Boost.Asio: synchroniczne i asynchroniczne programowanie serwisów sieciowych w C++

Wojciech Wiśniewski[1]

[1] Mobica

Luty 2016

Program prezentacji

Co to jest Boost.Asio?

Asio (Asynchronous input-output) jest biblioteką służącą do pisania przenośnych i wydajnych programów, które muszą współbieżnie reagować na zdarzenia pochodzące z wielu źródeł:

- gniazdek sieciowych (sockets)
- zegarów (timers)
- sygnałów
- portów szeregowych

(typowy przypadek: serwis sieciowy obsługujący wielu klientów równocześnie)
Obsługuje zarówno synchroniczny, jak i asynchroniczny styl programowania.

Synchroniczna i asynchroniczna obsługa wejścia/wyjścia: definicje

Synchroniczność i asynchroniczność dotyczą sposobu obsługi (potencjalnie) długo trwających zadań wykonywanych przez urządzenia zewnętrzne (wejścia/wyjścia, np. oczekiwania na dane z sieci).

Obsługa synchroniczna

W podejściu synchroniznym program inicjuje operację wejścia/wyjścia, a następnie zostaje zablokowany do czasu jej zakończenia (oddaje sterowanie do systemu operacyjnego, zostaje „uśpiony"). Obsługę wielu operacji jednocześnie uzyskujemy poprzez użycie wielu programów lub wątków.

Obsługa asynchroniczna

W podejściu asynchronicznym program inicjuje (w sposób nieblokujący) operację wejścia/wyjścia, a następnie może być powiadamiany (w sposób blokujący) o jej statusie. Obsługę wielu operacji jednocześnie uzyskujemy inicjując wiele operacji bez czekania na ich zakończenie.

Synchroniczna i asynchroniczna obsługa wejścia/wyjścia: systemy operacyjne

API do obsługi synchronicznego wejścia-wyjścia w głównych systemach operacyjnych jest podobne (`read`/`write`, `recv`/`send`).
API do obsługi asynchronicznej mocno różnią się między systemami:

- Linux – `epoll`
- *BSD, MacOS X – `kqueue`
- Windows – overlapped I/O, completion ports

Boost.Asio tworzy niezależną od sytemu warstwę abstrakcji ponad tymi niskopoziomowymi API i ustanawia model programowania asynchronicznego, w którym rejestrujemy funkcje (lub inne wywoływalne obiekty), które będą wywołane, gdy zainicjowana operacja się zakończy.

Nasz przykład

Napiszemy ten sam prosty program (serwer czatu) w trzech różnych stylach:

- synchronicznym wielowątkowym
- asynchronicznym z użyciem callbacków
- asynchronicznym z użyciem wątków kooperatywnych

i porównamy wydajność, łatowść pisania oraz podatność na modyfikacje programów napisanch w powyższych stylach.

Specyfikacja serwera czatu

- Serwer nasłuchuje na porcie TCP podanym z linii poleceń.
- Programem klienckim będzie `netcat`.
- Po połączeniu serwer pyta klienta nick. Serwer dba o unikalność nicków zalogowancych w danej chwili klientów, pytanie o nick będzie powtarzane dopóki klient nie poda unikalnego nicka.
- Zalogowani klienci dostają (na zasadzie jeden-do-wszystkich) komunikaty o zdarzeniach: logowania i rozłączenia innych klientów oraz wpisywane przez nich linijki tekstu.
- Specjalne komendy dostępne dla klientów:
  - `/quit` powoduje rozłączenie klienta
  - `/shutdown` powoduje zamknięcie całego serwera
- Serwer powinien obsłużyć niepowodzenie dowolnej operacji sieciowej.
- Serwer powinien być wolny od wyścigów i niepoprawnego użycia pamięci (sprawdzane przy pomocy `valgrinda`).

Klasy

Dwie klasy: `ChatServer` i `ClientSession`

ChatServer: interfejs pulibczny

```
class ChatServer {
  public:
  // interfejs dla programu glownego

  ChatServer(int port);
  ~ChatServer();

  // Petla glowna
  void run();

  // interfejs dla klasy ClientSession

  // Sprawdz, czy nick jest juz zarejestrowany, jesli nie, to zarejestruj go i ustaw w kliencie.
  // Zwraca true w przypadku sukcesu i false w przypadku porazki.
  bool setClientName(const std::shared_ptr<ClientSession>& client, const std::string &name);

  // Rozeslij wiadomosc do wszystkich klientow
  void broadcast(ClientSession& sender, const std::shared_ptr<std::string>& msg);

  // Usun klienta z listy zarejestrowanych klientow
  void removeClient(std::shared_ptr<ClientSession>&& client);

  // Zamknij serwer (zakoncz petle glowna)
  void shutdown();
};
```

ClientSession: interfejs publiczny (dla klasy ChatServer)

```
class ClientSession : public std::enable_shared_from_this<ClientSession> {
public:
  ClientSession(ChatServer& server, boost::asio::io_service &ioService);

  boost::asio::ip::tcp::socket& socket() {
    return _socket;
  }

  // Pobiez nicka (jesli ustawiony)
  const std::string* getName() const {
    return _nameValid ? &_name : nullptr;
  }

  // Ustaw nicka
  void setName(const std::string &name) {
    assert(! _nameValid);
    _name = name;
    _nameValid = true;
  }

  // Rozpocznij dialog z klientem
  void start();

  // Wyslij wiadomosc do klienta
  void sendMessage(const std::shared_ptr<std::string>& msg);

  // Zakoncz dialog z klientem
  void terminate();

private:
  ChatServer& _server;
  boost::asio::ip::tcp::socket _socket;
  std::string _name;
  bool _nameValid;
};
```

std::enable_shared_from_this

Dziedziczenie po `std::enable_shared_from_this` umożliwia pobieranie sprytnych wskaźników do instancji tej klasy wewnątrz jej metod (klasa dostarcza metodę `shared_from_this`).
Zastosowana przez `enable_shared_from_this` technika dodawania funkcjonalności do klas poprzez dziedziczenie z szablonowej klasy bazowej sparametryzowanej przez klasę pochodną nazywa się „curiously recurring template pattern".

boost::asio::io_service

`io_service` to klasa zarządzająca operacjami i obiektami wejścia/wyjścia. W wersji asynchronicznej dostarcza pętlę główną (`io_service::run()`), której w tej wersji nie będziemy używać.
Wszystkie obiekty wejścia/wyjścia w Asio (gniazdka, zegary, itd.) dziedziczą po `basic_io_object` i rejestrują się `io_service` podczas konstrukcji.

