Jędrzej Kuryło

Graficzna konsola sterownicza systemu MRROC++ stworzona w oparciu o platformę Java

Praca dyplomowa inżynierska pod kierunkiem
mgr inż. Tomasza Winiarskiego

Instytut Automatyki i Informatyki Stosowanej
Politechniki Warszawskiej

Warszawa, Wrzesień 2008

Streszczenie

Celem mojej pracy inżynierskiej było zaprojektowanie i implementacja nowej graficznej konsoli sterowniczej systemu MRROC++ na podstawie istniejącego już rozwiązania. W przeciwieństwie do pierwowzoru, miała ona umożliwiać zdalną pracę w systemie, być przenośna między różnymi platformami sprzętowymi i programowymi, a jej dystrybucja i aktualizacja powinna być możliwie nieskomplikowana.

Graficzna konsola sterownicza została zrealizowana w architekturze klient-serwer. Serwer aplikacji napisany został w języku C++ i działa jako jeden z procesów systemu MRROC++. Aplikacja kliencka to applet języka Java. Komunikacja pomiędzy klientem i serwerem odbywa się przy pomocy pary protokołów TCP/IP.

Powstała w ramach pracy konsola spełnia wszystkie stawiane przed nią wymagania, oferuje również możliwości niedostępne w pierwowzorze. Poprawność działania aplikacji zweryfikowana została w warunkach laboratoryjnych podczas sterowania rzeczywistymi robotami.

Abstract

Title: Java based MRROC++ graphical user Interface

The aim of this thesis was to design and implement a new graphical user interface for the MRROC++ system, based on the existing console. Unlike its prototype, the new interface had to provide remote access to the MRROC++ system. Moreover, it had to be a cross-platform and easy to distribute and update.

The new graphical user interface uses client-server model. The server was written in C++ and works as one of MRROC++ processes. The graphical user interface is a Java applet, communicating with server via TCP/IP protocols.

Application meets all the requirements, but it also offers some new functions. It has proved to work properly in the laboratory experimental setup.

Spis treści

1 Wstęp

1.1 Geneza pracy

Częstą sytuacją podczas programowania systemów robotowych jest wytwarzanie podobnych do siebię programów, jednak działających na różnym sprzęcie, realizujących nieco odmienne zadania. Wtedy w sukurs przychodzą programowe struktury ramowe, wspomagające tworzenie sterowników dla systemów robotowych. Zawierają one moduły i wzorce programowe oraz narzędzia przydatne przy realizacji programów sterujących.

Jedną z takich programowych struktur ramowych jest MRROC++[1] [1] stworzona w Instytucie Automatyki i Informatyki Stosowanej Politechniki Warszawskiej. Wspomaga ona tworzenie sterowników dla współbieżnych systemów wielorobotowych. Napisana w języku C++ działa pod kontrolą systemu operacyjnego czasu rzeczywistego QNX Neutrino [2]. Jednym z komponentów MRROC++ jest graficzna konsola sterownicza, umożliwiająca m.in. uruchamianie procesów sterujących i zadań, ręczne sterowanie manipulatorami oraz udostępniającą informacje o stanie systemu.

Po latach użytkowania istniejącej graficznej konsoli sterowniczej okazało się, iż nie do końca spełnia ona oczekiwania obecnych użytkowników. Rozwój informatyki i pokrewnych jej dziedzin przyniósł rozwój nowych technologii, a wraz z nim wzrost oczekiwań użytkowników stawianych aplikacjom. Część zastosowanych rozwiązań okazała się być przestarzała, a nowe technologie pozwalają obecnie na realizację pewnych aspektów działania konsoli w efektywniejszy sposób. Dlatego też pojawił się pomysł zrealizowania nowej wersji graficznej konsoli sterowniczej w oparciu o istniejące dotychczas rozwiązanie. Nie tylko realizowane funkcje, ale również wygląd konsoli, możliwie zbliżony do dotychczas stosowanej, umożliwić miały możliwie szybką i bezproblemową migrację użytkowników na nową wersję. Nowa wersja konsoli zaś miałaby stanowić także solidną bazę, rozszerzaną później o nowe funkcje.

Z mojej strony zainteresowanie projektem i realizacją nowej wersji graficznej konsoli sterowniczej systemu MRROC++ wynikało z umiejscowienia tego tematu na pograniczu kilku ciekawych dziedzin - robotyki i programowania systemu wieloorobotowego z jednej strony, a cieszącej się ogromną popularnością technologią Java z drugiej strony. Dodatkową motywacją był również fakt rzeczywistego wykorzystania konsoli w pracy z systemem wielorobotowym.

1.2 Cel pracy

Celem mojej pracy inżynierskiej było zaprojektowanie i realizacja graficznej konsoli sterowniczej systemu MRROC++ o możliwościach pierwotnej konsoli. Aby nadać jednak sens powielaniu już istniejącej i z powodzeniem używanej aplikacji, przed re-

[1] Multi-Robot Research Oriented Controller

alizowaną konsolą postawiony został szereg wymagań, których pierwotna konsola nie spełnia, a które uznano za istotne w obliczu rozwoju technologii i metodologii programowania, jak i zmian w systemie MRROC++.

Podstawowym z tych wymagań jest wieloplatformowość rozwiązania, rozumiana jako umożliwienie sterowania systemem wielorobotowym przy użyciu możliwie dowolnej sprzętowej i programowej konfiguracji komputera. Dotychczas używana konsola może być uruchamiana jedynie pod kontrolą systemu operacyjnego QNX Neutrino, jednak sama w sobie nie korzysta ze szczególnych cech tego systemu czasu rzeczywistego. Nie ma więc istotnych przeciwwskazań, by umożliwić uruchomienie jej w innych systemach, zwłaszcza że w obliczu dzisiejszej różnorodności używanych systemów operacyjnych taka możliwość jest wskazana. Wymaganie to dodatkowo zyskuje na ważności w obliczu trwających już od dłuższego czasu prac nad przeniesieniem na platformy inne niż QNX niektórych elementów struktury MRROC++ i środowiska programistyczno-developerskiego.

Równie pożądaną cechą, co wieloplatformowość rozwiązania, jest możliwość pracy zdalnej w systemie MRROC++ przy wykorzystaniu niezależnych od platformy protokołów sieciowych. Dotychczasowe rozwiązanie, mimo że działające w środowisku rozproszonym, dopuszcza pracę jedynie w obrębie sieci lokalnej QNET[2], a to ograniczenie jest nieuzasadnione w obliczu łatwego obecnie dostępu do sieci praktycznie z każdego komputera. Wymagana możliwość pracy zdalnej rozumiana jest więc jako umożliwienie pracy w systemie MRROC++ przy użyciu komputera w dowolnej lokalizacji z możliwie dowolnym uruchomionym systemem operacyjnym. Jest ono również konsekwencją wymaganej wieloplatformowości rozwiązania, gdyż dopuszczenie do działania komputerów z systemem operacyjnym innym niż QNX wymaga zastosowania innego niż QNET protokołu sieciowego.

Kolejną cechą, którą spełniać powinna realizowana aplikacja, jest łatwość dystrybucji. Instalacja i uruchomienie dotychczasowej konsoli, ze względu na ścisłą jej integrację ze strukturą MRROC++, wymaga instalacji całej struktury. Dlatego też kluczem do łatwiejszej dystrybucji i aktualizacji jest wyodrębnie graficznej konsoli sterowniczej z systemu MRROC++ i realizacja jej jako samodzielnej aplikacji. Takie rozwiązanie pozwoli też uczynić uruchomienie konsoli na wybranej maszynie mniej pracochłonnym i skomplikowanym.

Spełnienie wymienionych wymagań przez realizowaną konsolę zaowocuje aplikacją znacznie przewyższającą dotychczasowe rozwiązanie funkcjonalnością i wygodą użytkowania. Nowe możliwości wykorzystania graficznej konsoli sterowniczej wydają się być warte wysiłku włożonego w realizację aplikacji.

[2]natywny protokół sieciowy systemu QNX Neutrino, który czyni z wchodzących w skład sieci komputerów rozproszone środowisko o wspólnych zasobach

2 Wprowadzenie do systemu MRROC++

2.1 Struktura ramowa MRROC++

MRROC++ [1] to programowa struktura ramowa wspomagająca wytwarzanie sterowników dla systemów wielorobotowych. Powstała ona w Instytucie Automatyki i Informatyki Stosowanej Politechniki Warszawskiej. Historia struktury sięga początku lat dziewięćdziesiątych, kiedy to powstał jej pierwowzór RORC[3] [3], wspomagający tworzenie sterowników dla pojedynczych robotów. Następcą struktury RORC był powstały w 1995 roku MRROC [4], umożliwiający współpracę z systemami wielorobotowymi, jednak stworzony z wykorzystaniem paradygmatu proceduralnego, w przeciwieństwie do powstałego kilka lat później obiektowego MRROC++. Struktura MRROC++ jest rozwijana po dziś dzień, w dużej mierze dzięki temu, iż stanowi podstawowe narzędzie wykorzystywane w pracach dyplomowych realizowanych w Laboratorium Robotyki. Od chwili powstania została wykorzystana do sterowania wykonaniem różnorodnych zadań - m.in. polerowania i frezowania przez robota RNT czy gry w warcaby i układania kostki Rubika [5] przez zmodyfikowane manipulatory Irp6 (Rysunek 1).

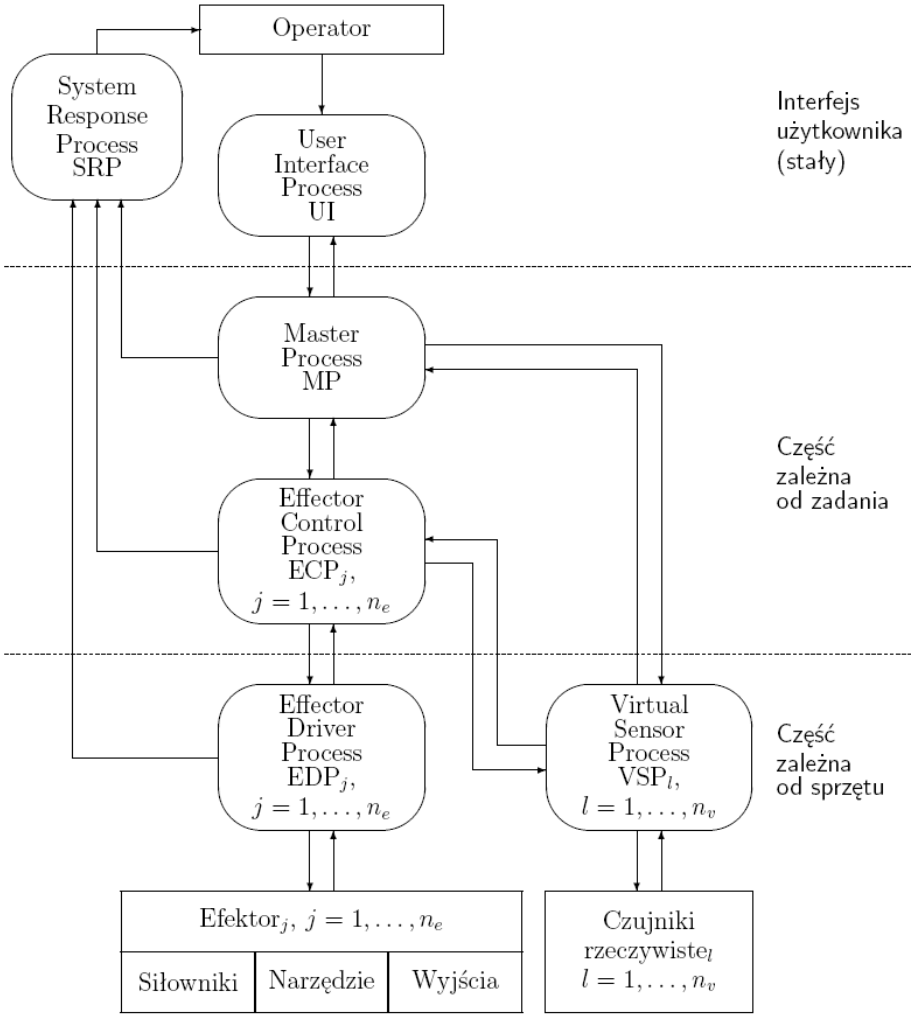
Rysunek 1: Dwa zmodyfikowane manipulatory Irp-6 układające kostkę Rubika

Struktura MRROC++ zawiera bibliotekę modułów oraz wzorców programów, na bazie których łatwo można zbudować dedykowany sterownik dla systemu wielorobotowego. Bibliotekę można rozszerzyć, tworząc nowe funkcje w C++, języku źródłowym struktury programowej, lub też modyfikując istniejące moduły. Powstały sterownik działa pod kontrolą systemu operacyjnego czasu rzeczywistego QNX.

Działający sterownik to w rzeczywistości zbiór procesów o ściśle określonych zadaniach, działających w węzłach sieci kolalnej. Sam sterownik ma budowę wielowarstwową

[3]Research-Oriented Robot Controller

i modularną (Rysunek 2), co upraszcza jego modyfikację, bowiem wymiana jednego z elementów nie pociąga za sobą wymiany czy modyfikacji pozostałych modułów.



Rysunek 2: Struktura sterownika MRROC++

Najniższą warstwą systemu jest warstwa sprzętowa. W tej warstwie działają procesy EDP[4], mające bezpośredni dostęp do sprzętu i sterujące ruchem pojedynczego efektora. Ich podstawowym zadaniem jest rozwiązywanie prostego i odwrotnego zadania kinematyki oraz realizacja funkcji charakterystycznych dla danego typu robota. Od strony programistycznej EDP stanowi wirtualny efektor, wykonujący zadane ruchy. Obok procesów EDP działają również procesy VSP[5] reprezentujące wirtualny sensor agregujący dane otrzymane z rzeczywistych czujników w wirtualny odczyt, stanowiący już informację użyteczną dla systemu. Odczyty czujników rzeczywistych wykonywane są co określony interwał czasu na żądanie procesu ECP lub MP.

