C++ONLINE

ANTHONY WILLIAMS

DESIGNING FOR CONCURRENCY USING MESSAGE PASSING

2024

Designing For Concurrency Using Message Passing

woven by TOYOTA

Designing For Concurrency Using Message Passing

- Why use message passing?

woven by TOYOTA

Designing For Concurrency Using Message Passing

- Why use message passing?
- Message passing frameworks

Designing For Concurrency Using Message Passing

- Why use message passing?
- Message passing frameworks
- Design Guidelines

woven by TOYOTA

Designing For Concurrency Using Message Passing

- Why use message passing?
- Message passing frameworks
- Design Guidelines
- Examples

woven by TOYOTA

Why use message passing?

The Hook

Eliminate all your concurrency bugs!

The Reality

You can:

The Reality

You can:

- Eliminate explicit synchronization

woven by TOYOTA

The Reality

You can:

- Eliminate explicit synchronization
- Eliminate OS-level deadlock

The Reality

You can:

- Eliminate explicit synchronization
- Eliminate OS-level deadlock
- Eliminate data races

The Cost

In exchange, you must:

The Cost

In exchange, you must:

- Use the framework for all synchronization

woven by TOYOTA

The Cost

In exchange, you must:

- Use the framework for all synchronization
- Avoid shared mutable state

The Cost

In exchange, you must:

- Use the framework for all synchronization
- Avoid shared mutable state
- Change your design approach

woven by TOYOTA

Remaining Problems

Problems still remain:

woven
by TOYOTA

Remaining Problems

Problems still remain:

- Still potential for race conditions
  $\Rightarrow$ Which message arrives first?

Remaining Problems

Problems still remain:

- Still potential for race conditions
  $\Rightarrow$ Which message arrives first?
- Still potential for message deadlock
  $\Rightarrow$ Two elements expecting messages from each other

Remaining Problems

Problems still remain:

- Still potential for race conditions
  $\Rightarrow$ Which message arrives first?
- Still potential for message deadlock
  $\Rightarrow$ Two elements expecting messages from each other
- The potential for dropped messages is a **new** problem

Remaining Problems

Problems still remain:

- Still potential for race conditions
  $\Rightarrow$ Which message arrives first?
- Still potential for message deadlock
  $\Rightarrow$ Two elements expecting messages from each other
- The potential for dropped messages is a **new** problem
- Still not a silver bullet

woven by TOYOTA

Message Passing Frameworks

What is a Message Passing Framework?

Tooling that manages the delivery of messages between independent elements

woven by TOYOTA

Benefits of an MPF

Developers can focus on:

Benefits of an MPF

Developers can focus on:

- the **content**,

Benefits of an MPF

Developers can focus on:

- the **content**,
- the **targets**, and

woven by TOYOTA

Benefits of an MPF

Developers can focus on:

- the **content**,
- the **targets**, and
- the **processing**

woven by TOYOTA

Benefits of an MPF

Developers can focus on:

- the **content**,
- the **targets**, and
- the **processing**

of messages, rather than the **delivery mechanism**.

woven by TOYOTA

Microservices

An MPF allows you to divide your application into **microservices**.

Microservices

An MPF allows you to divide your application into **microservices.**

The API for each microservice is the messages you can send it, and the messages it sends.

Guarantees on lack of reentrancy

An MPF will usually guarantee that each microservice only processes one message at a time.

woven by TOYOTA

Guarantees on lack of reentrancy

An MPF will usually guarantee that each microservice only processes one message at a time.

They may also guarantee that all messages for a given microservice will be handled on the same thread.

woven by TOYOTA

Guarantees on lack of reentrancy

An MPF will usually guarantee that each microservice only processes one message at a time.

They may also guarantee that all messages for a given microservice will be handled on the same thread.

They may give you control over which microservices share a thread.

woven by TOYOTA

Delivery Mechanisms

The MPF may give you a choice of delivery mechanism.

Delivery Mechanisms

The MPF may give you a choice of delivery mechanism.

The mechanism may be in-process, cross-process, or even cross-network.

woven by TOYOTA

Delivery Mechanisms

The MPF may give you a choice of delivery mechanism.

The mechanism may be in-process, cross-process, or even cross-network.

Ideally the microservice code will be isolated from that choice.

woven
by TOYOTA

I’ve got a theory

woven by TOYOTA

I’ve got a theory

In practice, widely available MPFs require you to write a lot of boilerplate: you often have to write code to drive the message loop.

woven by TOYOTA

I’ve got a theory

In practice, widely available MPFs require you to write a lot of boilerplate: you often have to write code to drive the message loop.

