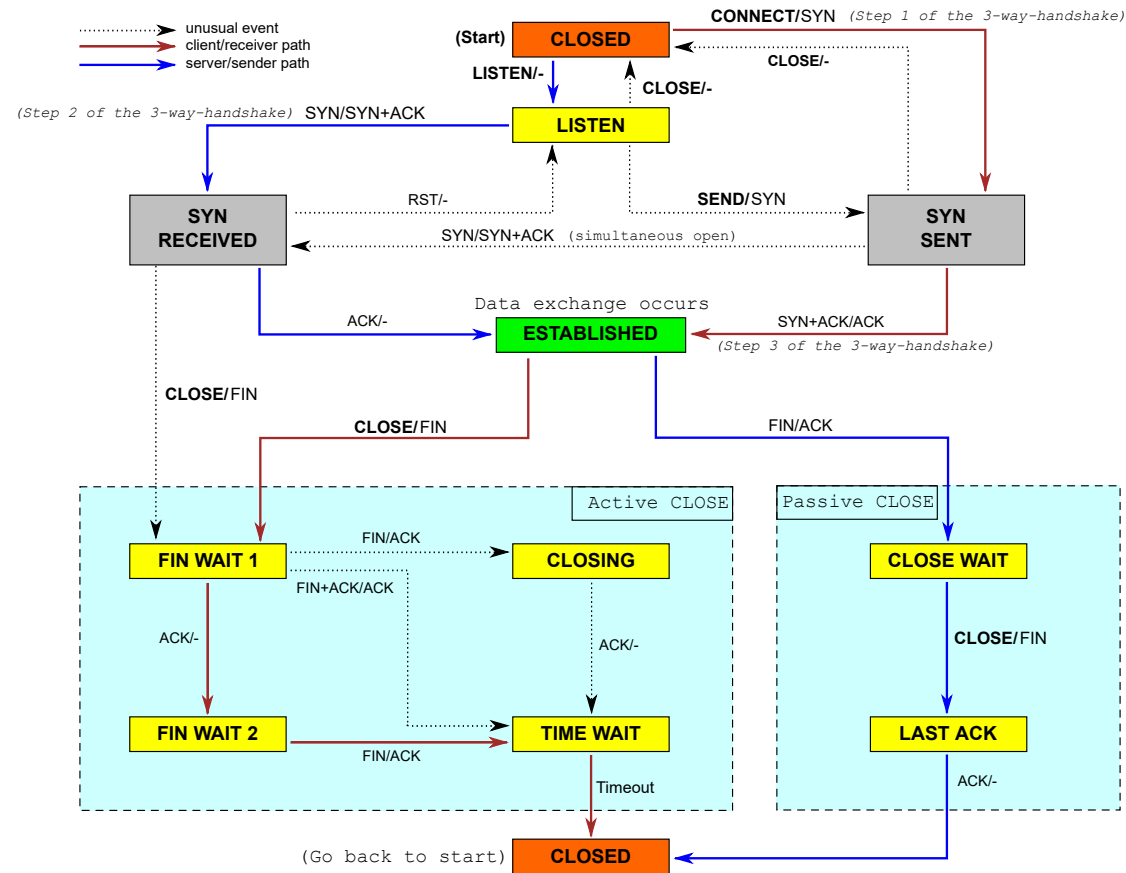
```
struct ignore_unknown_event {
    template<typename State, typename Event>
    auto operator()(const State&, const Event&)
    { return std::nullopt; }
};
```