Warstwa druga odpowiedzialna jest za wykonanie zadania na pojedynczym efekto-

[4]Effector Driver Process
[5]Virtual Sensor Process

rze. W jej skład wchodzą procesy ECP[6]. Ich ilość, podobnie jak procesów EDP, jest równa liczbie efektorów w systemie. Procesy te realizują algorytm sterowania pojedynczego manipulatora. W celu realizacji programu użytkowego komunikują się z procesem EDP sterującym odpowiadającym im robotom oraz procesami MP, VSP czy UI. Jednym z zadań pojedynczego procesu ECP może być generacja trajektorii dla robota pod jego kontrolą w przypadku niezależnej lub luźnej współpracy.

Najwyższą warstwę stanowi pojedynczy proces MP[7], koordynujący współpracę robotów i sterująca wykonaniem zadania. Synchronizuje w czasie działanie efektorów, a w przypadku ich ścisłej współpracy generuje trajektorie. Dodatkowo proces ten realizuje również polecenia użytkownika otrzymane z procesu UI.

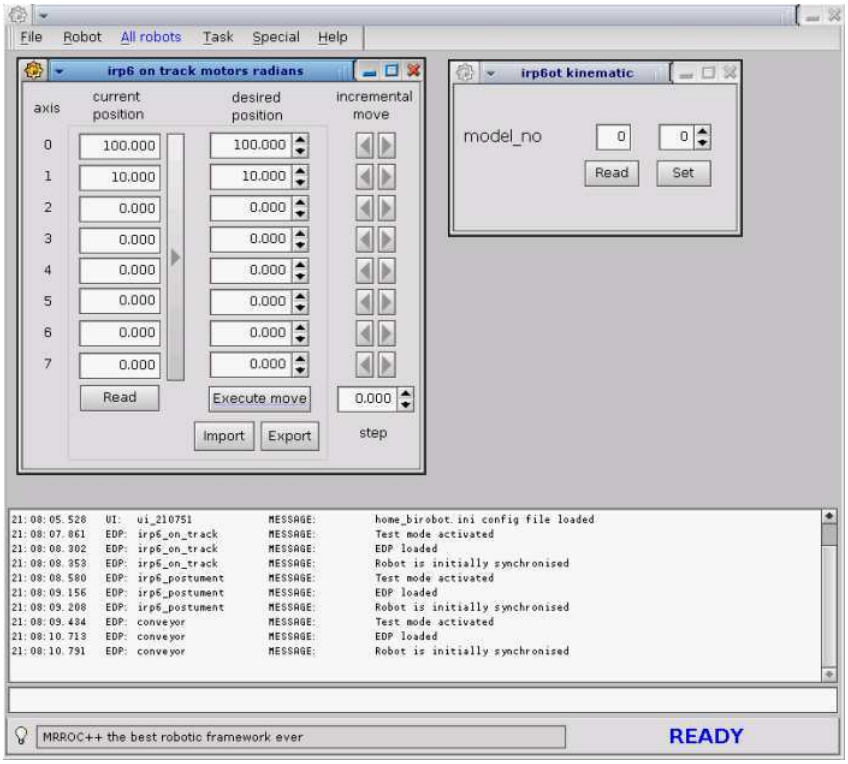
Oprócz wyżej wymienionych w systemie istnieje również warstwa niezależna od wykonywanego zadania, odpowiedzialna za komunikację z użytkownikiem. W warstwie tej pracuje proces UI[8]. Proces UI realizuje graficzny interfejs użytkownika, oparty na środowisku graficznym Photo MicroGUI, wchodzącym w skład systemu operacyjnego QNX. Jego wątek SR[9] odpowiedzialny jest za wyświetlanie stanu systemu sterującego.

Procesy ECP, MP i VSP są zależne od wykonywanego zadania, podczas gdy procesy EDP są ściśle związane ze sprzętem. Oznacza to, iż modyfikacja zadania sprowadza się do modyfikacji kodu procesów MP i ECP. Zmiany w sprzęcie z kolei pociągają za sobą konieczność modyfikacji kodu procesów EDP w przypadku efektora i VSP w przypadku czujnika. Taki rozdzial warstwy zadania od warstwy sprzętowej ułatwia tworzenie sterowników, gdyż umożliwia oprogramowanie zadania bez konieczności ingerencji w kod sterujący efektorów i czujników, a kod sterujący zadaniem uniezależnia od użytego sprzętu.

2.2 Graficzna konsola sterownicza

Podstawowym zadaniem graficznej konsoli sterowniczej jest umożliwienie sterowania systemem MRROC++ i poszczególnymi robotami w wygodny i intuicyjny sposób. Konsola została zbudowana w oparciu o środowisko graficzne Photon MicroGUI [6], wchodzące w skład systemu operacyjnego czasu rzeczywistego QNX. Środowisko to oferuje bogaty zestaw elementów graficznych (tzw. widgetów), obsługę wielu standardów kodowania tekstu i wsparcie dla aplikacji wielojęzycznych. W zależności od wykonywanej akcji konsola komunikuje się z odpowiednimi procesami EDP, ECP lub MP.

[6]Effector Control Process
[7]Master Process
[8]User Interface
[9]System Response



Rysunek 3: Graficzna konsola sterownicza systemu MRROC++ wykonana w Photon MicroGUI

2.2.1 Podstawowe elementy

Okno graficznej konsoli sterowniczej, przedstawione na rysunku 3 można podzielić na cztery główne elementy.

- menu górne — umożliwia wydawanie podstawowych komend dla systemu MRROC++, takich jak uruchomienie procesów EDP sterujących poszczególnymi robotami, uruchomienie procesu MP sterującego wykonaniem zadania czy wydanie rozkazów dla wszystkich uruchomionych robotów. W menu górnym zawarte są również indywidualne menu wszystkich robotów obsługiwanych przez system.

- panel główny — w obszarze tym otwierane są okna ruchów ręcznych poszczególnych robotów, umożliwiające m.in. podgląd położenia manipulatora czy też ruch wybranego robota do zadanej pozycji

- konsola — na niej wyświetlane są komunikaty systemowe, zawierające informacje o przebiegu zadania, potwierdzenia wykonania poleceń czy informacje o błędach

- pasek statusu — obok hasła głoszącego wyższość ramowej struktury programowej MRROC++ nad innymi podobnymi strukturami, wyświeta również informację

o aktualnej zajętości systemu, czy to wskutek powoływania właśnie procesów sterujących, komunikacji z procesami lub wykonywania rozkazu ruchu

2.2.2 Realizowane funkcje

Gdy w systemie MRROC++ nie są uruchomione żadne procesy sterujące, graficzna konsola sterownicza oferuje dostęp jedynie do podstawowych funkcji systemu. Umożliwia ona wtedy m.in. wczytanie konfiguracji z pliku wybranego z listy, wyświetlającej zawartość katalogu *configs/* oraz czyszczenie konsoli tekstowej. Najważniejszym jej zadaniem jest jednak uruchamianie procesu MP, sterującego wykonaniem zadania, oraz wcześniej procesów EDP poszczególnych robotów - pojedynczo dla każdego robota lub jednocześnie dla wszystkich robotów wymienionych w pliku konfiguracyjnym. Uruchomienie tych procesów umożliwia dostęp do dużo szerszego zakresu funkcji oferowanych przez system.

Uruchomienie procesu EDP daje dostęp do menu wybranego robota. Zawiera ono z reguły dostęp do polecenia synchronizacji oraz ruchu robota do jednej z predefiniowanych pozycji. Pas transmisyjny Conveyor udostępnia dodatkowo okno dialogowe Move, służące do zadawania pozycji przy pomocy położenia silników. Zmodyfikowane manipulatory Irp6 oferują o wiele więcej sposobów specyfikowania pożądanej pozycji. Może być ona zadana za pomocą położeń silników oraz stawów, jak również przy pomocy współrzędnych oś-kąt oraz współrzędnych Eulera. Dodatkowo możliwe jest zadawanie pozycji narzędzia w tych układach współrzędnych oraz wybór modelu kinematyki i algorytmów serworegulacji poszczególnych stawów. Każdy z robotów udostępnia również polecenie zakończenia działania procesu EDP.

Uruchomienie procesu MP daje dostęp do funkcji wywoływanych z okna Process Control, dostępnego w menu Task. Okno to umożliwia wstrzymywanie i ponowne uruchamianie procesu MP, jak również uruchamianie i zatrzymywanie wątków pomiarowych EDP_READER, przechowujących i zapisujących stan procesu EDP.

2.2.3 Wady dotychczasowego rozwiązania

Istniejąca graficzna konsola sterownicza, jako aplikacja nieco wiekowa, nie nadąża niestety za najnowszymi trendami. Dotychczasowa konsola nie korzysta z pełni możliwości, które oferuje wielowątkowość. Uniemożliwia ona przede wszystkim wykonywanie równoległych operacji na różnych robotach. Wskutek tego już sama synchronizacja robotów odbywa się sekwencyjnie i zajmuje kilkakrotnie więcej czasu niż mogłaby, mimo że nie ma przeciwwskazań, by w pewnych przypadkach wykonywać ją równolegle na wszystkich robotach.

Kolejnym problemem jest konieczność instalacji systemu operacyjnego QNX na maszynie rzeczywistej lub wirtualnej w celu uruchomienia dotychczasowej konsoli sterow-

niczej. Nie jest to wadą, gdy użytkownik planuje uruchamianie na tej samej maszynie aplikacji stworzonych dla systemu MRROC++, gdyż do ich uruchomienia system QNX jest i tak wymagany. Jednak w przypadku, gdy na wybranej maszynie uruchamiana ma być jedynie konsola sterownicza, koszt instalacji, mierzony poświęconym na to czasem, miejscem na dysku twardym komputera, czy też złożonością tej operacji, wydaje się być nieadekwatny do rzeczywistych wymagań sprzętowych i systemowych aplikacji, jaką jest konsola sterownicza.

Kolejną wadą dotychczasowego rozwiązania jest jego nieprzenośność na inne systemy operacyjne ze względu na ścisłą integrację środowiska Photon MicroGUI z systemem QNX. Spełnienie wymagania jak największej przenośności aplikacji między różnymi systemami operacyjnymi umożliwić ma pracę z systemem MRROC++ przy użyciu możliwie dowolnej platformy, bez ponoszenia wcześniej wymienionych kosztów.

Przywiązanie aplikacji do systemu operacyjnego QNX ogranicza również możliwość pracy zdalnej, bardzo pożądanej obecnie cechy, gdyż obszar roboczy aplikacji sprowadza się do obrębu sieci lokalnej QNET.

Celem pracy jest stworzenie konsoli wolnej od powyższych wad i ograniczeń.

3 Opis wybranych technologii

Zrealizowana przeze mnie graficzna konsola sterownicza systemu MRROC++ napisana została jako applet języka Java [7]. Do budowy interfejsu graficznego wykorzystałem bibliotekę Java Swing. Użyte do realizacji zadania technologie postaram się przybliżyć w tym rozdziale.

3.1 Platforma Java

Historia powstania i rozwoju języka Java ma swój początek w grudniu 1990 roku, kiedy to zespół inżynierów zatrudnionych w firmie Sun dostał za zadanie opracowanie technologii tworzenia przenośnych aplikacji, zapewniających wielowątkowość, automatyczne zarządzanie pamięcią i możliwość programowania rozproszonego. Projekt ten został nazwany Stealth Project. W 1992 roku zademonstrowany został nowy język Oak, dwa lata później przemianowany na Javę z powodu zastrzeżenia nazwy Oak przez inną firmę. Oficjalna premiera platformy Java miała miejsce 23 grudnia 1995 roku. Od 2006 roku platforma Java jest udostępniana prawie w całości na licencji GNU, z wyłączeniem kilku bibliotek z zamkniętym źródłem.

Platforma Java to zbiór narzędzi i bibliotek udostępnionych przez firmę Sun Microsystems w celu tworzenia aplikacji w języku Java. Dostępna jest ona w czterech wersjach:

Java Card umożliwia pisanie małych aplikacji uruchamianych na kartach elektronicznych (tzw. smart cards)

Java Micro Edition dedykowana dla urządzeń przenośnych z niewielką ilością dostępnych zasobów

Java Standard Edition stworzona z myślą o tworzeniu aplikacji dla komputerów osobistych

Java Enterprise Edition umożliwia tworzenie wielowarstwych, najczęściej rozproszonych aplikacji klasy Enterprise

Podstawowym narzędziem, na którym oparta jest platforma, jest język Java. Jest to silnie zorientowany obiektowo język wysokiego poziomu - kompilator Javy wymaga, by całość kodu aplikacji znajdowała się w klasach. Składnia języka wzorowana jest na języku C++, jednak posiada uproszczony model obiektowy i uniezależniona jest od niskopoziomowych operacji.

Najważniejszą cechą języka i platformy jest to, że zostały zaprojektowane jako całkowicie niezależne od platformy sprzętowej, na której uruchamiane mają być aplikacje. Jest to możliwe, ponieważ aplikacje napisane w Javie nie są uruchamiane bezpośrednio na maszynie, ale w ustandaryzowanym środowisku nazywanym wirtualną maszyną

Javy[10], zaimplementowanym pod różnorodne systemy operacyjne. Stworzone w języku Java programy są kompilowane do kodu pośredniego (tzw. bytecode), zawierającego instrukcje dla maszyny wirtualnej. Dystrybucja programu w postaci kodu pośredniego, wykonywanego na maszynie wirtualnej, daje niezależność od platformy, gdyż może być wykonany wszędzie tam, gdzie znajduje się wirtualna maszyna Java.