Implementacja ChatService::run

```
class ChatServer {
public:
  ChatServer(int port);
private:
  boost::asio::ip::tcp::acceptor _acceptor;
  std::mutex _clientsMutex;
  std::set<std::shared_ptr<ClientSession>> _clients;
  pthread_t _acceptingThreadId;
  std::atomic<bool> _isTerminating;
};

ChatServer::ChatServer(int port) :
_acceptor(_ioService, boost::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4(), port)),
_isTerminating(false) { }

void ChatServer::run() {
  _acceptingThreadId = pthread_self();
  while (!_isTerminating) {
    std::shared_ptr<ClientSession> client = std::make_shared<ClientSession>(*this, _ioService);
    _acceptor.accept(client->socket());
    std::lock_guard<std::mutex> guard(_clientsMutex);
    if (!_isTerminating) {
      auto p = _clients.insert(client);
      assert(p.second);
      client->start();
    }
  }
}
```

boost::asio::ip::tcp::acceptor

`acceptor` służy do przyjmowania nowych połączeń (wrapper na funkcje systemowe `bind`, `listen` i `accept`).
Konstruktor (oprócz `io_service`), przyjmuje obiekt typu `endpoint`, reprezentujący specyfikację nasłuchiwania – trójkę: protokół, adres (u nas pusty: słuchamy na wszystkich interface'ach), port.
Metoda `acceptor::accept` w blokujący sposób oczekuje na nowe przychodzące połączenie i umieszcza je w przekazanym `socket`'cie.

std::mutex

`mutex` to podstawowy sposób wykluczania wzajemnego między wątkami.
Standard C++11 oferuje cztery rodzaje mutexów: `mutex`, `recursive_mutex`, `timed_mutex` i `recursive_timed_mutex`. Wszystkie mutexy obsługują operacje `lock`, `try_lock` i `unlock`. Warianty `timed` obsługują dodatkowo próby blokowania z limitem czasu: `try_lock_for` i `try_lock_until`. Wersje `recursive` umożliwiają wielokrotne blokowanie przez ten sam wątek.
Używanie rekursywnych mutexów jest niezalecane przez niektóre poradniki dobrego stylu programowania.

std::lock_guard

`lock_guard` umożliwia zarządzanie blokowaniem mutexów poprzez RAII (Resource Acquisition Is Initialization): podczas konstrukcji blokuje mutex, przy destrukcji go zwalnia.

std::make_shared

Funkcja `make_shared<T>` alokuje przy pomocy jednego wywołania `new` miejsce na obiekt klasy `T` i obiekt licznik odwołań dla `shared_ptr`. Następnie wywołuje konstruktor klasy `T` z podanymi argumentami.
Jest to przykład tzw. „perfect forwarding".

Implementacja ClientSession::start

```
class ClientSession {
private:
  void readerThread();
  void writerThread();

  std::thread _readerThread;
  std::thread _writerThread;
};

void ClientSession::start() {
  _readerThread = std::thread(std::bind(&ClientSession::readerThread, this));
  _writerThread = std::thread(std::bind(&ClientSession::writerThread, this));
}
```

`std::thread`

`thread` reprezentuje wątek. Konstruktor pobiera dowolny wywoływalny obiekt o sygnaturze `void ()`. Wątek startuje od razu w momencie konstrukcji (nie ma metody `start`).

std::bind

`bind` pozwala nam „ustalić” niektóre (lub wszystkie) argumenty obiektu wywoływalnego i zwrócić nowy wywoływalny obiekt o sygnaturze z usuniętymi „ustalonymi” argumentami.
`bind` jako argumenty przyjmuje obiekt wywoływalny i takie argumenty, jakie przyjmowałoby jego wywołanie, albo specjalne obiekty `_1`, `_2`, itd., żyjące w namespace `std::placeholders`. Specjalne obiekty placeholders specyfikują, jak będzie wyglądało wywołanie nowego obiektu funkcyjnego.
W naszym przypadku obiektem wywoływalnym jest wskaźnik do bezargumentowej metody klasy `ClientSession`. Wywołanie takiego obiektu potrzebuje jednego argumentu: wskaźnika lub refererencji do obiektu `ClientSession`. `bind` zaspokaja więc wszystkie argumenty obiektu wywoływalnego i powstaje nowy obiekt o sygnaturze `void ()` – takiej, jakiej potrzebuje konstruktor `thread`.

Implementacja ClientSession::readerThread

```
class ClientSession {
private:
  enum {
    ALL_RUNNING = 0,
    READER_TERMINATED = 1,
    WRITER_TERMINATED = 2,
    READER_TERMINATION_REQUESTED = 4
  };

  std::string readLineFromClient();
  bool parseLine(const std::string& line);
  void onReaderShutdown();

  std::atomic<int> _state;
};

ClientSession::ClientSession(ChatServer& server, boost::asio::io_service &ioService) :
_state(ALL_RUNNING) { }
```

Implementacja ClientSession::readerThread ciąg dalszy

```
void ClientSession::readerThread() {
  try {
    bool loginSuccessfull = false;
    while (! loginSuccessfull && _state == ALL_RUNNING) {
      boost::asio::write(_socket, boost::asio::buffer("What's your name?\n"));
      std::string name = readLineFromClient();
      if ((loginSuccessfull = _server.setClientName(shared_from_this(), name))) {
        std::string response = "Welcome to the chat, " + name + "!\n";
        boost::asio::write(_socket, boost::asio::buffer(response));
      }
      else {
        std::string response = "Name '" + name + "' is already taken, invent another one.\n";
        boost::asio::write(_socket, boost::asio::buffer(response));
      }
    }

    while (parseLine(readLineFromClient()));
  }
  catch (std::exception& ex) {
    const std::string *name = getName();
    std::string formattedName = name ? "'" + *name + "'" : "(null)";
    std::cout << "Client " << formattedName << " reader thread exeption: " << ex.what() << std::endl;
  }

  onReaderShutdown();
}
```

boost::asio::write

`write` to przeciążona funkcja szablonowa. Tutaj używamy ją z minimalną liczbą argumentów. Pierwszym argumentem jest obiekt zapisywalny (implementujący metodę `write_some`), drugim jest bufor wyjściowy (koncept `ConstBufferSequence` – coś z czego można wyciągnąć listę obszarów pamięci zawierających dane do zapisania).

boost::asio::buffer

`buffer` to przeciążona funkcja szablonowa, przekształcająca różne obiekty (zwykła tablica, `std::array`, `boost::array`, wskaźnik + rozmiar, `std::string`, itd.) w obiekty odpowiadające konceptowi `MutableBufferSequence` lub `ConstBufferSequence`.

std::atomic

Obiekty typu `atomic<T>` w większości wypadków zachowują się jak zwykłe zmienne typu `T`: mają konwersję niejawną konwersję na `T` (wołającą `atomic<T>::load()`) i operator przypisania biorący `T` jako argument (wołający `atomic<T>::store(T)`). Używając obiektów `atomic` mamy pewność, że w przypadku, gdy jeden wątek ustawia zapisuje daną zamienną, a inny ją odczytuje, wątek czytający zawsze odczyta aktualną wartość. Nie jest to zagwarantowane w przypadku zwykłych wartości (nawet typów prostych).

