Bachelorarbeit

Till Kaiser

Loose Coupling and Communication in Reactive Systems in C++14

Till Kaiser

Loose Coupling and Communication in Reactive Systems in C++14

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Stephan Pareigis
Zweitgutachter: Prof. Dr. rer. nat. Martin Becke

Eingereicht am: July 15th 2016

Till Kaiser

Thema der Arbeit

Loose Coupling and Communication in Reactive Systems in C++14

Stichworte

Lose Kopplung, Reaktive Systeme, C++, Kommunikation, Design Patterns

Kurzzusammenfassung

In dieser Bachelorarbeit werden verschiedene Kommunikations- und Event-Distributions-Techniken untersucht, um lose Kopplung in reaktiven Systemen zu erreichen. Mehrere Implementierungen für verschiedene Kommunikationsmodelle sowie Variationen auf bewährte Kommunikations-Entwurfsmuster (Design Patterns) werden in modernem C++14 Code erschlossen. Const Correctness und konsequente Nutzung von Smart-Pointern sind Teil des Standards dieser Bachelorarbeit.

Till Kaiser

Title of the paper

Loose Coupling and Communication in Reactive Systems in C++14

Keywords

loose coupling, reactive systems, C++, communication, design patterns

Abstract

This bachelor’s thesis explores the use of different communication and event distribution techniques to achieve loose coupling in reactive systems. Several implementations for various communication models as well as variations on traditional communication design patterns are supplied in modern C++14 code. Const correctness and the use of smart pointers are part of the standard used in this bachelor’s thesis.

Contents

List of Tables

List of Figures

Listings

Introduction

1 Motivation

Modern object oriented software consists of many individual components which are required to communicate with each other. Especially in reactive systems communication is an essential part as every action triggers certain well-defined reactions from the entire system. External and internal events have to be distributed among different system components. However, the communication between those components should not be a direct communication because this couples the components together while they should be independent. If a system's components depend directly upon each other they cannot easily be designed, created, tested or replaced individually. Especially in larger scale systems this is problematic. So the goal is to achieve a high degree of loose coupling but still manage an efficient communication between the individual components.

The approach of this bachelor's thesis is to translate the set-up of communication from a digital electric circuit into object oriented code using the latest C++14 standard.

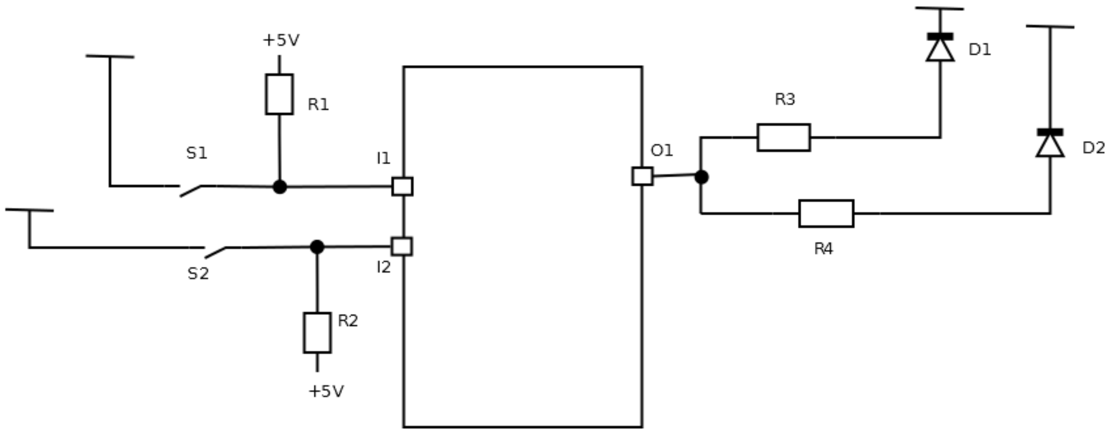


Figure 1.1: Electric circuit

Taking a look at a simple logic circuit (Figure 1.1) it becomes apparent, that an out-port can be connected to numerous entities (O1 is connected to the LEDs D1 and D2) but an in-port (I1, I2) can only be connected to a single entity (the switches S1 and S2 in the figure). This is reflected in the idea of ports and connectors presented in Chapter 3.

2 Coding Style

The coding style used in this bachelor's thesis is defined as follows:

Type	**Case**	**Separator**	**Leading**	**Trailing**
class names	lower case	underscore	-	-
struct names	lower case	underscore	-	-
function names	lower case	underscore	-	-
template types	upper camel case	-	-	underscore
private members	lower case	underscore	-	underscore
function variables	lower case	underscore	-	two underscores
typedefs	upper camel case	-	underscore	-
function arguments	lower case	underscore	-	-
static constants	all capitals	underscore	-	-
enums	all capitals	underscore	-	-

Table 2.1: Coding style used in this thesis

Indentation type	Spaces
Tabbing type	Spaces
Indentation size	4 spaces

Table 2.2: Indentation and Tabbing policy

Basic Communication Models

3 Ports and Connectors

Bran Selic invented the ROOM methodology in 1996 [3] which predates UML and introduces **ports** as bidirectional communication interfaces. The ROOM model is relatively complex and mostly applicable for large-scale systems but the basic idea of channelling all communications through ports can be applied to basic communication models as well.

In order to create a connection between two independent objects for communication three components must be designed (Figure 3.1):

- An **out-port** as the source of the data to be transferred
- An **in-port** as the destination
- A **connector** to forward the data from out-port to in-port

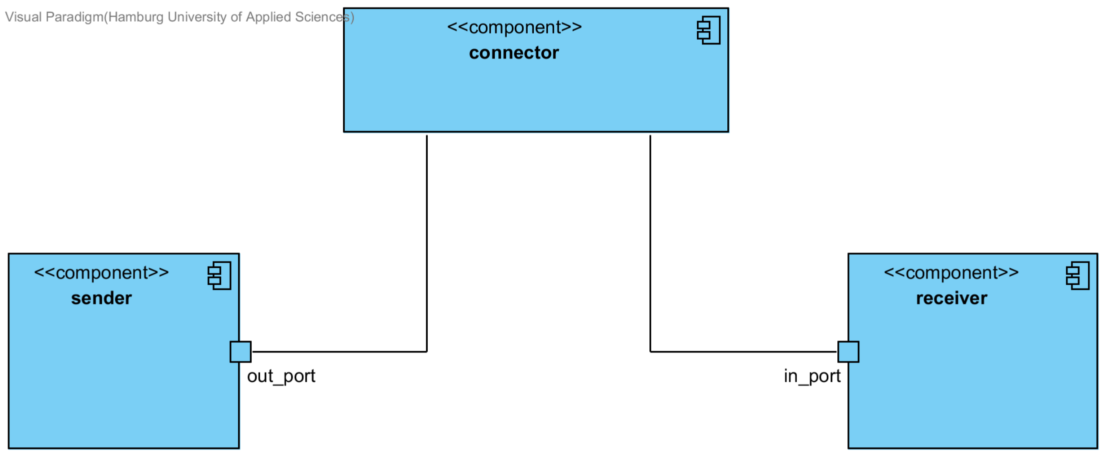


Figure 3.1: Connection between two components: component diagram

The *Single Responsibility Principle* dictates that all classes solely serve one purpose ("A class should have only one reason to change" [4, p. 95]). Thus the connector between the two ports should be implemented as a separate class. Objects can contain any number of in- and out-ports.

Only a maximum of one out-port can be connected to every in-port while an out-port can be connected to any number of in-ports.

The implementation now depends on the set-up of those three classes. The following cases have to be considered:

3.1 Single-Threaded Environment

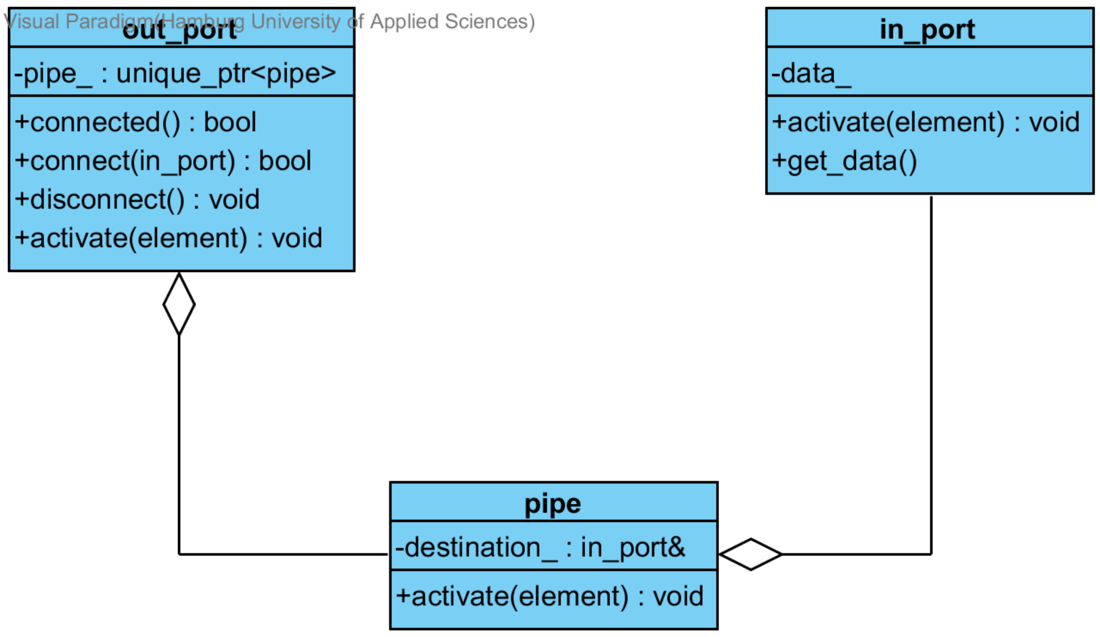

Visual Paradigm(Hamburg University of Applied Sciences)
out_port
-pipe_ : unique_ptr<pipe>
+connected() : bool
+connect(in_port) : bool
+disconnect() : void
+activate(element) : void
in_port
-data_
+activate(element) : void
+get_data()
pipe
-destination_ : in_port&
+activate(element) : void

Figure 3.2: Unidirectional communication using pipe: class diagram

If the two communicating objects exist in the same thread, the only way is to pass the data directly through the connector. This connector shall forth be called a **pipe**. Since all components run on the same thread, the out-port must call the pipe and the pipe must call the connected in-port (Figure 3.2). After the data has been processed on the receiver's (in-port) side the programme returns to the caller (out-port) and can continue.

```cpp
template <typename Type_>
class out_port {
    std::unique_ptr<pipe<Type_>> pipe_;
public:
    out_port()
        : pipe_{nullptr} {}

    bool connected(void) const {
        return pipe_ != nullptr;
    }

    bool connect(std::unique_ptr<pipe<Type_>> p) {
        if(connected())
            return false;
        pipe_ = std::move(p);
        return true;
    }

    bool connect(in_port<Type_>& in) {
        return connect(std::make_unique<pipe<Type_>>(in));
    }

    void disconnect(void) {
        pipe_ = nullptr;
    }

    void activate(const Type_& element) {
        if(connected()) {
            pipe_ -> activate(element);
        }
    }

    void activate(Type_&& element) {
        if(connected()) {
            pipe_ -> activate(std::forward<Type_>(element));
        }
    }
};
```

Listing 3.1: Implementation of an out-port class

The out-port class (Listing 3.1) holds a unique pointer to the connected pipe. Since a pipe is a direct forward connection between an out- and an in-port it can only ever be referenced by a single out-port. The unique pointer guarantees this.

An out-port can be connected directly to the in-port using the *connect(in_port<Type_>&)* function which creates the pipe between the ports on the fly. Alternatively the pipe to be used can be specified by using the *connect(unique_ptr<pipe<Type_> >)* function. This is useful when additional functionality for the pipe is desired. In this case a class derived from pipe can be passed as the argument.

```
template <typename Type_> class in_port {
    Type_ data_;
    int id_;
public:
    in_port()
        : data_{}
        , id_{}
    {   static int ID = 0;
        id_ = ID++;    }

    virtual void activate(const Type_& element) {
        data_ = element;
    }

    virtual void activate(Type_&& element) {
        data_ = std::forward<Type_>(element);
    }

    virtual auto get_id(void) const {
        return id_;
    }

    virtual Type_ get_data(void) const {
        return data_;
    }

    virtual ~in_port() {}
};
```

Listing 3.2: Implementation of an in-port class

It is generally helpful to be able to identify the ports with a unique ID. In this example the in-port has been fitted with a static ID counter which applies a unique integer identifier to each created in-port. Another common approach would be to name the objects with unique string identifiers. Maintaining these IDs can be done in a supervisor object which will be discussed in Chapter 4.

All three components have a member function *activate* both for const references and perfect forwarding. This is where the connection between the out- and in-port lies. When activated, the out-port calls the corresponding function on the pipe which in turn calls *activate* on the in-port. The argument is the data to be transferred and is passed from function to function using forward referencing.

```
template <typename Type_>
class pipe {
    in_port<Type_>& destination_;

public:
    explicit pipe(in_port<Type_>& destination)
        : destination_{destination}
    {}

    virtual ~pipe() {}

    virtual void activate(const Type_& element) {
        destination_.activate(element);
    }

    virtual void activate(Type_&& element) {
        destination_.activate(std::forward<Type_>(element));
    }

    virtual auto get_destination(void) const {
        return destination_;
    }
};
```

Listing 3.3: Implementation of a pipe

The pipe requires a valid reference to an in-port in order to be created. This prevents the creation of an unconnected pipe. While ports can exist without being connected to allow for connections at run time, pipes should only exist when connected on both sides.

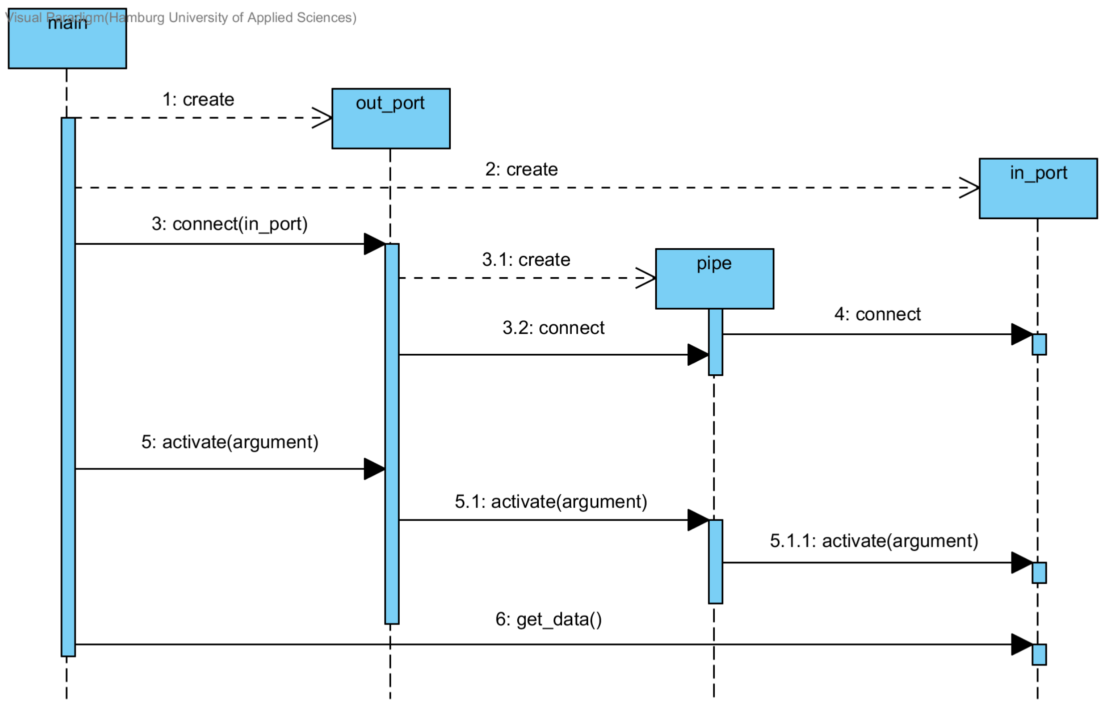


Figure 3.3: Unidirectional communication using pipe: sequence diagram

3.2 Multi-Threaded Environment

If the caller and receiver run in separate threads, the connector acts as a mediator [5, p. 273] between the caller and receiver threads. Both threads access the connector which in this context shall be called a **buffer**. The caller stores its data in the buffer and then continues running. The receiver collects the data from the buffer and then returns to process it.