Przenośność między systemami operacyjnymi Java zawdzięcza mnogości implementacji wirtualnej maszyny Java. Oprócz oficjalnej implementacji firmy Sun Microsystems dla systemów Windows, Linux i Solaris, stworzone zostały również implementacje innych dostawców, ale posiadające certyfikaty kompatybilności firmy Sun, co gwarantuje przenośność aplikacji między różnymi implementacjami. Nieoficjalne implementacje maszyny wirtualnej Javy powstały dla następujących platform:

Windows — implementacje firm IBM i Microsoft

Linux — implementacja firmy IBM, projekt Blackdown i Kaffe

OS/2 — implementacje firm IBM, GoldenCode Development i Innotek

MacOS — implementacja firmy Apple

HP-UX — implementacja firmy HP

Irix — implementacja firmy SGI

AIX,OS/400 — implementacja firmy IBM

Na rysunku 4 przedstawiono schemat platformy Java SE, wykorzystanej do realizacji zadania. Na samej górze znajduje sie język Java, na dole zaś platformy sprzętowe, na których wykonują się aplikacje w języku Java. Pomiędzy nimi umiejscowiono narzędzia i biblioteki, umożliwiające tworzenie tekstowych i graficznych aplikacji. Najważniejszymi elementami platformy są:

java — interpreter języka Java, wykonujący na maszynie wirtualnej rozkazy zawarte w kodzie pośrednim

javac — kompilator języka Java, tłumaczący kod w języku Java na kod pośredni

javadoc — narzędzie do wytwarzania dokumentacji na podstawie znaczników umieszczonych w kodzie programu

jar — narzędzie do tworzenia archiwów, zawierających klasy wykorzystywane przez aplikację

[10]Java Virtual Machine (JVM)

RMI — Remote Method Invocation - zbiór narzędzi umożliwiających interakcję aplikacji z innymi systemami za pośrednictwem sieci

AWT, Swing, Java2D — biblioteki graficzne, służące do budowy graficznego interfejsu użytkownika

Accessibility, Drag'n'Drop — biblioteki zapewniające dodatkowe funkcje ułatwiające pracę z graficznymi aplikacjami

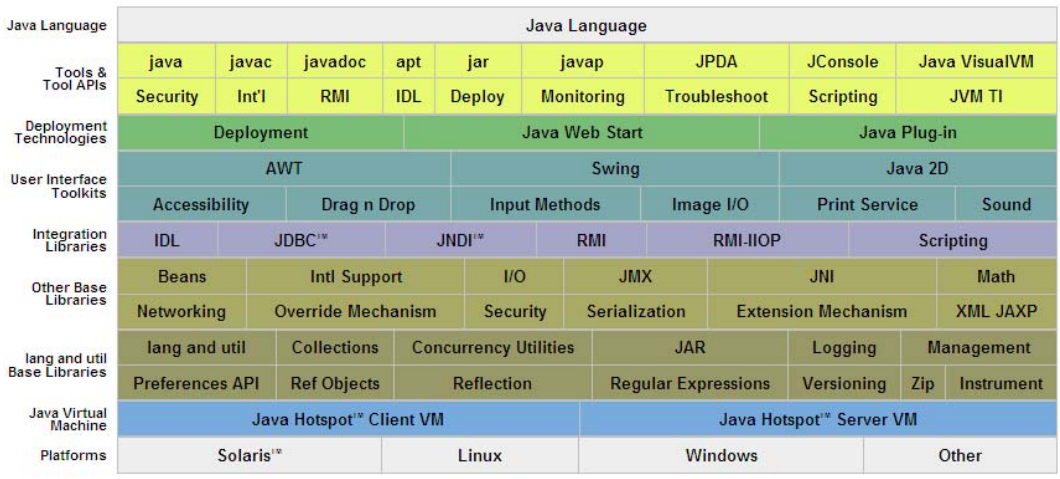
JDBC — Java Database Connectivity - warstwa abstrakcji baz danych

IDL — umożliwia tworzenie aplikacji rozproszonych w środowisku CORBA

JNI — Java Native Interface - interfejs umożliwiający korzystanie z funkcji natywnych dla danego systemu operacyjnego z poziomu aplikacji Java

XML JAXP — umożliwia przetwarzanie dokumentów XML

Wymienione elementy stanowią zaledwie ułamek możliwości wbudowanych w język Java, spora ich część przedstawiona została na rysunku 4, jednak to nie bogactwo bibliotek było powodem wyboru tejże platformy do realizacji zadania. Tworzona konsola sterownicza systemu MRROC++ wykorzystuje tylko podstawowe biblioteki oferowane przez Javę, w pełni czerpie za to z przenośności platformy, gdyż właśnie przenośność była jednym z wymagań stawianych aplikacji.



Rysunek 4: Schemat platformy Java SE w wersji 6

3.2 Biblioteka Java Swing

Biblioteka Swing [8] wchodzi w skład Java Foundation Classes (JFC), zbioru komponentów do tworzenia graficznego interfejsu użytkownika. Podstawowymi elementami

tej biblioteki są różnego rodzaju kontrolki, od najprostszych przycisków, poprzez pola tekstowe, na rozwijanych drzewach, listach i tabelach kończąc. Kontrolki te służą do budowy interaktywnego interfejsu graficznego. Oprócz biblioteki Swing wskład JFC wchodzi również pakiet Drag'n'Drop, udostępniający mechanizmy "przeciągnij i upuść", pakiet Accessibility, zawierający ułatwienia dla osób niedowidzących i niedosłyszących, oraz pakiet Java2D, umożliwiający tworzenie grafiki dwuwymiarowej.

Pierwotnie do tworzenia interfejsów graficznych Java korzystała z biblioteki AWT[11], zawierających tzw. komponenty ciężkie, które każde zdarzenie tłumaczyły na odpowiednie wywołanie systemu operacyjnego. Swing jest następcą biblioteki AWT, w przeciwieństwie jednak do niej stworzony został w całości w języku Java, dzięki czemu jest niezależny od platformy sprzętowej. Dodatkowo do rysowania interfejsu wykorzystywana jest biblioteka Java 2D, a nie funkcje systemu operacyjnego, czego skutkiem jest prawie identyczny (z dokładnością do kolorystyki i stylu) wygląd aplikacji pod różnymi systemami operacyjnymi.

Komunikacja pomiędzy interfejsem graficznym a aplikacją polega na obsłudze asynchronicznie wywoływanych zdarzeń. Zdarzenia mające źródło w interfejsie użytkownika (np. wciśnięcie przycisku) obsługiwane są po kolei przez pojedynczy wątek obsługi zdarzeń EDT[12], który przekazuje zdarzenie do metody actionPerformed() obiektów skojarzonych ze zdarzeniem, implementujących interfejs ActionListener, umożliwiający obsługę zdarzeń. Konsekwencją jednowątkowej obsługi zdarzeń jest ich kolejkowanie, dopóki wątek EDT nie zakończy obsługi poprzedniego zdarzenia. Rodzi to niebezpieczeństwo zakleszczeń oraz zawieszenia interfejsu w przypadku realizacji czasochłonnych operacji w wątku EDT podczas obsługi zdarzenia.

Projekt Java Swing zakładał wstępnie realizację biblioteki w architekturze Model-Widok-Kontroler:

Model przechowuje dane definiujące komponent

Widok tworzy graficzną reprezentacje komponentu

Kontroler obsługuje interakcję z użytkownikiem i w razie potrzeby modyfikuje model lub widok w odpowiedzi na akcję użytkownika

Zależności pomiędzy widokiem i kontrolerem były jednak zbyt duże, gdyż kontroler w dużym stopniu korzysta z implementacji widoku, dlatego też w rzeczywistości model MVC degenerował w model Dokument-Widok.

Oddzielenie widoku od modelu spowodowało, że Java Swing posiada jeszcze jedną, bardzo przydatną cechę - umożliwia zmianę wyglądu interfejsu bez modyfikacji kodu nim sterującego. Dlatego też możliwe jest definiowanie własnych schematów wyglądu

[11]Abstract Window Toolkit

[12]Event Dispatch Thread

aplikacji (tzw. look and feel). Co więcej, aplikacja uruchomiona pod kontrolą różnych systemów operacyjnych może wykorzystać schemat wyglądu tych właśnie systemów, dzięki czemu jej wygląd przypominać będzie natywne aplikacje danego systemu operacyjnego.

3.3 Applet języka Java

Applet języka Java to niewielki program osadzany w kodzie strony WWW i uruchamiany w kontekście przeglądarki internetowej. Pobierany jest on z serwera w postaci kodu pośredniego, a następnie wykonywany przez maszynę wirtualną Javy. Dystrybucja appletu w postaci kodu pośredniego zapewnia przenośność pomiędzy platformami oferowaną przez język Java.

Applet w kodzie HTML osadzić przy pomocy następującego prostego kodu:

```
<applet
    code="MrrocppUI"
    archive="JavaUISigned.jar">
</applet>
```

Applety Javy stosowane są w celu dodania dynamiki do statycznej strony HTML, zawierającej niezmienną treść, identyczną dla każdego użytkownika. Applety umożliwiają zmianę zawartości strony w zależności od akcji podejmowanych przez użytkownika, pozwalając nie tylko na czytanie treści strony, ale również na dwukierunkową komunikację pomiędzy serwerem, a użytkownikiem. Alternatywą dla appletu są strony wykonane przy wykorzystaniu DHTML, technologii Flash lub Silverlight, jednak ustępują one appletowi Javy pod względem oferowanych funkcji.

Maszyna wirtualna Javy uruchamia applety z innymi uprawnieniami niż samodzielne aplikacje. Projektanci JVM założyli, że użytkownik instalujący i uruchamiający samodzielną aplikację Javy bierze na siebie odpowiedzialność za ewentualne konsekwencje. Dlatego też takie aplikacje mają domyślnie przyznany dostęp do wszystkich zasobów komputera.

W przeciwieństwie do samodzielnych aplikacji, applety Javy pobierane są z internetu i uruchamiane automatycznie przez przeglądarkę. W celu zapewnienia bezpieczeństwa wykonywania kodu pobranego z internetu, przeglądarki internetowe uruchamiają applet w bardzo ograniczonym środowisku (tzw. sandbox), przez co ma on dostęp tylko do wybranych zasobów. Pobrany z internetu applet nie ma dostępu do plików lokalnej maszyny czy schowka, nie jest również uprawniony do nawiązywania połączeń sieciowych z adresem innym niż strona, z której został pobrany.

Ze względu na trudność pisania bardziej złożonych aplikacji przy tak surowych ograniczeniach, dopuszcza się rozszerzenie uprawnień appletu poprzez jego cyfrowe podpisanie. Przed uruchomieniem takiego appletu podpis zostanie zweryfikowany, a

użytkownik zapytany o to, czy dostawca appletu jest dla niego zaufaną stroną. Jeżeli użytkownik ufa dostawcy, to applet zyska możliwość dostępu do dodatkowych zasobów komputera, zdefiniowanych w pliku java.policy.

Realizacja graficznej konsoli sterowniczej jako appletu ma szereg zalet. Najważniejszą jest to, że applet spełnia stawiane aplikacji wymaganie łatwej dystrybucji. Dystrybucja jest ułatwiona, gdyż aplikację wystarczy pobrać z serwera korzystając z przeglądarki internetowej, w którą wyposażony jest praktycznie każdy system operacyjny. Applet nie wymaga instalacji, więc w razie potrzeby użytkownik bez większego wysiłku może uruchomić graficzną konsolę sterowniczą systemu MRROC++ na dowolnym komputerze podłączonym do internetu.

Taki sposób dystrybucji rodzi również pytanie, czy konieczność pobierania aplikacji przed uruchomieniem nie powoduje zbędnego ruchu sieciowego. W przypadku appletu jednak nie jest to problemem, gdyż po pobraniu przechowywany jest on w pamięci podręcznej przeglądarki, a ponowne pobranie następuje dopiero w przypadku pojawienia się na serwerze nowszej wersji appletu. Tu również objawia się kolejna cecha appletu - łatwość aktualizacji. Dzięki scentralizowanej instalacji aktualizację konsoli wystarczy przeprowadzić tylko w jednym miejscu, a aktualna wersja zostanie pobrana przez użytkowników w momencie próby uruchomienia w przeglądarce starszej wersji.

3.4 Protokół TCP/IP

Aby graficzna konsola sterownicza spełniała stawiane jej wymaganie możliwości pracy zdalnej w systemie, zdecydowałem się na realizację aplikacji w architekturze klient-serwer, konsekwencją czego była realizacja komunikacji za pomocą protokołów sieciowych. Do tego celu wybrana została para protokołów TCP/IP [9], wykorzystywana typowo do komunikacji w sieci internet. Niewątpliwą zaletą TCP/IP jest fakt, iż protokół ten jest dostępny w standardowych instalacjach większości systemów operacyjnych. Istnieją również implementacje tego protokołu dla większości języków programowania, co umożliwia niemal nieograniczone jego stosowanie.

Protokół IP[13] to protokół warstwy sieciowej modelu ISO/OSI, odpowiadający za dostarczenie pakietów od nadawcy do odbiorcy bazując jedynie na ich adresach. Do adresowania wykorzystywany jest obecnie najczęściej protokół IPv4, ale z powodu na wyczerpującą się pulę adresów coraz częściej do użycia wchodzi protokół IPv6.