Obsługa błędów

Ponieważ sterowanie przepływa „wprost” (stan programu jest odzwierciedlony przez stos), możemy sygnalizować błędy za pomocą wyjątków i łapać je w jednym miejscu.
Wyjątki nie powinny opuszczać głównej funkcji wątku (spowodowałoby to wywołanie `std::terminate()` i zakończenie programu).
Wszystkie operacje blokujące (widzieliśmy już `boost::asio::accept` i `boost::asio::write`) rzucają wyjątki typu `boost::system::system_error`, dziedziczące po `std::exception`.
Jeśli obsługa błędów poprzez wyjątki nam nie odpowiada, każda operacja blokująca ma wariant z dodatkowym parametrem wyjściowym typu `boost::system::error_code`, który można przetestować na okoliczność błędu.
Zobaczymy to w wersji asynchronicznej.

Implementacja ClientSession::readLineFromClient

```
class ClientSession {
private:
  boost::asio::streambuf _inputBuffer;
};

std::string ClientSession::readLineFromClient() {
  boost::asio::read_until(_socket, _inputBuffer, '\n');
  std::istream stream(&_inputBuffer);
  std::string line;
  std::getline(stream, line);
  return line;
}
```

boost::asio::streambuf

`boost::asio::steambuf` to rozszerzalny bufor dziedziczący z `std::streambuf`. Jak widzimy, można z niego w łatwy sposób wyciągać dane po linijce.

boost::asio::read_until

`read_until` jest operacją blokującą (przeciążoną funkcją szablonową), która czyta, dopóki dane znajdujące się w buforze nie spełniają określonego warunku.
Pierwszym argumentem jest obiekt odczytywalny, czyli posiadający metodę `read_some`.
Drugim argumentem jest `boost::asio::streambuf`.
Trzecim argumentem jest warunek końca. W naszym przypadku czytamy, aż w buforze nie znajdzie się co najmniej jeden znak końca linii, ale można też czekać na określony napis (`std::string`), aż dane spełnią zadane wyrażenie regularne (`boost::regex`) lub dowolny warunek, sprawdzany przy pomocy obiektu wywoływalnego.

Implementacja `ClientSession::parseLine`

Ta metoda jest identyczna we wszystkich implementacjach. Używam biblioteki Boost.DateTime, a nie `std::chrono::time_point`, ponieważ nie znalazłem prostego sposobu zamiany `std::chrono::time_point` na tekst ☹.

```
bool ClientSession::parseLine(const std::string& line) {
  if (line == "/quit") {
    return false;
  }
  else if (line == "/shutdown") {
    _server.shutdown();
    return false;
  }
  else {
    std::ostringstream stream;
    stream << boost::posix_time::microsec_clock::local_time() << ' '
           << _name << ": " << line << std::endl;
    _server.broadcast(*this, std::make_shared<std::string>(stream.str()));
    return true;
  }
}
```

Implementacja ChatServer::setClientName

```
template <class T>
struct PtrLess {
  bool operator()(const T *lhs, const T *rhs) const {
    if (lhs == nullptr) {
      return rhs != nullptr;
    }
    else if (rhs == nullptr) {
      return false;
    }
    else {
      return *lhs < *rhs;
    }
  }
};

class ChatServer {
private:
  typedef std::map<const std::string*, std::shared_ptr<ClientSession>,
                   PtrLess<std::string>> NamesToClientsMap;

  std::mutex _namesToClientsMutex;
  NamesToClientsMap  _namesToClients;
}

bool ChatServer::setClientName(const std::shared_ptr<ClientSession>& client,
                               const std::string &name) {
  std::lock_guard<std::mutex> guard(_namesToClientsMutex);
  NamesToClientsMap::iterator found = _namesToClients.find(&name);
  if (found == _namesToClients.end()) {
    client->setName(name);
    auto res = _namesToClients.insert(std::make_pair(client->getName(), client));
    assert(res.second);
    return true;
  }
  else {
    return false;
```

Implementacja ChatServer::broadcast

Jedyną ciekawostką jest tu, że wykopiowujemy listę zalogowanych klientów z mapy, aby krócej blokować mutex.

```
void ChatServer::broadcast(ClientSession& sender,
                           const std::shared_ptr<std::string>& msg) {
  std::vector<std::shared_ptr<ClientSession>> clients;
  {
    std::lock_guard<std::mutex> guard(_namesToClientsMutex);
    clients.reserve(_namesToClients.size());
    for (const auto& kvPair : _namesToClients) {
      clients.push_back(kvPair.second);
    }
  }
  for (const auto& receiver : clients) {
    if (receiver.get() != &sender) {
      receiver->sendMessage(msg);
    }
  }
}
```

Implementacja ClientSession::sendMessage

```
class ClientSession {
private:
  std::mutex _messagesMutex;
  std::deque<std::shared_ptr<std::string>> _messages;
  std::condition_variable _writerCondition;
};

void ClientSession::sendMessage(const std::shared_ptr<std::string>& msg) {
  std::lock_guard<std::mutex> guard(_messagesMutex);
  _messages.push_back(msg);
  _writerCondition.notify_one();
}
```

`std::condition_variable`: powiadamiane

`condition_variable` to „urządzenie”, które usypia wątek, dopóki nie zostanie spełniony zadany warunek. Tutaj widzimy funkcjonalność budzenia. Na jednej `condition_variable` może spać wiele wątków, metoda `notify_one` budzi tylko jeden z nich, `notify_all` – wszystkie.

