Low Orbit Task Cannon

Techniki Internetowe, Projekt
Dokumentacja końcowa

Tomasz Jakubczyk, Eryk Ratyński, Andrzej Roguski, Kacper Stachyra

7 stycznia 2016

Low Orbit Task Cannon na serwerze GitHub

1 Treść zadania

"W sieci jest zbiór zarządzanych węzłów, serwer zarządzający i stacja konsoli administratora. W węzłach pracują agenty zarządzające. Agent zarządzający może: załadować kod nowego procesu, usunąć kod procesu, uruchomić/zatrzymać/wznowić/zabić dany proces zgodnie z harmonogramem, wznowić proces nie raportujący swej żywotności, podać dane statystyczne serwerowi. System umożliwia administratorowi zarządzanie rozproszonymi procesami. System komunikacji powinien móc pracować w przestrzeni adresów IPv4 i IPv6. Ponadto należy zaprojektować moduł do Wireshark umożliwiający wyświetlanie i analizę zdefiniowanych komunikatów."

2 Założenia projektowe

2.1 Środowisko

- Low Orbit Task Cannon (LOTC) uruchamiany jest na systemie operacyjnym GNU/Linux
- Hosty systemu LOTC mają działającą usługę synchronizacji czasu

2.2 Zadania

- Zadania po wprowadzeniu do LOTC nie wymagają modyfikacji
- Wykonanie zadania wymaga uruchomienia wyłącznie jednego pliku wykonywalnego (może on jednak uruchamiać inne podprogramy)
- Zadania dają się uruchomić w systemie GNU/Linux bez GUI (w szczególności - bez X Window System)
- Zadania wykonywane są w trybie wsadowym, tj. nie wymagają interakcji z użytkownikiem

3 Struktura systemu

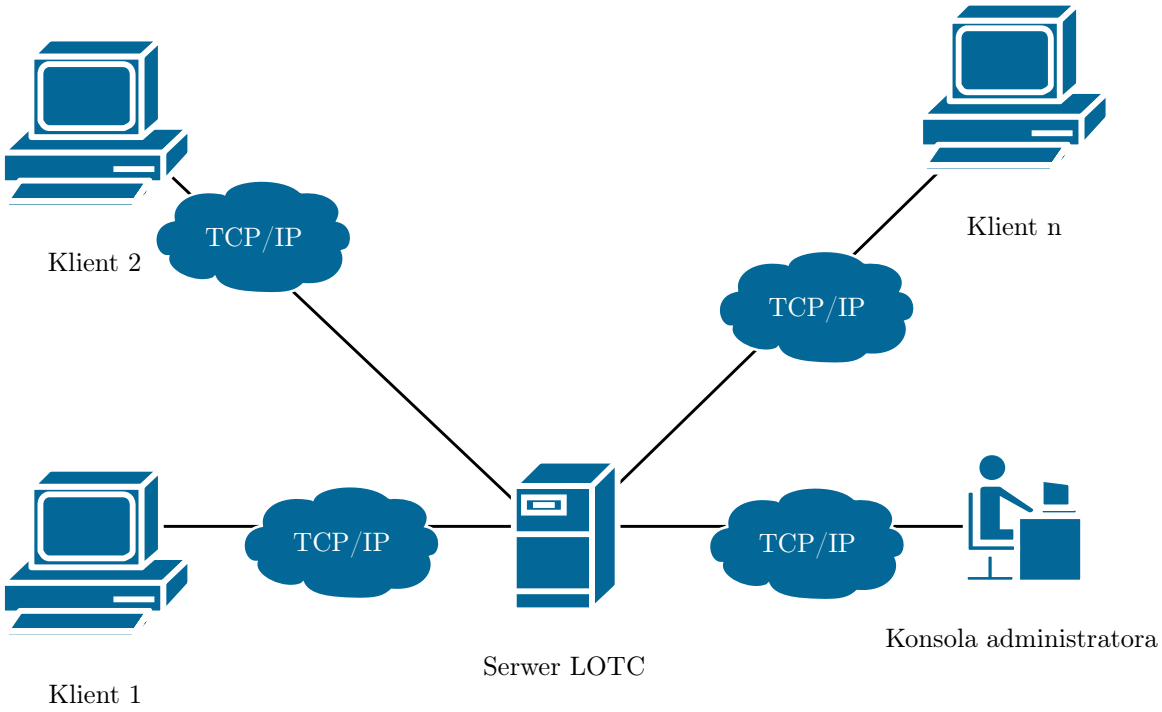
3.1 Moduły

Low Orbit Task Cannon zawiera następujące moduły:

1. Protokół LOTC
2. Serwer
3. Klient (agent)
4. Konsola administratora

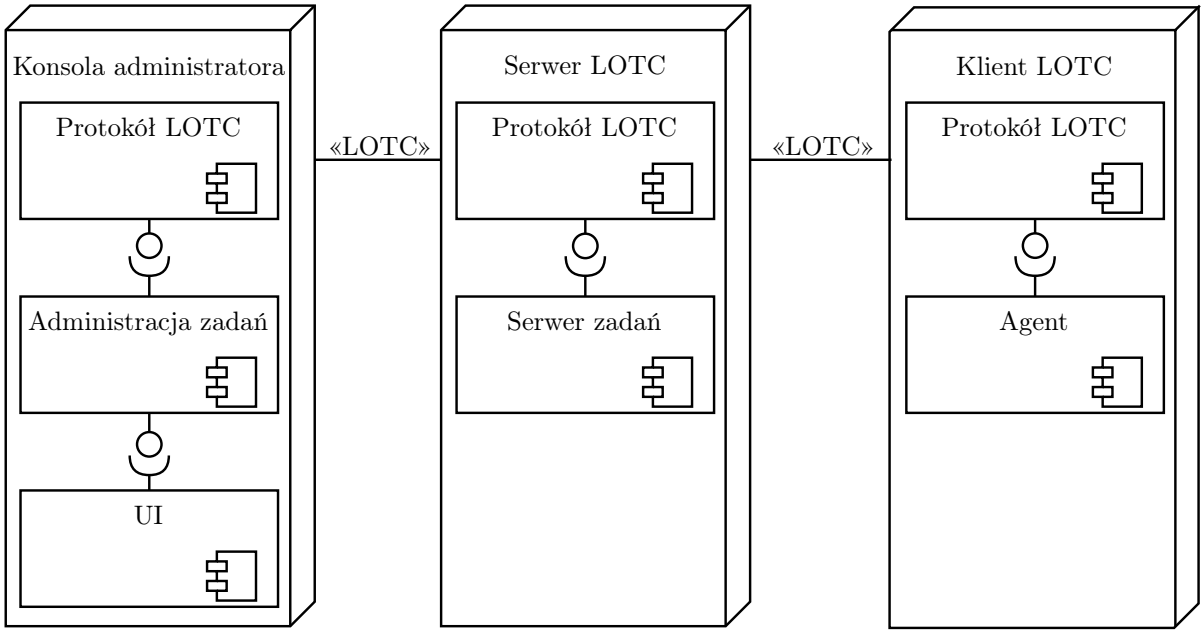
3.2 Topologia

- Każdy klient LOTC musi być połączony siecią TCP/IP z serwerem LOTC
- Konsola administratora musi być połączona siecią TCP/IP z serwerem LOTC
- Serwer LOTC musi być połączony siecią TCP/IP