This data exchange can happen in a lot of different ways and always has to be synchronised in some way to avoid race conditions:

- **fully synchronous**: The caller thread blocks when the buffer is filled, the receiver thread blocks when the buffer is empty
- **continuous write**: The caller continuously overwrites the existing data in the buffer, the receiver blocks on an empty buffer
- **continuous read**: The receiver always collects the latest data in the buffer even if it was already collected before

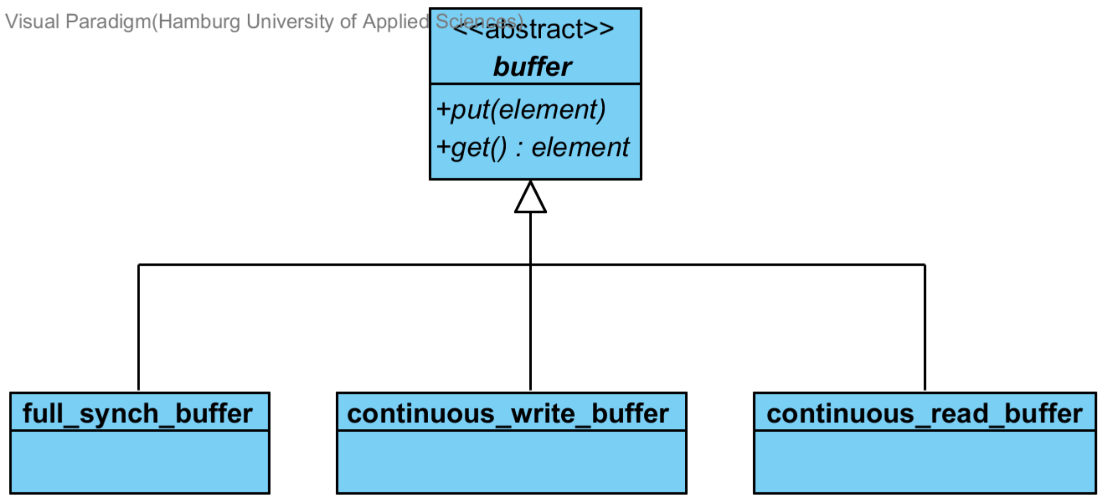


Figure 3.4: Unidirectional communication using buffer: class diagram

All these different buffers derive from the same abstract buffer struct as they all use the same two basic functions *put* and *get* (Figure 3.4). The buffer struct is implemented as seen in Listing 3.4.

```
template <typename Type_>
struct buffer {
    virtual ~buffer() {}

    virtual void put(Type_) = 0;

    virtual Type_ get(void) = 0;
};
```

Listing 3.4: Abstract buffer base struct

3.2.1 Full Synchronous Buffer

If a data consistent buffer is needed the fully synchronous approach is to be chosen. A full synchronous buffer blocks on both the *get* and the *put* call. If *get* is called on an empty buffer it blocks until new data is available. Every call of *get* flushes the buffer. If *put* is called on a buffer that already holds data the call blocks until the buffer is empty. In a full synchronous buffer every set of data is read exactly once. No data is overwritten (and lost) and none is read twice.

```cpp
template <typename Type_>
class full_synch_buffer : public buffer<std::unique_ptr<Type_>> {
public:
    full_synch_buffer()
        : element_{nullptr}
        , mtx_{}
        , cond_{}
    {
    }

    void put(std::unique_ptr<Type_> element) override {
        {
            std::unique_lock<std::mutex> lock__(mtx_);
            cond_.wait(lock__, [&]{return element_==nullptr;});
            element_ = std::move(element);
        }
        cond_.notify_one();
    }

    std::unique_ptr<Type_> get(void) override {
        decltype(element_) return__ = nullptr;
        {
            std::unique_lock<std::mutex> lock__(mtx_);
            cond_.wait(lock__, [&]{return element_!=nullptr;});
            return__ = std::move(element_);
        }
        cond_.notify_one();
        return return__;
    }

private:
    std::unique_ptr<Type_> element_;
    std::mutex mtx_;
    std::condition_variable cond_;
};
```

Listing 3.5: Full synchronous buffer implementation

Because the data in the buffer is always unique (it is moved out of the buffer when *get* is called) it derives from **buffer<std::unique_ptr<Type_> >** instead of **buffer<Type_ > >** (Listing 3.5). This makes *put* take a unique pointer as an argument and *get* return a unique pointer. A condition variable is used to synchronise access to the buffered data.

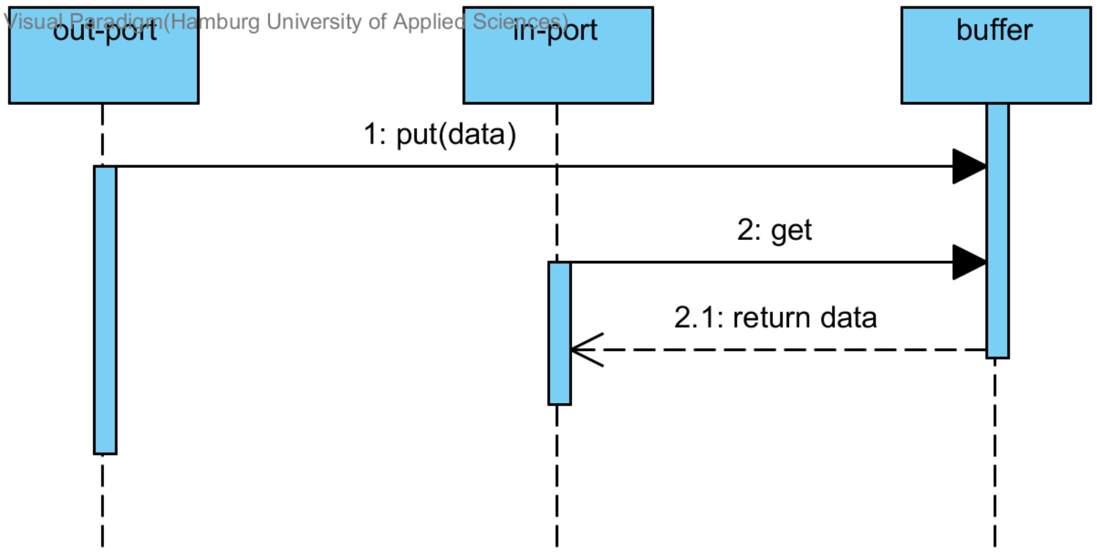


Figure 3.5: Unidirectional communication using buffer: sequence diagram

3.2.2 Continuous Write Buffer

The continuous write buffer is useful in contexts where old data becomes irrelevant as soon as new data is available. This is often the case when collecting and processing sensor data in reactive systems. It is usually sensible to always work on the latest set of sensor data so any old data in the buffer becomes irrelevant by the time new data becomes available. The continuous write buffer follows that principle by only blocking on the *get* call when the buffer is empty. This ensures that every set of data can only be collected from the buffer once but it is always possible to overwrite the current contents of the buffer.

The implementation is very similar to the full synchronous buffer and again uses a unique pointer to reference its data (Listing 3.6). Even though the data in the buffer is continuously overwritten every element is only read once which still makes the element in the buffer unique.

```
template <typename Type_>
class continuous_write_buffer: public buffer<std::unique_ptr<Type_>>
```

Listing 3.6: Continuous write buffer: definition

The buffer contains three member variables, a unique pointer to the stored element, a mutex to lock the buffer and a condition variable for organising access to the stored data.

```
private:
    std::unique_ptr<Type_> element_;
    std::mutex mtx_;
    std::condition_variable cond_;
```

Listing 3.7: Continuous write buffer, private members

The *put* function overwrites the current data in the buffer and then calls *notify_one* on the condition variable (Listing 3.8).

```
    void put(std::unique_ptr<Type_> element) override {
        {
            std::unique_lock<std::mutex> lock__(mtx_);
            element_ = std::move(element);
        }
        cond_.notify_one();
    }
```

Listing 3.8: Continuous write buffer, put function

```
    std::unique_ptr<Type_> get(void) override {
        decltype(element_) return__ = nullptr;
        {
            std::unique_lock<std::mutex> lock__(mtx_);
            cond_.wait(lock__, [&]{return element_!=nullptr;});
            return__ = std::move(element_);
        }
        return return__;
    }
```

Listing 3.9: Continuous write buffer, get function

The get operation (like in the full synchronous buffer) waits on the condition variable for the buffer to fill up and then empties it and returns the element (Listing 3.9). Unlike the full synch buffer the continuous write buffer does not need to call *notify_one* in the *get* call because *put* does not wait for the buffer to clear prior to writing.

3.2.3 Continuous Read Buffer

Some applications (like time-triggered systems) require a continuous flow of data. Especially in control engineering controllers run best at a fixed frequency. In such cases it is better to reuse old data instead of waiting for new data to become available as that would violate the fixed frequency principle. This is where a continuous read buffer is used.

In a continuous read buffer the buffer is never cleared once it has been filled. The *put* operation replaces the current buffer contents with the new data while *get* always returns the latest data regardless of whether it was read before.

```
    void put(Type_ element) override {
        std::lock_guard<std::mutex> lock__(mtx_);
        element_ = element;
    }

    Type_ get(void) override {
        std::lock_guard<std::mutex> lock__(mtx_);
        return element_;
    }
```

Listing 3.10: Continuous read buffer: *put* and *get* calls

The continuous read buffer takes a default value to be stored in the buffer until it is overwritten by the first *put* call. The put and get functions are just normal getter and setter calls in this implementation (Listing 3.10).

It is necessary to use a lock guard in this context because reading from or writing to the buffer content is not necessarily an atomic operation. Especially with primitive data types this can be a massive loss in performance. Table 3.1 shows that over 80 per cent of the time spent on a *put* call were spent on locking and unlocking.

Operation	**Average time in ns**	**Average time in %**
locking	83	34.8
storing	44	18.6
unlocking	111	46.6

Number of test runs: **1000**

Table 3.1: Average timing of the *put* operation on a continuous_read_buffer<int>

Atomic

A faster way to implement the continuous read buffer is to use a lock free three buffer system as proposed by Reto Carrara [6]. While this data structure uses slightly more memory space than the normal blocking buffer, it significantly reduces access times on the buffer.

Carrara's design uses three buffers to store data in. One buffer is used to read data from, the other two are written to in turn. Whenever data has been written to one buffer it becomes the new read buffer. The only thing that is critical to be synchronised is the update of changes on which buffer is used for which purpose, called the read- and write *consensus*. This can be done atomically by using the **test and set** method [7]. Carrara uses system specific code to implement test and set which usually requires to descend to assembly code level. C++11's new std::atomic types offer an abstraction on these atomic operations which makes the test and set function easy to implement (Listing 3.11).

```cpp
std::atomic<bool> touched;
bool test_and_set(void) {
    return touched.exchange(true);
}
```

Listing 3.11: Test and set function using std::atomic

The rest of Carrara's code is system independent as it only uses standard C and C++ features. With the test and set function written in std::atomic code it will compile on all systems which support C++11. The disadvantage of this implementation is that it explicitly requires a one consumer, one producer environment. Especially multiple producers would make it impossible to find a global consensus. Carrara's atomic three buffer code can also be wrapped to satisfy the buffer interface (Listing 3.12). The rest of the code can be taken from Carrara's paper [6].

Comparing the two implementations of the continuous read buffer a significant increase in speed can be registered when using the atomic version (Table 3.2.3).

Buffer implementation	**Total *put* time in µs**	**Total *get* time in µs**
Blocking	10 025	10 017
Atomic	1 808	3 825

Number of put / get operations: **100 000**

Table 3.2: Comparison of blocking and atomic buffer access speeds

One major drawback of atomic code at this point is that it heavily relies on the use of raw pointers which modern C++ paradigms try to avoid. Except for an experimental version of the shared pointer using atomics smart pointers cannot be used in atomic lock-free operations [8].

The reason is that atomic types are only supported for primitive data types and not for classes or structs. A raw pointer however is a primitive data type as it can be collapsed to an integer value which is why atomic code usually performs the synchronisation-critical operations on pointers and leaves the complex operations which actually work on the data the pointers point to out of the atomic context.

```cpp
template<typename Type_>
class three_buffer : public buffer<Type_> {
public:
    explicit three_buffer(Type_ default_val)
        : buffer_{default_val}
        , data_{}
        , write_{data_}
        , read_{data_}
    {
    }

    void put(Type_ element) override {
        int last__ = write_.get_read_consensus();
        int last_write__ = write_.get_last_written();
        int index__ = permutator_[last__][last_write__];
        buffer_[index__] = element;
        write_.setLastWritten(index__);
    }

    Type_ get(void) override {
        return buffer_[read_.get_read_consensus()];
    }

private:
    const int permutator_[3][3]
        = { { 1, 2, 1 }, { 2, 2, 0 }, { 1, 0, 0 } };
    data_buffer<Type_> buffer_;
    consensus_data data_;
    write_consensus write_;
    read_consensus read_;
};
```

Listing 3.12: Three buffer wrapper implementation

3.3 Connector Based

Another option is to make the transaction between two ports connector based. In this scenario the connector runs in its own thread and collects the data from the out-port and then dispatches it to the in-port. This type of connector will be called **collector** here.

```
template <typename Type_>
class collector {
public:
    collector(collector_out_port<Type_>& source,
                collector_in_port<Type_>& destination)
        : source_{source}
        , destination_{destination}
    {
        source_.connect(_self);
        destination_.connect(_self);
    }

    ~collector() {
        source_.disconnect(_self);
        destination_.disconnect();
    }

    void transfer(void) {
        auto collect__ = source_.collect();
        destination_.store(collect__);
    }

private:
    collector_out_port<Type_>& source_;
    collector_in_port<Type_>& destination_;
    std::weak_ptr<collector<Type_>> _self;
};
```

Listing 3.13: Collector style connector

Because of the inverted flow of control the collector requires its own implementation of all three components (in-port, out-port and connector). The collector itself holds references to both ports it is connected to. In addition it contains a std::weak_ptr to itself (Listing 3.13). The reason for this is that an object must never contain a self-pointing shared pointer or its destructor would never be called. The workaround is to use a weak pointer which can be converted to a shared pointer when referenced from outside of the object. The weak pointer is used to connect to the ports and allow for the ports to address the collector as well as the collector can address the ports.

```cpp
template <typename Type_>
class collector_out_port {
public:
    explicit collector_out_port(const Type_& default_val)
        : collectors_{}
        , data_{default_val}
    {}

    Type_ collect(void) const {
        return data_;
    }

    void set_data(const Type_& data) {
        data_ = data;
    }

    void connect(std::weak_ptr<collector<Type_>> coll) {
        collectors_.push_front(coll.lock());
    }

    void disconnect(std::weak_ptr<collector<Type_>> coll) {
        collectors_.remove(coll.lock());
    }

    bool connected(void) const {
        return !collectors_.empty();
    }
private:
    std::list<std::shared_ptr<collector<Type_>>> collectors_;
    Type_ data_;
};
```

Listing 3.14: Collector style out-port

The out-port contains a list of shared pointers to the connected collectors (Listing 3.14). These pointers are created from weak pointers passed by the collector and transformed into std::shared_ptr s forming a strong reference to the collector. The in-port only has a single shared pointer pointing to the collector (Listing 3.15) as there can only be one connection to an in-port. The collector collects the data from the out-port and transfers it to the in-port using the *transfer* function.

```
class collector_in_port {
public:
    explicit collector_in_port(Type_& default_val)
        : data_{default_val}
        , collector_{nullptr}
    {}

    void store(Type_& data) {
        data_ = data;
    }

    Type_ get_data(void) const {
        return data_;
    }

    void connect(std::weak_ptr<collector<Type_>> coll) {
        collector_ = coll.lock();
    }

    void disconnect(void) {
        collector_ = nullptr;
    }

    bool connected(void) const {
        return collector_ != nullptr;
    }
private:
    Type_ data_;
    std::shared_ptr<collector<Type_>> collector_;
};
```

Listing 3.15: Collector style in-port

3.4 Task Based

All of these different types of connections can be used in a task based system. Task based systems rely on splitting the programme into little chunks of code which logically must be executed in sequence. These tasks are then sent to a **thread pool** and executed at a future time. The implementation of this thread pool and task class are described in Chapter 11.4.