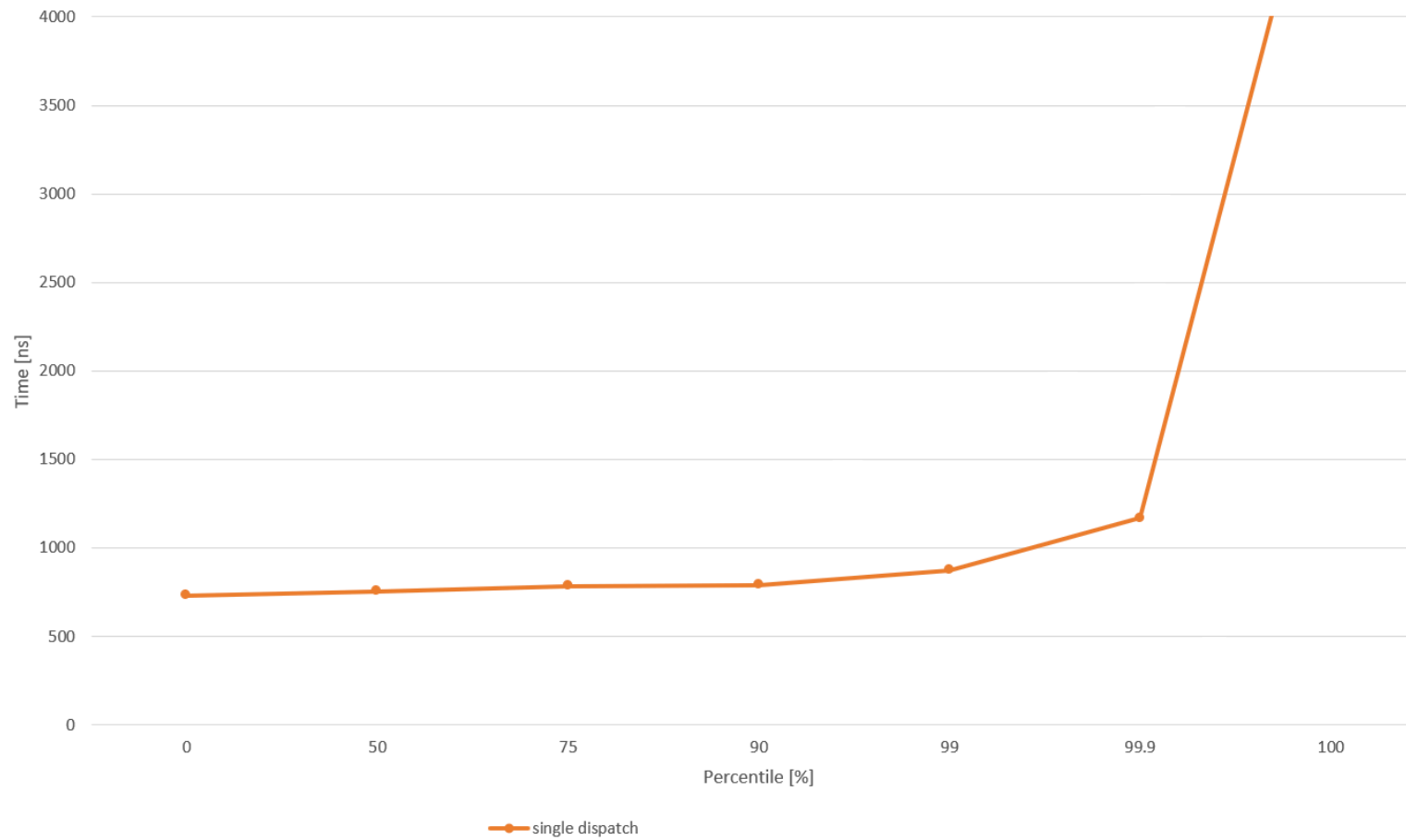


**WHAT ABOUT PERFORMANCE?**

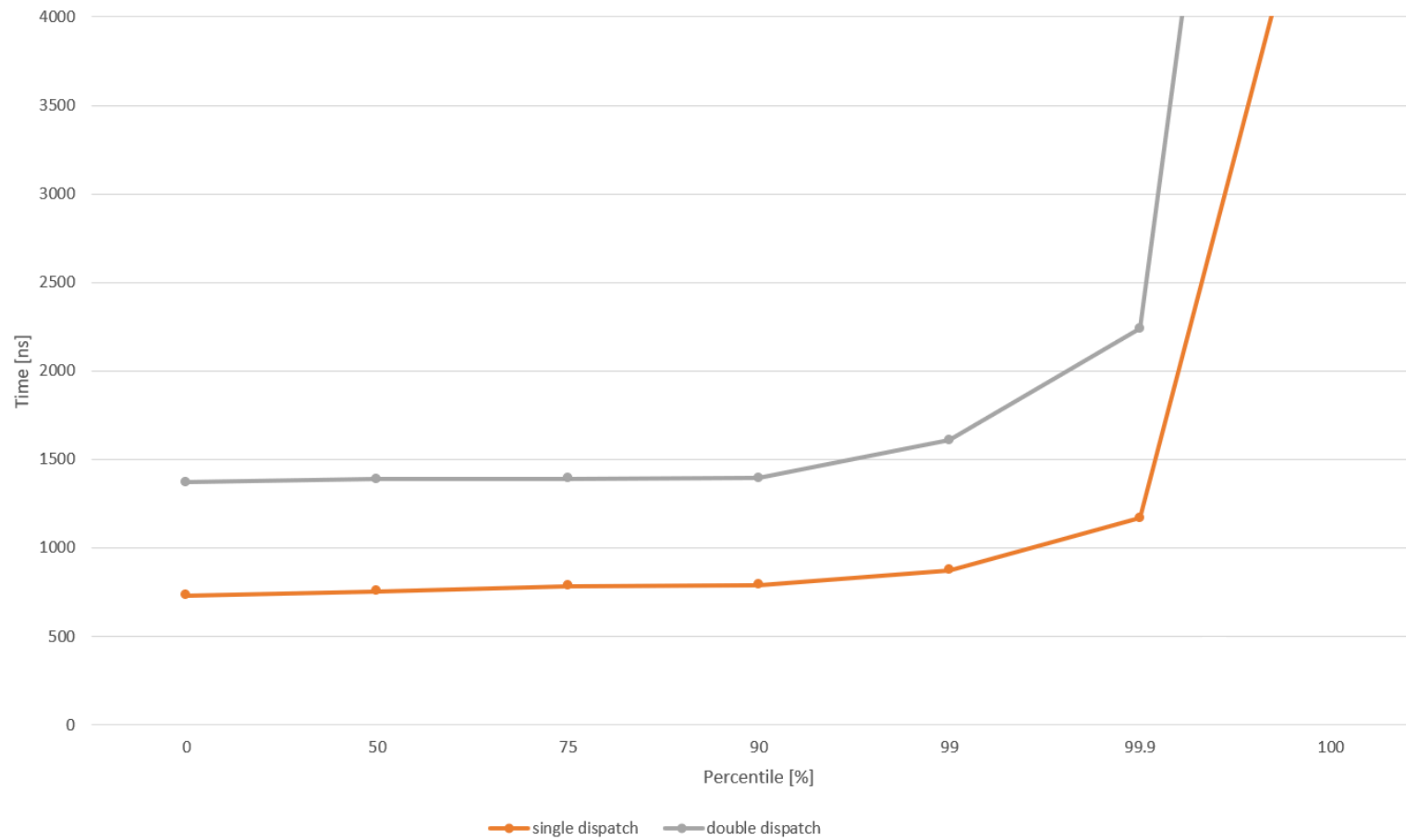
# TCP STATE DIAGRAM



# TCP FSM PERFORMANCE

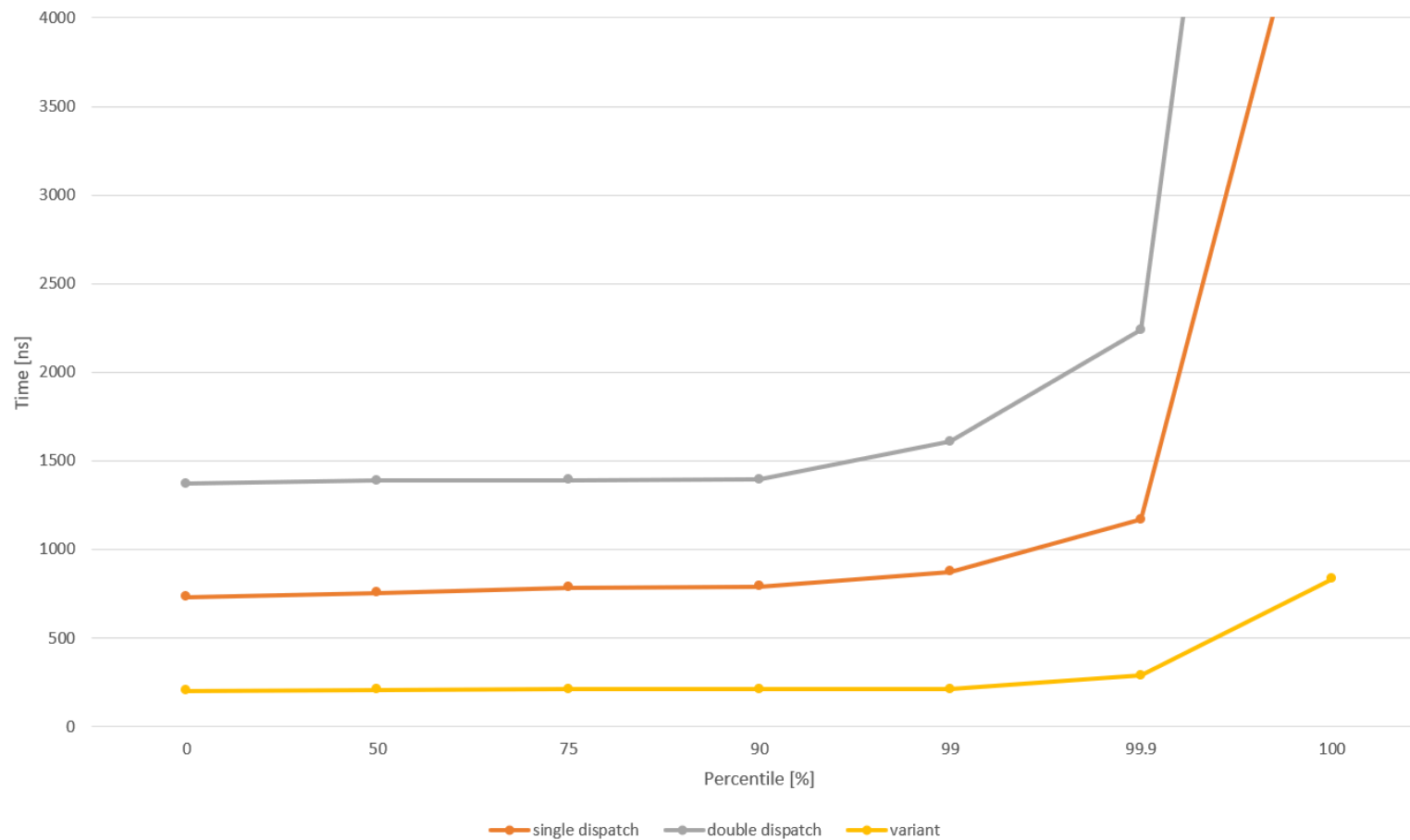


# TCP FSM PERFORMANCE





# TCP FSM PERFORMANCE



# SUMMARY



# `std::variant<Types...>` VS INHERITANCE

---

INHERITANCE

VARIANT

# `std::variant<Types...>` VS INHERITANCE

---

## INHERITANCE

- Open/Closed to new alternatives

## VARIANT

- Closed to new alternatives

# `std::variant<Types...>` VS INHERITANCE

## INHERITANCE

- Open/Closed to new alternatives
- Closed to new operations

## VARIANT

- Closed to new alternatives
- Open to new operations

# `std::variant<Types...>` VS INHERITANCE

## INHERITANCE

- Open/Closed to new alternatives
- Closed to new operations
- Multi-level

## VARIANT

- Closed to new alternatives
- Open to new operations
- Single level

# `std::variant<Types...>` VS INHERITANCE

## INHERITANCE

- Open/Closed to new alternatives