3.3 Diagram rozmieszczenia

Komunikacja między konsolą i serwerem oraz serwerem i klientem odbywa się poprzez protokół LOTC. Wykorzystywany jest do tego moduł implementujący protokół LOTC i wystawiający interfejs komunikacyjny.
Z racji potrzeby synchronizacji, serwer oraz klient wykorzystują moduł implementujący niezbędne minimum klienta NTP potrzebne do zapytania serwera NTP o aktualny czas.
Program do administracji zadaniami wystawia interfejs niezbędny do zbudowania UI, zarówno w wersji tekstowej jak i graficznej.



3.4 Diagramy klas

Diagramy klas głównych pod opisami odpowiednich modułów.
Pozostałe na GitHubie projektu.

4 Środowisko sprzętowo-programowe

- System operacyjny: GNU/Linux
- Język programowania: C++14 (system LOTC)
- Biblioteki: Boost $\geqslant$ 1.59 (z wyłączeniem nakładek na API gniazd BSD)
- Kompilator: GCC $\geqslant$ 5.2.0
- Debugger: GDB $\geqslant$ 7.10

5 Protokół LOTC

5.1 Założenia

Protokół LOTC zapewnia komunikację w warstwie aplikacji między hostami systemu. Podstawowymi zadaniami protokołu są:

1. Rejestracja i usuwanie agentów
2. Zarządzanie wykonaniem pojedynczych zadań
3. Zarządzanie zależnościami między zadaniami
4. Przesyłanie plików
5. Przekazywanie wyników zadań
6. Przekazywanie żądania synchronizacji
7. Monitorowanie responsywności agentów
8. Zgłaszanie błędów

Ponadto protokół powinien weryfikować stan komunikacji poprzez system potwierdzeń i raportów sukcesu/porażki.

5.2 Struktura protokołu

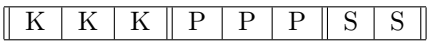
Protokół dzieli się na osiem kategorii, realizujących osiem wyżej wymienionych zadań. Ponadto niektóre z kategorii dzielą się dalej na podkategorie. Ortagonalnie do kategorii funkcjonuje podział komunikatów na:

- Żądania - komunikaty inicjujące wykonanie czynności
- Potwierdzenia - komunikaty potwierdzające otrzymanie żądania
- Raporty sukcesu - komunikaty informujące o pomyślnym wykonaniu czynności
- Raporty porażki - komunikaty informujące o błędzie w trakcie wykonaniu czynności

5.2.1 Kod komunikatu

Każdy komunikat LOTC rozpoczyna się ośmiobitowym kodem jednoznacznie informującym o kategorii, podkategorii i stanie czynności. Kod ma następującą strukturę:

- Kategoria [3 bity]
- Podkategoria [3 bity]
- Stan czynności [2 bity]



5.2.2 Kategorie

Kod binarnie	Kod dziesiątkowo	Kategoria	Opis
000	0	HOST	Rejestracja i usuwanie agentów
001	1	TASK	Zarządzanie wykonaniem pojedynczych zadań
010	2	DEP	Zarządzanie zależnościami między zadaniami
011	3	FILE	Przesyłanie plików
100	4	RET	Przekazywanie wyników zadań
101	5	SYN	Synchronizacja
110	6	PING	Monitorowanie responsywności agentów
111	7	ERR	Zgłaszanie błędów

5.2.3 Podkategorie

Kategoria HOST:

Kod binarnie	Kod dziesiątkowo	Podkategoria	Opis
000	0	H_ADD	Dodanie agenta
001	1	H_RM	Usunięcie agenta
010	2	H_STATE	Zapytanie o stan
011	3	x	[zabroniony]
100	4	x	[zabroniony]
101	5	x	[zabroniony]
110	6	x	[zabroniony]
111	7	x	[zabroniony]

Kategoria TASK:

Kod binarnie	Kod dziesiątkowo	Podkategoria	Opis
000	0	T_ADD	Dodanie zadania
001	1	T_RM	Usunięcie zadania
010	2	T_RUN	Wykonanie zadania
011	3	T_KILL	Zakończenie zadania
100	4	T_STOP	Wstrzymanie zadania
101	5	T_CONT	Kontynuowanie zadania
110	6	T_OK	Zadanie gotowe do przetwarzania
111	7	T_NOK	Zadanie niegotowe do przetwarzania

Kategoria ERR:

Kod binarnie	Kod dziesiątkowo	Podkategoria	Opis
000	0	E_HEAD	Błędne pola nagłówka
001	1	E_LGTH	Błędna długość nagłówka
010	2	E_CSUM	Błędna suma kontrolna
011	3	E_TASK	Błąd obsługi zadania
100	4	E_FILE	Błąd obsługi pliku
101	5	x	[zabroniony]
110	6	x	[zabroniony]
111	7	E_OTH	Inny błąd

Pozostałe kategorie:

Kod binarnie	Kod dziesiątkowo	Podkategoria	Opis
000	0	DEF	Wartość domyślna
001	1	x	[zabroniony]
010	2	x	[zabroniony]
011	3	x	[zabroniony]
100	4	x	[zabroniony]
101	5	x	[zabroniony]
110	6	x	[zabroniony]
111	7	x	[zabroniony]

5.2.4 Stany czynności

Kod binarnie	Kod dziesiątkowo	Stan	Opis
00	0	REQ	Inicjacja czynności
01	1	ACK	Potwierdzenie otrzymania komunikatu REQ
10	2	OK	Zakończenie czynności - sukces
11	3	ERR	Zakończenie czynności - porażka

5.2.5 Szczegóły protokołu

Każda z kategorii komunikatów posiada własny nagłówek.

Kategoria HOST:

Pozwala na dodanie albo usunięcie do 2^{16} agentów jednocześnie.

Pole	Długość [bit]	Opis
Kod komunikatu	8	
Wersja IP	1	Wersja adresów IP agentów 0 - v4, 1 - v6
Wyrównanie	7	Zera
Liczba agentów	16	Liczba agentów
Adres 1	32/128	Adres IP pierwszego agenta (długość zależna od wersji IP)
...		
Adres n	32/128	Adres IP n-tego agenta, gdzie n = liczba agentów

Kategoria TASK:

Pozwala na dodawanie, usuwanie i sterowanie wykonaniem zadań w systemie.