Abstract the boiler plate in a wrapper library or tool.

woven by TOYOTA

Design Guidelines

Focus

Each microservice should be focused on one task

woven by TOYOTA

Independence

Microservices should not overlap in their functionality

woven by TOYOTA

Value Types

Messages should be **value types**

Avoid blocking

Replace blocking calls with separate microservices

woven by TOYOTA

State Machines

Each microservice can be modelled as a single-thread state machine

woven by TOYOTA

Examples

Example: Dining Philosophers

Example: Dining Philosophers

- **N philosophers** sat round a table sharing a large plate of rice.
- Each philosopher wants to alternate between **thinking** and **eating**.
- There are **N chopsticks** around the table, with one between every pair of philosophers.
- All philosophers start thinking.
- After a random length of time thinking, a philosopher will try and eat.
- To eat, a philosopher must pick up the chopsticks either side. A philosopher cannot eat without both chopsticks.
- After a random length of time eating, a philosopher will put down both chopsticks and think.

Actors for Dining Philosophers

We can create multiple actors:

- An actor for each philosopher
- An actor for I/O
- An actor for Timing
- An actor for the “server” responsible for managing the chopsticks

woven by TOYOTA

The I/O Actor

The **I/O** actor just waits for incoming messages with text to print out.

```
using OutputMessage = std::string;

void IO::OnMessage(OutputMessage msg) {
  std::cout << msg << '\n';
}
```

The Timing Actor

The **Timing** actor is more involved. Incoming messages specify a time and another actor:

```
struct TimerRequest {
  std::chrono::steady_clock time;
  messaging::sender target;
};
```

The **Timing** actor sends a `TimerExpired` message to the `target` at the appropriate time.

woven by TOYOTA

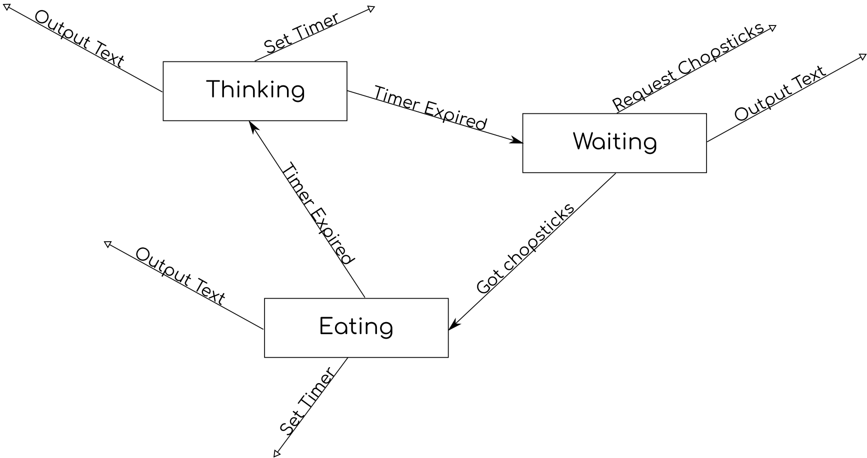
The Philosopher Actors

Each philosopher has 3 states: **thinking**, **waiting for chopsticks** and **eating**.

In each state we can process a particular set of incoming messages.

woven by TOYOTA

Philosopher States



woven by TOYOTA

A Thinking Philosopher

When the philosopher starts thinking, then we send a message to the **I/O** actor to say so, and to the **Timing** actor specifying when they're done thinking:

```
void Philosopher::OnThinking() {
  auto thinking_time = random_duration();
  io.Send(name + " thinking");
  timing.Send(
    TimerRequest{Clock::now() + thinking_time, GetSelf()});
}
```

woven by TOYOTA

A Thinking Philosopher

When the timer expires, the philosopher is waiting for chopsticks:

```
void Philosopher::OnMessage(TimerExpired) {
  if(current_state == states::thinking) {
    io.Send(name + " finished thinking");
    SetState(states::waiting_for_chopsticks);
  }
}
```

woven by TOYOTA

A Hungry Philosopher

When the philosopher starts waiting for chopsticks, they must be requested from the **server**, after sending a message to the **I/O** actor to be displayed:

```
void Philosopher::OnWaitingForChopsticks() {
  io.Send(name + " waiting for chopsticks");
  server.Send(RequestChopsticks{
    left_chopstick, right_chopstick,
    GetSelf()});
}
```

woven by TOYOTA

The philosopher gets chopsticks

When the philosopher gets chopsticks they start eating

```
void Philosopher::OnMessage(GotChopsticks) {
  io.Send(name + " received chopsticks");
  SetState(states::eating);
}
```