- Closed to new operations
- Multi-level
- OO

## VARIANT

- Closed to new alternatives
- Open to new operations
- Single level
- Functional

# `std::variant<Types...>` VS INHERITANCE

## INHERITANCE

- Open/Closed to new alternatives
- Closed to new operations
- Multi-level
- OO
- Pointer semantics

## VARIANT

- Closed to new alternatives
- Open to new operations
- Single level
- Functional
- Value semantics



# `std::variant<Types...>` VS INHERITANCE

## INHERITANCE

- Open/Closed to new alternatives
- Closed to new operations
- Multi-level
- OO
- Pointer semantics
- Design forced by the implementation details

## VARIANT

- Closed to new alternatives
- Open to new operations
- Single level
- Functional
- Value semantics
- Many design choices possible

# `std::variant<Types...>` VS INHERITANCE

## INHERITANCE

- Open/Closed to new alternatives
- Closed to new operations
- Multi-level
- OO
- Pointer semantics
- Design forced by the implementation details
- Forces dynamic memory allocations

## VARIANT

- Closed to new alternatives
- Open to new operations
- Single level
- Functional
- Value semantics
- Many design choices possible
- No dynamic memory allocations

# `std::variant<Types...>` VS INHERITANCE

## INHERITANCE

- Open/Closed to new alternatives
- Closed to new operations
- Multi-level
- OO
- Pointer semantics
- Design forced by the implementation details
- Forces dynamic memory allocations
- Strict interfaces

## VARIANT

- Closed to new alternatives
- Open to new operations
- Single level
- Functional
- Value semantics
- Many design choices possible
- No dynamic memory allocations
- Duck typing

# `std::variant<Types...>` VS INHERITANCE

## INHERITANCE

- Open/Closed to new alternatives
- Closed to new operations
- Multi-level
- OO
- Pointer semantics
- Design forced by the implementation details
- Forces dynamic memory allocations
- Strict interfaces
- Complex

