Roman Ulan

Wzorce silników zdarzeń w C++

Część III: ThreadPool – wielowątkowa alternatywa

W poprzednim artykule przedstawiłem wzorzec Reactor w wersji reużywalnej, na bazie której pokażę wielowątkowe podejście do obsługi zdarzeń z wzorcem ThreadPool.

SŁABOŚCI WZORCA REACTOR

Podstawową bolączką reaktywnego podejścia do obsługi zdarzeń jest jego jednowątkowość – zdarzenia są obsługiwane w tym samym wątku, w którym zostały otrzymane, w sposób szeregowy. W naszym przykładzie (serwer echo) sprawdza się znakomicie, gdyż raczej nikomu nie przyszło do głowy wysyłać gigabajtowych wiadomości z setek/tysięcy/milionów programów klienckich. Pytanie: co by się stało, gdyby jednak tak właśnie miało być (np. serwer HTTP)? Prawdopodobnie czas reakcji spowodowałby masę problemów, począwszy od opóźnionych wiadomości zwrotnych, poprzez brak ich dostarczenia, aż po frustrację klientów korzystających z usług naszego serwera.

Podsumowując w jednym zdaniu: skalowalność to nie najmocniejsza strona reaktora – nie skaluje się wcale, co dyskredytuje go w niektórych zastosowaniach.

WZORZEC THREADPOOL (TP)

Na początek drobne wyjaśnienie: *ThreadPool* w ramach tego artykułu to wzorzec obsługi zdarzeń i nie powinien być mylony z idiomem (modelem) wielowątkowości o tak samo brzmiącej nazwie (*thread-pool*).

Na wzorzec TP składają się dwa elementy:

1. pula wątków sensu stricto – wspomniany już wyżej idiom, stanowiący o grupie wątków realizujących ten sam typ zadania, zarządzaną ze wspólnego interfejsu;
2. reaktora, pod którego interfejsem się ukrywa, oraz który odbiera zdarzenia z demultipleksera i zleca ich obsługę podwładnej mu puli wątków.

Całość dobrze prezentują dwa tradycyjne diagramy UML (patrz rysunki obok).

CZĘŚĆ WIELOWĄTKOWA (T-P)

Podstawowa definicja puli wątków już właściwie padła – to grupa wątków realizująca zadanie tego samego rodzaju. Czym zatem może być zadanie? Otóż może to być każdy cel, jak choćby wykonanie „jakiejś" zadanej funkcji. Uwaga! Nie znaczy to automatycznie tej/takiej samej funkcji. Innymi słowy mówiąc, każdy z wątków w puli ma dokładnie tę samą implementację.

Druga ważna rzecz to sposób panowania nad wątkami – są razem uruchamiane i razem zakańczane.

Ponadto w puli wątków nie ma wątków ważniejszych czy też mniej ważnych. Każdy z nich jest równorzędnym wykonawcą zadań pobieranych z kolejki (bez wnikania kto zlecił dane zadanie), a dla użytkownika t-p nie ma znaczenia, który wątek wykona dane zadanie.

Taka postać rzeczy ma swoje konsekwencje:

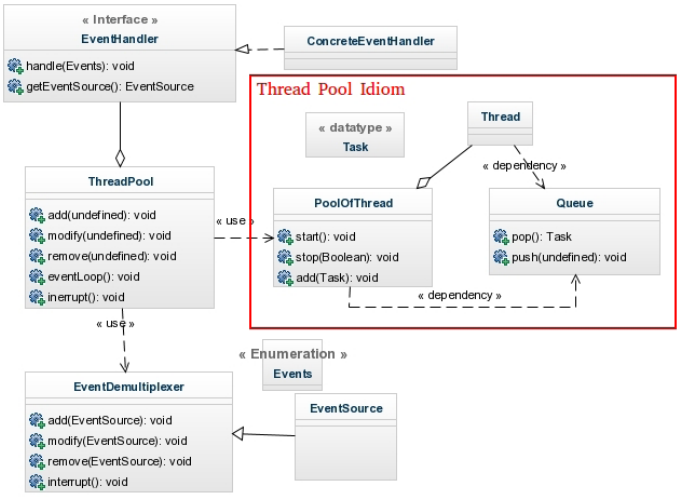
- zaszeregowanie zadań w kolejce nie daje gwarancji kolejności wykonywania zadań – nie mamy wiedzy, jak system będzie przyznawał kontekst poszczególnym wątkom, a jedyne, czego możemy się spodziewać, to kolejność, w jakiej zadania zostaną zdjęte z kolejki (to definiuje jej algorytm, np. FIFO);
- t-p jest zwyczajnym realizatorem zadań – nie przejmuje się spójnością danych współdzielonych pomiędzy zadaniami.