Pole	Długość [bit]	Opis
Kod komunikatu	8	
Flaga priorytetu	1	Określa, czy zatrzymywać zadania o niższym priorytecie
Wyrównanie	7	Zera
Priorytet	16	
ID zadania	32	
Znacznik czasu	32	określa, kiedy polecenie wchodzi w życie

Kategoria DEP:

Pozwala określić, jakie zadania muszą poprzedzać wykonanie zadania Z (maksymalnie 2^{16} zadań).

Pole	Długość [bit]	Opis
Kod komunikatu	8	
Wyrównanie	8	Zera
Liczba zadań poprzedzających	16	
ID zadania Z	32	
ID zadania poprzedzającego 1	32	
...		
ID zadania poprzedzającego n	32	Gdzie n to liczba zadań poprzedzających

Kategoria FILE:

Pozwala przesyłać pojedyncze pliki.

Pole	Długość [bit]	Opis
Kod komunikatu	8	
Flaga typu pliku	1	0 - główny plik zadania, 1 - plik pomocniczy
Długość nazwy pliku	7	Maksymalnie 128 znaków
Suma kontrolna	16	Wyliczana z nazwy i zawartości pliku
ID zadania	32	Określa, do jakiego zadania należy plik
Rozmiar pliku	32	Maksymalnie 2^{32} bajtów
Nazwa pliku	8 - $8 \cdot 128$	
Zawartość pliku	0 - $8 \cdot 2^{32}$	

Kategoria RET:

Pozwala zwracać wyniki działania zadań w postaci pliku.

Pole	Długość [bit]	Opis
Kod komunikatu	8	
Kod zakończenia zadania	8	POSIX exit status
Długość nazwy pliku	8	któryś z typów prostych lub ciąg bitów
Wyrównanie	8	Zera
ID Zadania	32	
Rozmiar pliku	32	Maksymalnie 2^{32} bajtów
Nazwa pliku	8 - $8 \cdot 128$	
Zawartość pliku	0 - $8 \cdot 2^{32}$	

Kategorie SYN, PING:

Pozwalają kolejno: zainicjować synchronizację, sprawdzić responsywność.

Z racji swojej prostoty, te nagłówki wymagają wyłącznie kodu komunikatu - stan czynności zawarty w kodzie wystarcza do przeprowadzenia niezbędnej komunikacji.

Pole	Długość [bit]
Kod komunikatu	8

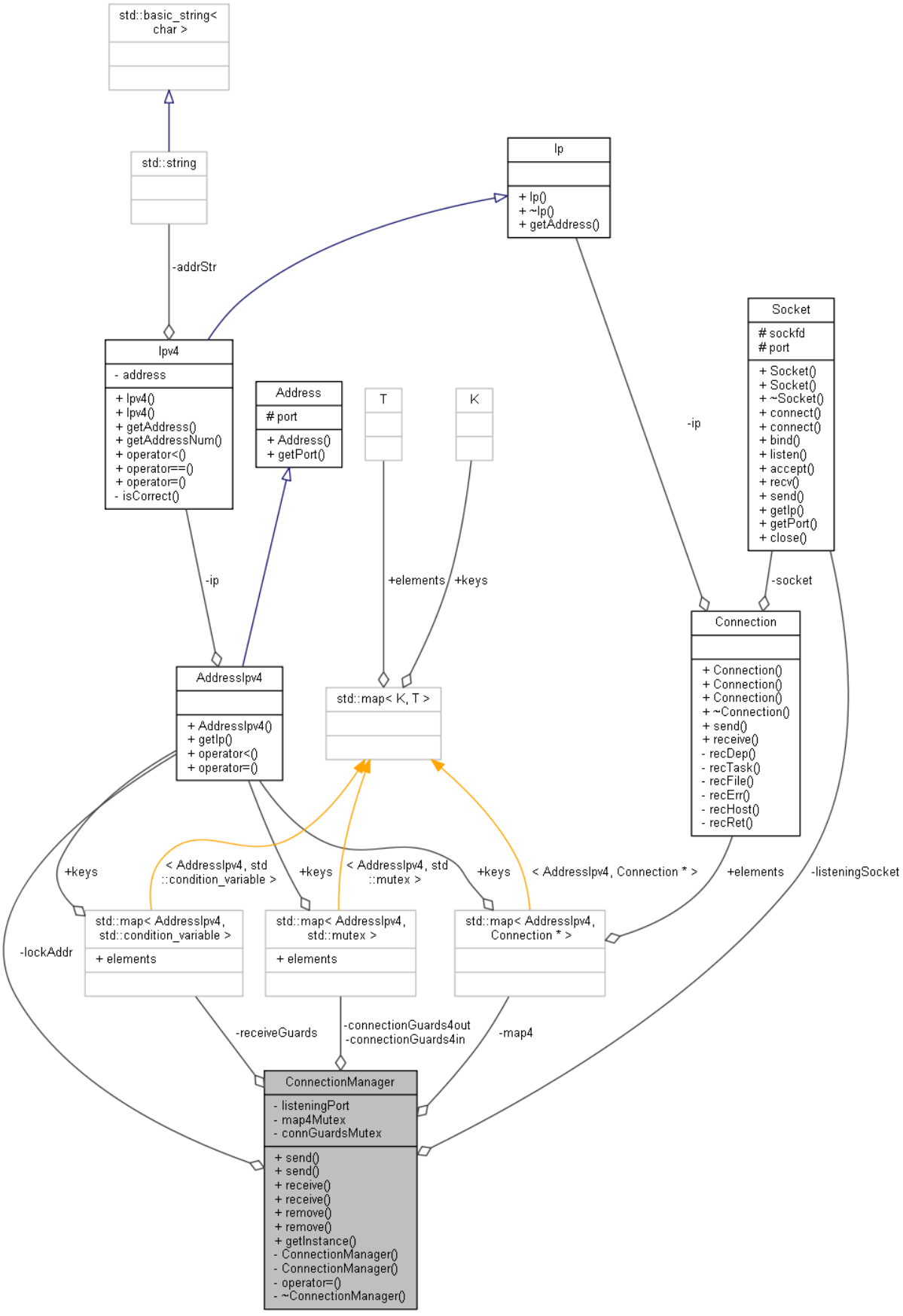
Kategoria ERR:

Pozwala na zgłoszenie błędu.

Komunikaty błędów dotyczą zawsze ostatnio otrzymanego polecenia. Komunikaty błędów dotyczących samego nagłówka zawierają wyłącznie odpowiedni kod. Błędy obsługi zadań i plików mają dodatkowe pole z kodem dalej specyfikującym rodzaj błędu.

Pole	Długość [bit]	Opis
Kod komunikatu	8	
Szczegółowy kod błędu	8	Tylko błędy obsługi zadań i plików

5.3 Diagram klas



6 Serwer

6.1 Opis

Serwer zarządza agentami, które kontrolują procesy na swojej platformie. Informacje o tym gdzie i co ma być zrobione dostaje z konsoli administratora. Kanałem komunikacyjnym z serwerem jest protokół LOTC działający na Ipv4 z wykorzystaniem mechanizmu gniazd sieciowych.