Implementacja ClientSession::writerThread

```
class ClientSession {
private:
  std::shared_ptr<std::string> getMessage();

  void onWriterShutdown();

};

void ClientSession::writerThread() {
  try {
    bool run = true;
    while (run) {
      auto msg = getMessage();
      if (msg) {
        boost::asio::write(_socket, boost::asio::buffer(*msg));
      }
      else {
        run = false;
      }
    }
  }
  catch (std::exception& ex) {
    const std::string *name = getName();
    std::string formattedName = name ? "'" + *name + "'" : "(null)";
    std::cout << "Client " << formattedName << " writer thread exception: " << ex.what() << std::endl;
  }

  onWriterShutdown();
}
```

Implementacja ClientSession::getMessage

```
std::shared_ptr<std::string> ClientSession::getMessage() {
  std::unique_lock<std::mutex> lock(_messagesMutex);
  _writerCondition.wait(lock, [this]() { return _state != ALL_RUNNING || !_messages.empty(); });
  if (_state != ALL_RUNNING) {
    return std::shared_ptr<std::string>();
  }
  else {
    auto msg = _messages.front();
    _messages.pop_front();
    return msg;
  }
}
```

std::unique_lock

`unique_lock` jest podobny do `lock_guard`. Różnicą jest to, że można używać go do wielokrotnego zamykania/zwalniania mutexu bez konstrukcji i destrukcji. `unique_lock` pamięta swój stan i podczas destrukcji zwolni mutex tylko jeśli będzie potrzeba. Można go też stworzyć w stanie niezablokowanym.

std::condition_variable: czekanie

Semantyka metody `wait`:

1. Blokujemy mutex broniący dostępu do danych, na których będziemy przeprowadzać test przy pomocy `unique_lock`
2. Do metody `wait` przekazujemy jako argumenty ten `unique_lock` i obiekt wywoływanly, który powinien zwrócić `true`, gdy oczekiwany warunek jest spełniony
3. `wait` sprawdza warunek, jeśli jest spełniony, wychodzi, jeśli nie, zwalnia mutex i usypia, czekając aż będie zawołane `notify_one` lub `notify_all`.
4. Po obudzeniu, mutex jest ponownie blokowany i wracamy do punktu 3

Kończenie wątków

W każdej chwili wątek czytający, jak i wątek piszący mogą zostać przerwane (np. na skutek wyjątku). Wątek kończący się wcześniej powinien spowodować przerwanie drugiego wątku, a drugi powinien poprosić serwer o derejestrację i usunięcie klienta. Odpowidzialne za to są metody `ClientSession::onReaderShutdown` i `ClientSession::onWriterShutdown`.

Implementacja ClientSession::onReaderShutdown i ClientSession::onWriterShutdown

```
class ClientSession {
  private:
  void interruptReader();
};

void ClientSession::onReaderShutdown() {
  int oldValue = _state.fetch_or(READER_TERMINATED);
  if (oldValue & WRITER_TERMINATED) {
    _server.removeClient(shared_from_this());
  }
  else {
    _writerCondition.notify_one();
  }
}

void ClientSession::onWriterShutdown() {
  int oldValue = _state.fetch_or(WRITER_TERMINATED);
  if (oldValue & READER_TERMINATED) {
    _server.removeClient(shared_from_this());
  }
  else {
    interruptReader();
  }
}
```

std::atomic::fetch_or

Operacja `fetch_or` jest równoznaczna z atomowym (nieprzerywalnym) wykonaniem takiego pseudokodu:

```
template <class T>
class atomic {
private:
  T _value;

public:
  T fetch_or(T mask) {
    T oldValue = _value;
    _value |= mask;
    return oldValue;
  }
};
```

Przerywanie operacji blokujących

Żeby zakończyć wątek czytelnika, musimy przerwać blokującą oprację `read_until`. Nestety, Boost.Asio nie zawiera API do przerywania operacji blokujących z innego wątku. Aby to zrobić na Linuxie musimy użyć niskopoziomowego API sygnałów. Tak więc nasz program nie będzie działał pod Windows ☹.

Implementacja ClientSession::interruptReader (i przy okazji main)

```
static void handler(int) { }

int main(int argc, char **argv) {
  struct sigaction action;
  memset(&action, 0, sizeof(action));
  action.sa_handler = handler;
  if (sigaction(SIGUSR1, &action, nullptr) != 0) {
    perror("sigaction");
    return 1;
  }

  try {
    if (argc < 2) {
      std::cerr << "Usage: " << argv[0] << " <port>\n";
      return 1;
    }
    ChatServer server(boost::lexical_cast<int>(argv[1]));
    server.run();
    return 0;
  }
  catch (std::exception &ex) {
    std::cerr << "Main thread exception: " << ex.what() << std::endl;
    return 1;
  }
}

void ClientSession::interruptReader() {
  _state.fetch_or(READER_TERMINATION_REQUESTED);
  pthread_kill(_readerThread.native_handle(), SIGUSR1);
}
```

Przerywanie operacji blokujących przez sygnały

Witajcie w latach '80! Oto przepis na przerywanie operacji blokujących.

1. Na początku życia programu przy pomocy funkcji `signal` lub `sigaction` rejestrujemy procedurę obsługi sygnału, która powinna mieć sygnaturę `void (int)`. Argumentem procedury obsługi jest numer obsługiwanego sygnału. Co robi ta procedura w naszym przypadku nie ma znaczenia: wystarczy pusta funkcja. Większość sygnałów jest przypisana zdarzeniom systemowym, więc program może je dostać z przyczyn niezależnych od nas, ale dwa są pozostawione do zastosowań zdefiniowanych przez użytkownika: `SIGUSR1` i `SIGUSR2`. Procedury obsługi sygnałów są globalne – dotyczą wszystkich wątków.
2. Pobieramy identyfikator wątku (`pthread_t`) przy pomocy metody `std::thread::native_id()`.
3. Wysyłamy do sygnał do wątku przy pomocy funkcji `pthread_kill`. Jeśli nie wyspecyfikujemy inaczej podczas rejestracji procedury, wykona się ona w kontekście wątku odbierającego sygnał i używając jego stosu. Jeśli wątek wykonuje kod użytkownika, nic specjalnego się nie wydarzy. Jeśli wykonuje funkcję systemową, zostanie ona przerwana, zwróci błąd, a `errno` zostanie ustawione na `EINTR` (interrupted system call). Boost.Asio przetłumaczy to na wyjątek.

Sprzątanie niepotrzebnych obiektów `ClientSession`

W jakim wątku powinny być niszczone obiekty `ClientSession`?