Z uwagi na fakt, że t-p jest osobnym bytem logicznym, do naszego projektu postanowiłem wprowadzić komponent `Threading`, w którym zaimplementowałem obiekt generycznej puli wątków (klasa `ThreadPool`) wykonującą zlecane jej funkcje o sygnaturze `void(void)` oraz obiekt bezpiecznej wielowątkowo kolejki, do której można wkładać zadania z obu końców, lecz pobierać tylko z jednego (klasa `TwoEndQueue`).

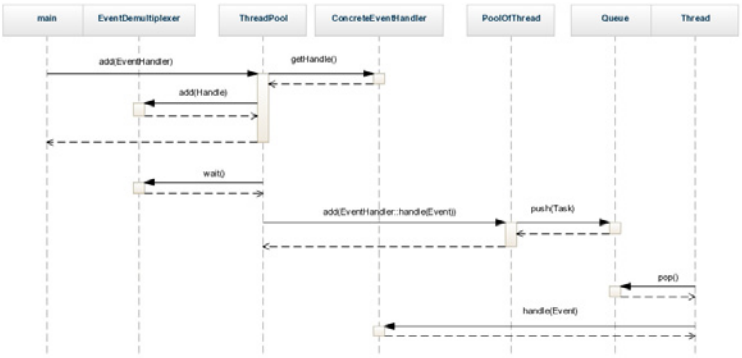
O serii „Wzorce silników zdarzeń"

Intencją serii jest dostarczenie usystematyzowanej wiedzy bazowej początkującym programistom, projektantom i architektom.

W kolejnych artykułach będę przedstawiał najbardziej użyteczne wzorce rozwiązujące znaczną część problemów powiązanych z tą niebanalną tematyką. Każdy z wzorców postaram się dokładnie opisać od strony technicznej (obiektów i ich powiązań na podstawie schematów UML oraz implementacji w C++), a także wad i zalet w konfrontacji wydajności i prostoty w fazie utrzymania.



Rysunek 1. Diagram klas wzorca ThreadPool



Rysunek 2. Diagram sekwencji wzorca ThreadPool

Listing 1. Klasy komponentu Threading

```
namespace Threading
{
template<class T>
class TwoEndQueue
{
public:
  TwoEndQueue();
  ~TwoEndQueue();

public:
  void pushFront(const T&);
  void pushBack(const T&);
  T pop();

private:
  typedef std::list<T> t_queue;
  std::mutex m_mutex;
  std::condition_variable m_condition;
  t_queue m_queue;
};
//class TwoEndQueue

template<class T>
TwoEndQueue<T>::TwoEndQueue() :
    m_mutex(), m_condition(), m_queue()
{
}

template<class T>
TwoEndQueue<T>::~TwoEndQueue()
{
}

template<class T>
void TwoEndQueue<T>::pushFront(const T& p_t)
{
  std::unique_lock < std::mutex > lock(m_mutex);
  m_queue.push_front(p_t);
  m_condition.notify_one();
}

template<class T>
void TwoEndQueue<T>::pushBack(const T& p_t)
{
  std::unique_lock < std::mutex > lock(m_mutex);
  m_queue.push_back(p_t);
  m_condition.notify_one();
}

template<class T>
T TwoEndQueue<T>::pop()
{
  std::unique_ptr<T> result;
  {
    std::unique_lock < std::mutex > lock(m_mutex);
    while (m_queue.empty())
      m_condition.wait(lock);

    result.reset(new T(m_queue.front()));
    m_queue.pop_front();
  }

  return *result;
}
} //namespace Threading

namespace Threading
{
class ThreadPool
{
private:
  struct Job
  {
  public:
    typedef std::function<void (void)> Task;
    enum Type {
      STOP,
      REGULAR
    };
    Job(Type, const Task&);
    Type m_Type;
    Task m_Task;

  private:
    Job();
  }; //struct Job

public:
  typedef Job::Task Task;
  explicit ThreadPool(size_t);
  ~ThreadPool();

public:
  void start();
  void stop(bool);
  void add(const Task&);

protected:
  static void threadFunc(TwoEndQueue<Job>&);

private:
  ThreadPool(const ThreadPool&);
  ThreadPool& operator=(const ThreadPool&);

private:
  typedef std::shared_ptr<std::thread> threadH;
  typedef std::vector<threadH> threads;

private:
  TwoEndQueue<Job> m_queue;
  size_t m_size;
  threads m_threads;
}; //class ThreadPool

ThreadPool::Job::Job(Type p_type, const Task& p_task)
  : m_Type(p_type), m_Task(p_task)
{
}

ThreadPool::ThreadPool(size_t p_size)
  : m_queue(), m_size(p_size), m_threads()
{
}

ThreadPool::~ThreadPool()
{
  //TODO check if kill all threads before if needed
}

void ThreadPool::start()
{
  if (!m_threads.empty())
  {
    throw std::runtime_error("ThreadPool started already");
  }

  std::string threadCreationError;
  for (size_t i = 0; (i < m_size) && threadCreationError.empty();
  ++i)
  {
    try
    {
      m_threads.push_back(threadH(new std::thread(std::bind(Thread
      Pool::threadFunc, std::ref(m_queue)))));
    }
    catch (const std::exception e)
    {
      threadCreationError = e.what();
      break;
    }
  }

  if (!threadCreationError.empty())
  {
    if (m_threads.size())
      stop(true);

    std::stringstream ss;
    ss << "ThreadPool thread creation error: " <<
    threadCreationError
      << ", ThreadPool not started at all";
    throw std::runtime_error(ss.str());
  }
}

void ThreadPool::stop(bool p_immediately)
{
  if (p_immediately)
    for (size_t i = 0; i < m_size; ++i)
      m_queue.pushFront(Job(Job::STOP, Task()));
  else
    for (size_t i = 0; i < m_size; ++i)
      m_queue.pushBack(Job(Job::STOP, Task()));

  for (auto i = m_threads.begin(); i != m_threads.end(); ++i)
    (*i)->join();

  m_threads.clear();
}

void ThreadPool::add(const Task& p_task)
{
  m_queue.pushBack(Job(Job::REGULAR, p_task));
}

void ThreadPool::threadFunc(TwoEndQueue<Job>& p_queue)
{
  while (1)
  {
    Job job = p_queue.pop();
    if (job.m_Type == Job::REGULAR)
      job.m_Task();
    else
      break;
  }
}

} //namespace Threading
```