The pipe communication works best with a task based system and will be used as an example in this chapter.

In order to get the pipe to work in a task based manner a new pipe class has to be created. It can be derived from the standard pipe and connected to the usual ports. The constructor must now take a reference to the system's thread pool and the *activate* function creates a task containing the actual transfer and adds it to the thread pool (Listing 3.16). Because the thread pool class is non-copyable a reference wrapper has to be used instead of a normal reference.

```
template <typename Type_>
class task_pipe : public pipe<Type_> {
    std::reference_wrapper<thread_pool<>> pool_;

public:
    task_pipe(in_port<Type_>& destination,
            thread_pool<>& pool)
        : pipe<Type_>{destination}
        , pool_{pool}
    {}

    void activate(Type_ element) override {
        pool_.get().add_task([&]{
            pipe<Type_>::get_destination()
                    .activate(element);
        });
    }
};
```

Listing 3.16: Pipe implementation using tasks

3.5 Summary

There are several different ways to achieve loosely coupled communication between two ports depending on the application and the concurrency model. The different ways to use the port and connector model can be seen in Table 3.3.

Connector Type	**in its own thread**	**potential data loss**	**potential duplicates**	**ports in different threads**
Pipe	no	implementation specific	implementation specific	usually not
Full Synchronous Buffer	no	no	no	yes
Continuous Write Buffer	no	yes	no	yes
Continuous Read Buffer	no	yes	yes	yes
Collector	yes	no	no	possibly

Table 3.3: Communication models using ports and connectors

4 Supervisor

A programme's life cycle can generally be divided into three sections: **initialisation**, **execution** and **termination** phase.

During initialisation the needed classes are set up, memory is allocated and connections are established. The execution phase is the main part of the programme in which the actual logic of the system is established. During the termination phase all connections must be closed, memory must be released so that the system can safely be shut down.

Especially during the initialisation and termination phases it is valuable to have a coordinating object which manages all these tasks. This object will be called a **supervisor**.

While the different components of the system all bring their own in- and out-ports as discussed in Chapter 3 the connections between these ports are not inherently created because that would violate the principle of loose coupling. Instead all connections are made by the supervisor during the initialisation phase and torn down during the termination phase. If the elements of a system can be predicted at compile-time the supervisor object can utilise the *static allocation pattern* [9, pp. 167-180] or *pool allocation pattern* to allocate memory beforehand during the initialisation phase.

Advanced Communication Models

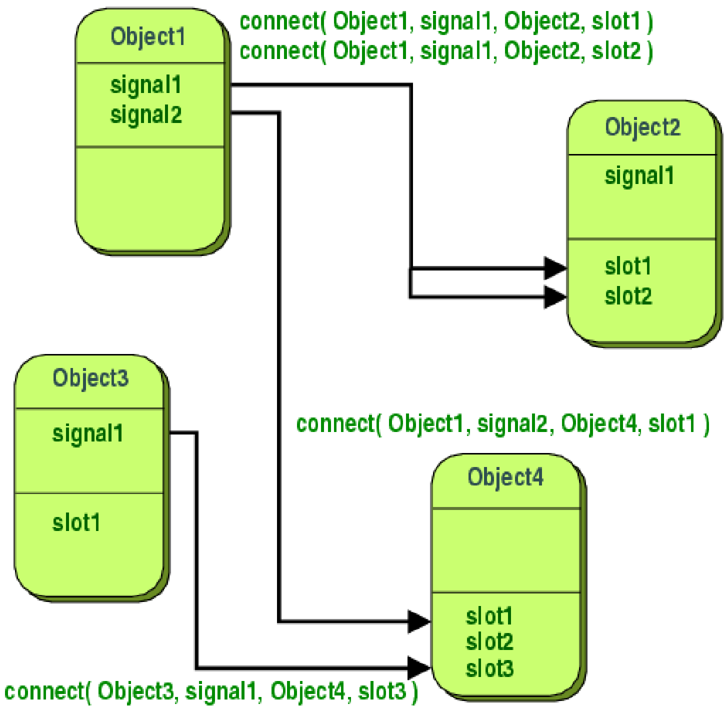
5 Signals and Slots

A special variation on the port approach is to use **signals and slots**. Signals and slots are a concept which was originally introduced as a feature in the C++ library Qt [10]. They facilitate an object-oriented type-safe callback mechanism.

5.1 Layout

A slot is a function object [11, p. 335] that holds a function with a specific signature. This function can be called directly or more typically from a signal object.

The signal is a generic manager object for slots. It can manage any number of slots and call them with the corresponding arguments.



Object1
signal1
signal2
connect(Object1, signal1, Object2, slot1)
connect(Object1, signal1, Object2, slot2)
Object2
signal1
slot1
slot2
Object3
signal1
slot1
connect(Object1, signal2, Object4, slot1)
Object4
slot1
slot2
slot3
connect(Object3, signal1, Object4, slot3)

Figure 5.1: Layout of signals and slots in Qt, source [10]

An object can contain any number of signals and slots, all with a well-defined purpose (Figure 5.1). Slots can connect to the signals of other objects and will then be called whenever the corresponding signal is activated.

5.2 Implementation

5.2.1 Signal

The signal class consists of a std::vector of references to slots. Since the std::vector collection cannot hold a direct reference to a class the std::reference_wrapper [12] is used (Listing 5.1).

```
    std::vector<std::reference_wrapper<slot<Args_...>>> slots_;
```

Listing 5.1: Signal class private member

The signal class has to be called **sig** because signal is a C keyword.

```
    virtual void connect(slot<Args_...>& sl) {
        slots_.push_back(std::reference_wrapper<slot<Args_...>>{sl});
        sl.connect(*this, true);
    }
```

Listing 5.2: Signal connect function

When connecting a signal to a slot a reference to the connected slot is added to the vector (Listing 5.2). The *connect* function of the slot class is also called because the slot has to know about the connected signal. The *connect* function of the slot class has to be called with an additional bool argument set to *true* which will be explained in Chapter 5.2.2. The disconnect function removes a slot from the vector. Since the order of the connected slots in the vector is irrelevant, the fastest way to remove an element is to first create an iterator using std::find which points to the slot to be removed, then *swap* it with the last element in the vector and then *pop* that last element (Listing 5.3). This way the vector does not have to be reordered.

```
    virtual void disconnect(slot<Args_...>& sl) {
        auto iterator__ = std::find(slots_.begin(), slots_.end(),
                std::reference_wrapper<slot<Args_...>>{sl});
        if(iterator__ != slots_.end())
            std::swap(*iterator__, slots_.back());
        slots_.back().get().disconnect();
        slots_.pop_back();
    }
```

Listing 5.3: Signal disconnect function

Instead of using the *std::weak_ptr* technique to connect the signal and slot to each other like in Chapter 3.3 this time a different solution is used. Instead of a smart pointer to the corresponding signal/slot the classes use references to each other. Since a reference can't be set to *nullptr* a custom *NULL type* has to be created for the signal. This is done by deriving from the sig class (Listing 5.4). A third way would be to use an option type which is not part of the C++ standard library but can be implemented as shown in Chapter 11.2.

```cpp
template <typename ... Args_>
struct null_signal : sig<Args_...> {
```

Listing 5.4: Signal NULL struct

The null_signal class uses a private constructor in combination with a static *get_null* function to supply its null type (Listing 5.5). This is similar to the *singleton pattern* [5, p. 127] but because of the way templates work in C++ a null signal will be created for every set of arguments. But since the null_signal struct is immutable and doesn't have any members this does not matter.

```cpp
    static null_signal<Args_...>& get_null(void) {
        static null_signal<Args_...> null_sig__;
        return null_sig__;
    }
```

Listing 5.5: Null signal getter function

The *null_signal* struct overrides all functions from the *sig* super class to throw exceptions when they are called (Listing 5.6).

```cpp
    void connect(slot<Args_...>& sl) override {
        throw "null signal cannot be connected";
    }
```

Listing 5.6: Null signal functions throw exceptions

The only function that can be called is the *is_null* function (Listing 5.7) which returns **true** when called from the null signal and **false** when called from the sig class (Listing 5.8).

```cpp
template <typename ... Args_>
bool null_signal<Args_...>::is_null(void) const {
    return true;
}
```

Listing 5.7: Null signal is_null function

This makes it easy to determine whether or not a signal is actually a null signal. The null signal can be applied to any sig variable because it is a subtype and can be used in a reference or std::reference_wrapper because is doesn't actually point to 0.

```cpp
template <typename ... Args_>
bool sig<Args_...>::is_null(void) const {
    return false;
}
```

Listing 5.8: Signal is_null function

5.2.2 Slot

The slot class holds a std::function and a reference pointing to the connected signal as the only private class members (Listing 5.9). Because of the way the *null_signal* was implemented in Chapter 5.2.1 the reference has to be formed with the std::reference_wrapper to work correctly.

```cpp
private:
    std::function<void(Args_...)> func_;
    std::reference_wrapper<sig<Args_...>> signal_;
```

Listing 5.9: Slot class private members

This function is initialised with the constructor argument as a std::function and the signal_ reference with the null signal (Listing 5.10).

```cpp
template <typename ... Args_>
class slot {
public:
    explicit slot(std::function<void(Args_...)> func)
        : func_{func}
        , signal_{null_signal<Args_...>::get_null()}
    {}
```

Listing 5.10: Slot class basic constructor

Initialising the slot class with a member function is more complicated. The approach used with the task class (chapter 11.4) using std::bind cannot be applied here. While std::bind can be used to partially bind parameters using std::placeholders [13], these place holders require the programmer to specify the exact number of arguments beforehand. There are different ways to solve this, explained in Chapter 11.1 from which the function *bind_function_to_object* (in any of the proposed implementations) can be used to bind the member function and create a std::function from it (Listing 5.11).

```
    template <typename Obj_, typename Func_>
    slot(Func_ (Obj_::*func)(Args_...), Obj_ & obj)
        : slot{bind_function_to_object(func, obj)}
    {}
```

Listing 5.11: Slot class member function constructor

This constructor works for non-const member function but must be overloaded to support cv-qualified [14] member functions (Listing 5.12).

```
    template <typename Obj_, typename Func_>
    slot(const Func_ (Obj_::*func)(Args_...), const Obj_ & obj)
        : slot{bind_function_to_object(func, obj)}
    {}
```

Listing 5.12: Slot class const member function constructor

To activate a slot the function call operator (Listing 5.13) is overloaded to launch the contained function with the provided arguments.

```
    inline void operator()(Args_... args) const {
        func_(args...);
    }
```

Listing 5.13: Slot class function call operator

The slot can connect to a signal using the *connect* function (Listing 5.14). The sig and slot classes are designed to allow connections from both sides, a signal can connect to a slot and a slot can connect to a signal with the same result. This requires the signal *connect* function to call *connect* on the slot and vice versa. To prevent infinite loops a bool argument "remote" (defaulted false) is added which states whether or not the function was called from the signal's *connect* function and end the loop.

```
    void connect(sig<Args_...>& sign, bool remote=false) {
        if(connected()) {
            signal_.get().disconnect(*this);
        }
        if(!remote) {
            sign.connect(*this);
        }
        signal_ = std::reference_wrapper<sig<Args_...>>(sign);
    }
```

Listing 5.14: Slots can connect to a signal

Equivalent to *connect* the *disconnect* function disconnects the slot from the signal. The function checks if the connected signal is null (Listing 5.15) and calls *disconnect* on the connected signal if it isn't.

```
    void disconnect(sig<Args_...>& sign) {
        if(!(sign.is_null()))
            sign.get().disconnect(*this);
    }
```

Listing 5.15: Slots can disconnect from a signal

Instead of using the method with the defaulted bool in the *connect* function the *disconnect* function is overloaded with a second function taking no arguments (Listing 5.16). This function is called from the signal class.

```
    void disconnect(void) {
        signal_ = null_signal<Args_...>::get_null();
    }
```

Listing 5.16: Slot's no-arg disconnect function

For the signal class to operate correctly the slot class needs to specify a custom equity operator (Listing 5.17). The reason is that the *disconnect* function removes the slot reference from a vector in the signal class which requires the slot classes to be comparable. The equity operator in this example simply evaluates that both slots point to the same place in memory.

```
    friend bool operator==(slot<Args_...>& sl1, slot<Args_...>& sl2)
    {
        return &sl1 == &sl2;
    }
```

Listing 5.17: Custom equity operator of the slot class

5.3 Summary

Signals and slots are type-safe callbacks which are wrapped in functor classes. Signals and slots can connect to each other which will make the signal call all the connected slots with their specific callback code. A big advantage of object-oriented callbacks like signals and slots is that they can easily be reused for many different purposes. C++ templates allow to specify the arguments to pass to the callback function which makes them even more versatile.

6 Channels

An advanced means for uncoupled communication can be achieved through **channels**. While the ports and connectors idea focuses on creating a distinct connection between two entities, the approach is converse with channels. A channel is a component that is completely detached from the programme logic. It is a synchronised queue that queues asynchronous data. Connectors never exist without their connected ports while channels exist *before* other components connect to them. A channel serves as a conduit between several senders and receivers and is not limited to a single sender.

6.1 Syntax

The syntax chosen for this implementation of channels has been loosely adapted from the **Go** programming language [15]. All operations on a channel are performed using the < < operator.

```
channel<int> ch(5); //channel which takes a maximum of 5 elements
ch << 12;           //add the number 12 to the channel
```

Listing 6.1: Channel: adding an element to a channel

Channels are created with a fixed maximum size (Listing 6.1). Elements are added to the channel with the < < operator which always points in the direction of the data flow.

```
int x;              //specify the target variable
x << ch;            //put the first element in ch in the variable x
```

Listing 6.2: Channel: retrieving an element from a channel

Retrieving an element from the channel works in the exact same way. The < < operator pops the first element from the channel and stores it in the variable (Listing 6.2).

```
channel<int> ch2(3);    //create another channel
ch2 << ch;              //add the first element in ch to ch2
```

Listing 6.3: Channel: transferring an element from one channel to another

The third operation on a channel is to transfer an element from one channel to another. This again is done by concatenating the two channels with < < (Listing 6.3).

6.2 Implementation

The channel requires semaphores to synchronise access and avoid over- and underflow. Since C++ does not offer an object-oriented semaphore a custom semaphore class has to be created (see Chapter 11.3).

The channel itself is implemented as a template class which takes the type to be stored as the template parameter (Listing 6.4).

```cpp
template <typename Type_>
class channel {
```

Listing 6.4: Channel template class

The channel class contains two semaphores to control the empty and full spaces in the channel, a mutex to synchronise access to the queue and the queue in which to store the actual data (Listing 6.5).

```cpp
private:
    semaphore sem_free_spaces_, sem_size_;
    std::queue<Type_> queue_;
    std::mutex mtx_;
```

Listing 6.5: Channel private members

The constructor (Listing 6.6) initialises one semaphore with zero and the other one with the maximum size of the channel.

```cpp
public:
    explicit channel(const size_t max_size)
        : sem_free_spaces_{max_size}
        , sem_size_{}
        , queue_{}
        , mtx_{}
    {}
```

Listing 6.6: Channel constructor

Internally there are two inline functions to enqueue (Listing 6.7) elements to and dequeue (Listing 6.8) elements from the queue.

```
    inline void enqueue(const Type_& element) {
        sem_free_spaces_.wait();
        mtx_.lock();
        queue_.push(element);
        mtx_.unlock();
        sem_size_.post();
    }
```

Listing 6.7: Channel enqueue function

These functions always increase the count of one semaphore and decrease the count of the other. The semaphore *sem_free_spaces_* counts the free spaces in the channel and blocks when the queue is filled making sure that no more than the specified number of elements can be stored. The semaphore *sem_size_* counts the elements currently in the queue and blocks on an empty queue. The mutex ensures that only one thread at a time can modify the queue.

```
    inline Type_ dequeue(void) {
        sem_size_.wait();
        mtx_.lock();
        auto return__ = queue_.front();
        queue_.pop();
        mtx_.unlock();
        sem_free_spaces_.post();
        return return__;
    }
```

Listing 6.8: Channel dequeue function

The *enqueue* and *dequeue* functions are called by the overloads of the < < operator to add data to the channel or retrieve it.

```
    void operator<<(const Type_& element) {
        enqueue(element);
    }
```

Listing 6.9: Data can be added to the channel using the < < operator

Elements can be enqueued with the function in Listing 6.9. The < < operator is used as a member function in this case and calls the enqueue function.