Protokół IP jest protokołem zawodnym - nie daje żadnych gwarancji poprawnego dostarczenia pakietu. W dostarczonych pakietach mogą znajdować się błędy, część pakietów może się powtarzać, a części może brakować, a kolejność ich dostarczania jest całkowicie niezależna od kolejności nadawania. Co więcej, protokół nie zapewnia żadnej sygnalizacji wystąpienia wyżej wymienionych błędów, dlatego też kontrolą poprawności

[13]Internet Protocol

transmisji muszą zajmować się protokoły warstwy wyższej.

Protokół TCP[14] to protokół warstwy transportowej modelu ISO/OSI i to na nim spoczywa odpowiedzialność za poprawność dostarczenia danych. Do najważniejszych cech tego protokołu można zaliczyć:

- dostarczanie datagramów do odbiorcy w takiej samej kolejności, w jakiej zostały nadane, poprzez buforowanie i ułożenie pakietów w odpowiedniej kolejnośći przed przekazaniem ich do warstwy wyższej
- gwarancja dostarczenia datagramu bez błędów, realizowane poprzez retransmisję błędnych lub zaginionych pakietów, a w najgorszym wypadku informacji zwrotnej o niemożności dostarczenia pakietu

Gwarancja dostarczenia pakietów była tym, co zdecydowało o wyborze tej pary protokołów do realizacji komunikacji pomiędzy klientem i serwerem. Specyfika komunikacji z systemem MRROC++ wymagała bowiem wysyłania przez serwer potwierdzeń wykonania ruchu, a błędy w ich dostarczeniu skutkowałyby niepoprawnym działaniem aplikacji.

3.5 Protokół Rsh

Do sterowania systemem MRROC++, oprócz graficznej konsoli klienckiej, niezbędny jest również serwer działający po stronie systemu QNX. Serwer nie zawsze jest uruchomiony, dlatego pojawiła się potrzeba zdalnego powołania procesu serwera. Do tego zadania wybrany został protokół Rsh.

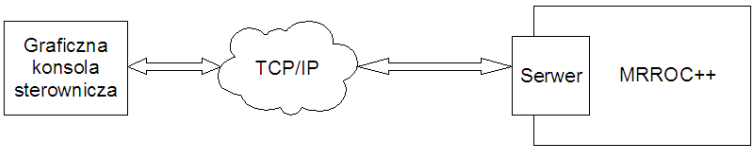
Rsh[15] to protokół wywodzący się z systemów BSD, wchodzący w skład pakietu rlogin. Służy do wykonywania poleceń powłoki systemowej na innej maszynie. Na zdalnej maszynie musi w tym celu zostać uruchomiony demon rshd, który nasłuchuje na porcie 514, a po nawiązaniu połączenia wykonuje otrzymane polecenie.

Implementację protokołu Rsh w Javie dostarczają biblioteki wchodzące w skład projektu Apache Commons. Protokół rsh dostarczany jest w pakiecie Net, który oprócz rsh, dostarcza on implementację większości podstawowych protokołów sieciowych, m.in. FTP, SMTP, POP3 oraz Telnet.

[14]Transport Control Protocol
[15]Remote Shell

4 Realizacja aplikacji

Wymaganą możliwość pracy zdalnej w systemie MRROC++ oferuje realizacja graficznej konsoli sterowniczej w architekturze klient-serwer (Rysunek 5). Realizowane przez aplikację funkcje zostały rozdzielone pomiędzy graficzną konsolę, uruchamianą po stronie klienta, oraz ściśle zintegrowany z systemem MRROC++ proces serwera.



Rysunek 5: Ogólna struktura systemu

Graficzna konsola, napisana w oparciu o technologię Java, umożliwia wydawanie podstawowych poleceń dla systemu MRROC++, rozkazów ruchów dla robotów oraz sterowanie przebiegiem wykonania zadania. Dodatkowo wyświetla informacje o stanie systemu i otrzymywane z serwera komunikaty. Pod względem oferowanych funkcji nie różni się ona od pierwotnej konsoli - z założenia bowiem miała ona być możliwie zbliżona do pierwowzoru.

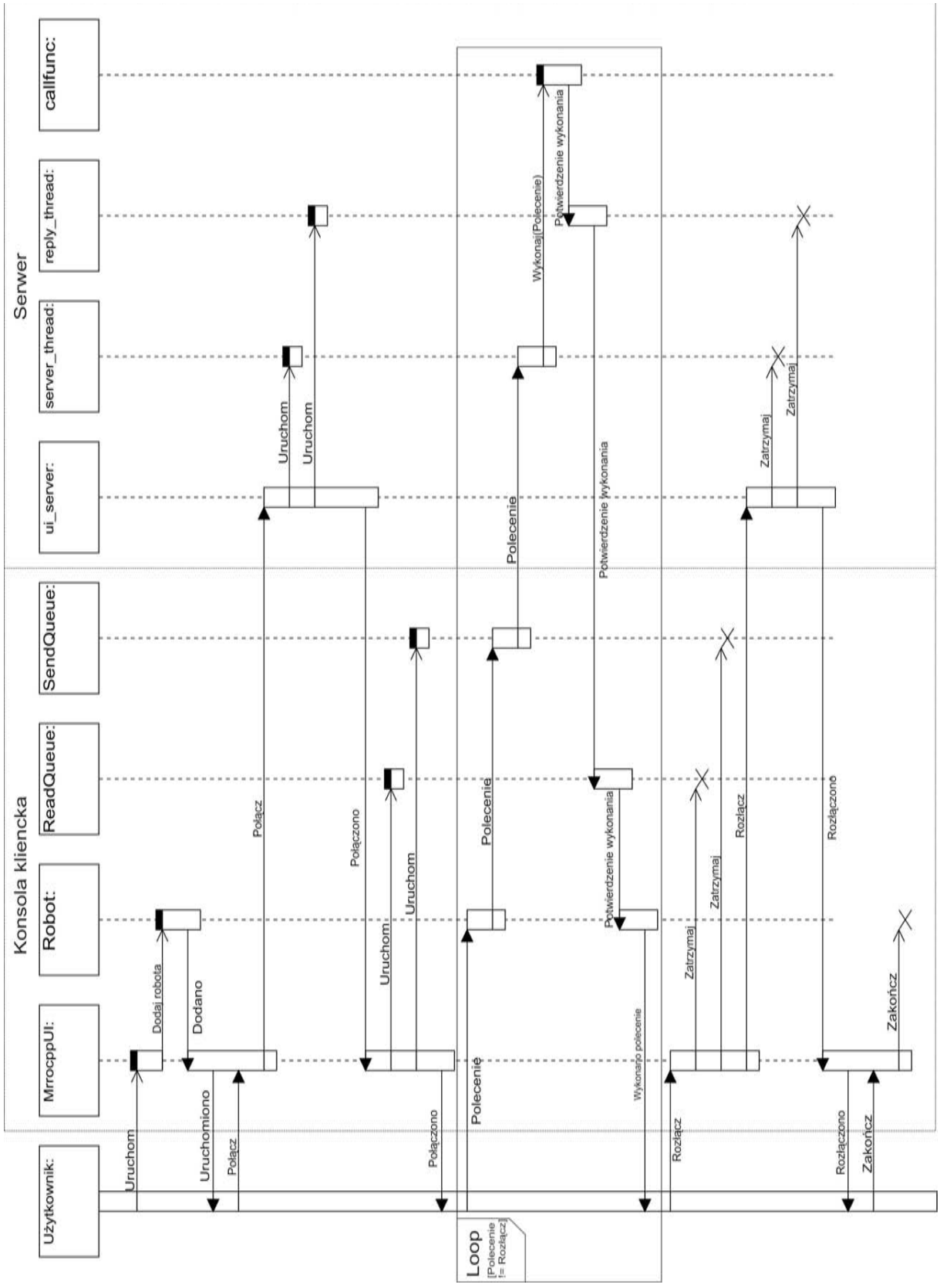
Serwer, działający jako jeden z procesów po stronie systemu MRROC++, napisany został w języku C++. Przyjmuje i wykonuje polecenia z aplikacji klienckiej, udostępnia też informację o stanie systemu i rozsyła komunikaty systemowe. Serwer komunikuje się z pojedynczą aplikacją kliencką, dlatego też próby nawiązania połączenia przez kolejne aplikacje są odrzucane.

Komunikacja pomiędzy klientem a serwerem odbywa się przy pomocy pary protokołów TCP/IP przy użyciu dedykowanej do tego struktury danych.

Scenariusz pracy z systemem MRROC++ przy pomocy graficznej konsoli sterowniczej przedstawiono na rysunku 6. Wszystkie przedstawione na nim elementy systemu zostaną szczegółowo przedstawione w tym rozdziale.

4.1 Identyfikatory operacji

W rozdziale 4.4 przedstawiony zostanie opracowany na potrzeby aplikacji protokół komunikacyjny służący do przesyłania informacji pomiędzy klientem i serwerem. Podstawą jego działania są trzy typy wyliczeniowe, zdefiniowane i używane przez większość klas aplikacji. Zakres ich użycia spowodował konieczność przedstawienia ich już teraz. Część z tych typów została zdefiniowana kilkukrotnie, bowiem w zależności od kontekstu zbiór dopuszczalnych wartości może być różny. W celu określenia robota będącego nadawcą lub odbiorcą komunikatu oraz identyfikacji funkcji, której wykonanie zlecił



Rysunek 6: Scenariusz pracy z systemem MRROC++

użytkownik, przesyłane są wartości zmiennych tych trzech typów.

Wspomniane typy wyliczeniowe to:

RobotId — identyfikator robota, zdefiniowany tylko w klasie MrrocppUI, określa robota, w którego kontekście nastąpiło zdarzenie.

DialogId — identyfikator okna dialogowego, zdefiniowany w klasie MrrocppUI obejmuje trzy okna dialogowe służące do sterowania systemem MRROCP++, zdefiniowany w klasach reprezentujących roboty obejmuje wszystkie okna dialogowe służące do sterowania danym robotem. Służy do identyfikacji okna dialogowego, zawierającego komponent, który wygenerował zdarzenie.

ActionId — identyfikator akcji, zdefiniowany w każdej klasie reprezentującej okno dialogowe, jak również w klasie MrrocppUI. Służy do określenia przycisku lub pozycji w menu, którego wciśnięcie wygenerowało zdarzenie.

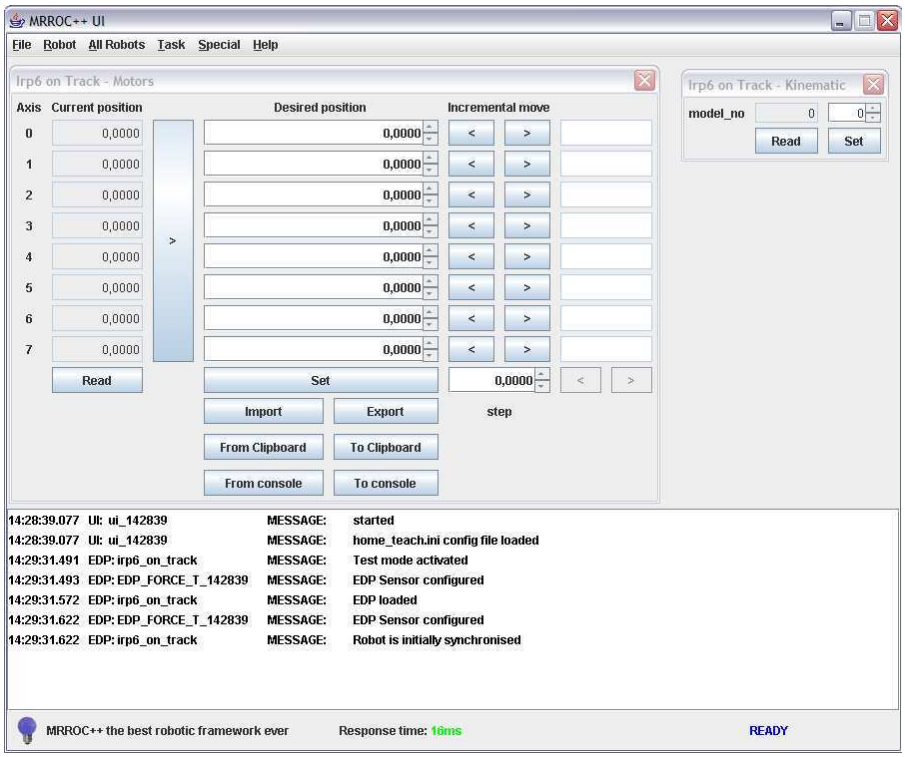
4.2 Klient

Kliencką część zrealizowanej aplikacji stanowi napisana w języku Java graficzna konsola sterownicza. Interfejs graficzny konsoli zaprojektowany został tak, by możliwie dokładnie odwzorowywał wygląd pierwotnej konsoli stworzonej w Photon MicroGUI. Wykorzystana do budowy interfejsu biblioteka Java Swing zawiera podobny zestaw kontrolek, jaki znaleźć można w Photon MicroGUI, udało się więc uzyskać wierne odwzorowanie dotychczasowego interfejsu. Na rysunku 7 przedstawiono nową wersję konsoli w stanie identycznym jak prezentowana na rysunku 3 pierwotna konsola. Funkcje oferowane przez zrealizowaną przeze mnie konsolę w pełni pokrywają te oferowane przez jej pierwowzór. Dodatkowo wprowadzone zostało kilka usprawnień. Oprócz eksportu pozycji robota do konsoli tekstowej, nowa wersja umożliwia eksport współrzędnych do schowka systemowego oraz zapamiętanie pozycji robota w programie, w celu późniejszego wybrania jej z listy i wczytania.