Nazywając wyżej obiektami obie klasy komponentu `Threading`, miałem na myśli obiekty w rozumieniu OOP:

- są to abstrakcje implementujące algorytmy i struktury danych w sposób spójny i zgodny z ich naturą;
- ukrywające swą implementację, a eksponując jedynie funkcje korzystające z obiektu i zmieniające go w sposób bezpieczny;
- klasa kolejki korzysta ze statycznego polimorfizmu za pośrednictwem szablonów.

Klasa `TwoEndQueue` implementuje klasyczny *Monitor*:

- metoda pop, jeśli kolejka (`m_queue`) jest pusta, oczekuje aktywnie na element na zmiennej warunkowej;
- metody `push*` po umiejscowieniu elementu w kolejce notyfikują wątki oczekujące na zadanie (te stojące na zmiennej warunkowej w metodzie pop);
- instancja klasy powinna być dzielona między wątkami przez użycie jednego z odniesień: wskaźnika lub referencji.

Ważnym elementem tej klasy jest odporność na zjawisko „nieoczekiwanej pobudki" (ang. *spurious weakup*) – w metodzie pop, jeśli wątek śpiący na zmiennej warunkowej zostanie obudzony z nieoczekiwanego powodu (`m_condition.wait(lock)` zostanie przerwane), wątek sprawdzi, czy kolejka jest pusta, i jeśli taka będzie, znów zaśnie w oczekiwaniu na zadanie.

Klasa `Threading::ThreadPool` pozwala uruchomić (metoda `start`) grupę wątków realizujących treść statycznej funkcji `threadFunc`, która jest reprezentantem obiektu `Thread` z diagramu klas przedstawionego wyżej.

Pula może być zatrzymana dwojako:

1. natychmiastowo – zadania typu STOP dla wątków są wkładane priorytetowo na początek kolejki;
2. zezwalając na realizację zadań będących już w kolejce – zadania typu STOP są wkładane na koniec kolejki.

Zwróćmy uwagę, że struktura `Job` jest prywatną właściwością puli wątków (rozumianej jako całość) pozwalającą w dynamiczny sposób zarządzać wątkami, a w publicznym API eksponowany jest jedynie typ zadania użytkowego puli wątków – funkcja do wykonania o sygnaturze `void(void)`.

Ważną sprawą, którą muszę tu nadmienić, jest fakt, iż w tej implementacji odszedłem od użycia biblioteki Boost na rzecz wykorzystania i wprowadzenia Cię w możliwości C++11 (wątki, muteksy, zmienne warunkowe, a także inteligentne wskaźniki i wyrażenia lambda). Jeśli martwisz się, że nie znasz tych komponentów, to zachowaj spokój – elementy te (poza lambdami) zostały wciągnięte z Boost i poza drobnymi szczegółami nie różnią się użyciem.

To, co ważne, to to, że gdy zechcesz kompilować podane przykłady, korzystaj z jak najaktualniejszych wersji kompilatorów (ja użyłem Gcc w wersji 4.9).