woven by TOYOTA

An Eating Philosopher

The philosopher eats for a random length of time, just like **thinking**, so we send a message to the **Timing** actor specifying when they're done eating:

```
void Philosopher::OnEating() {
  auto eating_time = random_duration();
  io.Send(name + " eating");
  timing.Send(
    TimerRequest{Clock::now() + eating_time, GetSelf()});
}
```

woven by TOYOTA

A Full Philosopher

When the philosopher is done eating they hand in the chopsticks and resume thinking.

```
void Philosopher::OnMessage(TimerExpired) {
  if(current_state == states::thinking) {
    // as before
  } else if (current_state == states::eating) {
    io.Send(name + " finished eating");
    server.Send(ReturnChopsticks{
      left_chopstick, right_chopstick});
    SetState(states::thinking);
  }
}
```

woven by TOYOTA

The Server Actor: Chopstick Requests

If chopsticks are requested, the server must hand them out, or add the philosopher to a waiting list:

```
void Server::OnMessage(RequestChopsticks request){
  if(ChopsticksAvailable(request)){
    UseChopsticksAndNotify(request);
  } else {
    waiting_list.push_back(request);
  }
}
```

woven by TOYOTA

The Server Actor: Returned Chopsticks

If chopsticks are returned, mark them free and check for waiting philosophers:

```
void Server::OnMessage(ReturnChopsticks request){
  MarkChopsticksAvailable(request);
  for(auto req=waiting_list.begin();
      req != waiting_list.end(); ) {
    if(ChopsticksAvailable(*req)) {
      UseChopsticksAndNotify(*req);
      it = waiting_list.erase(it);
    } else ++it;
  }
}
```

woven by TOYOTA

The Server Actor: Using Chopsticks

Notifying philosophers about chopsticks is just another message:

```
void Server::UseChopsticksAndNotify(
    RequestChopsticks request) {
  chopstick_free[request.left_chopstick] = false;
  chopstick_free[request.right_chopstick] = false;
  request.philosopher.Send(GotChopsticks{});
}
```

woven by TOYOTA

Example: A robot control system

woven by TOYOTA

Example: A robot control system

Moving a robot requires coordinating movements of multiple parts, each with their own actuator.

We want the control code for each actuator to run concurrently with the coordination code and the code for other actuators.

woven by TOYOTA

Example: Moving a robot

We have actors for:

- The central coordinator
- Each actuator
- Timing

woven by TOYOTA

The central coordinator

For each large-scale position the robot needs to be in, send messages to the actuators with the steps:

```
void Coordinator::OnMessage(SetPositionFoo msg) {
  SetState(states::waiting_for_position_foo);
  actuator1.Send(SetPosition{Ac1FooPos, msg.target_time});
  actuator2.Send(SetPosition{Ac2FooPos, msg.target_time});
  actuator3.Send(SetPosition{Ac3FooPos, msg.target_time});
  actuator4.Send(SetPosition{Ac4FooPos, msg.target_time});
}
```

woven by TOYOTA

Actuator actors

Each actuator has hardware-specific code to control it.

```
void Actuator1::OnMessage(SetPosition msg) {
  hardware.StopMoving();
  auto curpos = hardware.GetCurrentPosition();
  auto delta = msg.position - curpos;
  auto time_delta = msg.target_time - Clock::now();
  if(delta) {
    hardware.StartMoving({.speed = delta/time_delta});
    timing.Send(TimerRequest{msg.target_time, GetSelf()});
  }
}
```

Actuator actors

Notify the controller when target reached:

```
void Actuator1::OnMessage(TimerRequest) {
  hardware.StopMoving();
  controller.Send(Actuator1ReachedPosition{});
}
```

woven by TOYOTA

The central coordinator

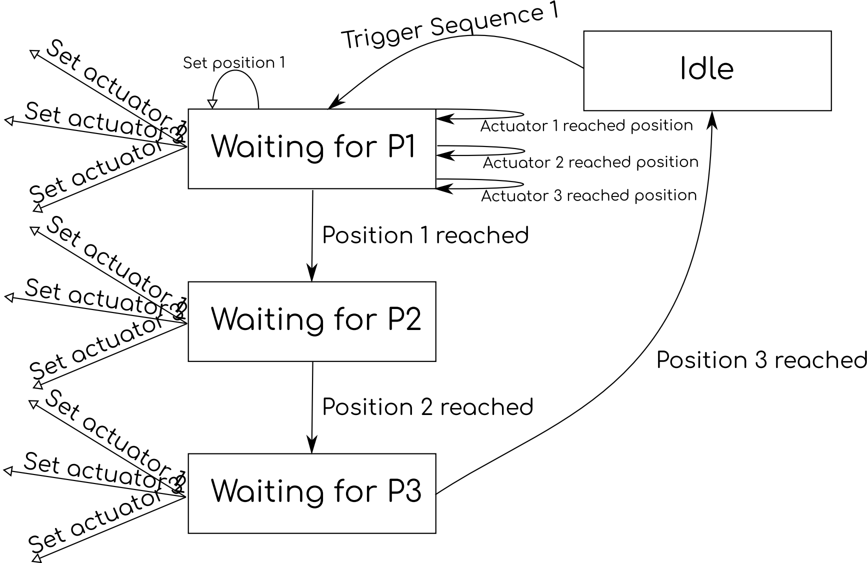
For movement through a sequence of positions, add intermediate states:

- Waiting for Position 1
- Waiting for Position 2
- Waiting for Position 3

Add messages for setting those positions, and notifying when they are reached.

woven by TOYOTA

Central Coordinator States



woven by TOYOTA

Starting a movement sequence

Start a movement sequence by moving to the first position:

```
void Coordinator::OnMessage(TriggerMotionSequence1 msg) {
  auto position1_time = Clock::now() + position1_duration;
  SetState(states::waiting_for_position1);
  GetSelf().Send(SetPosition1{position1_time});
}
```

woven by TOYOTA

Waiting to reach a position

As each actuator reaches position, check overall robot state:

```
void Coordinator::OnMessage(Actuator1ReachedPosition) {
  switch(state) {
  case states::waiting_for_position1:
    if(ActuatorsReachedPosition1()) {
      GetSelf().Send(Position1Reached{});
    }
    break;
  // code for other states
  }
}
```

woven by TOYOTA

Moving to the next position

When the robot is in the right position, move to the second position:

```
void Coordinator::OnMessage(Position1Reached) {
  auto position2_time = Clock::now() + position2_duration;
  SetState(states::waiting_for_position2);
  GetSelf().Send(SetPosition2{position2_time});
}
```

woven by TOYOTA

More robot states

For complex movements and multiple sequences, the state map can get large.

More robot states

For complex movements and multiple sequences, the state map can get large.

Making the state machine table driven can make this easier.

More robot states

For complex movements and multiple sequences, the state map can get large.

Making the state machine table driven can make this easier.

This is just classic state machine processing.

Summary

woven by TOYOTA

Summary

- A Message Passing Framework allows you to focus on the **messages** rather than **delivery**

Summary

- A Message Passing Framework allows you to focus on the **messages** rather than **delivery**
- An MPF can eliminate explicit synchronization

woven by TOYOTA

Summary

- A Message Passing Framework allows you to focus on the **messages** rather than **delivery**
- An MPF can eliminate explicit synchronization
- Each microservice should be focused on one task

Summary

- A Message Passing Framework allows you to focus on the **messages** rather than **delivery**
- An MPF can eliminate explicit synchronization
- Each microservice should be focused on one task
- Microservices should have minimal overlap in their responsibilities

woven by TOYOTA

Summary

- A Message Passing Framework allows you to focus on the **messages** rather than **delivery**
- An MPF can eliminate explicit synchronization
- Each microservice should be focused on one task
- Microservices should have minimal overlap in their responsibilities
- Messages should be value types

woven by TOYOTA

Summary

- A Message Passing Framework allows you to focus on the **messages** rather than **delivery**
- An MPF can eliminate explicit synchronization
- Each microservice should be focused on one task
- Microservices should have minimal overlap in their responsibilities
- Messages should be value types
- Replace blocking calls with separate microservices

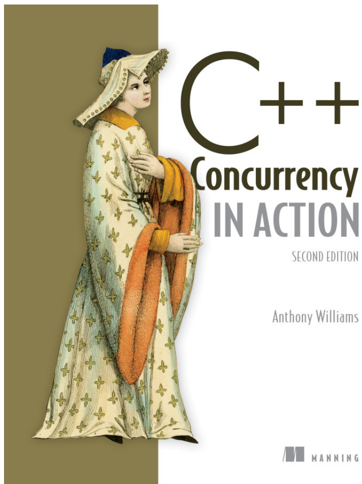
woven by TOYOTA

Summary

- A Message Passing Framework allows you to focus on the **messages** rather than **delivery**
- An MPF can eliminate explicit synchronization
- Each microservice should be focused on one task
- Microservices should have minimal overlap in their responsibilities
- Messages should be value types
- Replace blocking calls with separate microservices
- Each microservice can be modelled as a single-thread state machine

woven by TOYOTA

My Book



C++ Concurrency in Action
Second Edition

Covers C++17 and the
first Concurrency TS

cplusplusconcurrencyinaction.com

woven by TOYOTA

Questions?

Message passing frameworks

Some Choices:

- ZeroMQ
- CAF — The C++ Actor Framework
- The actor framework from my book
- You can write your own.

woven by TOYOTA