4.2.1 Struktura aplikacji

Strukturę aplikacji klienckiej przedstawiono na diagramie 8. Trzonem aplikacji jest obiekt klasy MrrocppUI, reprezentującej główne okno graficznej konsoli sterowniczej. Klasa ta realizuje dwukierunkową komunikację z systemem MRROC++, umożliwia również wydawanie podstawowych poleceń dla systemu.

Gamę funkcji oferowanych przez klasę MrrocppUI rozszerza szereg okien dialogowych umożliwiających wczytywanie konfiguracji czy sterowanie przebiegiem wykonania zadania. Okna dialogowe wraz z klasą MrrocppUI zapewniają realizację całości funkcji graficznej konsoli sterowniczej, za wyjątkiem rozkazów związanych ze sterowaniem pojedynczymi robotami.



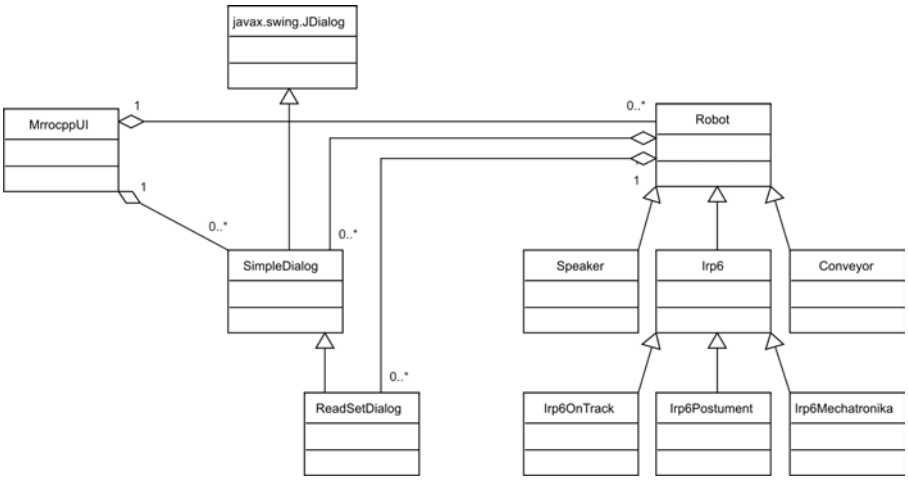
Rysunek 7: Graficzna konsola sterownicza systemu MRROC++ wykonana w Java Swing

Ostatnim elementem aplikacji są klasy reprezentujące roboty obsługiwane przez system MRROC++. Zapewniają one możliwość ręcznego sterowania robotem, udostępniają również informację o jego aktualnym stanie.

4.2.2 Klasa MrrocppUI

Graficzna konsola sterownicza została zrealizowana jako applet języka Java, dlatego też klasa MrrocppUI (Rysunek 9) dziedziczy po klasie javax.swing.JApplet. Jednym z zadań tej klasy jest stworzenie głównego okna aplikcji, dlatego też obsługuje ona zdarzenia wywoływane przez komponenty wchodzące wskład tego okna, co wymusiło implementację przez tę klasę interfejsu ActionListener.

Graficzna konsola sterownicza systemu MRROC++ rozpoczyna swoje działanie od statycznej funkcji MrrocppUI::main(), która tworzy instancję obiektu klasy MrrocppUI. Konstruktor klasy MrrocppUI w pierwszym kroku tworzy obiekty reprezentujące obsługiwane przez system MRROC++ roboty, a następnie wywoływana jest funkcja createGUI, tworząca i wyświetlająca główne okno aplikacji (rysunek 7). Dalsze działanie programu obejmuje obsługę zdarzeń wygenerowanych przez użytkownika, jak i obsługę komunikatów otrzymanych z systemu.



Rysunek 8: Struktura aplikacji klienckiej

W górnej części głównego okna znajduje się menu. Zawiera ono podstawowe polecenia dla systemu MRROC++, obsługiwane właśnie przez klasę MrrocppUI. Na rysunku 10 przedstawiono hierarchię menu górnego. Część pozycji otwiera okno dialogowe, pozostałe wysyłają do serwera odpowiednie żądanie. W menu górnym znajduje się również podmenu Robot. Zawiera ono menu służące do sterowania poszczególnymi robotami. Zawartość tych menu udostępniana jest przez klasy reprezentujące obsługiwane roboty za pomocą funkcji Robot::getMenu().

W dolnej części okna głównego znajduje się konsola tekstowa. Służy ona do wyświetlania zarówno komunikatów otrzymywanych z serwera, jak i generowanych przez aplikację kliencką. Każdy komunikat posiada jeden z predefiniowanych typów, przypisane jest mu również źródło oraz moment generacji. Wszystkie te informacje wyświetlane są właśnie w konsoli tekstowej. Dostęp do konsoli realizowany jest przy użyciu funkcji MrrocppUI::LogMessage(). Funkcja ta jako argumenty przyjmuje treść oraz typ komunikatu. Obsługiwane typy zdefiniowane są w globalnym typie wyliczeniowym MessageType.

Poniżej konsoli znajduje się pasek statusu. Wyświetla on aktualny stan pracy serwera (Rysunek 11), którym może być jeden z poniższych stanów:

Disconnected — aplikacja kliencka nie jest połączona z serwerem

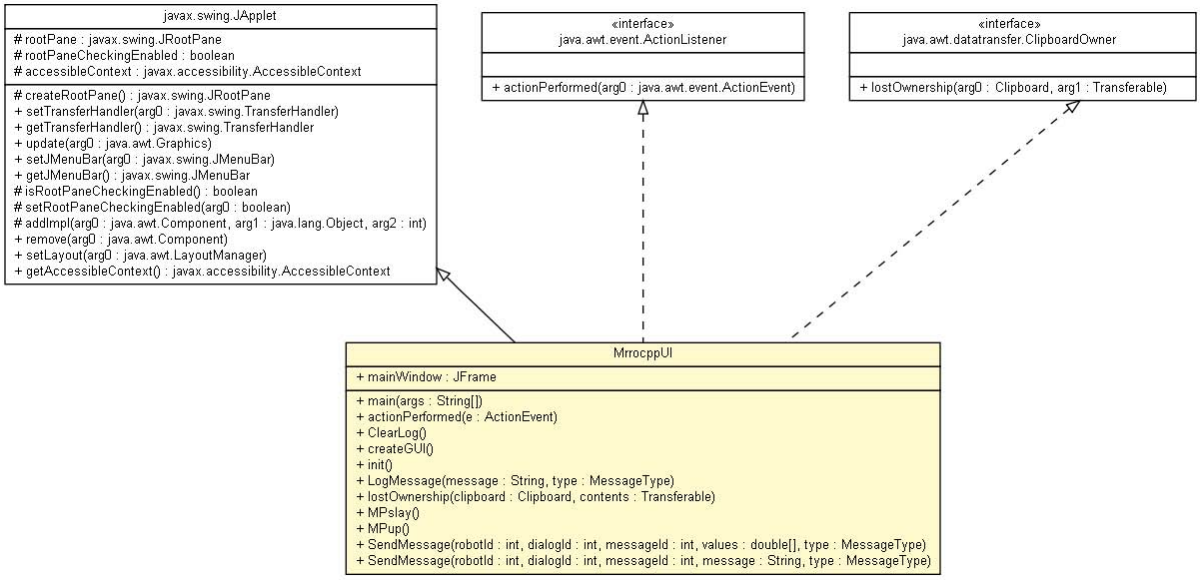
Starting — serwer jest w trakcie uruchamiania

Communication — komunikacja pomiędzy klientem i serwerem

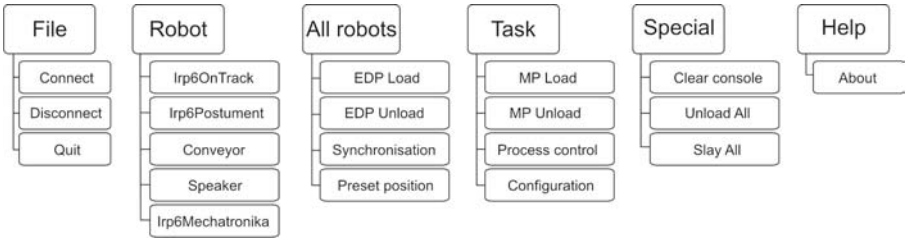
Ready — serwer jest gotowy do przyjęcia polecenia

Busy — serwer wykonuje polecenie użytkownika

Synchronisation — jeden z robotów jest synchronizowany



Rysunek 9: Klasa MrrocppUI



Rysunek 10: Hierarchia menu

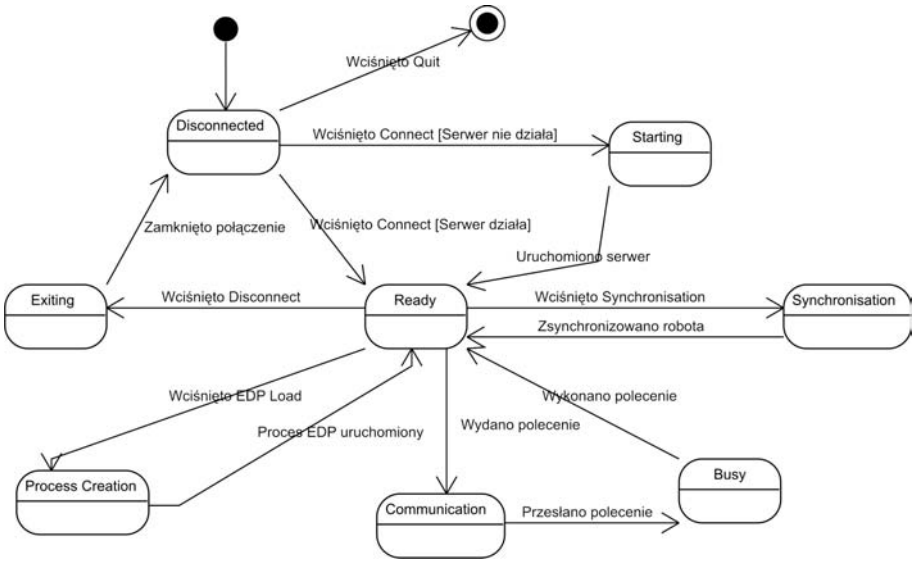
Process Creation — po strone serwera powoływany jest proces EDP

Exiting — zamykanie serwera

Na pasku statusu, podobnie jak w pierwotnej wersji konsoli sterowniczej, znajduje się również napis sławiący strukturę MRROC++ jako najlepszą strukturę ramową kiedykolwiek stworzoną.

Zdarzenia wygenerowane przez komponenty znajdujące się w głównym oknie aplikacji obsługiwane są przez klasę MrrocppUI w metodzie MrrocppUI::ActionPerformed(). Obsługa zdarzenia sprowadza się do wysłania odpowiedniego polecenia do serwera po stronie systemu MRROC++ przy użyciu funkcji *MrrocppUI::SendMessage()*. Komunikacja z serwerem opisana jest szerzej w rozdziale 4.4.

Klasa MrrocppUI w realizowaniu podstawowych funkcji systemu posiłkuje się trzema oknami dialogowymi reprezentowanymi przez klasy MrrocppUI::ConnectDialog, MrrocppUI::ConfigDialog oraz MrrocppUI::ControlDialog, opisanymi w dalszej części rozdziału. Klasa MrrocppUI wraz z tymi trzema klasami pokrywają całość możliwości



Rysunek 11: Diagram stanów aplikacji klienckiej

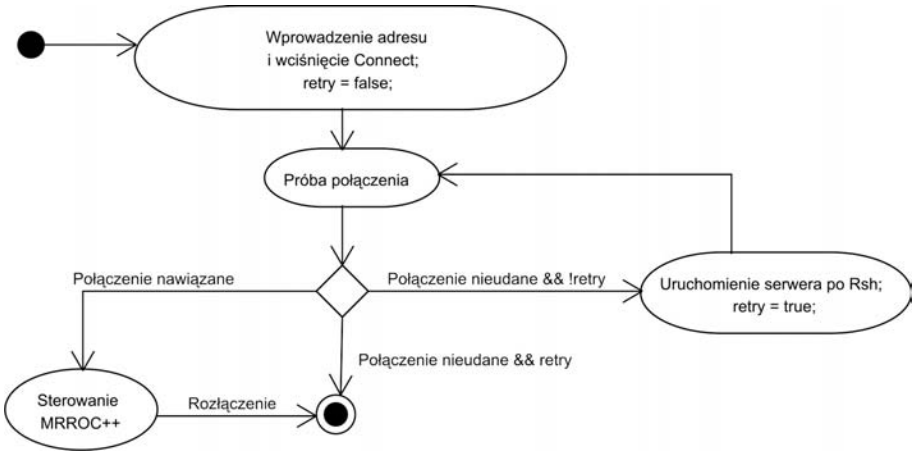
oferowanych przez konsolę sterowniczą za wyjątkiem sterowania pojedynczymi robotami.

MrrocppUI::ConnectDialog Klasa MrrocppUI::ConnectDialog odpowiada za nawiązanie połączenia z serwerem po stronie systemu MRROC++. Wyświetla ona okno dialogowe, umożliwiające wprowadzenie adresu maszyny, na której działa serwer, oraz ścieżkę do katalogu, w którym znajduje się plik wykonywalny serwera. Po wprowadzeniu parametrów połączenia aplikacja usiłuje nawiązać połączenie z serwerem przy pomocy gniazd TCP/IP. Możliwych jest kilka scenariuszy (Rysunek 12) nawiązywania połączenia z serwerem:

1. W przypadku poprawnego nawiązania połączenia uruchamiane są w osobnych wątkach funkcje odpowiadające za odbieranie i nadawanie komunikatów - odpowiednio MrrocppUI::ReadQueue::run() i MrrocppUI::SendQueue::run(). Klasy MrrocppUI::ReadQueue i MrrocppUI::SendQueue opisane są szerzej w rozdziale 4.4.