THREADPOOL W POSTACI DOCELOWEJ

Do implementacji silnika TP wykorzystałem ten sam dokładnie zestaw interfejsów pomocniczych, co w przypadku reaktora: `EventHandler` i `EventDemultiplexer`, zaś sama klasa `ThreadPool` eksponuje nieco rozszerzony interfejs publiczny klasy `Reactor` z poprzedniego artykułu.

Listing 2. Implementacja silnika ThreadPool

```
namespace EventEngines
{
class ThreadPool
{
public:
  typedef std::shared_ptr<ThreadPool> Ptr;

public:
  explicit ThreadPool(EventDemultiplexer::Ptr, size_t);
  ~ThreadPool();

  void add(EventHandler::Ptr);
  void modify(EventSource::Ptr);
  void remove(EventSource::Descriptor);
  void eventLoop();
  void interrupt(bool);

private:
  ThreadPool(const ThreadPool&);
  ThreadPool& operator=(const ThreadPool&);

private:
  typedef std::map<EventSource::Descriptor, EventHandler::Ptr>
  t_handlers;
  typedef std::pair<EventHandler::Ptr, EventSource::EventTypes>
  t_toHandle;
  typedef std::vector<t_toHandle> t_toHandles;

  std::mutex m_mutex;
  bool m_run;
  t_handlers m_handlers;
  EventDemultiplexer::Ptr m_eventDemultiplexer;
  Threading::ThreadPool m_threadPool;
}; //class ThreadPool

ThreadPool::ThreadPool(EventDemultiplexer::Ptr p_
eventDemultiplexer, size_t p_tpSize)
  : m_run(false), m_eventDemultiplexer(p_eventDemultiplexer),
  m_threadPool(p_tpSize)
{
  m_threadPool.start();
}

ThreadPool::~ThreadPool()
{
  m_threadPool.stop(true);
}

void ThreadPool::add(EventHandler::Ptr p_eventHandler)
{
  std::unique_lock<std::mutex> lock(m_mutex);
  m_eventDemultiplexer->add(p_eventHandler->getEventSource());
  m_handlers.insert(std::make_pair(p_eventHandler-
  >getEventSource()->getDescriptor(), p_eventHandler));
}

void ThreadPool::modify(EventSource::Ptr p_eventSource)
{
  std::unique_lock<std::mutex> lock(m_mutex);
  m_eventDemultiplexer->modify(p_eventSource);
}

void ThreadPool::remove(EventSource::Descriptor p_descriptor)
{
  std::unique_lock<std::mutex> lock(m_mutex);
  t_handlers::iterator i = m_handlers.find(p_descriptor);
  if (i != m_handlers.end())
  {
    m_handlers.erase(i);
    m_eventDemultiplexer->remove(p_descriptor);
  }
}

void ThreadPool::eventLoop()
{
  m_run = true;
  while (m_run)
  {
    EventDemultiplexer::Events events;
    m_eventDemultiplexer->wait(events);

    t_toHandles toHandles;
    {//scope of m_mutex lock begin
      std::unique_lock<std::mutex> lock(m_mutex);
      for (auto i = events.begin(); i < events.end(); ++i)
      {
        auto ih = m_handlers.find(i->descriptor);
        if (ih != m_handlers.end())
        {
          toHandles.push_back(t_toHandle(ih->second,
          i->eventTypes));
        }
        else
        {
          throw std::runtime_error("EventDemultiplexer returned
          event for \ unfounded handler");
        }
      }
    }//scope of m_mutex lock end

    for (auto i = toHandles.begin(); i != toHandles.end(); ++i)
    {
      t_toHandle th = *i;
      Threading::ThreadPool::Task t = [th](void)->void { th.first-
      >handle(th.second); };
      m_threadPool.add(t);
    }
  } //while (1)
}

void ThreadPool::interrupt(bool p_immediately)
{
  m_run = false;
  m_eventDemultiplexer->interrupt();
}
} //namespace EventEngines
```

Zastanawiające może być pytanie, w jakim celu pojawiły się dwie nowe metody:

- `modify`
- `interrupt`

Pierwsza pozwala na modyfikację maski zdarzeń dla danego źródła i związanego z nim handler'em, ale jej cel rozjaśnię nieco później.

Druga pozwala na przerwanie pętli reaktywnego oczekiwania na zdarzenia i funkcji `wait` obiektu `EventDemultiplexer`. Jest konieczna, gdyż obsługa błędów przetwarzania zdarzeń nie może odbyć się przez proste rzucenie wyjątku tak, jak to miało miejsce w reaktorze, gdyż samo zdarzenie jest obsługiwane poza główną pętlą silnika (`eventLoop`) w innym wątku!

Taka postać rzeczy wymaga wprowadzenia równoważnych metod do interfejsu (i jego implementacji) `EventDemultiplexer` (`Epoll` i `EpollED`), co uczyniłem, ale zaprezentuję dosłownie za chwilę.