In the case of dequeuing an element from the channel the < < operator can not be implemented using a member function because the channel is on the right hand side (Listing 6.2). To solve this the operator has to be declared as a friend function of the channel class (Listing 6.10).

```
Type_ friend operator<<(Type_& target, channel<Type_>& source) {
    return target = source.dequeue();
}
```

Listing 6.10: Data can be retrieved from the channel using the < < operator

To allow a channel to pass data directly into another channel another friend overload of the < < operator has to be created. This function simply dequeues an element from the source channel and enqueues it to the destination channel (Listing 6.11).

```
void friend operator<<(channel<Type_>& target,
        channel<Type_>& source) {
    target.enqueue(source.dequeue());
}
```

Listing 6.11: Data can be passed directly from one channel to another

The channel also supplies a *size* function to check its current size (Listing 6.12). The function returns the current value of the semaphore *sem_size_* which counts the elements in the queue.

```
auto size(void) const {
    return sem_size_.get_value();
}
```

Listing 6.12: Channel size function

There is a problem with this implementation when terminating the system. Dequeuing threads will block on an empty channel and enqueuing threads will block on a full channel. If the system is shut down while there are still threads waiting on the channel dead-lock scenarios can arise. To avoid this the channel offers a *destroy* function which destroys the two semaphores and thus releases any waiting threads (Listing 6.13). This function should be called by the system's *supervisor* (Chapter 4) during the termination phase or by an encapsulating object in its destructor.

```
void destroy(void) {
    sem_free_spaces_.destroy();
    sem_size_.destroy();
}
```

Listing 6.13: Channel destroy function

6.3 Unlimited Channels

A variation on the standard channel is the **unlimited channel**. It behaves like the standard channel but does not have a limited size and thus never blocks on an enqueue operation. This can be useful when data comes in great bursts and many elements have to be added to a channel in a very short time before the consumer can dequeue and process them all. If these burst don't happen too frequently the data can still be processed in reasonable time but the channel will temporarily be filled with a lot of data which a standard limited channel would not allow. The implementation of an unlimited channel is very similar to the one of the standard channel except that it does not have the *sem_free_spaces_* semaphore which blocks on a filled queue.

6.4 Summary

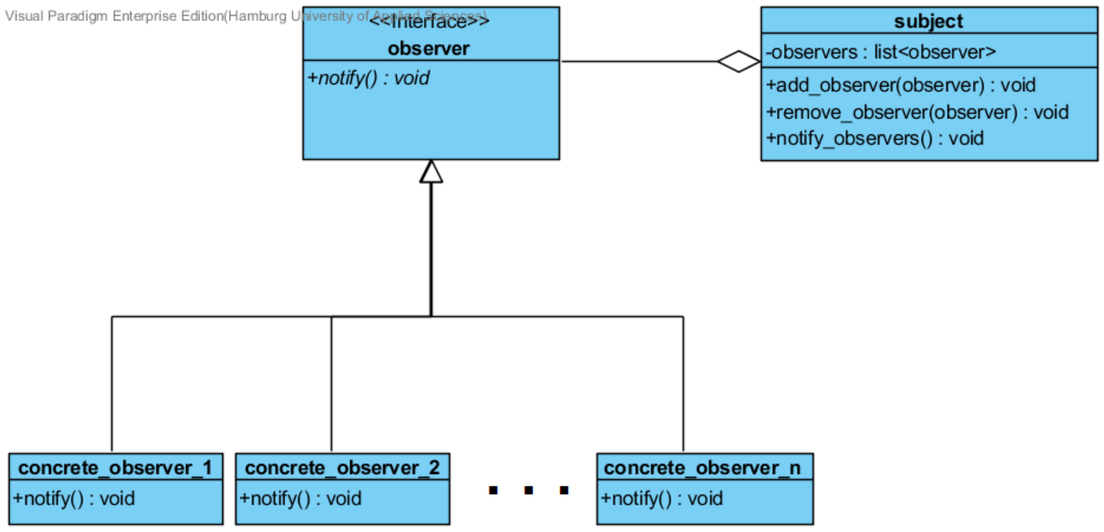
A channel is a synchronised queue which allows communication between different components of the system. The channel will work exactly the same way regardless of how many different components or threads use it.

Channel often have a limited capacity which makes them block all enqueuing threads when the maximum capacity is reached. They also block dequeuing threads when there is no data in the channel. Unlimited channels are a variation which does not have a maximum capacity and doesn't block enqueuing threads.

Impact on Design Patterns

7 Observer Pattern

The **observer pattern** is a very common design pattern for uncoupling communication. It can be used to make several objects (called observers) listen for changes on a specific object (called subject in [5, p. 293]). The observers all implement the same abstract interface which supplies a *notify* function which each observer must implement (Figure 7.1). The observers can register to the subject which will call *notify* on all the connected observers whenever the need arises.



Visual Paradigm Enterprise Edition(Hamburg University of Applied Sciences)
<<Interface>>
observer
+notify() : void
subject
-observers : list<observer>
+add_observer(observer) : void
+remove_observer(observer) : void
+notify_observers() : void
concrete_observer_1
+notify() : void
concrete_observer_2
+notify() : void
concrete_observer_n
+notify() : void

Figure 7.1: Observer pattern: class diagram

7.1 Observer Implementation Using Signals and Slots

The introduction of signals and slots (Chapter 5) allows to cut out the observer interface completely. Instead the observers can directly connect their custom defined *notify* functions directly to the subject slot. Besides greatly reducing boilerplate code a major advantage of using signals and slots is that it allows every subject to define its own template type for the notification if it is necessary to directly pass information to the observers.

```cpp
class subject {
public:
    subject()
        : topic1{}
        , topic2{}
        , topic3{}
    {}

    inline void update_topic1(int i) {
        topic1(i);
    }

    inline void update_topic2(std::string s) {
        topic2(s);
    }

    inline void update_topic3(double d1, double d2) {
        topic3(d1, d2);
    }

    sig<int>            topic1;
    sig<std::string>    topic2;
    sig<double, double> topic3;
};
```

Listing 7.1: Subject implemented using signals and slots

The subject can now offer different topics to observe in the form of signals as public members (Listing 7.1). If desired the subject class can utilise *register* and *unregister* functions to manage the access to the topics (which can then be private) and to update the topics like it is done in the example in Listing 7.1. Any class interested in these topics can now easily register to these topics by connecting their slots to the supplied signals.

The observer class can contain several slots for handling different signals (Listing 7.2). When using the standard observer implementation this would have required an abstract observer interface for each handler or to create a separate observer class and have the main class instantiate it several times.

```cpp
class observer {
public:
    observer()
        : handler1{[](std::string s){
                std::cout << s << std::endl;
            }}
        , handler2{&observer::print_sum, *this}
    {}

    void print_sum(double d1, double d2) {
        std::cout << (d1 + d2) << std::endl;
    }

    slot<std::string>    handler1;
    slot<double, double> handler2;
};
```

Listing 7.2: Observer implemented using signals and slots

Listing 7.3 shows an example of what the usage of a signal and slot based observer might look like.

```cpp
    subject s;
    observer obs;

    obs.handler1.connect(s.topic2);
    obs.handler2.connect(s.topic3);

    s.update_topic2("notified!");
    // prints:  notified!

    s.update_topic3(5.0, 2.5);
    // prints:  7.5
```

Listing 7.3: Usage of signal and slot based observer pattern

7.2 Publish and Subscribe

[5, p. 293] states **Publish-Subscribe** as an alternate name for the observer pattern. Several other source [16] [17] use the name publish-subscribe for an asynchronous version of the observer pattern. The latter definition will be used here. In [17, pp. 341] the publish and subscribe variation is proposed using an *event channel* to mediate between the subject (called publisher) and the observer (called subscriber). This channel uses its own publisher and subscriber and thus decouples the communicating objects further (Figure 7.2).

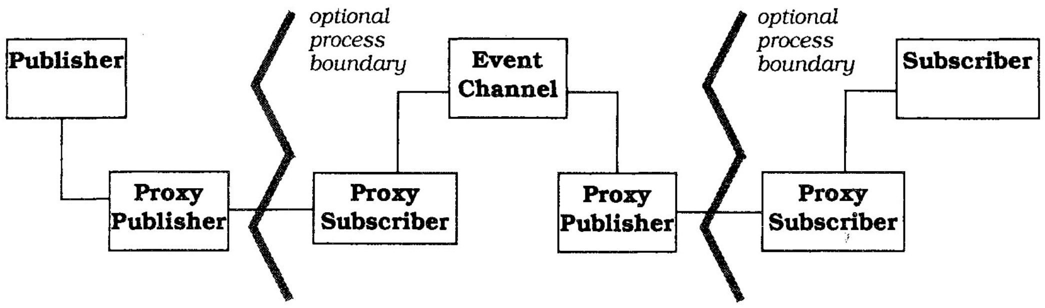


Figure 7.2: Uncoupled publisher-subscriber using an event channel, source: [17]

7.2.1 Publish-Subscribe Channel

Using signals and slots as publishers and subscribers and a channel between them to demultiplex the communication a publish-subscribe channel can be created.

```
private:
    channel<Type_> channel_;
    sig<Type_> publisher_;
    slot<Type_> subscriber_;
    std::mutex mtx_;
```

Listing 7.4: Publish-subscribe channel: private members

The implementation requires a signal serving as a proxy publisher, a slot as a proxy subscriber and a channel to store and transfer the data between them (Listing 7.4). Additionally a mutex has been added to synchronise operations. This is optional because synchronisation could be realised outside the publish-subscribe channel.

```
    void connect_publisher(sig<Type_>& sign) {
        std::lock_guard<decltype(mtx_)> lock__(mtx_);
        subscriber_.connect(sign);
    }

    void disconnect_publisher(sig<Type_>& sign) {
        std::lock_guard<decltype(mtx_)> lock__(mtx_);
        subscriber_.disconnect(sign);
    }
```

Listing 7.5: Connect and disconnect publisher and pub-sub channel

When a publisher connects to a pub-sub channel it effectively connects to the internal slot (Listing 7.5). The slot transfers the data to the channel and calls the internal signal to distribute the latest data in the channel (Listing 7.6).

```
    void transfer(Type_ arg) {
        channel_ << arg;
        Type_ event__;
        event__ << channel_;
        std::lock_guard<decltype(mtx_)> __lock(mtx_);
        publisher_(event__);
    }
```

Listing 7.6: Transfer function to activate the channel

In order to receive the data from the publisher subscribers must subscribe to the channel (Listing 7.7) which will connect them to the internal signal.

```
    void subscribe(slot<Type_>& sl) {
        std::lock_guard<decltype(mtx_)> __lock(mtx_);
        publisher_.connect(sl);
    }

    void unsubscribe(slot<Type_>& sl) {
        std::lock_guard<decltype(mtx_)> __lock(mtx_);
        publisher_.disconnect(sl);
    }
```

Listing 7.7: Subscribers can subscribe to the channel

7.2.2 Asynchronous Publish-Subscribe Channel

The publish-subscribe channel still has to wait for very subscriber to finish its process before it can continue which can be problematic in systems where the subscribers can take a long time to finish their task or in distributed systems where a fully synchronous approach would mean that several computers would spend a lot of time doing nothing.

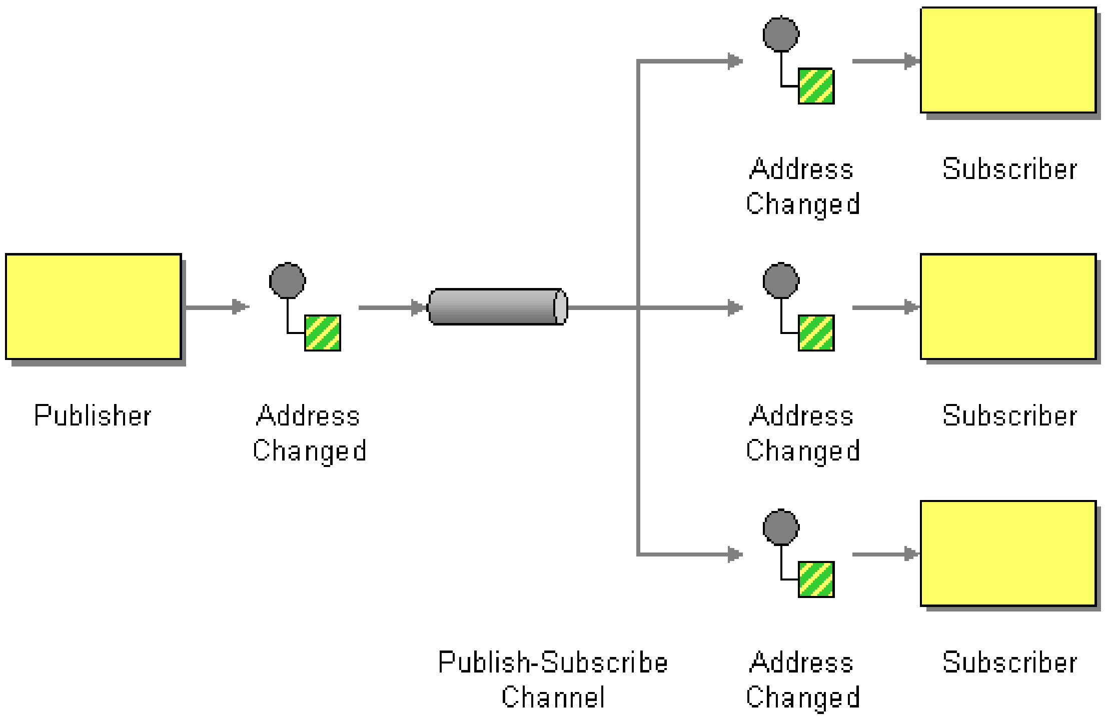


Figure 7.3: Uncoupled publisher-subscriber using separate output channels, source: [18]

This can be solved by creating "one input channel that splits into multiple output channels, one for each subscriber" [16, p. 107]. The contents of the input channel are duplicated for every output channel which allows the subscribers to dequeue and process the data from the channels at their own speed without slowing down other processes (Figure 7.3).

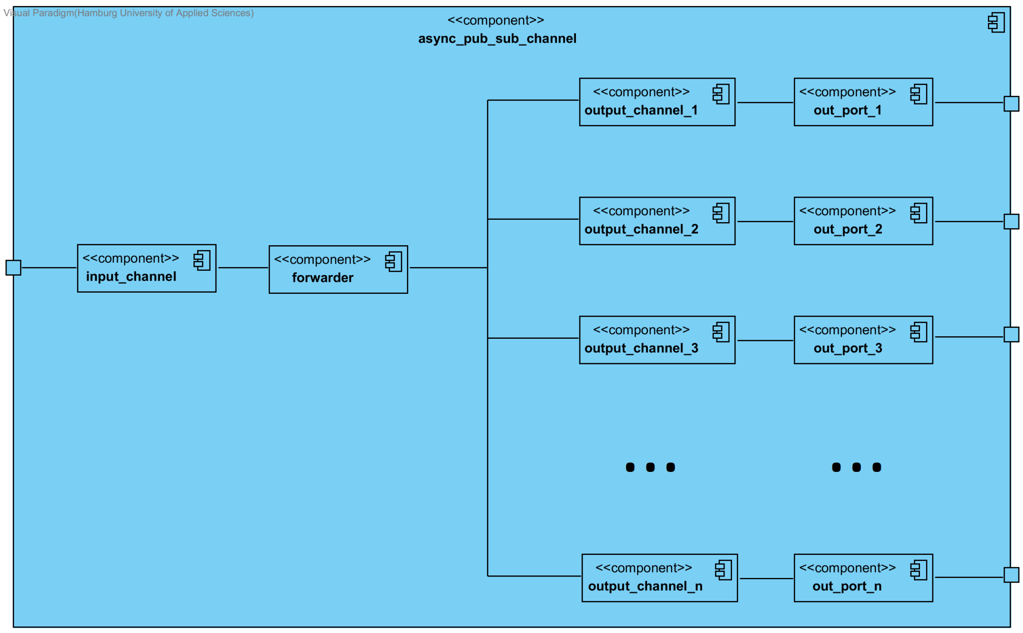


Figure 7.4: Asynchronous publish-subscribe channel: component diagram

The asynchronous publish-subscribe channel consists of multiple components (Figure 7.4):

- an input channel which publishers can add their data to
- a forwarder function which takes the data from the input channel and duplicates it for the output channels
- several output channels, one for each subscriber
- out-ports which the subscribers can connect to

The publishers in this implementation do not need to register to the channel but can simply publish their data on the channel. This allows the channel to be used by any number of publishers at once (Figure 7.5).