- Nie może to być ani wątek czytający, ani piszący `ClientSession`, ponieważ niszczenie obiektu `std::thread`, kiedy przebieg jego funkcji się jescze nie zakończył powoduje wywołanie `std::terminate` i zakończenie programu.
- Wątki innych `ClientSessions` mogą nie istnieć (jeśli jest tylko jeden połączony klient).
- Wątek główny może być dowolnie długo zablokowany w `acceptor::accept`.

Potrzebujemy jeszcze jednego wątku!

Implementacja ChatServer::removeClient

```
class ClientSession {
public:
  void waitToFinish();
};

void ClientSession::waitToFinish() {
  int state = _state.load();
  assert(state & READER_TERMINATED);
  assert(state & WRITER_TERMINATED);

  _readerThread.join();
  _writerThread.join();
}

class ChatServer {
private:
  std::mutex _clientsToRemoveMutex;
  std::deque<std::shared_ptr<ClientSession>> _clientsToRemove;
  std::condition_variable _reaperCondition;
};

void ChatServer::removeClient(std::shared_ptr<ClientSession>&& client) {
  std::lock_guard<std::mutex> guard(_clientsToRemoveMutex);
  _clientsToRemove.push_back(std::move(client));
  _reaperCondition.notify_one();
}
```

`std::thread::join` i `std::atomic<T>::load`

Metoda `thread::join` czeka, aż funkcja wątku się zakończy.
Metoda `atomic<T>::load` explicite ładuje aktualną wartość zmiennej atomowej z pamięci (przeprowadza synchronizację).

Implementacja ChatServer::reaperThread

```
class ChatServer {
private:
  void reaperThread();
  std::shared_ptr<ClientSession> getClientToRemove();

  std::thread _reaperThread;
  std::atomic<bool> _isTerminating;
};


void ChatServer::reaperThread() {
  try {
    while (true) {
      std::shared_ptr<ClientSession> client = getClientToRemove();
      client->waitToFinish();
      {
        std::lock_guard<std::mutex> guard(_namesToClientsMutex);
        _namesToClients.erase(client->getName());
      }
      {
        std::lock_guard<std::mutex> guard(_clientsMutex);
        size_t res = _clients.erase(client);
        assert(res == 1);
        if (_isTerminating && _clients.empty()) {
          break;
        }
      }
    }
  }
  catch (std::exception& ex) {
    std::cout << "Reaper thread exception: " << ex.what() << std::endl;
  }
}
```

Implementacja ChatServer::reaperThread cd

```
std::shared_ptr<ClientSession> ChatServer::getClientToRemove() {
  std::unique_lock<std::mutex> lock(_clientsToRemoveMutex);
  _reaperCondition.wait(lock, [this]() { return !_clientsToRemove.empty(); });
  std::shared_ptr<ClientSession> client = _clientsToRemove.front();
  _clientsToRemove.pop_front();
  return client;
}

ChatServer::ChatServer(int port) :
  _reaperThread(std::bind(&ChatServer::reaperThread, this)),
  _isTerminating(false) { }
```

Implementacja ChatServer::shutdown

```
void ChatServer::shutdown() {
  std::unique_lock<std::mutex> lock(_clientsMutex);
  for (const auto& client : _clients) {
    client->terminate();
  }
  _isTerminating = true;
  lock.unlock();
  pthread_kill(_acceptingThreadId, SIGUSR1);
}

ChatServer::~ChatServer() {
  _reaperThread.join();
}
```

Serwer synchroniczny wielowątkowy – podsumowanie

Zalety:

- „Prosty" przepływ sterowania.
- Niewielki narzut kodu na obsługę błędów dzięki wyjątkom.
- Program automatycznie korzysta z wielu rdzeni.

Wady:

- Skomplikowana procedura zamykania („zwijanie" drzewa wątków).
- Duża możliwość popełnienia subtelnych błędów związanych ze współbieżnością.
- Kiepska wydajność i skalowalność.

Subtelne błędy

Jest tak:

```
void ChatServer::run() {
  _acceptingThreadId = pthread_self();
  while (!_isTerminating) {
    std::shared_ptr<ClientSession> client = std::make_shared<ClientSession>(*this, _ioService);
    _acceptor.accept(client->socket());
    std::lock_guard<std::mutex> guard(_clientsMutex);
    if (!_isTerminating) {
      auto p = _clients.insert(client);
      assert(p.second);
      client->start();
    }
  }
}
```

Dlaczego nie tak?

```
void ChatServer::run() {
  _acceptingThreadId = pthread_self();
  while (!_isTerminating) {
    std::shared_ptr<ClientSession> client = std::make_shared<ClientSession>(*this, _ioService);
    _acceptor.accept(client->socket());
    if (!_isTerminating) {
      std::lock_guard<std::mutex> guard(_clientsMutex);
      auto p = _clients.insert(client);
      assert(p.second);
      client->start();
    }
  }
}
```

Subtelne błędy cd

Jest tak:

```
void ChatServer::run() {
  _acceptingThreadId = pthread_self();
  while (!_isTerminating) {
    std::shared_ptr<ClientSession> client = std::make_shared<ClientSession>(*this, _ioService);
    _acceptor.accept(client->socket());
    std::lock_guard<std::mutex> guard(_clientsMutex);
    if (!_isTerminating) {
      auto p = _clients.insert(client);
      assert(p.second);
      client->start();
    }
  }
}
```

Dlaczego nie tak?

```
void ChatServer::run() {
  _acceptingThreadId = pthread_self();
  while (!_isTerminating) {
    std::shared_ptr<ClientSession> client = std::make_shared<ClientSession>(*this, _ioService);
    _acceptor.accept(client->socket());
    std::unique_lock<std::mutex> lock(_clientsMutex);
    if (!_isTerminating) {
      auto p = _clients.insert(client);
      assert(p.second);
      lock.unlock();
      client->start();
    }
  }
}
```

Implementacja ChatServer::run

```
class ChatServer {
private:
  void startAccept();
  void onAccept(std::shared_ptr<ClientSession> client,
                const boost::system::error_code& error);
};

void ChatServer::run() {
  startAccept();
  _ioService.run();
}

void ChatServer::startAccept() {
  std::shared_ptr<ClientSession> client = std::make_shared<ClientSession>(*this, _ioService);
  _acceptor.async_accept(client->socket(),
                         std::bind(&ChatServer::onAccept, this, client, _1));
}

void ChatServer::onAccept(std::shared_ptr<ClientSession> client,
                          const boost::system::error_code& error) {
  if (! error) {
    client->start();
  }
  else {
    std::cout << "Accept error: " << error << std::endl;
  }

  startAccept();
}
```

boost::asio::ip::tcp::acceptor::async_accept

Szablonowa metoda `async_accept` jako ostatni argument bierze wywoływalny obiekt o sygnaturze `void (const boost::system::error_code&)`, który zostanie zawołany, gdy nadejdzie nowe połączenie lub wystąpi błąd.
Już na tym przykładzie widzimy, że jedna metoda rozbiła się nam na trzy i jesteśmy zmuszeni do jawnej obsługi błędów.

boost::asio::io_service::run

Metoda `run` dostarcza pętlę główną obsługi zdarzeń. Będzie się ona kręcić, dopóki będą aktywne jakieś asynchroniczne wywołania lub do momentu, w której powiemy jej, żeby przerwała.