6.2 Reagowanie na zdarzenia

Zdarzeniami przychodzącymi do serwera mogą być polecenia z konsoli administratora, odpowiedzi od agentów lub zdarzenia wewnętrzne serwera np. time-out. Zdarzenia są obiektami z różnych klas typów zdarzeń dziedziczących po bazowej klasie zdarzeń, pojawiającymi się w kolejce zdarzeń. Zostają one obsłużone na podstawie mapy strategii i zdjęte z kolejki. Prawdopodobnie będzie około trzech różnych typów zdarzeń: komunikat z agenta, polecenie od administratora, zdarzenie wewnętrzne.

6.3 Moduły serwera

Serwer da się podzielić na kilka wyraźnych modułów:

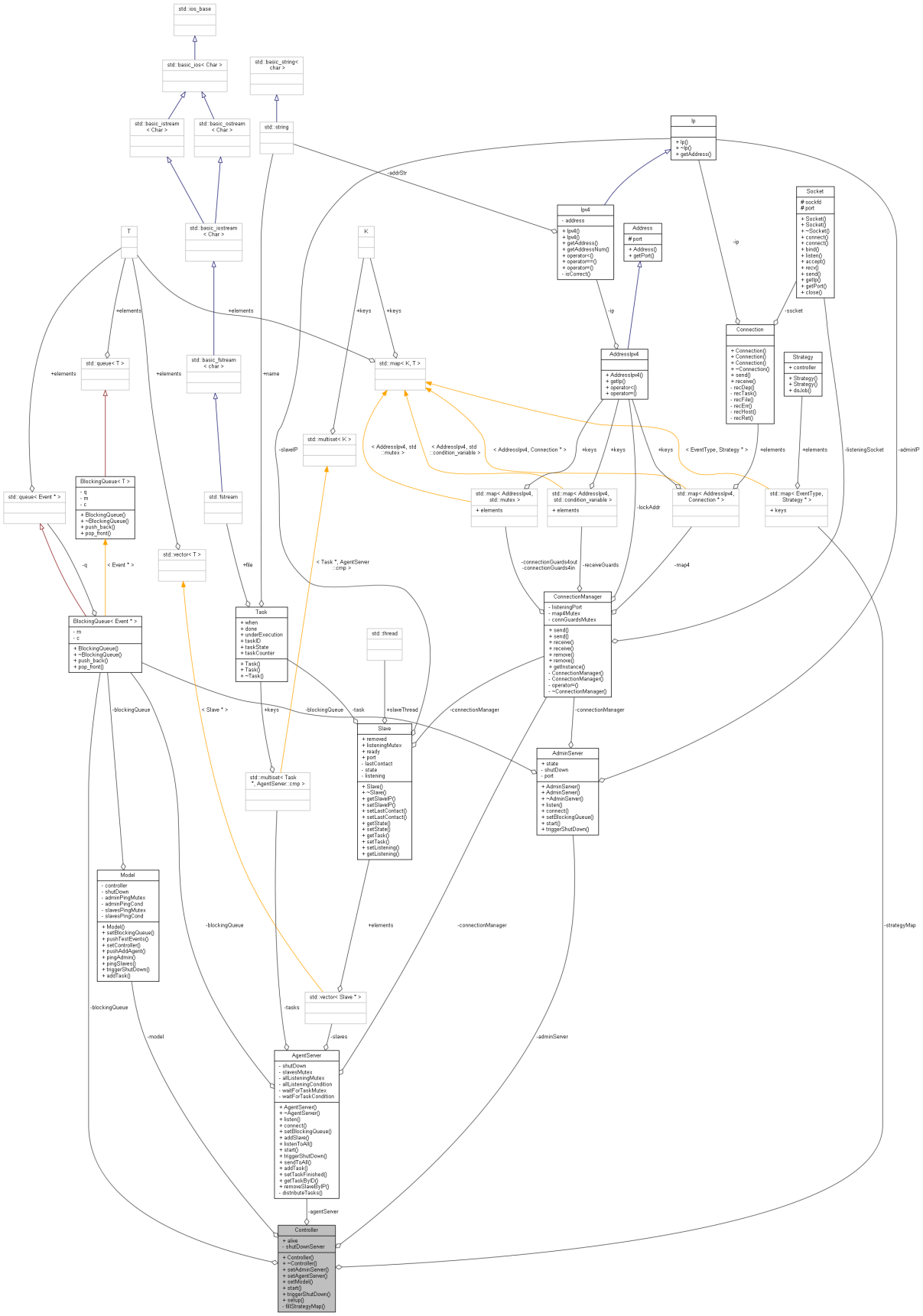
- Kontroler – przetwarza zdarzenia przychodzące z modułów i wykonuje lub zleca wykonanie odpowiednich metod w modułach. Odpowiada też za uruchomienie usług modułów szczególnie przy starcie serwera.
- Serwer klienta – odpowiada za komunikację z agentami. Serwer nawiązuje połączenie ze słuchającym agentem, wysyła polecenie, czeka na odpowiedź, otrzymaną odpowiedź wrzuca na kolejkę zdarzeń i zamyka połączenie. Może też nasłuchiwać na połączenia od agenta z wiadomością o zmianie statusu lub okresowym raportem.
- Serwer administratora – odpowiada za komunikację z konsolą administratora. Serwer czeka na polecenie od administratora, wrzuca na kolejkę zdarzeń i zwraca odpowiedź. Konsola administratora może sama nawiązać połączenie i wysłać polecenia, ale jeśli serwer ma po jakimś czasie odesłać raport, to konsola powinna też nasłuchiwać.
- Model – zbiór metod wywoływanych przez kontroler w celu realizacji wybranej strategi z mapy. Może też zawierać maszynę stanów i informacje o agentach.

Poszczególne moduły mogą pracować w osobnych wątkach lub procesach. Kontroler powinien być w wątku głównym.
Model, o ile będzie zgłaszał jakieś zdarzenia (zapewne związane z upływem czasu), powinien mieć osobny wątek. Jeśli nie, to będzie wywoływany (jego metody) w tym samym wątku co kontroler.
Serwer klienta, a właściwie każda jego instancja, powinien być w osobnym wątku. Wątki te powinny być tworzone wtedy gdy wyniknie to ze strategii działania.
Serwer administratora również powinien być w osobnym wątku.

6.4 Komunikacja z konsolą i agentami

Komunikacja z konsolą i agentami odbywać będzie się poprzez protokół LOTC, przy wykorzystaniu modułu dostarczającego interfejs będący nakładką na protokół.