Kolejną ważną zmianą jest obsługa samych zdarzeń. Odbywa się ona w takiej samej pętli jak w przypadku reaktora, z tym że nie jest w niej bezpośrednio wołana funkcja obsługi zdarzenia z odpowiedzialnego handler'a, lecz jest to delegowane jako zadanie do puli wątków (część funkcji `eventLoop` zaznaczona na czerwono w Listingu 2). Do tego celu wykorzystałem wyrażenia lambda, których składnia bazowa prezentuje się następująco:

Listing 3. Składnia wyrażeń lambda

```
[/*zmienne lokalne przekazywane do ciała lambdy*/]
(/*argumenty wejściowe*/)
-> /*typ zwracany*/
{/*ciało funkcji definiowanej jako lambda*/};
```

Istotnym detalem jest fakt, że pula wątków będąca składową silnika jest startowana w konstruktorze TP i zatrzymywana w destruktorze TP, zachowując idiom RAII, co jest zbieżne z implementacją poprzednika (`Ractor`).

ZMIANY W IO

Napisałem już wyżej, że nasz obiekt `Epoll` musi być rozszerzony o dwie metody: `interrupt` i `modify`.

Tej pierwszej uzasadnienie wydaje się być oczywiste i właściwie takie jest: w niektórych wypadkach musi istnieć konieczność kontrolowanego przerwania blokującego wywołania funkcji systemowej (np. na okoliczność zakończenia aplikacji). Jest to właściwie dyktowane logiką i funkcjonowaniem wzorca TP.

W przypadku `interrupt` sprawa jest podyktowana samym demultiplekserem systemowym (epoll'em). Sedno tkwi w odpowiedzi na pytanie, jak zachowa się drugie wywołanie funkcji systemowej `epoll_wait`, w sytuacji gdy zdarzenie zgłoszone w poprzednim jej wywołaniu nie zostało jeszcze obsłużone. Otóż w przypadku dotychczas używanych masek zdarzeń, takie drugie wywołanie zostanie przerwane natychmiast, zgłaszając te same zdarzenia, co za pierwszym razem. Jak temu zaradzić? Tak się składa, że systemowy epoll daje nam dwie możliwości:

1. do maski zdarzeń dodajemy typ `EPOLLET`, który powoduje, że epoll „zapomina na chwilkę" o danym gnieździe, dopóki ostatnio zgłoszone zdarzenie nie zostanie obsłużone
2. lub dokładamy do maski zdarzeń `EPOLLONESHOT`, który powoduje, że dla danego gniazda epoll zgłosi zdarzenie jednorazowo, a by zgłaszał kolejne, trzeba dla danego gniazda zmodyfikować maskę zdarzeń (stąd konieczność `modify`).

Ważne jest to, że w obu przypadkach, gdy epoll zapomina na chwilę o danych gniazdach, a w tym czasie przyjdą nowe, epoll natychmiast je zgłosi: w pierwszym przypadku po obsłużeniu uprzednich zdarzeń oraz w drugim zaraz po modyfikacji maski.

Dla uproszczenia w listingu poniżej zamieściłem jedynie kod ważny z punktu widzenia zmian, jakie nastąpiły.