2. W przypadku, gdy nie uda się nawiązać połączenia, aplikacja zakłada, że nie został uruchomiony po stronie systemu MRROC++ serwer. Aplikacja spróbuje w takim wypadku uruchomić serwer, korzystając w tym celu z protokołu RExec, a następnie spróbuje ponownie nawiązać połączenie.

3. W przypadku nieudanej próby ponownego nawiązania połączenia, po uprzedniej próbie uruchomienia serwera na zdalnej maszynie, kolejna próba połączenia nie zostanie podjęta. Nieudane uruchomienie procesu serwera może być przede wszystkim spowodowane brakiem plików wykonywalnych.



Rysunek 12: Scenariusz nawiązywania połączenia z serwerem

Dostęp do okna dialogowego MrrocppUI::ConnectDialog jest możliwy po wybraniu opcji *Configuration* z menu *Task*.

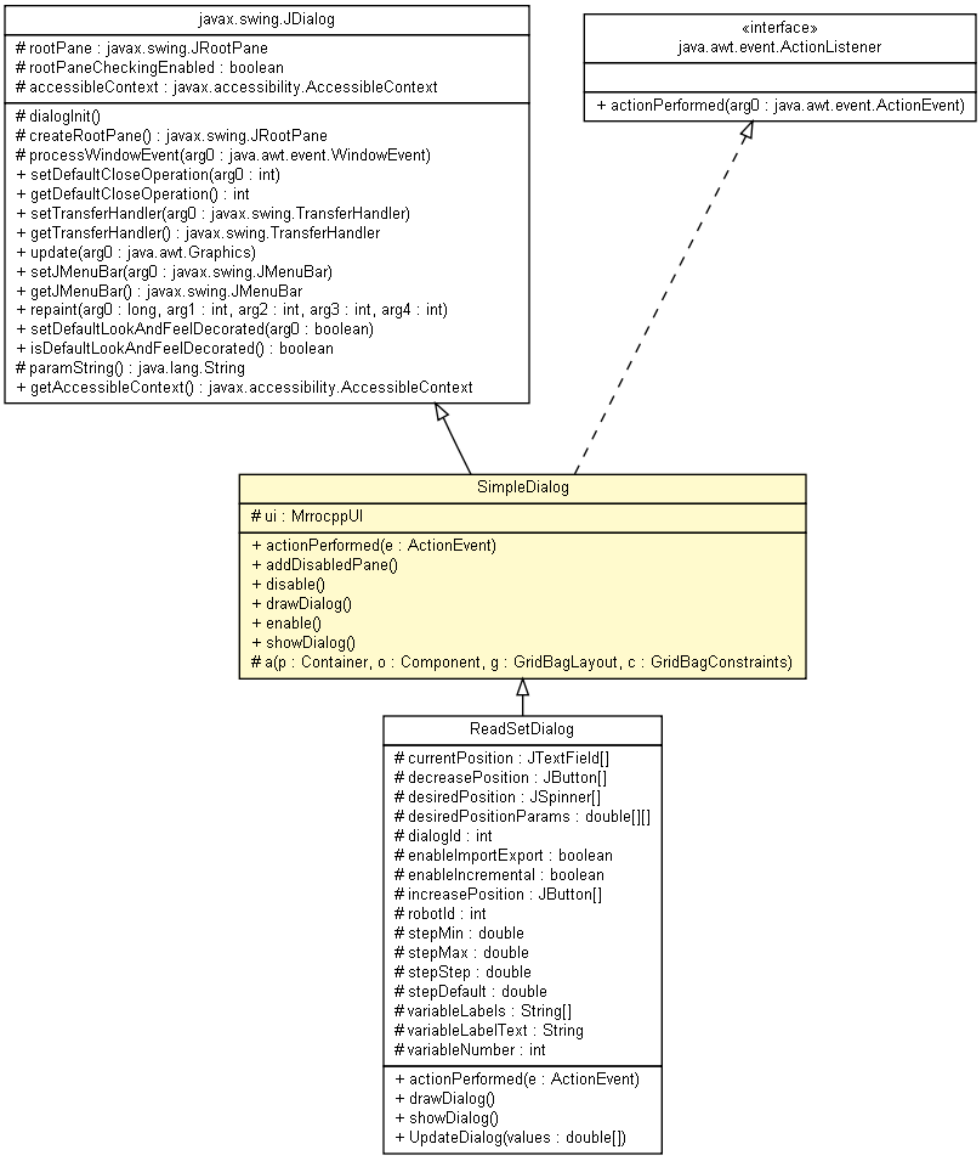
Klasa MrrocppUI::ConfigDialog Klasa ta umożliwia wybranie pliku konfiguracyjnego spośród dostępnych na serwerze. Konstruktor tej klasy wysyła do serwera polecenie przesłania zawartości katalogu *configs/* w katalogu domowym systemu MRROC++, a następnie wyświetla okno dialogowe, zawierające listę dostępnych plików konfiguracyjnych. W przypadku wyboru pliku przez użytkownika, jego nazwa przesyłana jest do serwera przy użyciu metody *MrrocppUI::SendMessage()*.

Dostęp do tego okna dialogowego jest możliwy po wybraniu opcji *Configuration* z menu *Task*.

Klasa MrrocppUI::ControlDialog Klasa ta reprezentuje okno sterowania wykonaniem zadania - dostępne po wybraniu opcji *Process Control* z menu *Task*. Umożliwia ona uruchamianie, zatrzymywanie i wznawianie procesu MP odpowiadającego za sterowanie przebiegiem zadania, oraz uruchamianie i zatrzymywanie procesów odpowiedzialnych za dokonywanie odczytów położeń poszczególnych robotów oraz sterowanie procesami ECP. Jak w całej aplikacji również i tutaj wysyłanie komunikatów odbywa się przy użyciu metody *MrrocppUI::SendMessage()*.

4.2.3 Okna dialogowe

Zdecydowana większość funkcji oferowanych przez graficzną konsolę sterowniczą, a w szczególności ruchy ręczne pojedynczymi robotami, realizowana jest przy pomocy okien dialogowych. W celu ułatwienia tworzenia nowych okien stworzone zostały dwie klasy przedstawione na rysunku 13, stanowiące podstawę do budowy okien dialogowych.



Rysunek 13: Klasy wykorzystywane do budowy okien dialogowych

Klasa SimpleDialog Klasa ta stanowi abstrakcyjną klasę bazową, po której dziedziczą wszystkie okna dialogowe używane przez aplikację. Skupia ona funkcje wspólną dla tych okien. Klasa ta dziedziczy po klasie *javax.swing.JDialog*. Implementuje ona interfejs *ActionListener*, może więc obsługiwać zdarzenia generowane przez komponenty które zawiera. Obiekty klasy *SimpleDialog* udostępniają następujące funkcje:

SimpleDialog(MrrocppUI, String) — Konstruktor klasy SimpleDialog. Jako argumenty przyjmuje referencje do obiektu klasy MrrocppUI, reprezentującej główne okno aplikacji, w którego kontekście wyświetlać będzie się okno dialogowe, oraz tytuł, który wyświetli się w nagłówku okna.

disable() — Umożliwia zablokowanie okna, wskutek czego niemożliwe będzie wydawanie systemowi poleceń. Z funkcji tej korzystają przede wszystkim okna służące do sterowania pojedynczymi robotami, ze względu na fakt, iż system MRROC++

nie dopuszcza równoległego wykonywania przez robota więcej niż jednego rozkazu.

enable() — Umożliwia odblokowanie okna, zablokowanego przy użyciu funkcji *disable()*

drawDialog() — Jest to abstrakcyjna funkcja, odpowiadająca za zbudowanie okna dialogowego. Implementację tej funkcji zapewniają konkretne klasy reprezentujące poszczególne okna dialogowe.

showDialog() — Powoduje wyświetlenie na ekranie okna dialogowego.

actionPerformed(ActionEvent) — Funkcja abstrakcyjna wchodząca wskład interfejsu ActionListener, służąca do obsługi zdarzeń generowanych przez komponenty należące do okna dialogowego. Jako argument przyjmuje referencję do obiektu klasy ActionEvent, reprezentującego zdarzenie i udostępniającego informację na temat źródła zdarzenia. Implementację tej metody zapewniają konkretne klasy reprezentujące poszczególne okna dialogowe.

Klasa ReadSetDialog Dziedzicząca po SimpleDialog klasa ReadSetDialog reprezentuje okna ruchów ręcznych pojedynczych robotów. Przykładowe okno typu ReadSetDialog przedstawiono na rysunku 14.

Rysunek 14: Przykładowe okno klasy ReadSetDialog

Okna typu ReadSetDialog umożliwiają między innymi:

- wyświetlanie aktualnej pozycji manipulatora
- wprowadzenie ręcznie żądanej pozycji
- krokowa zmiana wartości pojedynczych współrzędnych

- eksport i import pozycji manipulatora do i ze schowka systemowego
- zapamiętanie nazwanych zestawów współrzędnych i ich przywracanie
- wysłanie do serwera żądania odczytu pozycji
- zadanie nowej pozycji manipulatora

W skład klasy *ReadSetDialog* wchodzą następujące funkcje publiczne:

ReadSetDialog(MrrocppUI, String) — konstruktor klasy ReadSetDialog. Jako argumenty przyjmuje referencje do obiektu klasy MrrocppUI, reprezentującej główne okno aplikacji, w którego kontekście wyświetlać będzie się okno dialogowe, oraz tytuł, który wyświetli się w nagłówku okna.

showDialog() — powoduje wyświetlenie okna dialogowego. Dodatkowo przesyła do serwera zapytanie o aktualną pozycję manipulatora i przepisuje te współrzędne w pola zawierające aktualną i żądaną pozycję robota.

updateDialog(double []) — funkcja przyjmuje jako argument tablicę zawierającą poszczególne współrzędne robota i wpisuje je w odpowiednie pola.

Stworzenie nowego okna dialogowego tego typu sprowadza się do implementacji klasy dziedziczącej po klasie ReadSetDialog. Konstruktor nowego okna powinien ustawić odpowiednio wartości zmiennych wymienionych poniżej. Zostaną one wykorzystane do budowy okna dialogowego. Obsługa zdarzeń i przesyłanie poleceń do serwera zostało już zaimplementowane w klasie ReadSetDialog.

DialogId dialogId — identyfikator okna dialogowego. Typ DialogId opisany jest szerzej w rodziale 4.4.

String variableLabelText — etykieta kolumny zawierającej nazwy współrzędnych

int variableNumber — liczba współrzędnych

String [] variableLabels — nazwy współrzędnych

boolean enableImportExport — włącza/wyłącza import i eksport pozycji

boolean enableIncremental — włącza/wyłącza możliwość krokowej zmiany pojedynczych współrzędnych

double stepMin, double stepMax — minimalna i maksymalna wartość kroku

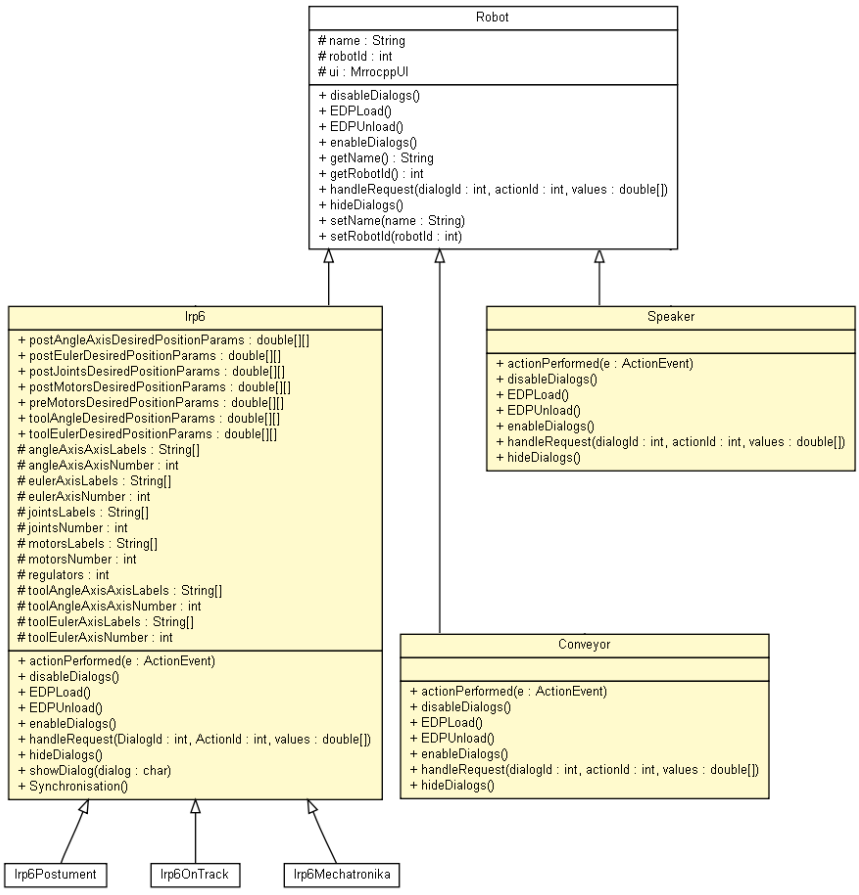
double stepDefault — domyślna, początkowa wartość kroku

double stepStep — wartość, o jaką zmieniać można krok

double[][][] desiredPositionParams — tablica zawierająca dziedziny poszczególnych współrzędnych, wartość domyślną oraz wartość kroku. Dla każdej współrzędnej w tablicy desiredPositionParams powinna znajdować się czteroelementowa tablica liczb typu *double*, odpowiadających odpowiednio domyślnej początkowej wartości współrzędnej, minimalnej wartości, maksymalnej wartości oraz wartości kroku, o jaki można zmieniać tę współrzędną.