6.5 Diagram klas



7 Agent (klient LOTC)

7.1 Opis

Agent to autonomiczny program zarządzający przydzielonymi przez serwer zadaniami. Agent zarządzający może: załadować/usunąć zadanie, uruchomić/zatrzymać/wznowić/zabić dany proces zgodnie z harmonogramem, podać dane statystyczne serwerowi.
Agent komunikuje się z serwerem (protokołem działającym na IPv4) wykorzystując mechanizm gniazd sieciowych.

7.2 Zasady działania agenta

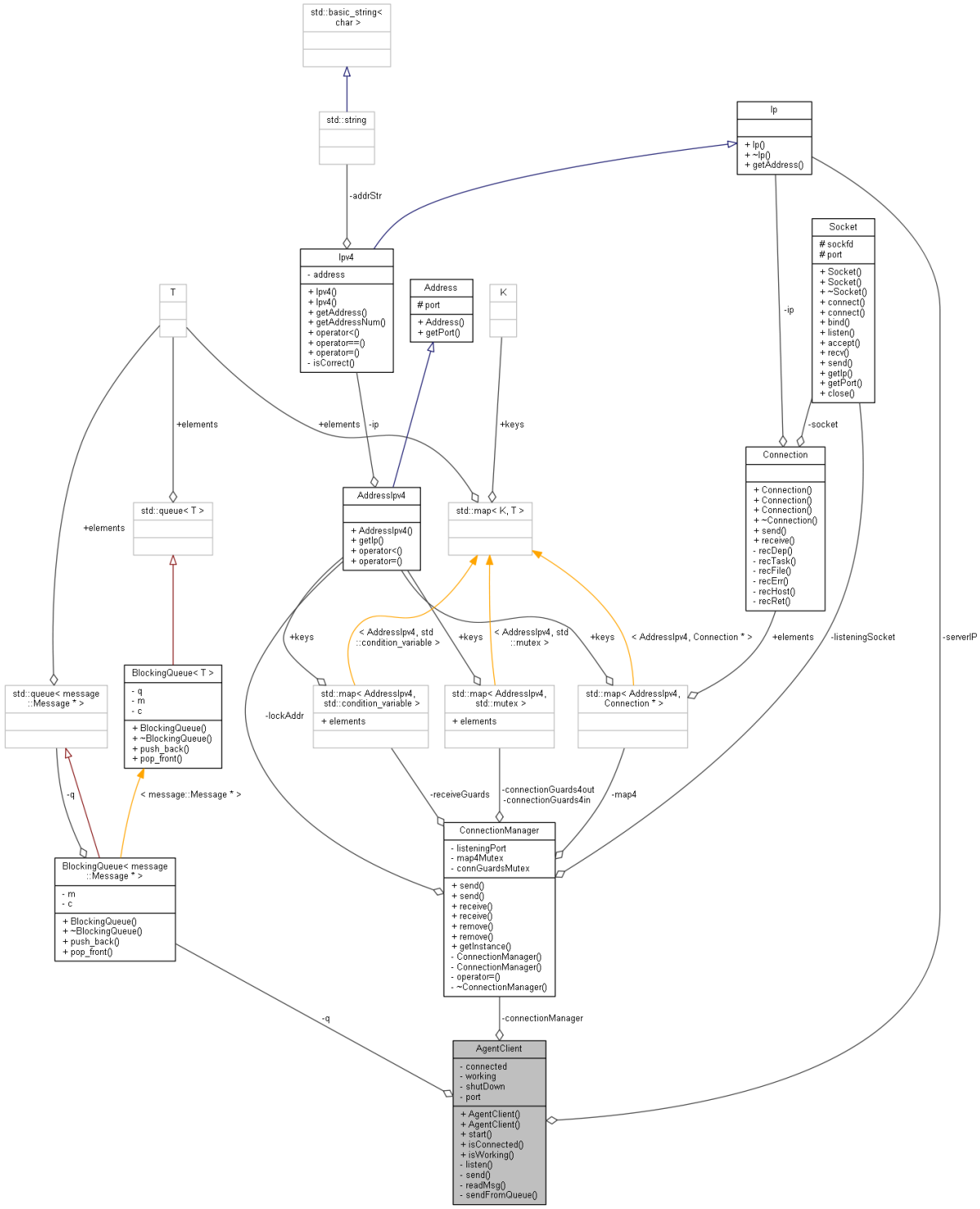
Agent po starcie łączy się z serwerem, następnie stale oczekuje na komunikaty wysyłane przez serwer. Po przyjściu komunikatu potwierdza jego otrzymanie a następnie analizuje jego treść według ustalonego protokołu. Potem wywołuje odpowiednią metodę w nowym wątku. Główny wątek dalej czeka na komunikaty od serwera, oraz wysyła raporty z wykonania innych wątków.

7.3 Zarządzanie zadaniami

Agent ma zdefiniowane metody, które będą wykonywane po otrzymaniu odpowiedniego polecenia z serwera. Lista poleceń:

- Załadowanie nowego zadania
- Usunięcie zadania
- Uruchomienie zadania
- Zabicie zadania
- Zatrzymanie wykonywania zadania
- Wznowienie wykonywania zadania
- Synchronizacja czasu
- Potwierdzenie żywotności

7.4 Diagram klas



8 Konsola administratora

8.1 Opis

Konsola administratora stanowi interfejs, który umożliwia użytkownikowi (administratorowi) sterowanie systemem komunikacji zarządzania rozproszonymi procesami. Poprzez wysyłanie odpowiednich komend, użytkownik powinien mieć możliwość pełnej konfiguracji systemu, wydawania konkretnych poleceń związanych z pracą systemu oraz otrzymywania informacji zwrotnych o statusie całego systemu. Informacje

zwrotne można wykorzystać do generowania raportów. Dodatkowo konsola administratora powinna zadbać o poprawność wprowadzanych komend i danych, a także sprawdzać parametry komunikatów otrzymywanych z sieci.

8.2 Komunikacja z serwerem

Komunikacja konsoli administratora z serwerem odbywa się dwustronnie, tj. do serwera są wysyłane komendy i polecenia, a od niego otrzymywane są raporty i informacje zwrotne, a także komunikaty o błędach. Wymaga to ciągłego nasłuchiwania komunikatów z serwera (informację można dostać w dowolnym momencie) przy jednoczesnym rozpoznawaniu komend wydanych przez użytkownika i, po sprawdzeniu ich poprawności, wysłaniu ich na serwer. Wymaga to powołania do pracy dwóch wątków (nasłuchującego i wysyłającego).

8.3 Sprawdzanie poprawności

W programie konsoli administratora będzie sprawdzana poprawność wpisywanych komend, ewentualnych argumentów programu podawanych z linii poleceń (do czego to będzie potrzebne i czy w ogóle?) oraz parametrów komunikatów odebranych z sieci, w celu uniknięcia błędów związanych z działaniem systemu lub celowego złośliwego działania i ataków na system. Na przykład gdy informacje zwrotne z serwera będą zawierały nieprawidłowy identyfikator węzła/procesu zostaną zignorowane, a próba powtórzona.