The asynchronous publish-subscribe channel is again implemented as a template class (Listing 7.8). The input channel is a normal channel from Chapter 6. The subscribers subscribe to the pub-sub channel with an in-port (Chapter 3). In order to store references to these ports they are mapped to the out-ports created by the channel (Listing 7.10).

```
template <typename Type_>
class async_pub_sub_channel {
```

Listing 7.8: Asynchronous publish subscribe channel template class

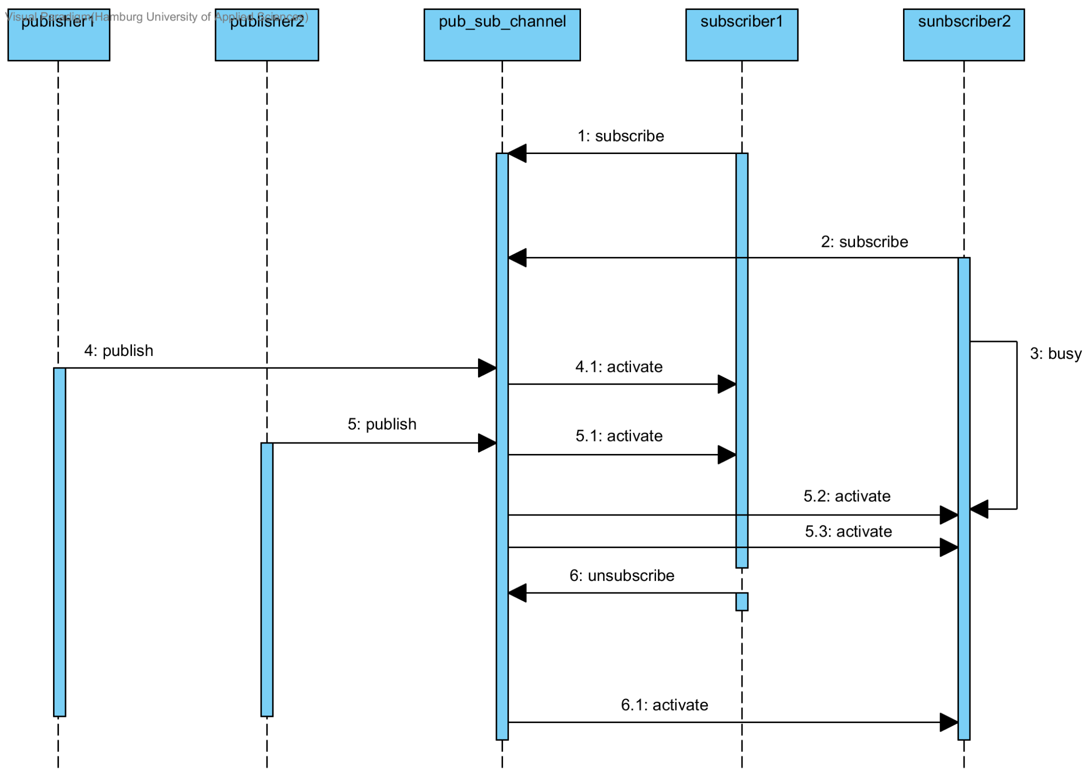


Figure 7.5: Asynchronous publish-subscribe channel: sequence diagram

The asynchronous pub-sub channel could be implemented using threads for forwarding the data to the subscribers. But since the number of threads in a system is limited and large numbers of threads are inefficient because of scheduling overhead, for this implementation a (better scaling) task based approach has been chosen. For that a thread pool is required which is passed as a reference in the constructor (Listing 7.9).

```
    async_pub_sub_channel(size_t max_size, thread_pool<>& pool)
        : input_channel_{max_size}
        , out_ports_{}
        , pool_{pool}
        , ochann_mtx_{}
    {}
```

Listing 7.9: Asynchronous publish subscribe channel: constructor

The connected subscribers are stored in a *std::unordered_map* [19] and mapped to the corresponding out-ports. An unordered map requires a hash function and an equity function for the keys. These can either be defined as a standard for the class which will be used globally or it can be passed to the map as a functor object. To allow for more flexibility the second approach is chosen here (Listing 7.10).

```
private:
    channel<Type_> input_channel_;
    typedef std::reference_wrapper<in_port<Type_>> _InportRef;
    std::unordered_map<_InportRef
        , pub_sub_out_port<Type_>
        , in_port_hash<Type_>
        , in_port_equals<Type_>> out_ports_;
    std::reference_wrapper<thread_pool<>> pool_;
    std::mutex ochann_mtx_;
```

Listing 7.10: Asynchronous publish subscribe channel: private class members

The equity function object is implemented to compare the id (Chapter 3.1) of the in-ports (Listing 7.11). Alternatively it could compare the addresses and check for identity.

```
template <typename Type_>
struct in_port_equals {
    auto operator()(const in_port<Type_>& in1
            , const in_port<Type_>& in2) const {
        return in1.get_id() == in2.get_id();
    }
};
```

Listing 7.11: Equity functor for the in-port class

The hash function object has to be implemented accordingly which in this case means returning the in-port id as the hash (Listing 7.12).

```
template <typename Type_>
struct in_port_hash {
    auto operator()(const in_port<Type_>& in) const {
        return (size_t) in.get_id();
    }
};
```

Listing 7.12: Hash functor for the in-port class

For the asynchronous publish-subscribe channel a specialised out-port has to be created. This out-port encapsulates a standard (pipe) out-port as described in Chapter 3.1 and the actual out-port channel.

It connects the out-port to the specified in-port (the subscriber) using the *task pipe* from Chapter 3.4. Activating the pub-sub out-port creates a task of forwarding an element in the channel to the in-port. To allow the asynchronous pub-sub channel the manipulation of its private members the *async_pub_sub_channel* is declared as a *friend* (Listing 7.13).

```
template <typename Type_>
class pub_sub_out_port {
    friend class async_pub_sub_channel<Type_>;
public:
    pub_sub_out_port(size_t max_size, in_port<Type_>& destination,
            thread_pool<>& pool)
        : port_{}
        , channel_{max_size}
    {
        task_pipe<Type_> pipe__(destination, pool);
        auto tp__ = std::make_unique<pipe<Type_>>(pipe__);
        port_.connect(std::move(tp__));
    }

    inline void activate(const Type_& element) {
        port_.activate(element);
    }

private:
    out_port<Type_> port_;
    channel<Type_> channel_;
};
```

Listing 7.13: Specialised sub-sub out-port

Subscribing to the channel works by passing the in-port by reference to the *subscribe* function. If the reference is not in the subscribers map yet an output channel and out-port is created.

In order to create the out-port in-place in the unordered map the *emplace* [20] call is used (Listing 7.14). Because of the non-unary constructor of the out-port class this has to be done using *std::forward_as_tuple* [21] and the *std::piecewise_construct* mechanism.

```
void subscribe(in_port<Type_>& in) {
    _InportRef in__{in};
    std::lock_guard<decltype(ochann_mtx_)> lock__(ochann_mtx_);
    auto iterator__ = out_ports_.find(in__);
    if(iterator__ == out_ports_.end()) {
        out_ports_.emplace(std::piecewise_construct
            , std::forward_as_tuple(in__)
            , std::forward_as_tuple(
                  input_channel_.max_size(), in, pool_)
        );
    }
}
```

Listing 7.14: Asynchronous publish subscribe channel: subscribe function

Unsubscribing from the channel works accordingly by removing the in-port reference from the unordered map. Before this can be done the output channel has to be destroyed (Listing 7.15). Otherwise potentially unprocessed data in the output channel would remain orphaned in memory.

```
void unsubscribe(in_port<Type_>& in) {
    _InportRef in__{in};
    std::lock_guard<decltype(ochann_mtx_)> lock__(ochann_mtx_);
    out_ports_[in__].channel_.destroy();
    out_ports_.erase(in__);
}
```

Listing 7.15: Asynchronous publish subscribe channel: unsubscribe function

The *publish* call allows any publisher to add data to the channel which will be published to all subscribers. If no subscriber is registered to the channel the data remains in the channel until there is at least one subscriber present (Listing 7.16). A task is created which uses the *forwarder* function (Listing 7.17) to distribute the data to the output channels.

```
void publish(Type_ data) {
    input_channel_ << data;
    if(!(out_ports_.empty())) {
        pool_.get().add_task(
          task{&async_pub_sub_channel<Type_>::forwarder, this});
    }
}
```

Listing 7.16: Asynchronous publish subscribe channel: publish function

```cpp
void forwarder(void) {
    Type_ element__;
    while(input_channel_.size() > 0) {
        element__ << input_channel_;
        std::lock_guard<std::mutex> lock__(ochann_mtx_);
        for(auto& port__ : out_ports_) {
            port__.second.activate(element__);
        }
    }
}
```

Listing 7.17: Asynchronous publish subscribe channel: forwarder function

7.3 Summary

The observer pattern is a versatile and widely spread communication design pattern. Through the use of the techniques and implementations demonstrated in this thesis it can be transferred to modern C++14 code and altered to support additional use cases. The different variations and their usage can be seen in Table 7.1.

Variation	**synchronous**	**multiple subjects**	**event queues**
Traditional (GoF) Observer	yes	no	none
Using Signals and Slots	yes	yes	none
Publish-Subscribe Channel	yes / no	yes	one
Async Pub-Sub Channel	no	yes	one per subscriber

Table 7.1: Overview of different observer implementations

8 Reactor Pattern

Another common communication design pattern is the **reactor pattern** [22, pp. 178]. Like the observer pattern (Chapter 7) it allows synchronous distribution of events. But contrary to the observer pattern the reactor design pattern aims at dispatching events system-wide through a single object called the *dispatcher*. Because events usually come from many different sources these sources have to connect and register to the reactor.

In order to separate the different event sources and dispatch the events individually and synchronously they are routed through a synchronous *event demultiplexer* (Figure 8.1).

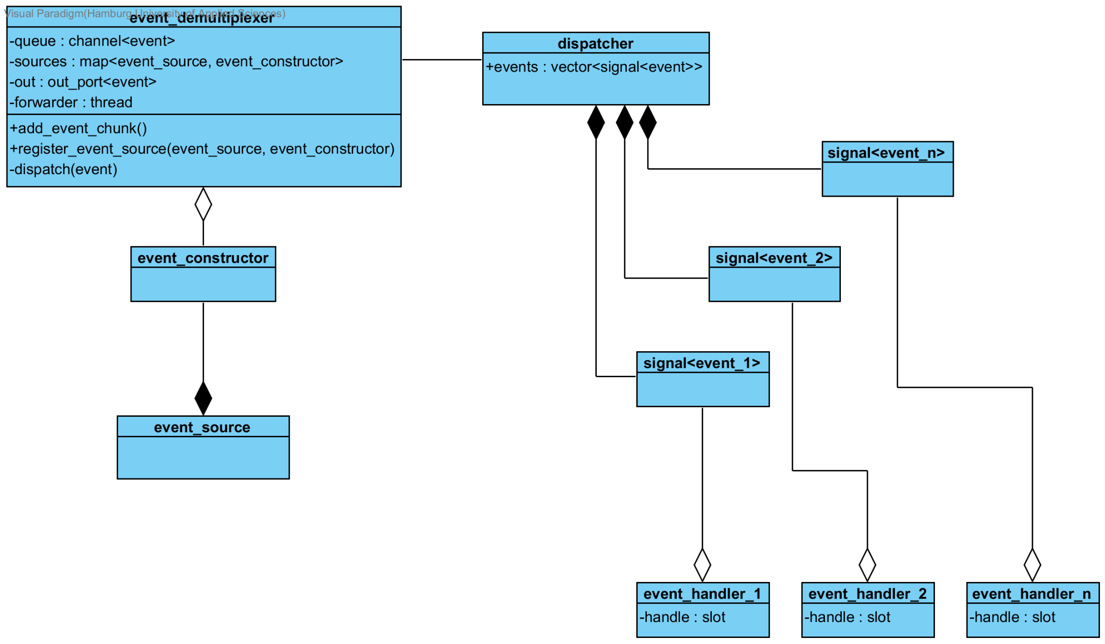


Figure 8.1: Reactor pattern: class diagram

The event demultiplexer is especially necessary when events or messages do not arrive in a whole but in little chunks. This can often be the case when receiving large messages byte-wise from a network socket. In this case an event constructor has to be defined for the event source (Listing 8.1).

```cpp
template <typename EventType_, typename ChunkType_>
struct event_constructor {
    virtual ~event_constructor() {}

    virtual optional<EventType_> construct(const ChunkType_&) = 0;
};
```

Listing 8.1: Event constructor interface

The event constructor will reconstruct the event from the individual chunks passed to it and return the constructed event when it is complete or an empty optional (Chapter 11.2) otherwise. An example implementation for creating an int from byte-chunks can be seen in Listing 8.2.

```cpp
typedef uint8_t _Byte;

struct byte_to_int : public event_constructor<int,_Byte> {
    byte_to_int()
        : chunks_{}
    {}

    optional<int> construct(const _Byte& chunk) override {
        chunks_.push_back(chunk);
        if(chunks_.size() < NUM_OF_CHUNKS)
            return {};
        int return__ = 0;
        for(size_t i=0; i<chunks_.size(); i++) {
            return__ |= chunks_[i] << (i<<3);
        }
        chunks_.clear();
        return optional<int>{std::move(return__)};
    }

    static constexpr size_t NUM_OF_CHUNKS = 4;
private:
    std::vector<_Byte> chunks_;
};
```

Listing 8.2: Event constructor example

8.1 Event Demultiplexer

The event demultiplexer is set up as a template class using the same template types as the event constructor (Listing 8.3). Variations to support different types of event templates would also be possible.

```
template <typename EventType_, typename ChunkType_>
class event_demultiplexer {
```

Listing 8.3: Event demultiplexer as a template class

The demultiplexer consists of a channel to forward the completed events to the dispatcher, a map which maps the event sources to the corresponding event constructors and an out-port which the dispatcher will be connected to (Listing 8.4).

```
private:
    typedef std::reference_wrapper<event_source<EventType_
            , ChunkType_>> _SourceRef;
    std::unordered_map<
        _SourceRef, event_constructor<EventType_, ChunkType_>
    > sources_;
    channel<EventType_> chann_;
    out_port<EventType_> out_;
    std::thread forwarder_;
```

Listing 8.4: Event demultiplexer: private members

Additionally the demultiplexer runs a thread which dequeues completed events from the channel and publishes them on the out-port (Listing 8.5).

```
    void forwarder_func(void) {
        EventType_ event__;
        while(true) {
            event__ << chann_;
            out_.activate(event__);
        }
    }
```

Listing 8.5: Event demultiplexer: event forwarder function

When an event source registers to the event demultiplexer a corresponding event constructor has to be specified. After checking for duplicates the source and constructor are added to the unordered map (Listing 8.6). Every source has its own event constructor stored in the demultiplexer's map and filled progressively with the event chunks.

```
    void register_source(
            const event_source<EventType_, ChunkType_>& source,
            const event_constructor<EventType_, ChunkType_>& cons) {
        _SourceRef src__{source};
        auto iterator__ = sources_.find(src__);
        if(iterator__ == sources_.end()) {
            sources_.emplace(std::piecewise_construct
                , std::forward_as_tuple(src__)
                , std::forward_as_tuple()
            );
        }
    }
```

Listing 8.6: Event demultiplexer: register function

Using the *add_event_chunk* function event sources can add their events piece by piece and the demultiplexer then adds them to the event constructor. Once the event is complete it is enqueued to the channel (Listing 8.7) and the forwarder thread will synchronously send it to the dispatcher (Listing 8.5).

```
    void add_event_chunk(
            const event_source<EventType_, ChunkType_>& source,
            ChunkType_ chunk) {
        _SourceRef src__{source};
        auto event__ = sources_[src__].construct(chunk);
        if(event__.has_content()) {
            chann_ << event__.get();
        }
    }
```

Listing 8.7: Event demultiplexer: adding an event chunk

8.2 Dispatcher

The dispatcher is the component of the reactor pattern that can be compared to the observer pattern's subject (Chapter 7). Its purpose is to supply the events that event handlers can register to and then call the registered handlers on the event's occurrence. The dispatcher is activated only by the event demultiplexer. For this it has to be connected to the demultiplexer's out-port.