Listing 4. Rozszerzona implementacja klasy `Epoll`

```
class Epoll
{
public:
  typedef std::shared_ptr<Epoll> Ptr;

  struct EventType
  {
    static int Error;
    static int In;
    static int Out;
    static int Hup;
    static int RdHup;
    static int EdgeTriggered;
    static int OneShot;
  };
  typedef std::set<int> EventTypes;

  struct Event
  {
    Event(const EventTypes&, Socket::Descriptor);
    EventTypes eventTypes;
    Socket::Descriptor descriptor;
  };
  typedef std::vector<Event> Events;

public:
  explicit Epoll(int p_size = 100);
  virtual ~Epoll();

private:
  Epoll(const Epoll&);
  Epoll& operator=(const Epoll&);

public:
  virtual void add(Socket::Ptr, const EventTypes&) const;
  virtual void modify(Socket::Ptr, const EventTypes&) const;
  virtual void remove(Socket::Descriptor) const;
  virtual void wait(Events&) const;
  virtual void interrupt() const;

private:
  void epollCtl(int, int, const EventTypes&) const;

private:
  const int m_size;
  const int m_epollFd;
  int m_selfStopPipes[2];
};

Epoll::Event::Event(const EventTypes& p_ets, Socket::Descriptor p_fd)
  : eventTypes(p_ets), descriptor(p_fd)
{
}

Epoll::Epoll(int p_size)
  : m_size(p_size), m_epollFd(::epoll_create(m_size))
{
  if (m_epollFd < 0)
  {
    throw std::runtime_error("Epoll create failed");
  }

  int pipeResult = pipe(m_selfStopPipes);
  if (pipeResult < 0)
  {
    ::close(m_epollFd);
    throw std::runtime_error("Pipe create failed");
  }

  int readPipeFlags = fcntl(m_selfStopPipes[0], F_GETFL, 0);
  fcntl(m_selfStopPipes[1], F_SETFL, readPipeFlags | O_NONBLOCK);
  EventTypes eventTypesReadPipe;
  eventTypesReadPipe.insert(Epoll::EventType::In);
  try
  {
    epollCtl(EPOLL_CTL_ADD, m_selfStopPipes[0], eventTypesReadPipe);
  }
  catch (const std::runtime_error& e)
  {
    ::close(m_selfStopPipes[0]);
    ::close(m_selfStopPipes[1]);
    ::close(m_epollFd);
    throw e;
  }
}

Epoll::~Epoll()
{
  ::close(m_epollFd);
}

void Epoll::add(Socket::Ptr p_socket, const EventTypes& p_ets) const
{
  int option = EPOLL_CTL_ADD;
  epollCtl(option, p_socket->getDescriptor(), p_ets);
}

void Epoll::modify(Socket::Ptr p_socket, const EventTypes& p_ets)
```

```
const
{
  int option = EPOLL_CTL_MOD;
  epollCtl(option, p_socket->getDescriptor(), p_ets);
}

void Epoll::remove(Socket::Descriptor p_fd) const
{
  if (::epoll_ctl(m_epollFd, EPOLL_CTL_DEL, p_fd, 0) < 0)
  {
    throw ::std::runtime_error("Cannot remove fd from epoll");
  }
}

void Epoll::wait(Events& p_events) const
{
  boost::scoped_array<epoll_event> events(new epoll_event[m_size]);
  ::memset(events.get(), 0, m_size * sizeof (epoll_event));

  int eventsLen = ::epoll_wait(m_epollFd, events.get(), m_size, -1);
  if (eventsLen < 0)
  {
    throw ::std::runtime_error("Epoll wait failed");
  }

  for (int i = 0; i < eventsLen; ++i)
  {
    if (events[i].data.fd == m_selfStopPipes[0])
    {
      p_events.clear();
      break;
    }
    else
    {
      EventTypes ets;
      nativeEvents(events[i].events, ets);
      p_events.push_back(Event(ets, events[i].data.fd));
    }
  }
}

void Epoll::interrupt() const
{
  char c = 'x';
  if (write(m_selfStopPipes[1], &c, 1) < 1)
  {
    throw std::runtime_error("Interrupt failed since write to pipe
    failed");
  }
}

void Epoll::epollCtl(int p_option, int p_fd, const EventTypes&
p_ets) const
{
  epoll_event e;
  ::memset(&e, 0, sizeof(e));
  e.data.fd = p_fd;
  e.events = nativeEvents(p_ets);

  if (::epoll_ctl(m_epollFd, p_option, p_fd, &e) < 0)
  {
    throw std::runtime_error("Epoll ctl failed");
  }
}
```

Warto zwrócić uwagę, w jaki sposób została wykonana funkcja `interrupt` – jest ona zrealizowana z użyciem prywatnego potoku, którego zdarzenia zna i rozpoznaje sama klasa `Epoll` bez żadnej ekspozycji do publicznego API.

Zmiany w aplikacji docelowej

Teraz już główne zmiany w kodzie aplikacji docelowej:

Listing 5. Implementacja klas aplikacyjnych (obsługujących logikę protokołu warstwy aplikacji i logiki aplikacji)

```
void EpollED::modify(EventSource::Ptr p_es)
{
  SocketES::Ptr sES = std::dynamic_pointer_cast<SocketES>(p_es);
  m_epoll->modify(sES->getSocket(), p_es->getEventTypes());
}

void EpollED::interrupt()
{
  m_epoll->interrupt();
}

ListenerES::ListenerES(TcpSocket::Ptr p_tcpSocket, int p_port)
  : SocketES(p_tcpSocket)
{
  m_eventTypes.insert(Epoll::EventType::Error);
  m_eventTypes.insert(Epoll::EventType::In);
  m_eventTypes.insert(Epoll::EventType::Hup);
  m_eventTypes.insert(Epoll::EventType::RdHup);
  m_eventTypes.insert(Epoll::EventType::OneShot);

  p_tcpSocket->setNonBlocking();
  p_tcpSocket->bind(p_port);
  p_tcpSocket->listen();
}

MessageES::MessageES(TcpSocket::Ptr p_tcpSocket)
  : SocketES(p_tcpSocket)
{
  m_eventTypes.insert(Epoll::EventType::Error);
  m_eventTypes.insert(Epoll::EventType::In);
  m_eventTypes.insert(Epoll::EventType::Hup);
  m_eventTypes.insert(Epoll::EventType::RdHup);
  m_eventTypes.insert(Epoll::EventType::OneShot);

  m_socket->setNonBlocking();
}

KeyboardES::KeyboardES(KeyboardSocket::Ptr p_keybSocket)
  : SocketES(p_keybSocket)
{
  m_eventTypes.insert(Epoll::EventType::Error);
  m_eventTypes.insert(Epoll::EventType::In);
  m_eventTypes.insert(Epoll::EventType::OneShot);

  m_socket->setNonBlocking();
}

void AcceptorEH::handle(const EventSource::EventTypes&
p_eventTypes)
{
  EventSource::EventTypes::const_iterator iIn = p_eventTypes.
  find(Epoll::EventType::In);
  if ( (iIn != p_eventTypes.end()) && (p_eventTypes.size() == 1) )
  {
    ListenerES::Ptr listenerES =
    std::dynamic_pointer_cast<ListenerES>(m_eventSource);
    MessageES::Ptr msgES = listenerES->accept();
    EchoResponderEH::Ptr erEH(new EchoResponderEH(msgES,
    m_threadPool));
    m_threadPool.add(erEH);
    m_threadPool.modify(m_eventSource);
  }
  else
  {
    throw std::runtime_error("Bad event for for acceptor");
  }
}

void EchoResponderEH::handle(const EventSource::EventTypes&
p_eventTypes)
{
  EventSource::EventTypes::const_iterator iIn =
    p_eventTypes.find(Epoll::EventType::In);
  if ( (iIn != p_eventTypes.end()) && (p_eventTypes.size() == 1) )
  {
    MessageES::Ptr mES =
    std::dynamic_pointer_cast<MessageES>(m_eventSource);
    std::string data;
    while (!boost::ends_with(data, "\n"))
    {
      std::string part;
      mES->read(part);
      data.append(part);
    }
    mES->getSocket()->write(data);
    m_threadPool.modify(m_eventSource);
  }
  else
  {
    m_threadPool.remove(m_eventSource->getDescriptor());
  }
}

void KeyboardEH::handle(const EventSource::EventTypes& p_eventTypes)
{
  EventSource::EventTypes::const_iterator iIn = p_eventTypes.
  find(Epoll::EventType::In);
  if ( (iIn != p_eventTypes.end()) && (p_eventTypes.size() == 1) )
  {
    SocketES::Ptr socketES =
    std::dynamic_pointer_cast<SocketES>(m_eventSource);
    KeyboardES::Ptr keyboardES =
    std::dynamic_pointer_cast<KeyboardES>(socketES);
    std::string data;
    keyboardES->read(data);
    m_file << data;

    if (boost::istarts_with(data, "exit"))
    {
      m_threadPool.interrupt(true);
    }
  }
  else
  {
    throw std::runtime_error("Bad event for for acceptor");
  }
}
```

Pierwsza widoczna zmiana to oczywiście rozszerzenie klasy reprezentującej `EventDemultiplexer` w oparciu o `Epoll` (`EpollED`) o wymagane nowe dwie proxy-metody: `modify` i `interrupt`, omówione już wyżej. Dodatkowa zmiana, którą zaznaczyłem, to wykorzystanie funkcji `std::dynamic_pointer_cast`, która zastąpiła swój ekwiwalent z `boost`, co znaczy ni mniej, ni więcej, że inteligentny wskaźnik `boost::shared_ptr` zastąpiłem jego odpowiednikiem z biblioteki standardowej C++11: `std::shared_ptr` o tych samych właściwościach.

Kolejna seria zmian to konstruktory klas źródeł zdarzeń, w których każde ze źródeł dokłada typ zdarzenia `Epoll::EventType::OneShot` z przyczyn podanych przy okazji omawiania zmian w `Epoll`.

Zwieńczeniem zmian związanych z wielowątkową obsługą zdarzeń są zmiany w klasach `*EH` (`EventHandler`'y):

1. Każdy z EH trzyma referencję do silnika obsługi zdarzeń (`ThreadPool`), co jest naturalną konsekwencją zmiany typu silnika z `Reactor` na `ThreadPool`.
2. Każdy z EH po rdzennej obsłudze zdarzeń wywołuje akcję modyfikacji maski zdarzeń dla trzymanego źródła, co jest wprost związane z wykorzystaniem typu `Epoll::EventType::OneShot`.
3. Zmieniła się nieco obsługa błędów – zamiast naiwnego rzucania wyjątków, główna pętla silnika obsługi zdarzeń przerywana jest głównie przez dedykowaną metodę `ThreadPool::interrupt` (rzucenie wyjątku doprowadziłoby do nieoczekiwanego zamknięcia programu, gdyż wątki przetwarzające zdarzenia nie spodziewają się ich zupełnie).