Implementacja ClientSession::start

```
class ClientSession : public std::enable_shared_from_this<ClientSession> {
public:
  void start();
  void terminate();
private:
  void askForUserName();
  void readUserName();
  void handleUserName(const std::string& userName);
  void startReceivingAndSendingMessages();
  void messageOutputFinished();
  void handleInputLine(const std::string& line);
  bool parseLine(const std::string& line);

  void asyncReadLine(void (ClientSession::*handler)(const std::string& line));
  template <class Buffer>
  void asyncWrite(const Buffer& buffer, void (ClientSession::*handler)());

  ChatServer& _server;
  boost::asio::ip::tcp::socket _socket;
  std::string _outputBuffer;
  bool _sendingAllowed;
};

ClientSession::ClientSession(ChatServer& server, boost::asio::io_service &ioService) :
  _sendingAllowed(false) { }

void ClientSession::start() {
  askForUserName();
}

void ClientSession::askForUserName() {
  asyncWrite(boost::asio::buffer("What's your name?\n"), &ClientSession::readUserName);
}

void ClientSession::readUserName() {
  asyncReadLine(&ClientSession::handleUserName);
```

Implementacja ClientSession::start cd

```
void ClientSession::handleUserName(const std::string& userName) {
  if (_server.setClientName(shared_from_this(), userName)) {
    _outputBuffer = "Welcome to the chat, " + userName + "!\n";
    asyncWrite(boost::asio::buffer(_outputBuffer), &ClientSession::startReceivingAndSendingMessages);
  }
  else {
    _outputBuffer = "Name '" + userName + "' is already taken, invent another one.\n";
    asyncWrite(boost::asio::buffer(_outputBuffer), &ClientSession::askForUserName);
  }
}

void ClientSession::startReceivingAndSendingMessages() {
  _sendingAllowed = true;
  if (! _messages.empty()) {
    asyncWrite(boost::asio::buffer(*_messages.front()), &ClientSession::messageOutputFinished);
  }
  asyncReadLine(&ClientSession::handleInputLine);
}

void ClientSession::handleInputLine(const std::string& line) {
  if (parseLine(line)) {
    asyncReadLine(&ClientSession::handleInputLine);
  }
  else {
    terminate();
  }
}
```

Trudności programowania asynchronicznego

- Jedna metoda `ClientSession::readerThread` została rozbita na 4. Metoda może zainicjować tylko jedną operację asynchroniczną na gniazdku; po zrobieniu tego musi wyjść. W szczególności na jednym gniazdku może być aktywna co najwyżej jedna operacja zapisu i co najwyżej jedna operacja odczytu.
- Komunikaty o powodzeniu lub niepowodzeniu logowania w wersji synchronicznej trzymaliśmy na stosie. W wersji asynchronicznej musimy trzymać je w polu klasy, ponieważ zmienne lokalne nie są zachowywane.

Implementacja ClientSession::asyncReadLine

```
class ClientSession {
private:
  void handleReadError(const boost::system::error_code& error);
  std::string readLineFromClient();

  friend class ReadHandler;
}

void ClientSession::asyncReadLine(void (ClientSession::*handler)(const std::string&)) {
  boost::asio::async_read_until(_socket, _inputBuffer, '\n',
                                ReadHandler(handler, shared_from_this()));
}
```

Implementacja ClientSession::asyncReadLine cd

```
class ReadHandler {
public:
  ReadHandler(void (ClientSession::*handler)(const std::string&),
              const std::shared_ptr<ClientSession>& client) :
    _handler(handler),
    _client(client) { }

  ReadHandler(void (ClientSession::*handler)(const std::string&),
              std::shared_ptr<ClientSession>&& client) :
    _handler(handler),
    _client(client) { }

  void operator()(const boost::system::error_code& error, size_t) {
    if (error) {
      _client->handleReadError(error);
      return;
    }
    std::string line = _client->readLineFromClient();
    ((*_client).*_handler)(line);
  }

private:
  void (ClientSession::*_handler)(const std::string&);
  std::shared_ptr<ClientSession> _client;
};

void ClientSession::handleReadError(const boost::system::error_code& error) {
  std::cout << "Client reading error: " << error.message() << std::endl;
  terminate();
}

std::string ClientSession::readLineFromClient() {
  std::istream stream(&_inputBuffer);
  std::string line;
  std::getline(stream, line);
  return line;
```

boost::asio::async_read_until

`async_read_until` to asynchroniczny odpowiednik funkcji `read_until`. Oprócz tych samych argumentów co `read_until`, przyjmuje do dodatkowo wywoływalny obiekt o sygnaturze `void (const boost::system::error_code&, size_t)`. Po zakończeniu operacji obiekt ten jest wołany z kodem błędu i liczbą przeczytanych bajtów, odpowiednio.

Implementacja ClientSession::asyncWrite

```
class ClientSession {
private:
  void handleWriteError(const boost::system::error_code& error);

  friend class WriteHandler;
};

template<class Buffer>
void ClientSession::asyncWrite(const Buffer& buffer, void (ClientSession::*handler)()) {
  boost::asio::async_write(_socket, buffer, WriteHandler(handler, shared_from_this()));
}
```

Implementacja ClientSession::asyncWrite cd

```
class WriteHandler {
public:
  WriteHandler(void (ClientSession::*handler)(), const std::shared_ptr<ClientSession>& client) :
    _handler(handler),
    _client(client) { }

  WriteHandler(void (ClientSession::*handler)(), std::shared_ptr<ClientSession>&& client) :
    _handler(handler),
    _client(client) { }

  void operator()(const boost::system::error_code& error, size_t) {
    if (error) {
      _client->handleWriteError(error);
      return;
    }
    ((*_client).*_handler)();
  }

private:
  void (ClientSession::*_handler)();
  std::shared_ptr<ClientSession> _client;
};

void ClientSession::handleWriteError(const boost::system::error_code& error) {
  std::cout << "Client writing error: " << error.message() << std::endl;
  terminate();
}
```

boost::asio::async_write

`async_write` to asynchroniczny odpowiednik funkcji `write`. Oprócz tych samych argumentów co `write`, przyjmuje do dodatkowo wywoływalny obiekt o sygnaturze `void (const boost::system::error_code&, size_t)`. Po zakończeniu operacji obiekt ten jest wołany z kodem błędu i liczbą zapisanych bajtów, odpowiednio.