Because activating the dispatcher's in-port has to trigger an immediate reaction (if it is not supposed to run in a separate thread) a custom in-port implementation is required.

The *forward in-port* can be seen as a very lightweight slot implementation (Chapter 5.2.2). It is derived from the standard in-port used with pipes (Chapter 3.1) but instead of storing data it forwards them directly to a function (Listing 8.8).

```cpp
template <typename Type_>
class forward_in_port : public in_port<Type_> {
public:
    explicit forward_in_port(std::function<void(Type_)> func)
        : func_{func}
    {}

    template <typename Obj_, typename Func_>
    forward_in_port(Func_ (Obj_::*func)(Type_), Obj_ & obj)
        : forward_in_port{bind_function_to_object(func, obj)}
    {}

    void activate(Type_ element) override {
        func_(element);
    }

    Type_ get_data(void) const = delete;
private:
    std::function<void(Type_)> func_;
};
```

Listing 8.8: Forward in-port class

The forward in-port class takes a lambda, function object or (member) function pointer as the constructor argument and collapses it to a std::function. The *activate* function is overridden to execute that function.

Because the *get_data* call used in the original in-port (Listing 3.2) is not required in the forward in-port it is explicitly deleted (Listing 8.8).

The dispatcher itself uses this forward in-port to connect to the event demultiplexer (Listing 8.9). It also has to define the available events and create the corresponding signals. This can either be done statically in the class itself (Listing 8.10) or it can be defined dynamically.

```
template <typename EventType_>
class dispatcher {
    forward_in_port<EventType_> in_;
public:
    template <typename ChunkType_>
    dispatcher(event_demultiplexer<EventType_, ChunkType_>& demux)
        : event1{}
        , event2{}
        , event3{}
        , in_{&dispatcher::dispatch, *this}
    {
        demux.out_.connect(in_);
    }
```

Listing 8.9: Dispatcher: initialisation

```
    static constexpr EventType_ EVENT_1(/*define*/);
    static constexpr EventType_ EVENT_2(/*define*/);
    static constexpr EventType_ EVENT_3(/*define*/);

    sig<EventType_> event1;
    sig<EventType_> event2;
    sig<EventType_> event3;
```

Listing 8.10: Dispatcher: event definition

The event handlers must all have a handler *slot* which implements the actual handle function. This slot can be connected to the corresponding signal of the dispatcher. The *dispatch* function which is called by the in-port chooses which event signal to activate (Listing 8.11).

```
    void dispatch(EventType_ event) {
        if(event==EVENT_1)
            event1(event);
        else if(event==EVENT_2)
            event2(event);
        else if(event==EVENT_3)
            event3(event);
        else
            throw "Unsupported";
    }
```

Listing 8.11: Dispatcher: dispatch call

8.3 Summary

The reactor pattern is used when different event handlers must handle events synchronously and the events are distributed through a single object (the dispatcher). If necessary an event demultiplexer can be used which demultiplexes overlapping incoming events from different sources and synchronously forwards them to the dispatcher.

Implementing the event handlers which traditionally is realised through a common abstract interface can be achieved easily through the use of signals (for the events) and slots (for the handlers) without the need for interface inheritance.

9 Pro-Actor Pattern

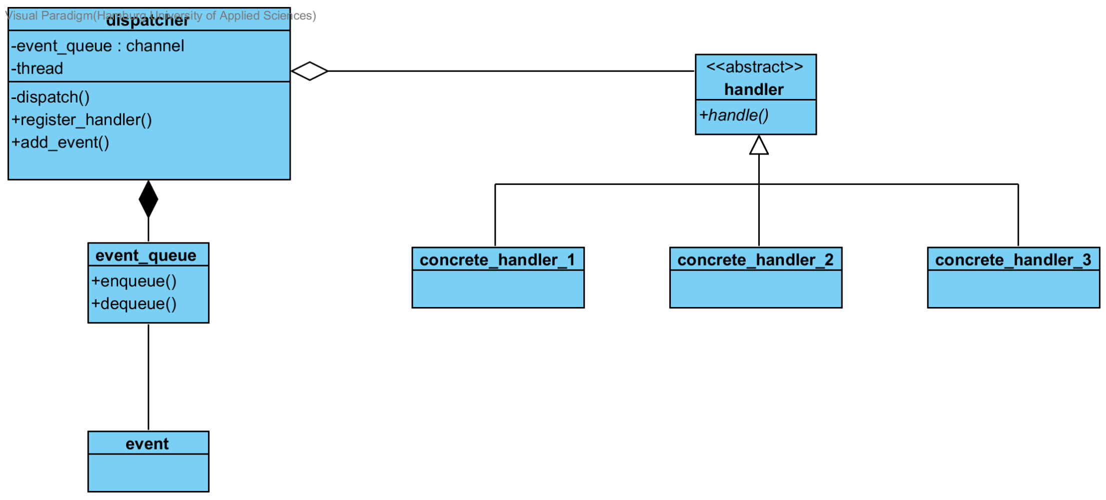


Figure 9.1: Pro-actor pattern: class diagram *adapted from [23]*

The pro-actor pattern is very similar to the reactor pattern (Chapter 8) except that is runs asynchronously instead of synchronously.

At the heart of the pro-actor pattern is the dispatcher just like it is in the reactor pattern. The pro-actor pattern runs a single thread but instead of using this thread to execute the different event handlers it just delegates the events to the handler and continues with the other handlers. This leaves the event handlers to execute their *handle* function themselves.

There are many variations to this pattern and different ways to implement its functionality:

9.1 Using a Buffer

The simplest way to implement the asynchronous nature of the pro-actor is to use a buffer as described in Chapter 3.2 to be positioned between the dispatcher and every event handler.

There are two drawbacks of this approach. The first is that this requires every event handler to run in a separate thread which is bad for large scaling systems.

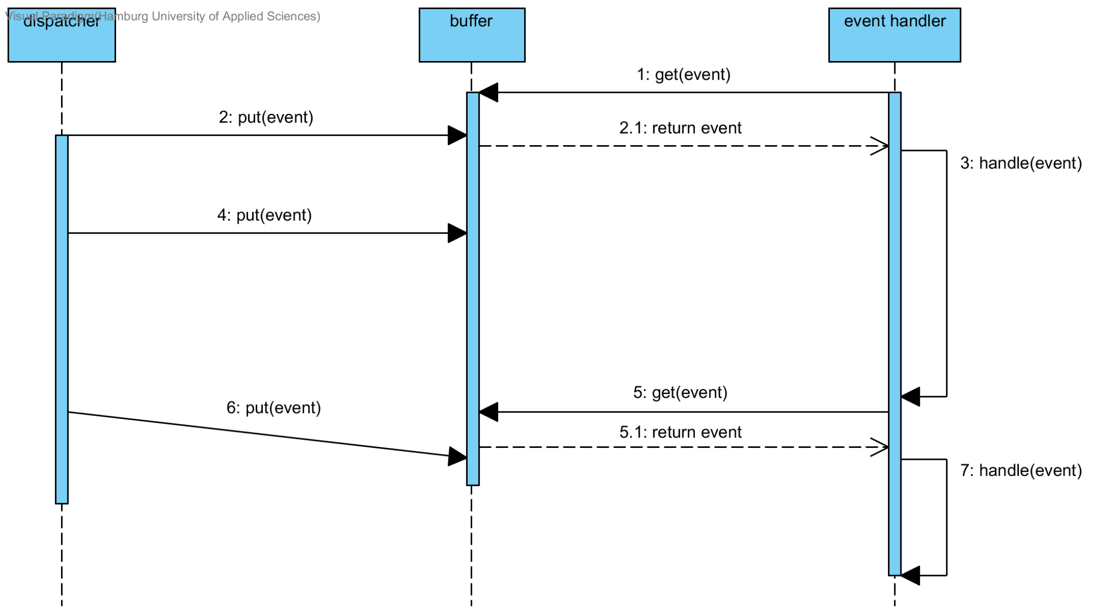


Figure 9.2: Pro-actor pattern using buffer: sequence diagram

The second is that this can still block the dispatcher if a blocking buffer (Chapter 3.2.1) is used. In Figure 9.2 the dispatcher blocks on the *put* call at **6** because the event is not processed yet. This can be avoided by placing a channel (Chapter 6) instead of a buffer. This approach is similar to the one used for the *asynchronous publish-subscribe channel* in Chapter 7.2.2.

9.1.1 Event Handler

The abstract handler interface (Listing 9.1) supplies the internal thread and a reference to the buffer (fully synchronous in this example) which will be supplied by the dispatcher.

```
struct handler {
    explicit handler(full_synch_buffer<event>& buff)
        : buff_{buff}
        , thread_{&handler::run, this} {}
protected:
    virtual void handle(event) = 0;
private:
    full_synch_buffer<event>& buff_;
    std::thread thread_;
```

Listing 9.1: Pro-actor pattern: buffer based handler interface

The thread runs the private *run* function which continuously pulls events from the buffer and passes them to the pure virtual handle function (Listing 9.2).

```
    void run(void) {
        while(true) {
            auto event__ = buff_.get();
            handle(*event__);
        }
    }
```

Listing 9.2: Pro-actor pattern: buffer based handler interface, run function

The concrete handlers then derive from the abstract handler and must also delegate their constructor to the superclass. The pure virtual *handle* call must be overridden (Listing 9.3) and will automatically be called by the thread encapsulated in the abstract superclass.

```
class concrete_handler final : private handler {
public:
    explicit concrete_handler(full_synch_buffer<event>& buff)
        : handler{buff}
    {}

private:
    void handle(event e) override {
        // implement handling
    }
};
```

Listing 9.3: Pro-actor pattern: buffer based event handler

9.1.2 Dispatcher

The dispatcher in this implementation holds a std::unordered_map like it already did in the reactor implementation (Chapter 8), an unlimited channel which events can be added to and a thread which dequeues the events from the channel and passes them to the event handlers (Listing 9.4).

```
private:
    std::thread dispatcher_thread_;
    unlimited_channel<event> event_queue_;
    std::unordered_map<
        event
        , std::vector<buffer<event>>
    > handlers_;
```

Listing 9.4: Pro-actor pattern: buffer based dispatcher, private members

In order to fill the unordered map with events the *add_event* function must be called (Listing 9.5). This function scans the unordered map for the specified event and emplaces an empty std::vector of buffers if the event key is not already present.

```
void dispatcher::add_event(event e) {
    auto iterator__ = handlers_.find(e);
    if(iterator__ == handlers_.end()) {
        handlers_.emplace(std::piecewise_construct
            , std::forward_as_tuple(e)
            , std::forward_as_tuple()
        );
    }
}
```

Listing 9.5: Pro-actor pattern: buffer based dispatcher, adding events

Event handlers don't register to the dispatcher directly but can order the buffer from which to retrieve the events by using the *register_handler* call (Listing 9.6). This will create a buffer in the vector the event points to and return it to the caller which can then pass it to the constructor of the handler (Listing 9.1).

```
buffer<event>& dispatcher::register_handler(event e) {
    if(handlers_.find(e) == handlers_.end())
        throw "Unknown event!";
    full_synch_buffer<event> buff__;
    handlers_[e].push_back(buff__);
    return buff__;
}
```

Listing 9.6: Pro-actor pattern: buffer based dispatcher, register a handler

An event can be added to the process by using the *activate* function (Listing 9.7). This will enqueue the event to the event queue to be distributed by the dispatcher.

```
void dispatcher::activate(event e) {
    event_queue_ << e;
}
```

Listing 9.7: Pro-actor pattern: buffer based dispatcher, activating an event

The dispatcher's thread then processes the events present in the queue one by one, blocking on an empty queue (Listing 9.8).

```
void dispatcher::process_queue(void) {
    event event__;
    while(true) {
        event__ << event_queue_;
        dispatch(event__);
    }
}
```

Listing 9.8: Pro-actor pattern: buffer based dispatcher, event processing thread

The events are dispatched to the registered handlers through the previously created buffer. The full synchronous buffers in the example in Listing 9.9 use unique pointers to their data so the events must be *moved* into the buffer.

```
void dispatcher::dispatch(event e) {
    for(auto buff__ : handlers_[e]) {
        auto event__ = std::make_unique<decltype(e)>(e);
        buff__.put(std::move(event__));
    }
}
```

Listing 9.9: Pro-actor pattern: buffer based dispatcher, dispatching events

9.2 Using Tasks

The task based approach is similar to the buffer approach but instead of placing a buffer between the dispatcher and the handler, the abstract handler base class defines a public *handle* function which creates a task from the pure virtual *handle_internal* function (Listing 9.10). The abstract handler base class uses a thread pool (Chapter 11.4) to handle the tasks so it takes this thread pool by reference to add the tasks to.

```cpp
struct handler {
    explicit handler(thread_pool<>& pool)
        : pool_{pool}
    {}

    virtual ~handler() {}

    void handle(event e) {
        pool_.get().add_task(&handler::handle_internal, this, e);
    }

protected:
    virtual void handle_internal(event) = 0;
private:
    std::reference_wrapper<thread_pool<>> pool_;
};
```

Listing 9.10: Pro-actor pattern: task based handler interface

The concrete handler must override the *handle_internal* function which is packaged into a task by the base class. The actual event handling is implemented in this function (Listing 9.11).

```cpp
class concrete_handler final : private handler {
public:
    explicit concrete_handler(thread_pool<>& pool)
        : handler{pool}
    {}

private:
    void handle_internal(event e) override {
        // implement handling
    }
};
```

Listing 9.11: Pro-actor pattern: task based event handler

The use of tasks is this context can cause a problem: because tasks should not contain any blocking elements as that would potentially harm the thread pool it must always be possible to execute them in parallel. This means that they can not depend on the state of the event handler. The event handler therefore should not be stateful or it might lead to undefined behaviour. This problem can be circumvented by using the variant described in Chapter 9.3.

9.3 Using std::async

C++11 offers another built-in task implementation called **std::async**. Async tasks are executed automatically on either a new thread or an existing system thread. The decision which of the two options is to be used is made within the C++ standard library hidden inside the std::async context [24, p. 228]. A std::async returns a **std::future** (or std::shared_future) which contains the results of the operation defined in the asynchronous task.

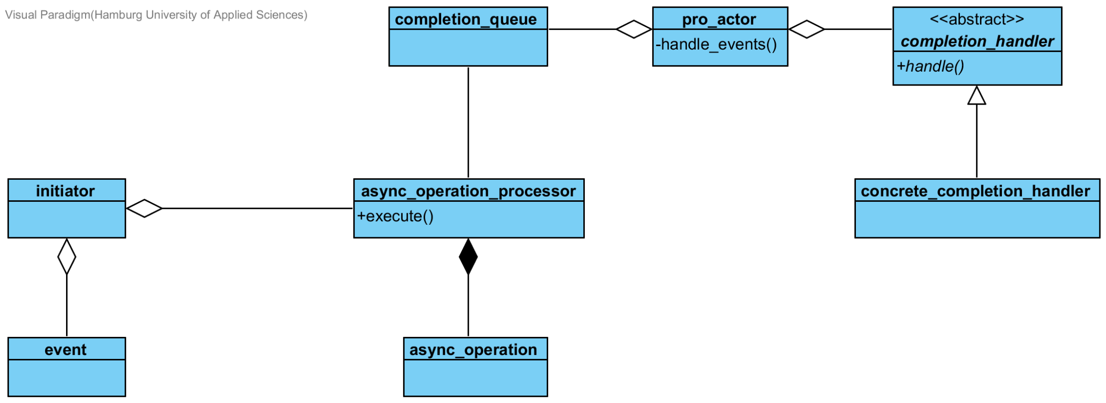


Figure 9.3: Pro-actor pattern using std::async: class diagram

Event handling benefits from the pro-actor pattern especially when a complex operation can be split into a (usually time-consuming) asynchronous part and a (faster) synchronous part. This is often the case when calculating complex results and writing them to disk. If the calculation is parallelisable it can be split into several tasks to be executed in parallel. Only writing to disk needs to be executed synchronously using the results of the asynchronous operations.

The setup proposed for the pro-actor pattern proposed by [22, p. 223] can be modelled well using std::async and std::future (Figure 9.3). In this implementation the queue is moved behind the actual event handling and only the synchronous part of the handling which is done by a completion handler is executed in order by dequeuing handled events from the completion queue.