4.2.4 Klasy robotów

Hierarchia klas reprezentujących roboty znajduje się na rysunku 15.



Rysunek 15: Hierarchia klas reprezentujących obsługiwane przez MRROC++ roboty

Graficzna konsola sterownicza systemu MRROC++ stworzona została z myślą o przyszłym rozszerzeniu gamy obsługiwanych robotów. Klasy reprezentujące obsługiwane roboty muszą dziedziczyć po abstrakcyjnej klasie *Robot*. Interfejs klasy *Robot* zawiera następujące funkcje:

Robot(MrrocppUI, String) — konstruktor klasy Robot. Jako argumenty przyjmuje referencje do obiektu reprezentującego główne okno aplikacji i nazwę robota.

getMenu() — abstrakcyjna metoda, zwracająca obiekt klasy javax.swing.JMenu, reprezentujący menu robota. Menu to zostanie dodane do menu Robot.

getName() — zwraca nazwę robota.

getRobotId() — zwraca identyfikator robota.

hideDialogs() — powoduje zamknięcie wszystkich okien służących do sterowania danym robotem.

EDPLoad() — obsługuje uruchomienie procesu EDP wybranego robota po stronie systemu MRROC++. Funkcja ta powinna przede wszystkim odblokować menu robota.

EDPUnload() — obsługuje zakończenie działania procesu EDP wybranego robota po stronie systemu MRROC++.

disableDialogs() — blokuje menu i wszystkie okna służące do sterowania danym robotem. Uniemożliwia to wydawanie robotowi jakichkolwiek poleceń. Najczęściej metoda ta stosowana jest po wydaniu robotowi rozkazu, a przed otrzymaniem z serwera potwierdzenia jego wykonania. Ma to zapobiec próbie wykonania równolegle dwóch lub więcej rozkazów przez pojedynczego robota.

enableDialogs() — odblokowuje menu i okna dialogowe danego robota.

handleRequest(DialogId, ActionId, double[]) — abstrakcyjna metoda obsługująca komunikaty otrzymane z serwera, a skierowane do danego robota. Typy wyliczeniowe *DialogId* i *ActionId* opisane są szerzej w rozdziale 4.4.

Najważniejszą funkcją, której implementację dostarczyć mają konkretne klasy reprezentujące rzeczywiste roboty, jest funkcja *handleRequest()*. Jako argumenty przyjmuje identyfikatory okna dialogowego i akcji oraz tablicę liczb rzeczywistych. Obsługa komunikatu otrzymanego z serwera realizowana jest w całości przez funkcję handleRequest(), a główna aplikacja jedynie pośredniczy w przekazaniu komunikatu. Dlatego też dodanie obsługi nowego robota nie wymaga ingerencji w kod aplikacji, a jedynie implementacji tejże funkcji.

W czasie pisania pracy system MRROC++ przystosowany był do sterowania pięcioma typami robotów. Trzy z nich to zmodyfikowane manipulatory Irp-6, dodatkowo obsługiwany jest pas transmisyjny (Conveyor) oraz Speaker. W aplikacji roboty te reprezentowane są przez klasy dziedziczące po klasie Robot, opisane w dalszej części rozdziału.

Klasa Irp6 Klasa ta dziedziczy po klasie Robot, sama zaś stanowi klasę bazową dla zmodyfikowanych manipulatorów Irp-6. Klasy reprezentujące te zmodyfikowane wersje robotów różnią się przede wszystkim liczbą stopni swobody, mogą również dodawać funkcje specyficzne dla danego typu manipulatora. Zmodyfikowane manipulatory Irp6 reprezentowane są przez klasy Irp6OnTrack, Irp6Postument i Irp6Mechatronika.

Oprócz zapewnienia implementacji abstrakcyjnych metod klasy Robot, klasa Irp6 definiuje dziewięć okien dialogowych, służących do sterowania pojedynczym robotem. Siedem z nich to okna typu ReadSetDialog, pozostałe to proste okna dialogowe dziedziczące po klasie SimpleDialog. Wszystkie zdefiniowane okna zostały przedstawione poniżej:

PostAngleAxisDialog — sterowanie manipulatorem za pomocą współrzędnych w układzie oś-kąt

PostEulerDialog — sterowanie manipulatorem za pomocą współrzędnych Eulera

PostJointsDialog — sterowanie manipulatorem za pomocą bezwzględnych położeń stawów po synchronizacji

PostMotorsDialog — sterowanie manipulatorem za pomocą bezwzględnych położeń silników po synchronizacji

PreMotorsDialog — sterowanie manipulatorem za pomocą względnych położeń silników przed synchronizacją

ToolAngleDialog — zadawanie pozycji narzędzia we współrzędnych oś-kąt

ToolEulerDialog — zadawanie pozycji narzędzia we współrzędnych Eulera

KinematicDialog — służy do podania numeru modelu kinematyki, który ma być używany

ServoAlgorithmDialog — wybór algorytmu serworegulacji

Klasa Conveyor Klasa ta reprezentuje pas transmisyjny. Dostarcza ona implementacje abstrakcyjnych metod klasy Robot. Dodatkowo definiuje dwa okna dialogowe, dziedziczące po klasie SimpleDialog:

MoveDialog — służy do zadawania pozycji pasa transmisyjnego

ServoAlgorithmDialog — wybór algorytmu

Klasa Speaker Klasa ta reprezentuje syntezator mowy. Dostarcza ona implementacje abstrakcyjnych metod klasy Robot. Dodatkowo definiuje okno dialogowe PlayDialog, pozwalające na wprowadzenie tekstu i przesłanie go do urządzenia.

4.3 Serwer

Napisany w C++ serwer, działający po stronie systemu MRROC++, odbiera komunikaty z konsoli graficznej i wykonuje zawarte w nich polecenia. Wysyła również do konsoli klienckiej potwierdzenia wykonania polecenia oraz komunikaty otrzymane z serwera, dotyczące zmiany jego stanu i przebiegu wykonania zadania.

Zaimplementowany serwer posiada niebanalną cechę, która sama w sobie czyni zrealizowaną aplikację lepszą od pierwowzoru. Serwer umożliwia mianowicie równoległe wykonywanie poleceń na poszczególnych robotach. Dotychczasowa konsola sterownicza nie przyjmowała żadnych rozkazów ruchu, jeżeli jakiś rozkaz już był realizowany przez dowolny manipulator. W nowej wersji konsoli ograniczenie to dotyczy tylko pojedynczych robotów - które faktycznie mogą wykonywać w danej chwili tylko jeden rozkaz. Nie ma natomiast nieuzasadnionego ograniczenia równoległego działania kilku robotów.

Źródła serwera znajdują się w pliku *ui_ init.cc*. Prototypy funkcji oraz wykorzystywane struktury danych zawarte są z kolei w pliku *gcc_ ntox86/proto.h*.

Dostęp do funkcji systemu MRROC++, z których korzysta serwer, możliwy jest przy użyciu funkcji zawartych w pliku fun.cc. Zdefiniowano tu odpowiedniki funkcji, z których korzysta pierwotna konsola stworzona w Photon MicroGUI. zdefiniowanych również w pliku fun.cc w katalogu tejże konsoli. Funkcje te nie zawierają jednak kodu modyfikującego interfejs użytkownika, wzbogacone są za to o możliwość przesyłania potwierdzenia wykonania polecenia.

Funkcje każdego z robotów obsługiwanych przez system zdefiniowane są plikach fun_r_NAZWAROBOTA.cc. Są to odpowiedniki plików wykorzystywanych w pierwotnej konsoli, a różnice między nimi są podobne, jak dla funkcji zdefiniowanych w pliku fun.cc. Dodatkowo funkcje ruchu oprócz potwierdzenia wykonania rozkazu przesyłają nowe współrzędne manipulatora.

Funkcja init Działanie serwera rozpoczyna się wraz z wywołaniem funkcji *init*. Realizuje ona funkcje swojego odpowiednika w pierwotnej konsoli. Inicjalizuje system MRROC++, wczytuje wartości zmiennych globalnych systemu, uruchamia serwer gns i dodaje obsługę sygnałów systemowych. Uruchamia również w osobnych wątkach funkcje server_thread i comm_thread.

Ze względu na potrzebę synchronizacji dostępu do flag typu *int*, określających zdolność robota do wykonania kolejnego polecenia, funkcja *init* inicjalizuje służące do tego

semafory typu *sem_t*. Ponieważ wykorzystywane są one w kilku różnych funkcjach, zostały zrealizowane jako zmienne globalne. Aktualnie zdefiniowane semafory i flagi zebrano w tabeli 1.

Semafor	*Flaga*	*Znaczenie*
sem_ui	block_ui	Dostęp do funkcji przesyłających informację do interfejsu graficznego
sem_all	block_all	Dostęp do dowolnego robota
sem_mp	block_mp	Dostęp do procesu MP
sem_irp6_on_track	block_irp6_on_track	Dostęp do manipulatora Irp6OnTrack
sem_irp6_mechatronika	block_irp6_mechatronika	Dostęp do manipulatora Irp6Mechatronika
sem_irp6_postument	block_irp6_postument	Dostęp do manipulatora Irp6Postument
sem_conveyor	block_conveyor	Dostęp do manipulatora Conveyor
sem_speaker	block_speaker	Dostęp do robota Speaker

Tablica 1: Semafory i flagi używane do synchronizacji dostępu do robotów

Podczas projektowania aplikacji rozważane było rozwiązanie problemu dostępu do pojedynczych robotów poprzez kolejkowanie rozkazów, jednak ostatecznie zwyciężyła koncepcja ignorowania poleceń otrzymanych w trakcie wykonywania wcześniejszych rozkazów ruchu. Wykonanie części bowiem poleceń zajmuje sporo czasu, dopuszczenie więc do kolejkowania poleceń skutkowałoby jeszcze większym opóźnieniem pomiędzy nadaniem rozkazu a potwierdzeniem jego wykonania.

Blokada wykonania więcej niż jednego polecenia przez pojedynczego robota zaimplementowana jest po stronie konsoli klienckiej. Implementacja po stronie serwera stanowi jedynie dodatkowe zabezpieczenie.

Funkcja server_thread Funkcja ta, działająca w osobnym wątku, stanowi najważniejszą część serwera. Nasłuchuje ona na wybranym porcie na połączenia przychodzące z graficznej konsoli klienckiej. Po nawiązaniu połączenia sterowanie przekazywane jest do głównej pętli programu, w której funkcja ta czeka na przesłanie polecenia. Po otrzymaniu komunikatu od użytkownika, funkcja server_thread uruchamia w nowym wątku funkcję obsługi zdarzeń *callfunc*, do której przekazuje tenże komunikat, a następnie przechodzi w stan oczekiwania na kolejne polecenia.

Funkcja comm_thread Funkcja ta również działa w osobnym wątku. Podstawowym jej zadaniem jest oczekiwanie na komunikaty przychodzące od systemu MRROC++. Otrzymane polecenia przesyłane są do aplikacji klienckiej.

Obsługiwane przez funkcję comm_thread komunikaty to m.in. zapytania wysyłane przez procesy ECP i MP, sterujące realizacją zadania lokalnie przez pojedynczy manipulator oraz globalnie, przez cały system. Służą one do komunikacji z użytkownikiem aplikacji klienckiej, a bardzo często uzyskania od niego informacji niezbędnej do wykonania zadania. Obsługiwane typy poleceń, identyfikowane przez wartości typu wyliczeniowego *ECP_TO_UI_COMMAND*:

YES_NO — pytanie do użytkownika, oczekujące na decyzję typu Tak/Nie

MESSAGE — komunikat dla użytkownika

DOUBLE_NUMBER — pytanie o wartość typu double

INTEGER_NUMBER — pytanie o wartość typu int

CHOOSE_OPTION — prośba o wybór jednej z możliwych opcji

LOAD_FILE — pytanie o nazwę pliku do wczytania

SAVE_FILE — pytanie o nazwę pliku do zapytania

Pierwsze cztery polecenia zawierają treść pytania lub komunikatu, piąte zawiera dodatkowo listę opcji do wyboru. Dwa pozostałe wykorzystują komunikat predefiniowany po stronie klienta.

Funkcja callfunc Funkcja ta odpowiada za obsługę zdarzeń przychodzących od użytkownika. Wywoływana jest w osobnym wątku dla każdego otrzymanego z konsoli klienckiej polecenia. Jako argument przyjmuje treść komunikatu.

Obsługa zdarzenia realizowana jest w kilku krokach:

1. *Sprawdzenie typu komunikatu* — Jeżeli zawiera on informacje liczbowe, to funkcja przydziela pamięć na tablicę typu *double*, a następnie wypełnia ją otrzymanymi liczbami. Jeżeli zawiera on jedynie informację tekstową, to jest ona zachowywana bez zmian.

2. *Określenie obiektu docelowego polecenia* — Jest on identyfikowany przez zmienną RobotId. Adresatem polecenia może być pojedynczy robot lub system MRROC++.

3. *Sprawdzenie dostępności adresata* — Ze względu na niemożność równoległego wykonywania kilku poleceń przez pojedynczego robota, nie są realizowane polecenia otrzymane przed zakończeniem realizacji poprzedniego rozkazu ruchu.

4. *Wykonanie rozkazu w przypadku dostępności adresata* — Funkcja obsługująca rozkaz identyfikowana jest za pomocą wartości zmiennych DialogId i ActionId. Przed uruchomieniem funkcji ustawiana jest flaga blokująca dostęp do adresata. Po wykonaniu rozkazu flaga jest kasowana. Przekazywany do funkcji jest otrzymany od konsoli klienckiej komunikat tekstowy lub tablica liczb.