`ClientSession::asyncReadLine` i `ClientSession::asyncWrite` – podsumowanie

Dzięki sprytnej implementacji uniknęliśmy manualnej obsługi błędów po każdym asynchronicznym wywołaniu i udało się oddzielić kod obslugujący błędy od kodu obsługującego zwykłą logikę.

Implementacja ClientSession::sendMessage i ChatServer::broadcast

```
void ClientSession::sendMessage(const std::shared_ptr<std::string> &msg) {
  bool startSending = _messages.empty() && _sendingAllowed;
  _messages.push_back(msg);
  if (startSending) {
    asyncWrite(boost::asio::buffer(*_messages.front()), &ClientSession::messageOutputFinished);
  }
}

void ClientSession::messageOutputFinished() {
  assert(_sendingAllowed);
  assert(!_messages.empty());
  _messages.pop_front();
  if (!_messages.empty()) {
    asyncWrite(boost::asio::buffer(*_messages.front()), &ClientSession::messageOutputFinished);
  }
}

void ChatServer::broadcast(ClientSession& client, const std::shared_ptr<std::string>& msg) {
  for (const auto& kvPair : _namesToClients) {
    ClientSession* receiver = kvPair.second.get();
    if (receiver!= &client) {
      receiver->sendMessage(msg);
    }
  }
}
```

Implementacja ClientSession::terminate i ChatServer::removeClient

```
void ClientSession::terminate() {
  _socket.cancel();
  _server.removeClient(*this);
}

void ChatServer::removeClient(ClientSession& client) {
  _namesToClients.erase(client.getName());
}
```

boost::asio::basic_io_object::cancel

Metoda `cancel` z klasy `basic_io_object`, po której dziedziczy `socket`, usuwa wszystkie zarejestrowane handlery operacji asynchronicznych związane z tym obiektem.

Implementacja ChatServer::shutdown()

```
void ChatServer::shutdown() {
  _ioService.stop();
}
```

boost::asio::io_service::stop

Metoda stop powoduje zakończenie pętli głównej w jej następnym obrocie – po zakończeniu handlera, który zawołał `stop`, `run` zakończy się.

Serwer asynchroniczny z callbackami – podsumowanie

Zalety:

- Brak problemów ze współbieżnością.
- Prosta sekwencja zakończenia programu.
- Duża wydajność.

Wady:

- „Odwrócone" sterowanie utrudnia implementację skomplikowanych algorytmów dialogu z klientem. Szczególnie niewdzięczne do implementacji są zagnieżdżone pętle.
- Stan programu musi być explicite przechowywany w polach klasy.
- Standardowo używany jest tylko jeden rdzeń. Jest możliwość uruchomienia wielu pętli `io_service::run` w wielu wątkach, ale wtedy problemy ze współbieżnością wracają.

Wątki kooperatywne: definicja

Wątki kooperatywne (green threads) to wątki zrealizowane przestrzeni użytkownika, niezależnie od systemu operacyjnego. Podobnie jak zwykły wątek, wątek kooperatywny posiada swoją kopię stanu procesora i stos. Wątki kooperatywne przełączane są manualnie, na własne żądanie. Stan procesora jest wtedy zachowywany, a sterowanie jest oddawane do schedulera działającego w przestrzeni użytkownika, który wybiera wątek, który zostanie wznowiony.

Implementacja ChatServer::run

```
class ChatServer {
private:
   void ChatServer::acceptThread(boost::asio::yield_context yield);
}

void ChatServer::run() {
  boost::asio::spawn(_ioService, std::bind(&ChatServer::acceptThread, this, _1));
  _ioService.run();
}

void ChatServer::acceptThread(boost::asio::yield_context yield) {
  boost::system::error_code ec;
  while (true) {
    std::shared_ptr<ClientSession> client = std::make_shared<ClientSession>(*this, _ioService);
    _acceptor.async_accept(client->socket(), yield[ec]);
    if (ec) {
        std::cout << "Accept error: " << ec.message() << std::endl;
        shutdown();
    }
    client->start();
  }
}
```

boost::asio::spawn

Funkcja `spawn` uruchamia nowy wątek kooperatywny. Pierwszym argumentem jest `io_service`, a drugim obiekt wywoływalny o sygnaturze `void (boost::asio::yield_context)`.

boost::asio::yield_context

`yield_context`, to klasa wykonująca całą magię: potrafi wstrzymywać i wznawiać nasz wątek.
`yield_context::operator[](boost::system::error_code&)` zachowuje stan wątka i zwraca obiekt wywoływalny pasujący do wszystkich operacji asynchronicznych. Ten wywoływalny obiekt po zawołaniu obudzi nasz wątek, a kod błędu przekazany jako argument do `operator[]` zostanie ustawiony na kod błędu zwrócony przez operację asynchroniczną.
Naszym schedulerem jest więc `io_service`, który przełącza wątki wołając handlery dla zdazeń.

Implementacja ClientSesion::start

```
class ClientSession {
private:
  void readerThread(boost::asio::yield_context yield);
  void writerThread(boost::asio::yield_context yield);
};

void ClientSession::start() {
  boost::asio::spawn(_ioService, std::bind(&ClientSession::readerThread, shared_from_this(), _1));
  boost::asio::spawn(_ioService, std::bind(&ClientSession::writerThread, shared_from_this(), _1));
}
```

Implementacja ClientSession::readerThread

```
class ClientSession {
private:
  std::string readLineFromClient(boost::asio::yield_context yield);
  bool parseLine(const std::string& line);
  template <class Buffer>
  void asyncWrite(const Buffer& buffer, boost::asio::yield_context yield);
};

void ClientSession::readerThread(boost::asio::yield_context yield) {
  try {
    bool loginSuccessfull = false;
    while (! loginSuccessfull) {
      asyncWrite(boost::asio::buffer("What's your name?\n"), yield);
      std::string name = readLineFromClient(yield);
      if ((loginSuccessfull = _server.setClientName(shared_from_this(), name))) {
        std::string response = "Welcome to the chat, " + name + "!\n";
        asyncWrite(boost::asio::buffer(response), yield);
      }
      else {
        std::string response = "Name '" + name + "' is already taken, invent another one.\n";
        asyncWrite(boost::asio::buffer(response), yield);
      }
    }

    while (parseLine(readLineFromClient(yield)));
  }
  catch (std::exception& ex) {
    std::cout << "Client reader thread exeption: " << ex.what() << std::endl;
  }

  onReaderShutdown();
}
```