The participants in this pro-actor setup are listed in Table 9.1. The main idea is to receive synchronous results from asynchronous operations which is done by putting a synchronised queue between the asynchronous operation and the completion handler which is then called synchronously. The final results of a handled event always end up with the pro-actor class (Figure 9.4). A valid variation would be to add another dispatcher which distributes the results from the pro-actor.

Component	Purpose
Initiator	initiates the handling of events
Asynchronous Operation Processor	creates asynchronous operations and enqueues the results in the completion queue
Asynchronous Operation	asynchronous part of the event handling. Result is returned to the asynchronous operation processor
Completion Queue	stores the results of the asynchronous operations to be completed synchronously
Pro-Actor	dequeues results from the completion queue and sends them to the completion handler
Completion Handler	synchronously completes the handling of the events and returns the results to the pro-actor

Table 9.1: Components of the pro-actor pattern adapted from [22]

9.3.1 Initiator

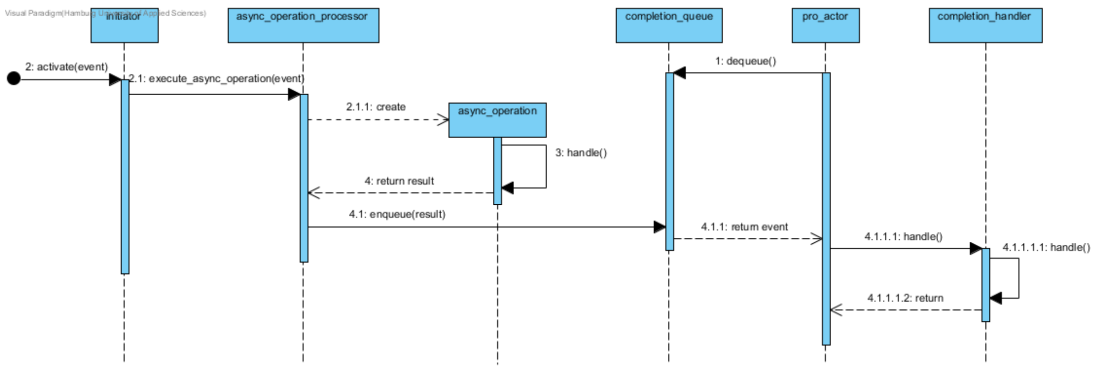


Figure 9.4: Pro-actor pattern using std::async: sequence diagram

The initiator class initiates the event handling process. It contains a reference to the asynchronous operation processor and a map which maps the possible event types to the corresponding asynchronous operations (Listing 9.12). In this example a std::map is used which requires a comparator to be specified for the key type (event in this case). Using a std::unordered_map would be another possibility.

```
private:
    async_operation_processor& processor_;
    std::map<event, async_operation, event_comparator> operations_;
```

Listing 9.12: Pro-actor pattern: initiator class, private members

The initiator's interface consist of a single function *activate* which is overloaded to take either a const reference or an rvalue reference to an event (Listing 9.13). The activate call takes the corresponding operation for the event from the map and passes them to the asynchronous operation processor.

```
    void activate(const event& e) {
        processor_.execute(operations_[e], e);
    }

    void activate(event&& e) {
        processor_.execute(operations_[e], std::forward<event>(e));
    }
```

Listing 9.13: Pro-actor pattern: initiator class, activate function

9.3.2 Asynchronous Operation Processor

The asynchronous operation processor is the component which launches the asynchronous operations and stores their results in the completion queue. The completion queue in this instance is simply represented by an unlimited channel (Chapter 6.3) which the async operation processor holds a reference to (Listing 9.14).

```
private:
    std::reference_wrapper<
        unlimited_channel<completion_event>> completion_queue_;
```

Listing 9.14: Pro-actor pattern: asynchronous operation processor class, private members

To start an asynchronous operation the *execute* function must be called on the asynchronous operation processor. This function expects two arguments: the asynchronous operation in the form of a std::function<completion_event(event)>and the event itself. The *execute* function is overloaded to support perfect forwarding (Listing 9.15). The asynchronous operation is defined as a standard function in this context which will allow the use of any function or functor object which meets the requirements. To use members functions the technique from Chapter 11.1 can be used. If desired it is of course possible to wrap the asynchronous operation in a separate class.

The *execute* call wraps the function passed as an argument into a lambda expression which also enqueues the result to the completion queue. This lambda is called from a std::async which is launched with the option *std::launch::async* which ensures that the operation is started immediately. The function returns after launching the operation but does not wait for its completion.

```
    void execute(std::function<completion_event(event)> async_op,
            event&& e) {
        std::async(std::launch::async, [&]{
           completion_queue_.get()<<async_op(std::forward<event>(e));
        });
    }

    void execute(std::function<completion_event(event)> async_op,
            const event& e) {
        std::async(std::launch::async, [&]{
           completion_queue_.get() << async_op(e);
        });
    }
```

Listing 9.15: Pro-actor pattern: asynchronous operation processor class, execute function

However this does not work correctly in C++14 as explained in Chapter 9.3.5.

9.3.3 Pro-Actor

The pro-actor is the central component of the pro-actor pattern. It runs in its own thread which continuously dequeues completion events from the completion queue (Figure 9.4) and passes them to the completion handler.

The pro-actor class needs references to both the completion queue and the completion handler. In addition it holds the thread which manages the execution of completion events (Listing 9.16).

```
private:
    std::reference_wrapper<
        unlimited_channel<completion_event>> completion_queue_;
    completion_handler& completion_handler_;
    std::thread thread_;
```

Listing 9.16: Pro-actor pattern: pro-actor class, private members

The thread runs the *handle_events* function which can be seen in Listing 9.17. It continuously and synchronously dequeues from the completion queue and then passes the events to the completion handler. Since this is the only place the completion events are actually needed, they can be passed as rvalue references using std::move.

```
    void handle_events(void) {
        completion_event event__;
        while(true) {
            event__ << completion_queue_.get();
            completion_handler_.handle(std::move(event__));
        }
    }
```

Listing 9.17: Pro-actor pattern: pro-actor class, handle_events function

Because this synchronous handling of completion events can potentially form a bottleneck if the synchronous operations are slow and long-duration it is better to use the *unlimited channel* as a completion queue in this context. Many asynchronous operations may finish at the same time and a limited channel would then block them which is against the idea of asynchronousity. But if synchronous operations (completion handling) become to expensive the pro-actor pattern is the wrong pattern to use as it profits from the parallel execution of asynchronous operations. In this case it should be resorted to the reactor pattern (chapter 8).

9.3.4 Completion Handler

```
struct completion_handler {
    virtual ~completion_handler() {}
    virtual void handle(completion_event&&) = 0;
};
```

Listing 9.18: Pro-actor pattern: completion handler interface

The completion handler interface offers the pure virtual *handle* function which takes an rvalue reference to a completion event (Listing 9.18). Depending on the implementation and the event types used it can be possible to handle all events with the same completion handler or it might be necessary to define individual ones.

The pro-actor class implementation in Chapter 9.3.3 only uses a single reference to a completion handler instead of maintaining a list of handlers. The reason is that this keeps the pro-actor as simple as possible. If multiple completion handlers are required they can easily be chained using the **chain of responsibility** pattern as described by [5, pp.223-228]. This pattern really benefits from using move semantics as following the chain of handlers would normally result in a lot of copies.

9.3.5 Note: Problems and Deprecation

While it offers a lot of new possibilities the use of std::async is still error prone. Especially calling a std::future's destructor can cause problems which have led to discussions of deprecating std::async [25]. While no conclusion has been reached on this topic as yet, at least changes to the way std::async and std::future work are very likely.

The implementation shown in Listing 9.15 does not work in the desired way. Because the std::future destructor blocks if *get* has not been called. This means that *execute* cannot return until the asynchronous operation has been completed which makes it a synchronous call. The implementation will work in future versions of C++ if the blocking nature of std::future's destructor is removed. For now the workaround in Listing 9.19, which uses a detached std::thread instead of std::async, can be used.

```
void execute(std::function<completion_event(event)> async_op,
        event&& e) {
    std::thread([&]{
        completion_queue_.get()<<async_op(std::forward<event>(e));
    }).detach();
}
```

Listing 9.19: Workaround for std::async using std::thread

Alternatively it is possible to implement a custom non-blocking version of the std::async call as described by [26].

9.4 Summary

The pro-actor design pattern is used when complex event handling operations can be split into an asynchronous and a synchronous part to increase performance. Unlike the other patterns examined in this paper the pro-actor pattern does not have clearly defined universal set-up but many different implementations can be found.

Three different ways of implementing the pro-actor pattern have been discussed in this paper:

The buffer based pro-actor is easy to implement as it is mainly an asynchronous variation on the reactor pattern. The problem is that the asynchronous operations can still block each other.

The task based pro-actor uses a thread pool to handle the events asynchronously. This generally works well but can lead to problems when blocking operations are used in the asynchronous operations.

The std::async based implementation is the most complex one in this context. It clearly splits responsibilities in event handling among the different components of the pattern. But this implementation relies on std::async which does not (yet) work in the expected way. Using the std::async implementation with tasks and a thread pool instead of async can be a good combination to avoid the current problems with std::async.

Conclusion

10 Conclusion

Loose coupling in communication can be achieved in numerous ways in C++. C++11 and 14 offer a great deal of new technologies which allow for more versatile and more generic implementations. Communications can be broken down into small components which support a high grade of reusability. C++11's variadic templates allow for even more generic implementations. Communication can be established between these basic components (Chapter 3) and then transferred to a higher level like the implementation of design patterns.

The use of a supervisor (Chapter 4) is useful for managing the connections of the individual components at run-time. It also eradicates the need for singletons by managing the creation of unique objects.

Communication design patterns benefit from the use of basic communication components like *ports* or more complex parts like *signals and slots* (Chapter 5). Especially the asynchronous pro-actor pattern benefits from the new concurrency API and std::async. The std::thread class allows for more system independent concurrency implementations.

Even though C++11 and 14 have introduced many new convenient features to the standard library there are still some common types missing. C++ still does not offer object-oriented semaphores, sockets, option types and others. The user has to implement these individually. Some of these implementations which have been used in the course of this paper can be found in Chapter 11.

Appendix

11 Utilities

This chapter explains the utility classes created for the projects in this paper. Some components had to be created because they are not (or not sufficiently) provided by the standard library but are not directly part of the technique or project explained in the chapter. These classes and their implementations can be found in this chapter.

11.1 Function Binder

In C++ a member function can not be passed as an argument without also referencing the parent object. When referencing a member function from another object this is not very practical because it requires the calling object to store a reference to the function's parent object. Often the calling object should not even need to know whether it is executing a member function, a global function, a lambda or a function object. To allow this kind of encapsulation it is necessary to bind the member function and the object together.

C++11 offers the new *std::bind* function [27] which allows to bind parameters to a function and return a new function taking no parameters.

```
auto func1 = [](int i, int j){return i+j;};
auto func2 = std::bind(func1, 12, 5);
```

Listing 11.1: std::bind usage example

The example in Listing 11.1 takes the parameters *i* and *j* from *func1* and binds the values 12 and 5 to them returning a new function (*func2*) which does not take any parameters but always executes with i=12 and j=5.

For std::bind to work all parameters must be bound explicitly. If only some parameters are meant to be bound, the unbound parameters must be specified using *placeholders* [13]. Listing 11.2 shows how to bind only a single parameter of a function taking two.

```
auto func3 = [](int i, bool b){
                 if(b)
                    return i;
                 return -1;
                 }
auto return5 = std::bind(func3, 5, std::placeholders::_1);

// Usage
return5(true);  //returns 5
return5(false); //returns -1;
```

Listing 11.2: Binding a parameter with placeholders

11.1.1 Bind Using std::placeholders

The std::placeholders namespace supplies placeholders ranging from _1 t _29. The type of these placeholders is *const _Placeholder<1>* to *const _Placeholders<29>* (in header <functional>) so the placeholder index is a template type and has to be created at compile time. This is problematic because it means that to create a function that binds all parameters to placeholders for any function it is not possible to iterate over the placeholders in a loop. Even when trying to achieve this using variadic template recursion [28, p. 222] the function will be limited to a maximum of 29 parameters. Each of these 29 parameters would need its own function which would have to be created manually because the placeholders are individual objects. Even though most functions will take a lot less than 29 parameters this is not an elegant solution.

But std::placeholders also allows programmers to write their own placeholders [29]. To get individual placeholders for all parameters the custom placeholder must be a template struct which takes the parameter index as the template argument (Listing 11.3).

```
template <int Num_>
struct place_holder {};
```

Listing 11.3: Individual template placeholder

In order to declare the new placeholder struct as a placeholder for a specific index the template struct *is_placeholder* must be declared in the std namespace and derived from *std::integral_constant* with the placeholder index (Listing 11.4).

```
namespace std {
    template <int Num_>
    struct is_placeholder<::place_holder<Num_>>
        : integral_constant<int, Num_> {};
}
```

Listing 11.4: Initialise new struct as placeholder

In order to get the placeholders for every argument a template instance of the place_holder struct has to be created for every index that is being used. This can be achieved using a *std::integer_sequence* [30] which was introduced in C++14.

The *std::make_integer_sequence* function creates a sequence of integers counting up to the specified number. This is used to create a template based sequence [31] of all the indices of the arguments which are to be bound (Listing 11.5).

```
template <typename Obj_, typename Func_,
    typename... Args_, int... Idx_>
std::function<Func_(Args_...)> bind_arg_seq_(
        Func_(Obj_::*func)(Args_...), Obj_& obj,
        std::integer_sequence<int, Idx_...>) {
    return std::bind(func, obj, place_holder<Idx_ + 1>{}...);
}
```

Listing 11.5: Binding parameters based on placeholder indices

Since placeholder indices start at 1 not 0 the placeholders have to be created using an incremented index <Idx_ + 1>. The function in Listing 11.5 is called from the *bind_function_to_object* function which creates the integer sequence (Listing 11.6) and returns a std::function with the parent object internally bound with the member function.

```cpp
template <typename Obj_, typename Func_, typename... Args_>
std::function<Func_(Args_...)> bind_function_to_object(
        Func_(Obj_::*func)(Args_...), Obj_& obj) {
    return bind_arg_seq_(func, obj,
            std::make_integer_sequence<int, sizeof...(Args_)>{});
}
```

Listing 11.6: Create an integer sequence counting arguments

11.1.2 Bind Derived Class Member Function

While this technique will work in most cases, it does not work when binding a member function to an object derived from the original parent class. In this case another template parameter (*Base_*) is introduced which represents the base class [32].

```cpp
template <typename Obj_, typename Base_,
    typename Func_, typename... Args_, int... Idx_>
std::function<Func_(Args_...)> bind_arg_seq_(
        Func_(Base_::*func)(Args_...), Obj_&& obj,
        std::integer_sequence<int, Idx_...>) {
    return std::bind(func, std::forward<Obj_>(obj),
            place_holder<Idx_ + 1>{}...);
}

template <typename Obj_, typename Base_,
    typename Func_, typename... Args_>
std::function<Func_(Args_...)> bind_function_to_object(
        Func_(Base_::*func)(Args_...), Obj_&& obj) {
    return bind_arg_seq_(func, std::forward<Obj_>(obj),
            std::make_integer_sequence<int, sizeof...(Args_)>{});
}
```

Listing 11.7: Bind member function to derived class

The object can now be passed as an rvalue reference to allow perfect forwarding [28, p. 181] of the object (Listing 11.7).

11.1.3 Bind Using Lambda Expression

Scott Meyers proposes using lambda expressions instead of std::bind [24, pp. 217-223] which makes the binding process a lot simpler. This will remove the need for working with placeholders altogether.

```
template <typename Obj_, typename Func_, typename... Args_>
auto bind_function_to_object(
        Func_(Obj_::*func)(Args_...), Obj_& obj) {
    return [&](Args_&&... args) mutable {
        return (obj.*func)(std::forward<Args_>(args)...);
    };
}
```

Listing 11.8: Bind member function using lambda

The lambda used for the bind encapsulates the member function call [33] and passes the (unbound) arguments as parameters (Listing 11.8). Because lambdas standardly capture variables as immutable the *mutable* keyword must be used to allow the use of perfect forwarding. In order to bind cv-qualified objects [14] an overload is necessary (Listing 11.9).

```
template <typename Obj_, typename Func_, typename... Args_>
auto bind_function_to_object(
        Func_ (Obj_::*func)(Args_...) const , Obj_ const& obj) {
    return [&](Args_&&... args) mutable {
        return (obj.*func)(std::forward<Args_>(args)...);
     };
}
```

Listing 11.9: Lambda bind for cv-qualified members

11.2 Optional

Sometimes it is required for a function to return a result only when certain requirements are met and return nothing otherwise. The standard approach to this is to pass a pointer to the function as a parameter in which the result is stored in the case of success. If the function is not successful it could either store a null-pointer or leave the pointer untouched and use the return type to return an error code.