8.4 Komendy

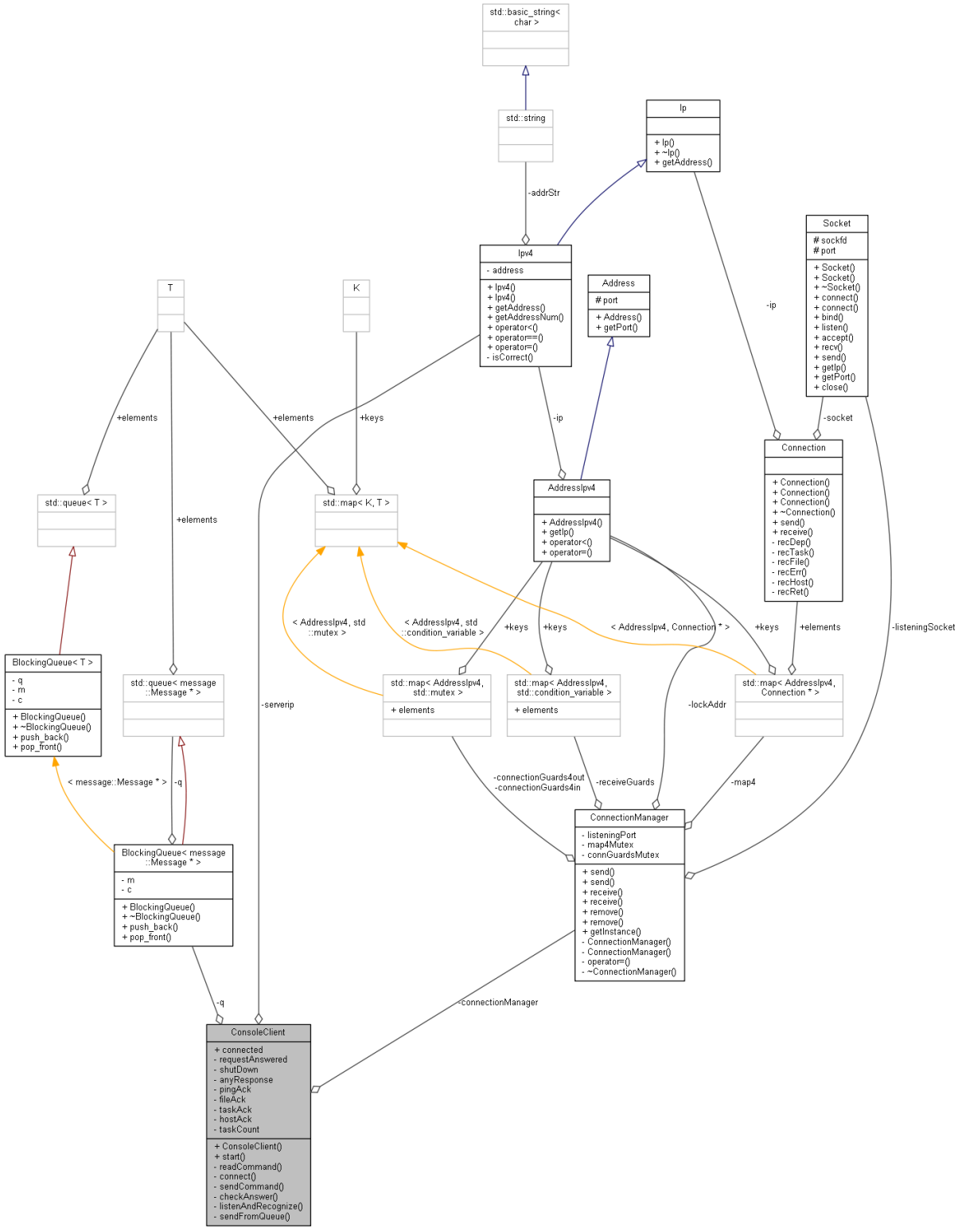
Komendy wydawane w konsoli administratora powinny umożliwiać:

- Dodanie/usunięcie/wstrzymanie/wznowienie zadania
- Nadanie zadaniu priorytetu podczas dodawania
- Definiowanie kolejności następowania zadań
- Oznaczenie zadania jako gotowego do przetwarzania
- Dodanie/usunięcie agenta
- Żądanie otrzymania raportu o pracy systemu (jednorazowo bądź cyklicznie)

8.5 Raporty

Konsola administratora, dzięki otrzymywaniu raportów z serwera, w tym raportów cyklicznych, może generować zbiorczy raport z całej pracy systemu (przez określony czas lub do zakończenia pracy) w oddzielnym pliku. Format tych raportów pozostaje do ustalenia (jakie informacje, kiedy, jak często, w jakiej formie to przedstawić).

8.6 Diagram klas



9 Przypadki użycia

Aby zmniejszyć objętość przypadków użycia, a zwłaszcza scenariuszy alternatywnych, powtarzające się, schematyczne działania zostaną opisane osobno w sposób ogólny.

9.1 Wysłanie polecenia

Proces zlecenia wykonania jakiegoś działania wygląda następująco:

1. Zlecającemu wysyła polecenie i czeka na odpowiedź
2. Odbiorca wysyła potwierdzenie odebrania polecenia
3. Odbiorca wykonuje polecenie
4. Odbiorca wysyła komunikat o sukcesie/porażce
5. Zlecający, po otrzymaniu komunikatu o sukcesie/porażce wysyła potwierdzenie otrzymania
6. Zlecający podejmuje dalsze działania

9.2 Błędy

Wystąpienie w trakcie wykonywania polecenia błędu zawsze wiąże się z następującym schematem działania:

1. Wykonawca zgłasza zlecającemu porażkę w wykonaniu czynności
2. Wykonawca wysyła zlecającemu stosowny komunikat ERR określający przyczynę błędu
3. Zlecający wysyła potwierdzenie otrzymania komunikatu o błędzie
4. Zlecający podejmuje dalsze działania

Błędy protokołu:

- Za długi/za krótki nagłówek
- Zabronione wartości pól

Błędy logiki:

- Dodawanie istniejących już zadań/plików/agentów
- Usuwanie nieistniejących zadań/agentów
- Oznaczenie jako gotowych nieistniejących zadań
- Uruchamianie niegotowych zadań
- Zabicie nieuruchomionych zadań
- Zatrzymywanie nieuruchomionych zadań
- Wznowienie niezatrzymanych zadań
- Ustalanie kolejności wykonywania nieistniejących zadań
- Ustalanie kolejności wykonywania zadań oznaczonych jako gotowe

9.3 Przekroczenie czasu oczekiwania na odpowiedź

Jeśli jedna strona nie odpowiada, strona oczekująca próbuje kilkukrotnie nawiązać komunikację komendą PING (protokołu LOTC). Jeśli uzyska odpowiedź, powtarza polecenie na które nie otrzymała odpowiedzi jeśli nie, dalsze działania zależą od tego, kim jest strona oczekująca:

- Agent: kontynuuje wykonywanie przydzielonych już zadań
- Serwer: oznacza agenta jako nieresponsywnego, przekazuje zadania tego agenta innym agentom
- Konsola: powiadamia użytkownika o utracie łączności z serwerem

Ponadto strona oczekująca okresowo próbuje nawiązać komunikację za pomocą komendy PING.