Implementacja ClientSession::readerThread cd

```
std::string ClientSession::readLineFromClient(boost::asio::yield_context yield) {
  boost::system::error_code ec;
  boost::asio::async_read_until(_socket, _inputBuffer, '\n', yield[ec]);
  if (ec) {
    throw boost::system::system_error(ec);
  }
  std::istream stream(&_inputBuffer);
  std::string line;
  std::getline(stream, line);
  return line;
}

template<class Buffer>
void ClientSession::asyncWrite(const Buffer& buffer, boost::asio::yield_context yield) {
  boost::system::error_code ec;
  boost::asio::async_write(_socket, buffer, yield[ec]);
  if (ec) {
    throw boost::system::system_error(ec);
  }
}
```

`ClientSession::readerThread` – wnioski

- Kod bardzo podobny jak w wersji wielowątkowej. Jedyne, co musimy zmienić, to gdy chcemy w jakieś metodzie wykonać „blokującą” operację, musimy przepropagować do niej `yield_context`.
- Wyjątki znów działają.
- Znów możemy mieć bufory na stosie.

Implementacja ClientSession::writerThread i ClientSession::sendMessage

```
class ClientSession {
private:
  ConditionVariable _writerCondition;
};

void ClientSession::writerThread(boost::asio::yield_context yield) {
  try {
    boost::system::error_code ec;
    bool run = true;
    while (run) {
      auto msg = getMessage(yield);
      if (msg) {
        asyncWrite(boost::asio::buffer(*msg), yield);
      }
      else {
        run = false;
      }
    }
  }
  catch (std::exception& ex) {
    std::cout << "Client writer thread exception: " << ex.what() << std::endl;
  }

  onWriterShutdown();
}
```

Implementacja ClientSession::writerThread i ClientSession::sendMessage cd

```
std::shared_ptr<std::string> ClientSession::getMessage(boost::asio::yield_context yield) {
  _writerCondition.wait(yield,
                        [this]() { return _state != ALL_RUNNING || !_outputData.empty(); });
  if (_state != ALL_RUNNING) {
    return std::shared_ptr<std::string>();
  }
  else {
    auto msg = _outputData.front();
    _outputData.pop_front();
    return msg;
  }
}

void ClientSession::sendMessage(const std::shared_ptr<std::string>& msg) {
  _outputData.push_back(msg);
  _writerCondition.notify_one();
}
```

Co to jest `ConditionVariable`?

Jak dotąd kod używający wątków kooperatywnych był bardzo podobny do kodu zwykłego programu wielowątkowego.
Nie potrzebujemy mutexów, ponieważ nie mamy „prawdziwej" współbieżności (przełączamy wątki na żądanie, kiedy stan pól w klasie jest spójny).
Potrzebujemy jednak funkcjonalności `condition_variable`: możliwości usypiania i wznawiania wątku w zależności od zadanego warunku.
Musimy napisać to sami!

Implementacja ConditionVariable

```
class ConditionVariable {
public:
  ConditionVariable(boost::asio::io_service &ioService);

  template <class Predicate>
  void wait(boost::asio::yield_context yield, Predicate pred);

  void notify_one();
  void notify_all();
private:
  boost::asio::deadline_timer _timer;
};

ConditionVariable::ConditionVariable(boost::asio::io_service& ioService) :
  _timer(ioService) { }

template <class Predicate>
void ConditionVariable::wait(boost::asio::yield_context yield, Predicate pred) {
  boost::system::error_code ec;
  while (! pred()) {
    _timer.async_wait(yield[ec]);
    if (ec == boost::asio::error::operation_aborted) {
      return;
    }
    else {
      throw boost::system::system_error(ec);
    }
  }
}
```

Implementacja ConditionVariable cd

```
void ConditionVariable::notify_one() {
  _timer.cancel_one();
}

void ConditionVariable::notify_all() {
  _timer.cancel();
}
```

boost::asio::deadline_timer

`boost::asio::deadline_timer` to klasa implementująca budzik. Przy pomocy metody `async_await` można zarejestrować callback, który zostanie wywołany po określonym czasie lub w zadanym momencie czasowym (mniej-więcej).
`ConditionVariable` nie nakręca budzika (nie precyzuje, kiedy ma być zawołany callback), więc będzie spać w nieskończoność (callback nigdy nie zostanie wywołany), chyba że zostanie zawołana metoda `cancel` lub `cancel_one`, która powoduje natychmiastowe wywołanie callbacka z błędem `boost::asio::error::operation_aborted`.
Ponieważ naszym callbackiem jest `yield_context`, wywołanie go spowoduje przełączenie kontekstu i obudzenie wątku śpiącego wątku, czyli to o co nam chodziło.

Implementacja sprzątania

```
void ClientSession::onReaderShutdown() {
  int oldState = _state;
  _state = oldState | READER_TERMINATED;
  if (oldState & WRITER_TERMINATED) {
    _server.removeClient(*this);
  }
  else {
    _writerCondition.notify_one();
  }
}

void ClientSession::terminate() {
  _state |= TERMINATE_REQUESTED;
  _socket.cancel();
  _writerCondition.notify_all();
}

void ClientSession::terminate() {
  _state |= TERMINATE_REQUESTED;
  _socket.cancel();
  _writerCondition.notify_all();
}

void ChatServer::shutdown() {
  _ioService.stop();
}
```

Serwer asynchroniczny z wykorzystaniem wątków kooperatywnych – podsumowanie

Zalety:

- „Prosty" przepływ sterowania, kod podobny do programu wielowątkowego.
- Stan może być przechowywany na stosie.
- Brak problemów ze współbieżnością.
- Prosy sposób wyjścia z programu.
- Niezła wydajność.

Wady:

- Standardowo używa tylko jednego rdzenia procesora. Można temu zaradzić podobnie jak w przypadku wersji asynchronicznej z użyciem callbacków, poprzez uruchomienie `io_service::run` w wielu „prawdziwych" wątkach jednocześnie, ale problemy ze współbeżnością powrócą.