A better alternative is to use an optional type which works as a wrapper for the return type. It can either contain the actual return value or it can be empty (Listing 11.10).

```
    bool has_content(void) const {
        return !(content_.empty());
    }

    Type_ get(void) const throw (empty_optional_exception) {
        if(has_content())
            return content_.front().get();
        throw empty_optional_exception(
           "optional::get can not be called on empty optional");
    }
```

Listing 11.10: Optional class: get call

If *get* is called on an empty optional it throws an *empty optional exception* which is implemented in Listing 11.11.

```
struct empty_optional_exception : public std::exception {
    empty_optional_exception(std::string message) noexcept
        : message_(message)
    {}

    virtual const char* what() const noexcept override {
        return message_.c_str();
    }
private:
    std::string message_;
};
```

Listing 11.11: Empty optional exception class

In order to achieve an "empty" class the contained type is added to a std::vector *content_* which can only ever contain a maximum of one element. It is initialised as an empty vector in the no argument constructor and initialised with a single member in the second constructor taking an initial value for the optional (Listing 11.12). The *reverse* call initialises the vector with preallocated memory for one element.

```
template <typename Type_>
struct optional {
    optional()
        : content_{}
    {
        content_.reverse(1);
    }

    explicit optional(Type_&& content)
        : optional{}
    {
        set(std::forward<Type_>(content));
    }
```

Listing 11.12: Optional class: constructors

Using a vector instead of a pointer or smart pointer to allocate memory and store the contained data may seem an unusual decision. However it is actually faster because the std::vector pre-allocates memory so it does not have to delete and re-allocate memory at every clear and set operation. Alternatively a std::array could be used which takes the size (in this case 1) as a template argument which is more intuitive to the fact that no more than one element can be contained. However it is not possible to empty an array so an additional flag would be required to state the validity of the contained data.

```
    void set(Type_&& element) {
        clear();
        content_.push_back(std::forward<Type_>(element));
    }

    void set(std::nullptr_t) {
        clear();
    }
```

Listing 11.13: Optional class: setting the content

The optional can always be set with a new value which will first clear (Listing 11.14) the contained vector to make sure it can never hold more than one value. It also allows to set *nullptr* as the contained value which will simply clear the vector (Listing 11.13).

```
    void clear(void) {
        if(has_content())
            content_.clear();
    }
```

Listing 11.14: Optional class: clearing the optional

Additionally a second class can be implemented to hold references instead of values. The only variation here is to store std::reference_wrappers to the supplied types (Listing 11.15)

```
template <typename Type_>
class optional_reference {
    typedef std::reference_wrapper<Type_> _TypeRef;
    std::vector<_TypeRef> wrapped_type_;
```

Listing 11.15: Optional reference class

11.3 Semaphore

Semaphores are not part of the C++ standard at this point. While it would be possible to use POSIX semaphores [34] this approach is not ideal as it is system dependant (not all systems conform to the POSIX standard). The goal is to create a semaphore class using only the C++ standard library. This can be done using a **mutex** and a **condition variable** which were added to the standard in C++11. Following the function names of the POSIX semaphores [34] the supplied functions can be seen in Listing 11.16.

```cpp
class semaphore {
public:
    semaphore(const size_t count = 0);

    void wait(void);

    void post(void);

    bool try_wait(void);

    size_t get_value(void) const;

    void destroy(void);

    bool valid(void) const;

private:
    size_t count_;
    std::condition_variable condition_;
    std::mutex mtx_;
    bool valid_;
};
```

Listing 11.16: Semaphore class declaration

The *post* function increases the count of the semaphore and notifies a thread that might be waiting on the semaphore if its count is at zero (Listing 11.17).

```
inline void semaphore::post(void) {
    if(!valid_)
        return;
    std::lock_guard<std::mutex> lock__(mtx_);
    count_++;
    condition_.notify_one();
}
```

Listing 11.17: Semaphore: post function

The *wait* function decreases the semaphore count and blocks if the count reaches zero. Any thread calling *wait* when the count is at zero waits on the condition variable until notified that the count has increased. A variation on this function is the *try_wait* function which does not block but simply return its success in acquiring the semaphore in the form of a bool (Listing 11.18).

```
inline void semaphore::wait(void) {
    std::unique_lock<std::mutex> lock__(mtx_);
    condition_.wait(lock__, [&]{
        return count_>0 || !valid_;
    });
    if(valid_)
        count_--;
}

inline bool semaphore::try_wait(void) {
    if(!valid_)
        return false;
    std::unique_lock<std::mutex> lock__(mtx_);
    if(count_ > 0) {
        count_--;
        return true;
    }
    return false;
}
```

Listing 11.18: Semaphore: wait and try_wait function

In addition it is sometimes necessary to destroy the semaphore and release all waiting threads. This can be done with the *destroy* call (Listing 11.19). When destroyed the *valid_* flag is cleared and all threads which are currently waiting on the semaphore are notified to stop waiting.

```
void semaphore::destroy(void) {
    valid_ = false;
    condition_.notify_all();
}
```

Listing 11.19: Semaphore: destroy function

Because *destroy* has to be called manually it is the caller's responsibility to check that the destroyed semaphore is not used any more. Otherwise calling *wait* will not block and the counter cannot be changed any more.

11.4 Thread Pool and Tasks

In some cases - especially when the creation of a lot of threads would harm the system's performance or when an application scales beyond the maximum number of threads the operating system can supply - it is often a good alternative to choose task-based programming.

To be able to use tasks the first thing to create is a task class. The purpose of this class is to wrap and store the actual task to be performed later. In order to do this it has to take a function describing the task as the constructor argument.

```cpp
class task {
public:
    task()
        : func_{[]{}}
    {}
```

Listing 11.20: Task class: default constructor

The default (no argument) constructor initialises the function contained within the task class with an empty lambda expression (Listing 11.20).

```cpp
    template<typename Func_, typename ... Args_>
    task(Func_ && func, Args_ &&... args)
        : func_{std::bind(func,std::forward<Args_>(args)...)}
    {}
```

Listing 11.21: Task class: function constructor

To be able to get any function into the task class two additional constructors are required. All lambdas, function pointers and functors can be wrapped in the task class using the template constructor in Listing 11.21. This constructor uses two template types: *Func_* which specifies the function to be wrapped and *Args_...* as a variadic template [28, p. 221] parameter specifies additional arguments.

```cpp
void task_function(void) {
    std::cout << "hello world" << std::endl;
}

struct functor {
    functor() {}
    void operator()(double d) {
        std::cout << d << std::endl;
    }
};

int main(void) {
    task t1([](int i){std::cout << i << std::endl;},12);
    task t2(&task_function);
    functor f;
    task t3(f,5.0);
}
```

Listing 11.22: Task class: constructor usage

Listing 11.22 shows how to use the task class with lambdas, function pointers and functors. However the current constructor does not allow member function pointers.

```cpp
    template<typename Obj_, typename Func_, typename ... Args_>
    task(Func_ (Obj_::*func)(Args_...), Obj_ & obj, Args_ &&... args)
        : func_{std::bind(func,obj,std::forward<Args_>(args)...)}
    {}
```

Listing 11.23: Task class: member function constructor

For this particular use case a new constructor has to be written which specifies the function's parent object (Listing 11.23). Both the function constructor and the member function constructor call std::bind on the arguments and convert them to a std::function<void(void)>.

```cpp
    inline void operator()(void) {
        func_();
    }
```

Listing 11.24: Task class: function call operator

To execute a task the function call (parentheses) operator is overloaded, making the task class a functor class. Since calling the contained function from another function is inefficient as it causes the programme to double branch the *inline* keyword can be used to resolve that inefficiency (Listing 11.24).

The second class necessary for the thread pool system is the thread pool itself. In this example the thread pool is implemented as a template class which takes the task implementation as the template type. It is defaulted to use the previously defined task class (Listing 11.25).

```
template <typename Callable_ = task>
class thread_pool
```

Listing 11.25: Thread pool class declaration

The thread pool contains a std::vector of std::threads, a std::queue and a std::mutex to lock the queue. In addition a std::condition_variable is used to signal the arrival of new tasks (Listing 11.26).

```
    std::vector<std::thread> threads_;
    std::queue<Callable_> tasks_;
    std::mutex queue_mtx_;
    std::condition_variable queue_cond_;
```

Listing 11.26: Thread pool class: private class members

The constructor takes the thread pool size (the number of threads to be used) as a single argument (Listing 11.27). This size should usually be set to the number of CPU cores. The threads are then all initialised with the same function (Listing 11.28). If the thread pool is supposed to run forever it is sensible to detach the threads so run independent from the main thread and don't have to be joined. Alternatively a *running* flag could be defined which would allow to shut down the threads and join them manually into the main thread. This operation would have to be carried out by a *supervisor* object (Chapter 4).

```
    explicit thread_pool(size_t size)
        : threads_{size}
        , tasks_{}
        , queue_mtx_{}
        , queue_cond_{}
    {
        for(auto i=0ull; i<size; i++)
            threads_[i] = std::thread(
                &thread_pool<Callable_>::thread_function, this)
              .detach();
    }
```

Listing 11.27: Thread pool class: constructor

The thread function waits on the condition variable to signal that there are elements in the queue and then executes them.

```
void thread_function(void) {
    Callable_ task__;
    while(true) {
        {
            std::unique_lock<std::mutex> lock__(queue_mtx_);
            queue_cond_.wait(lock__, [&]{
                    return !(tasks_.empty());});
            task__ = tasks_.front();
            tasks_.pop();
        }
        task__();
    }
}
```

Listing 11.28: Thread pool class: thread function

Tasks can be added to the pool by calling the *add_task* function (Listing 11.29) which notifies the condition variable and causes a sleeping thread to wake up.

```
void add_task(Callable_ && task) {
    std::lock_guard<std::mutex> __lock(queue_mtx_);
    tasks_.push(std::forward<Callable_>(task));
    queue_cond_.notify_one();
}
```

Listing 11.29: Thread pool class: add tasks

Since a system should not contains multiple thread pools it could be implemented as a*singleton* or be created exactly once by the system's supervisor (Chapter 4). Under no circumstances must a thread pool be copied as this will lead to undefined behaviour. Therefore all copy (and move) operations on the thread pool are to be prohibited (Listing 11.30).

```
thread_pool<Callable_>& operator=(const thread_pool<_Callable>&)
                                              = delete;
thread_pool<Callable_>& operator=(thread_pool<_Callable>&&)
                                              = delete;
thread_pool(const thread_pool<Callable_>&) = delete;
thread_pool(thread_pool<Callable_>&&) = delete;
```

Listing 11.30: Thread pool class: deleted move and copy operations

Bibliography

[1] B. Stroustrup, *Einführung in die Programmierung mit C++*. Pearson Studium - IT, Pearson Deutschland, 2010.

[2] B. Stroustrup, *The C++ Programming Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2000.

[3] B. Selic, "Tutorial: real-time object-oriented modeling (ROOM)," in *Real-Time Technology and Applications Symposium, 1996*, pp. 214–217, June 1996.

[4] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 32nd ed., April 2005.

[6] R. Carrara, "Asynchronous lock-free Three-Buffer Data-Sharing for Embedded Systems using C/C++," April 2004.

[7] G. Kendall, "G53OPS - Operating Systems - Test and Set Lock." `http://www.cs.nott.ac.uk/~pszgxk/courses/g53ops/Processes/proc08-tsl.html`, January 2002. URL date: 05/07/2016.

[8] H. Sutter, "N4058: Atomic Smart Pointers." `https://isocpp.org/blog/2014/06/n4058`, June 2014. URL date: 05/07/2016.

[9] B. P. Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., September 2002.

[10] "Signals & Slots | Qt4.8." `http://doc.qt.io/qt-4.8/signalsandslots.html`, 2016. URL date: 12/05/2016.

[11] B. Eckel and C. Allison, *Thinking in C++, Volume 2: Practical Programming*. Upper Saddle River, NJ, USA: Pearson Education, Inc., 2004.

[12] cppreference.com, “std::reference_wrapper.” http://en.cppreference.com/w/cpp/utility/functional/reference_wrapper, March 2016. URL date: 20/05/2016.

[13] cppreference.com, “std::placeholders::_1, std::placeholders::_2, ..., std::placeholders::_N.” http://en.cppreference.com/w/cpp/utility/functional/placeholders, May 2015. URL date: 01/06/2016.

[14] cppreference.com, “cv (const and volatile) type qualifiers.” http://en.cppreference.com/w/cpp/language/cv, April 2016. URL date: 28/05/2016.

[15] “Effective Go - The Programming Language.” https://golang.org/doc/effective_go.html#channels. URL date: 25/05/2016.

[16] G. Hohpe and B. Woolf, *Enterprise Integration Patterns.* Addison-Wesley, 4th ed., August 2004.

[17] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns.* Chichester, West Sussex PO19 8SQ, England: John Wiley & Sons Ltd., 1996.

[18] G. Hohpe, “Enterprise Integration Patterns - Home.” http://www.enterpriseintegrationpatterns.com/, 2016. URL date: 15/06/2016.

[19] cppreference.com, “std::unordered_map.” http://en.cppreference.com/w/cpp/container/unordered_map, June 2016. URL date: 14/05/2016.

[20] cppreference.com, “std::map::emplace.” http://en.cppreference.com/w/cpp/container/map/emplace, May 2016. URL date: 08/06/2016.

[21] C++ Truths, “Perfect Forwarding of Parameter Groups in C++11.” http://cpptruths.blogspot.de/2012/06/perfect-forwarding-of-parameter-groups.html, June 2012. URL date: 08/06/2016.

[22] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2 - Patterns for Concurrent and Networked Objects.* Chichester, West Sussex PO19 8SQ, England: John Wiley & Sons Ltd., 2000.

[23] H. Ebrahimmalek, "Proactor Pattern - CodeProject." http://www.codeproject.com/Articles/33011/Proactor-Pattern, February 2009. URL date: 13/07/2016.

[24] S. Meyers, *Effektives modernes C++ - 42 Techniken für besseren C++11- und C++14-Code*. Köln: O'Reilly Verlag GmbH & Co. KG, 1st ed., 2015.

[25] H. Sutter, C. Carruth, and N. Gustafsson, "async and future (Revision 4)." http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3773.pdf, September 2013. URL date: 04/07/2016.

[26] Stack Overflow, "c++ - Workaround for blocking async?." http://stackoverflow.com/questions/16296284, April 2013. URL date: 09/07/2016.

[27] cppreference.com, "std::bind." http://en.cppreference.com/w/cpp/utility/functional/bind, May 2016. URL date: 01/06/2016.

[28] T. T. Will, *C++11 programmieren - 60 Techniken für guten C++11-Code*. Bonn: Galileo Press, 1st ed., 2012.

[29] cppreference.com, "std::is_placeholder." http://en.cppreference.com/w/cpp/utility/functional/is_placeholder, October 2015. URL date: 02/06/2016.

[30] cppreference.com, "std::integer_sequence." http://en.cppreference.com/w/cpp/utility/integer_sequence, April 2016. URL date: 14/06/2016.

[31] Lounge<C++>, "Indices." http://loungecpp.wikidot.com/tips-and-tricks%3aindices. URL date: 08/06/2016.

[32] Stack Overflow, "c++ - How to bind a member function to an object in C++14." http://stackoverflow.com/questions/37456513, May 2016. URL date: 26/05/2016.

[33] Stack Overflow, "c++ - Short way to std::bind member function to object instance, without binding parameters." http://stackoverflow.com/questions/14803112, February 2013. URL date: 24/05/2016.

[34] "Overview of POSIX Semaphores." http://linux.die.net/man/7/sem_overview. URL date: 20/05/2016.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, July 15th 2016 Till Kaiser