## VARIANT

- Closed to new alternatives
- Open to new operations
- Single level
- Functional
- Value semantics
- Many design choices possible
- No dynamic memory allocations
- Duck typing
- Simple

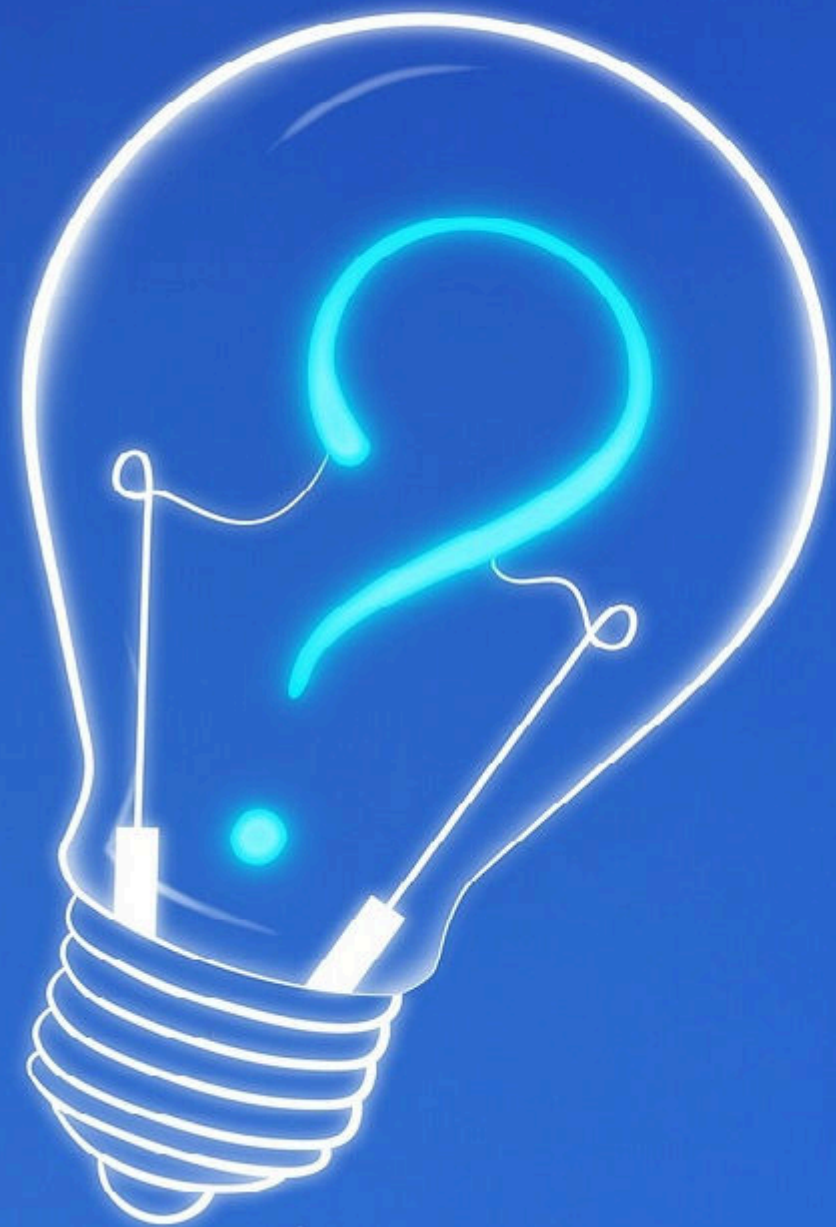
# `std::variant<Types...>` VS INHERITANCE


## INHERITANCE

- Open/Closed to new alternatives
- Closed to new operations
- Multi-level
- OO
- Pointer semantics
- Design forced by the implementation details
- Forces dynamic memory allocations
- Strict interfaces
- Complex
- Slower

## VARIANT

- Closed to new alternatives
- Open to new operations
- Single level
- Functional
- Value semantics
- Many design choices possible
- No dynamic memory allocations
- Duck typing
- Simple
- Faster



The background is a solid yellow color. It is decorated with several black geometric shapes, primarily parallelograms and triangles, arranged in a pattern that suggests a 3D perspective or a stylized architectural design. These shapes are positioned around the edges and corners of the frame.

**CAUTION**  
**Programming**  
**is addictive**  
**(and too much fun)**