9.4 Przypadek użycia: obsługa zadań i agentów

1. Administrator wydaje polecenie dodania/usunięcia/wykonania/zabicia/wstrzymania/wznowienia zadania, oznaczenia zadania jako gotowego, ustalenia kolejności wykonywania zadań lub dodania/usunięcia agenta
2. Konsola przesyła odpowiedni komunikat serwerowi
3. Serwer aktualizuje stan zadań/agentów zgodnie z poleceniem
4. Serwer przesyła konsoli odpowiedni raport i aktualizuje strategię wykonywania zadań
5. Konsola wyświetla raport administratorowi

9.5 Przypadek użycia: żądanie raportu

1. Administrator wydaje żądanie raportu
2. Konsola przesyła odpowiedni komunikat serwerowi
3. Serwer zbiera niezbędne dane, odpytując agentów jeśli to konieczne
4. Serwer przesyła konsoli odpowiedni raport
5. Konsola wyświetla raport administratorowi

10 Sposób testowania aplikacji

Testowanie działania systemu odbywać się będzie etapami:

1. Kompilacja i wykrycie błędów składniowych poszczególnych modułów
2. Testy jednostkowe dla poszczególnych metod klas (tam, gdzie to zasadne)
3. Testy integracyjne z wykorzystaniem pluginu do Wiresharka sprawdzające poprawność wysyłanych komunikatów i wywołań powiązanych z nimi metod
4. Uruchomienie skryptu do konsoli administratora symulującego przypadki użycia

Wykorzystanie metodykę zwiną i iteracyjnego modelu wytwarzania oprogramowania pozwoli na przeprowadzenie testów już po implementacji minimum funkcjonalności systemu. Testy będą stopniowo rozszerzane wraz z rozbudową systemu.

11 Sposób demonstracji rezultatów

Demonstracja rezultatów będzie się opierała na realizacji wybranych (lub wszystkich) przypadków użycia przy wykorzystaniu napisanego wcześniej skryptu testującego.

12 Wnioski

12.1 Tomasz Jakubczyk - wnioski z testowania

Niezaprzeczalnie poprawnie działa kolejka blokująca BlockingQueue. Jest to bardzo ważny element projektu, gdyż zapewnia niezawodną synchronizację wątków. Bez tego nie było by szans, żeby coś działało. Każdy z modułów, czyli serwer, konsola, agent i protokół, same z siebie wydają się działać poprawnie, jednak testowanie wykazało, że moduły te zapałały do siebie gorącą nienawiścią.
Ponadto wykorzystywane kompilatory g++ i biblioteki obsługujące standard c++11 i c++14 wykazują się różnymi dziwnymi zachowaniami. O ile kompilacja i linkowanie przebiega pomyślnie, to nie można powiedzieć, że wykonanie programów takie jest.
Na Ubuntu występuje błąd

```
*** buffer overflow detected ***:  terminated
```

wydaje się, że jest to wina bibliotek dystrybucji Linuxa.
Inne problemy występują pod Cygwinem, który teoretycznie powinien być zgodny z Linuxem. Tam wydaje się, że Istnieją pewne ograniczenia co do otwierania i zamykania gniazd. Objawia się to, niemożliwością powtórnego nawiązania połączenia. Nie da się też wykluczyć, że powyższe błędy są związane z testowaniem poszczególnych, modułów aplikacji naraz na jednym komputerze bez ustawionych maszyn wirtualnych. Wydaje się, że biblioteki socket nadal są niedopracowane i zawierają błędy.
Nie mniej udaje się wykonywać programy aplikacji LOTC na Linux Arch, bez wyżej wymienionych błędów.
Mimo, że prawie od samego początku projektu zaczęliśmy testować protokół aplikacji, to i tak pod koniec, okazało się, że w połączeniu z pozostałymi częściami aplikacji wszystko nagle przestało działać. Być może było to spowodowane zbyt słabym opisaniem metod publicznych klas protokołu. W trakcie integrowania ze sobą modułów wyszły na jaw różne nieporozumienia co do działania protokołu. Być może przy protokole powinno pracować więcej osób, albo najlepiej wszyscy, ale to by oznaczało wchodzenie sobie w drogę.
Jeśli jeszcze raz miał bym robić ten projekt, to wstępnie zarządził bym, żeby każdy napisał własną obsługę gniazd do przesyłania komunikatów protokołu i wtedy dużo łatwiej było by wykrywać błędy, bo wyraźnie były by widoczne wizje działania komunikacji sieciowej poszczególnych programistów i dało by się ich w porę naprostować.
Testowanie okazało się też niesamowicie czasochłonne i nieprzyjemne.
Mimo usilnych starań nie udało się przekonać członków zespołu do napisania odpowiedniej liczby testów jednostkowych i postało ich zaledwie kilka.
Zostało napisanych kilka skryptów testowych, ale raczej w niczym one nam nie pomogły, a próby doprowadzenia ich do działania na niektórych maszynach i systemach zajęły sporo czasu. Oczywiście niezwykle przydatne okazały się ogromne ilości komunikatów debugowania oraz GDB pozwoliło szybko znaleźć kilka poważnych błędów, ale nie wszystkie.
GDB wykazuje się złym zachowaniem, jeśli w programie zwracane są wyjątki, które nie mają zakończyć programu.
Być może powinno się wyznaczyć jedną osobę, która zajmowała by się tylko testowaniem tego co inni napisali.

12.2 Tomasz Jakubczyk - opis doświadczeń z projektu

Po pierwsze lepiej dobierać współpracowników.
Zwykle ten obowiązek spada na head hunterów, a przynajmniej można zobaczyć CV. Niestety, osoby preferowane na członków projektu postanowiły nie zapisać się na przedmiot TIN. Nie uzyskałem też zgody na realizowanie projektu w mniejszym zespole, co niewątpliwie dało by lepszy efekt.
Bardzo zauważalny okazał się brak środków nacisku na członków zespołu. Niektórzy członkowie mimo napomnień odkładali pisanie projektu, a na końcu zamiast wziąć się w garść i zarwać trzy noce odpuścili sobie zupełnie. Być może częściowo ponosi za to winę okres świąteczny, zalew kolokwiami i innymi projektami przez czas trwania projektu od prezentacji wstępnej do terminu oddania.
Być może jeśli cały projekt byłby zadany już w pierwszym lub drugim tygodniu semestru, kiedy jest jeszcze dużo czasu i termin oddania byłby gdzieś w pierwszej połowie semestru, przebieg projektu byłby zgoła inny. Nie widzę też żadnego dobrego uzasadnienia, czemu tak by nie miało być, bo wiedzę potrzebną do zakodowania aplikacji albo w większości już posiadaliśmy, albo i tak musieliśmy doczytać bezpośrednio ze źródeł (internetu).