Main

Tradycyjnie na końcu prezentuję zwieńczenie naszych starań, czyli treść funkcji `main`:

Listing 6. Implementacja funkcji main

```
int main(int, char**)
{
  try
  {
    Epoll::Ptr epoll(new Epoll());
    EpollED::Ptr epollED(new EpollED(epoll));
    EventEngines::ThreadPool tp(epollED, 2);

    KeyboardSocket::Ptr keybSocket(new KeyboardSocket());
    KeyboardES::Ptr keybES(new KeyboardES(keybSocket));
    EventHandler::Ptr keybEH(new KeyboardEH("log.txt", keybES, tp));

    TcpSocket::Ptr listenerSokcet(new TcpSocket());
    ListenerES::Ptr listenerES(new ListenerES(listenerSokcet, 5050));
    EventHandler::Ptr acceptorEH(new AcceptorEH(listenerES, tp));

    tp.add(keybEH);
    tp.add(acceptorEH);
    tp.eventLoop();
  }
  catch (const std::runtime_error& rte)
  {
    std::cout << "Runtime exception: " << rte.what() << std::endl;
  }
  catch (const std::exception& e)
  {
    std::cout << "STD exception: " << e.what() << std::endl;
  }
  return 0;
}
```

Zwróćmy uwagę: zmieniło się właściwie tylko 6 linijek funkcjonalnych! I są one związane tylko i wyłącznie ze zmianą silnika obsługi zdarzeń z `Reactor` na `ThreadPool`:

- fabrykacja silnika `ThreadPool`;
- przekazanie referencji do TP
- dodanie do TP odpowiednich EH
- wystartowanie głównej pętli obsługi zdarzeń TP.

WNIOSKI

Mocne strony

- Dzięki separacji logiki domenowej i aplikacyjnej wykonanej w poprzednim artykule, zmiany w projekcie (mimo tego, że są znaczące, bowiem zmienia się cały model wielowątkowości) były mało inwazyjne, a starcie z podatnym na błędy natywnym API C zostały ograniczone do minimum (dodanie dwóch metod).
- Dzięki wykorzystaniu wzorca TP aplikację można spokojnie wyskalować, dostosowując ją do średniego natężenia ruchu zależnego od ilości klientów i wielkości wymienianych wiadomości.
- Dzięki nienaruszeniu bazowej architektury obiektowej utrzymaliśmy proste zależności między obiektami i poziom testowalności kodu.
- Kod nie stracił (przynajmniej w porównaniu do poprzedniej wersji) na przejrzystości i czytelności.

Słabe strony

- Może się wydawać, że takie podejście do tak prostego zadania (serwer echo) jest nieco przesadzone, niemniej jednak jest to tylko pokaz koncepcji, w którą bardzo łatwo wszczepić dowolny protokół powyżej warstwy czwartej modelu OSI (np. HTTP).
- Standardowo stan obsługi błędów nadal pozostawia sporo do życzenia, choć nadal w 100% się sprawdza.
- Rozmiar TP jest podany na sztywno, a mógłby być przynajmniej wczytywany z linii argumentów wywołania programu.

ZAKOŃCZENIE

Tradycyjnie, zainteresowanych zachęcam do pobrania kodu (link w ramce „W sieci") i jego przetestowania. Gorąco polecam przeprowadzenie testów porównawczych wszystkich przedstawianych do tej pory rozwiązań przynajmniej w trzech wariantach:

1. małego natężenia ruchu
2. ekstremalnie dużego natężenia ruchu
3. optymalnego – średnia wyliczona z dwóch powyższych wariantów.

W następnym artykule do naszego projektu dołożę kolejny z ważnych wzorców będącym połączeniem podejścia reaktywnego (prezentowanego uprzednio) i wielowątkowego (przedstawionego dziś).

W sieci

- Kod źródłowy aplikacji z części I:
  https://github.com/RomanUlan/ReactorBase
- Kody źródłowe aplikacji z części II:
  https://github.com/RomanUlan/ReactorInheritance
  https://github.com/RomanUlan/ReactorTemplates
- Kod źródłowy aplikacji prezentowanej w niniejszej części:
  https://github.com/RomanUlan/ThreadPool

Roman Ulan roman.ulan@gmail.com

Senior Software Architect w Tieto Poland Sp. z o.o. Trener C++ w Bottega IT Solutions. Specjalizuje się w rozwiązaniach backend'owych w języku C++. Entuzjasta czystego i testowalnego kodu osadzonego w przemyślanej architekturze opartej na sprawdzonych wzorcach. Interesuje go programowanie równoległe i rozproszone oraz zagadnienia „software design".