Problemem okazało się też zbyt mało precyzyjne opisanie szczegółów implementacji protokołu w dokumentacji wstępnej. Uważam, że opis pól komunikatów był niewystarczający. Powinniśmy byli napisać jak dokładnie będziemy wykorzystywać gniazda, z pewnością oszczędziło by to wiele czasu przy próbach integracji modułów aplikacji.

Okazało się, że niektórzy członkowie mają skłonność do rzadkiego commitowania swoich postępów na githuba, co utrudniało kontrolę sytuacji. Przydało by się jakieś oprogramowanie do podglądania, czym właśnie zajmują się członkowie zespołu. Niezwykle przydatny okazał się komunikator Skype w fazie integracji i debugowania aplikacji. Pozwoliło to szybko zorientować się w sytuacji co do projektu i nakierować tok prac na właściwe tory.
Dla odmiany Facebook nie okazał się zbyt dobrym sposobem komunikacji, chociaż mogło to wynikać z

tego którzy członkowie zespołu wykorzystywali go do komunikacji. Komunikacja przez Facebooka prowadziła często do tego, że pytania bądź polecenia pozostawały bez odpowiedzi. Może być to też związane ze zbyt dużym natłokiem informacji na Facebooku.
Zauważalna była duża rozbieżność w umiejętnościach programistycznych członków zespołu. Jest to zapewne związane z tym, że program studiów informatyki na Wydziale Elektroniki Politechniki Warszawskiej przewiduje stanowczo za mało godzin poświęconym nauce języków programowania.
Testowanie regresyjne oraz integracyjne powinno być przeprowadzane co najmniej raz w tygodniu, niestety, niektórzy członkowie aż do końca projektu nie napisali nic co mogło by się do tego nadawać.
Sądzę, że wprowadziliśmy zbyt rozbudowaną strukturę logiczną projektu, który tego na prawdę nie wymagał i przez to nie starczyło czasu na pełne zrealizowanie projektu. Jeśli projekt zostałby napisany strukturalnie w języku c przez jedną osobę, to wyszło by to dużo prościej, dużo szybciej, zostały by zrealizowane wszystkie założenia projektowe i debugowanie okazało by się dużo prostsze.
Ubolewamy, też nad tym, że nie udało się napisać pluginu Wiresharka do podglądania komunikacji sieciowej, jednak w naszym przypadku dużo by nie pomógł i uznaliśmy, że w zbyt ograniczonym czasie potrzeba zająć się raczej usuwaniem błędów krytycznych wywołania programu.
W ostatnich dniach projektu byliśmy o krok od uzyskania działającego jednocześnie IPv4 i IPv6, jednak zabrakło na to czasu i w ogromie innych problemów musieliśmy zadowolić się działającym IPv4. Gdyby na początku projektu klasa Ip nie była by klasą wirtualną po której dziedziczą Ipv4 i Ipv6, to niewątpliwie to by się nam udało. Niestety na chwilę obecną Ipv6 jest tylko częściowo zrealizowane w kodzie.
Już tydzień przed końcem projektu zorientowaliśmy się, że raczej nie ma szans na realizacje synchronizacji zegarów przez pobranie z serwera NTP, niemniej został napisany kod do protokołu który może bez problemu obsługiwać tą opcję.
Okazało się też, że i tak testujemy naszą aplikację w jednym środowisku i sens tego jest znikomy.

12.3 Andrzej Roguski

Jeśli miałbym wskazać największe gwoździe w trumnie projektu, byłyby nimi faza projektowania i testowania.

Znaczącym problemem okazała się rzecz prozaiczna - aplikacja docelowo mająca działać na wielu hostach, była testowana na pojedynczej maszynie.
Aby w ogóle umożliwić uruchomienie na jednym hoście serwera, konsoli i agentów, moduł komunikacyjny musiał poradzić sobie z przetwarzaniem komunikatów nie na podstawie samych adresów IP, ale także i portów.
Każdy moduł musiał nasłuchiwać na innym porcie. Serwer musiał też po porcie rozpoznawać rozmówców - a ten domyślnie był losowany. Przypisanie konkretnych portów wyjściowych prowadziło do błędów funkcji związanych z socketami, mimo użycia `SO_REUSEADDR` - i na tej zasadzie bardzo duża część czasu poświęcona została na walkę z funkcjonalnością niezwiązaną z samym zadaniem, który to czas mógłby zostać wykorzystany na pisanie testów, implementację obsługi IPv6 czy stworzenie modułu Wireshark.
Co gorsza, z nieznanych nam przyczyn aplikacje inaczej zachowywały się (czyt. działały bądź nie) u różnych osób - mimo środowisk teoretycznie spełniających wymogi dotyczące kompilacji i uruchomienia.

Mam też wrażenie, że wielu problemów można by uniknąć dzięki większej liczbie spotkań projektowych i większej aktywności zespołu przed przystąpieniem do "prac głównych". Jeszcze na etapie integracji pokutowały nieporozumienia odnośnie działania poszczególnych modułów albo głupie przeoczenia oczywiste z perspektywy programisty-odbiorcy danej funkcjonalności.

Na pewno nie pomogła znikoma aktywność aż połowy zespołu - mimo usilnych nalegań teamleadera, połowa(!) modułów powitała nowy rok w stanie niemal dziewiczym, co doprowadziło do panicznego wyścigu z czasem w ostatnim tygodniu przerwy świątecznej. Dodając do tego niezrozumiałe różnice w działaniu aplikacji, pod koniec prac tylko ja miałem możliwość testowania aplikacji, a Tomek dokładną znajomość kodu serwera, konsoli i agenta, co skutkowało tym, że żaden z nas nic sensownego nie mógł w rozsądnym czasie osiągnąć. Pierwotny zamysł przydzielania sił zespołu tam, gdzie jest to potrzebne runął w gruzach.

13 Wkład

13.1 Podział prac

Osoba odpowiedzialna	Zadania
Tomasz Jakubczyk	Teamleader, serwer
Eryk Ratyński	Agent
Andrzej Roguski	Protokół, klient NTP, plugin Wireshark, dokumentacja
Kacper Stachyra	Konsola administratora

13.2 Praca przy zadaniach

Osoba	Zadania
Tomasz Jakubczyk	Teamleader, serwer, konsola, agent, testowanie
Eryk Ratyński	Agent
Andrzej Roguski	Protokół (IPv4), dokumentacja, testowanie
Kacper Stachyra	Konsola administratora

13.3 Czas

Osoba	Zadania
Tomasz Jakubczyk	około 100 h
Eryk Ratyński	około 25 h
Andrzej Roguski	około 100 h
Kacper Stachyra	około 100 h