5. *Pominięcie rozkazu w przypadku zajętości adresata* — Funkcja kończy swoje działanie, nie wywołując żadnej funkcji.

4.4 Komunikacja

Komunikacja pomiędzy aplikacją kliencką, a serwerem działającym po stronie systemu MRROC++ odbywa się dwukierunkowo. Informacja przesyłana przez graficzną konsolę sterowniczą do serwera obejmuje:

- polecenia dla systemu MRROC++, np. uruchomienie procesu MP, zabicie procesów sterujących robotami czy wczytanie pliku konfiguracyjnego
- rozkazy ruchu dla pojedynczych robotów, zawierające docelowe współrzędne
- polecenia dla poszczególnych robotów, np. żądanie synchronizacji
- żądanie przesłania stanu robota, m.in. jego aktualnych współrzędnych

Serwer do klienta może przesyłać:

- potwierdzenia wykonania operacji
- informację zwrotną w odpowiedzi na żądanie przesłania stanu robota
- komunikaty generowane przez działające procesy, wyświetlane później w konsoli tekstowej w dolnej części głównego okna
- komunikaty o błędach

Ważnym aspektem komunikacji między klientem a serwerem było rozpoznawanie zerwania połączenia lub nieprawidłowego działania jednej ze stron. W przypadku błędu krytycznego po stronie systemu MRROC++ konsola czekałaby w nieskończoność na potwierdzenie wykonania rozkazu. Dlatego też opisane w dalszej części rodziału klasy i funkcje implementują zabezpieczenia, pozwalające szybko stwierdzić zerwanie połączenia lub nieprawidłowe działanie jednej ze stron.

4.4.1 Klient

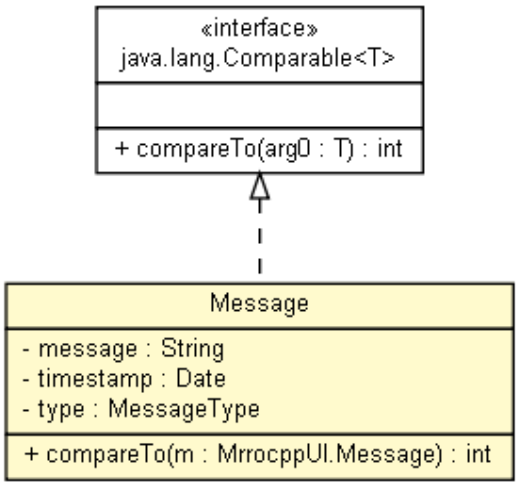
Po stronie graficznej konsoli sterowniczej wysyłanie i odbieranie komunikatów odbywa się za pośrednictwem działających w osobnych wątkach funkcji *run* klas *MrrocppUI::SendQueue* i *MrrocppUI::ReadQueue*. Przesyłane wiadomośći reprezentowane są w programie przez obiekty klasy *MrrocppUI::Message*

Klasa MrrocppUI::Message Klasa ta (Rysunek 16)reprezentuje komunikaty przesyłane z konsoli sterowniczej do serwera. Obiekty klasy *Message* posiadają następujące atrybuty:

type — rodzaj wiadomości - wartość typu wyliczeniowego *MessageType* — określająca priorytet wysyłania wiadomośći

message — treść wiadomości

timestamp — znacznik czasu, określający moment wygenerowania wiadomości



Rysunek 16: Klasa MrrocppUI::Message

Klasa *Message* implementuje interfejs *Comparable*, umożliwiający porównywanie i sortowanie obiektów tej klasy względem ich typu i czasu wygenerowania. Cecha ta pozwala na implementację kolejek priorytetowych przechowujących obiekty klasy *Message*. Tego rodzaju kolejka używana jest w programie do nadawania wiadomości w kolejności uwzględniającej ich priorytet. Priorytet ten określany jest przez typ komunikatu - jedną z wartości *FatalError*,*NonFatalError*,*SystemError*,*Message* lub *UnknownError* - a następnie przez czas generacji. Takie rozwiązanie gwarantuje sekwencyjne wykonywanie poleceń, umożliwia jednak natychmiastowe dostarczenie komunikatów o błędach istotnych dla działania systemu.

Funkcja MrrocppUI::SendMessage() Do wysyłania komunikatów - reprezentowanych przez obiekty klasy *Message* - służy przeładowana funkcja *MrrocppUI::SendMessage()*. Przyjmuje ona następujące argumenty:

RobotId — identyfikator systemu lub robota, który wygenerował komunikat

DialogId — identyfikator okna dialogowego, zawierającego komponent, który wygenerował zdarzenie będące źródłem komunikatu

ActionId — identyfikator akcji

string | double[] — ciąg znaków w przypadku komunikatu tekstowego lub tablica liczb rzeczywistych

MessageType — typ wiadomości

Funkcja ta tworzy nowy obiekt klasy *Message*, a następnie wstawia go do kolejki komunikatów do wysłania za pomocą funkcji *send()* klasy *MrrocppUI::SendQueue*.

Klasa MrrocppUI::SendQueue Klasa ta reprezentuje kolejkę priorytetową komunikatów do wysłania. Przechowuje ona komunikaty oraz wysyła je do serwera w kolejności wynikającej z priorytetu wiadomości.

Klasa *MrrocppUI::SendQueue* dziedziczy po klasie *java.lang.Thread*, dzięki czemu jej metoda *run()* może działać w osobnym wątku. Takie rozwiązanie pozwala na swobodne wysyłanie komunikatów nawet w przypadku wykonywania przez aplikację czasochłonnych operacji.

Ze względu na wspomnianą wcześniej konieczność kontroli poprawności działania połączenia pomiędzy klientem a serwerem, funkcja *run()* wysyła co ustalony interwał czasu do serwera pusty komunikat, pozwalający po stronie serwera stwierdzić, że konsola pracuje poprawnie.

Klasa MrrocppUI::ReadQueue Klasa ta obsługuje komunikaty przychodzące z serwera. Podobnie jak klasa *MrrocppUI:SendQueue* dziedziczy po *java.lang.Thread* i działa w osobnym wątku.

Metoda *run()* czyta komunikaty otrzymane z serwera i na podstawie przesłanych identyfikatorów - *RobotId,DialogId* i *ActionId* - rozpoznaje adresata wiadomości. Jeżeli adresatem jest system - wywoływana jest odpowiednia metoda klasy *MrrocppUI*. Jeżeli zaś adresatem jest któryś z obsługiwanych robotów, wtedy komunikat oraz wartości *DialogId* i *ActionId* przekazywane są do funkcji *handleRequest()* obiektu reprezentującego danego robota.

Funkcja *run()* tej klasy stanowi kolejny element systemu rozpoznawania niepoprawnego działania któregoś z elementów. Mierzy ona czas pomiędzy kolejnymi otrzymanymi komunikatami. Jeżeli przekroczy on określoną wartość, połączenie zostanie zerwane.

4.4.2 Serwer

Komunikacja po stronie serwera zrealizowana została podobnie, jak w konsoli klienckiej. Komunikaty trzymane są w kolejce i wysyłane na bieżąco w osobnym wątku. Po stronie serwera również zaimplementowane zostały mechanizmy służące wykryciu nieprawidłowego działania aplikacji klienckiej.

Klasa Message Klasa ta zdefiniowana jest w pliku *gcc_ntox86/proto.h*. Stanowi odpowiednik klasy *MrrocppUI::Message* po stronie klienckiej, zawiera nawet takie same atrybuty:

RobotId — identyfikator systemu lub robota, który wygenerował komunikat

DialogId — identyfikator okna dialogowego, zawierającego komponent, który wygenerował zdarzenie będące źródłem komunikatu

ActionId — identyfikator akcji

char[] — ciąg znaków stanowiący komunikat

MessageType — typ wiadomości

Reprezentuje ona komunikat przesyłany z serwera do klienta. Zdecydowana większość wysyłanych przez serwer wiadomości to potwierdzenia wykonania rozkazu oraz komunikaty tekstowe, generowane przez procesy sterujące wykonaniem zadania, wyświetlane później na konsoli tekstowej po stronie klienta.

Funkcja replySend Funkcja ta służy do wstawiania obiektów klasy *Message* do kolejki komunikatów. Przyjmuje jeden argument - wskaźnik na obiekt klasy *Message*.

Synchronizacja dostępu Kolejka komunikatów używana jest równolegle przez wiele wątków - wątek *reply_thread* wysyłający komunikaty do klienta, wątek *comm_thread* wstawiający komunikaty systemowe oraz wątki funkcji *callfunc*, wysyłającej potwierdzenia wykonania otrzymanych od użytkownika rozkazów. Pojawiła się więc potrzeba synchronizacji dostępu do kolejki, co zrealizowane zostało przy pomocy semaforów binarnych. Wstawianie do kolejki jest czynnością bardzo krótką, dlatego też oczekiwanie na zwolnienie zasobu nie powoduje zauważalnego spowolnienia działania aplikacji.

Funkcja reply_thread Funkcja ta, działająca jako oddzielny wątek, odpowiada za wysyłanie komunikatów do klienta. Pobiera ona z kolejki komunikatów wszystkie dostępne w danej chwili wiadomości, po czym wysyła je do klienta

Wątek comm_thread Funkja ta odpowiedzialna jest za generowanie wiadomości zawierających komunikaty systemowe, wyświetlane później w konsoli tekstowej po stronie aplikacji klienckiej.

Funkcja pobiera za pomocą wywołania systemowego *MsgReceive* wiadomości z systemu, a następnie tworzy obiekt klasy *Message*. Wygenerowana wiadomość wstawiana jest następnie do kolejki komunikatów przy pomocy funkcji *ReplySend*.

4.5 Testy

Graficzna konsola sterownicza została sprawdzona podczas testów na rzeczywistych robotach, znajdujących się w laboratorium robotyki Instytutu Automatyki i Informatyki Stosowanej Politechniki Warszawskiej. Poprawność jej działania wykazana została podczas wykonywania przez roboty następujących zadań:

- uczenia robota trajektorii — doprowadzanie manipulatora do wybranych położeń i ich zapamiętanie, a następnie odtwarzanie przez robota trajektorii
- układania kostki Rubika — współpraca dwóch zmodyfikowanych manipulatorów Irp6 wyposażonych w system wizyjny i chwytak
- sprzężenia haptycznego dwóch manipulatorów Irp6

5 Podsumowanie

5.1 Wnioski

W ramach pracy powstała sprawnie działająca nowa wersja graficznej konsoli sterowniczej systemu MRROC++. Aplikacja została zrealizowana w architekturze klient-serwer, a jej funkcje rozdzielone zostały pomiędzy graficzą konsolę uruchamianą po stronie klienta oraz proces serwera zintegrowany z systemem MRROC++. Komunikacja pomiędzy klientem i serwerem odbywa się przy wykorzystaniu pary protokołów TCP/IP.

Nowa wersja aplikacji spełnia postawione przed nią we wstępie wymagania. Do realizacji aplikacji klienckiej wybrana została technologia Java firmy Sun Microsystems, sama zaś konsola działa jak applet języka Java. Pozwala to na uruchomienie konsoli pod kontrolą większości współczesnych systemów operacyjnych, a w szczególności Linux, Windows, Unix i MacOS. Obok wieloplatformowości takie rozwiązanie zapewnia również łatwość dystrybucji i aktualizacji oprogramowania.

Powstała graficzna konsola ma wygląd, zgodnie z założeniami, bardzo podobny do dotychczas używanej, stworzonej w oparciu o Photon MicroGUI. Nowa konsola realizuje również w całości funkcje pierwowzoru, pojawiło się jednak kilka istotnych usprawnień. Zbliżony wygląd i identyczny sposób pracy z konsolą umożliwia łatwą i bezproblemową migrację użytkowników na nową wersję konsoli sterowniczej.

Proces serwera napisany został w języku C++, co umożliwiło jego ścisłą integrację ze strukturą MRROC++. Serwer udostępnia na bieżąco informację o aktualnym stanie systemu oraz wykonuje otrzymywane od użytkownika polecenia, czy to polecenia dla systemu MRROC++, czy też rozkazy ruchu dla poszczególnych robotów. Jego niewątpliwą zaletą jest niezależność od stworzonej konsoli klienckiej, co pozwala na współpracę serwera z innymi aplikacjami, niekoniecznie powielającymi jedynie funkcje graficznej konsoli sterowniczej.

Poprawność działania graficznej konsoli sterowniczej została potwierdzona nie tylko testami przeprowadzanymi przeze mnie podczas implementacji aplikacji, jak i po jej zakończeniu - prawdziwą próbą dla nowej wersji było praktyczne jej zastosowanie do sterowania systemem wielorobotowym w wydziałowym labolatorium 012. Próba ta zakończyła się sukcesem, a szeroki wachlarz realizowanych w jej trakcie zadań pozwoliło dogłębnie przetestować poprawność działania aplikacji.

5.2 Perspektywy rozwoju

Zrealizowana aplikacja stworzona została pod kątem współpracy ze wszystkimi robotami, które obsługiwane są przez pierwotną konsolę sterowniczą, tj. trzy wersje robota przemysłowego Irp-6, pas transmisyjny Conveyor oraz efektor do emisji dźwięku

Speaker. Projektowana była ona jednak z myślą o jak najłatwiejszym rozszerzaniu gamy obsługiwanych robotów, więc dodanie nowego robota jest nieskomplikowane, a jedyny wymagany nowy kod źródłowy dotyczy działań specyficznych dla tegoż robota (Dodatek A).

Dzięki realizacji aplikacji w architekturze klient-serwer możliwe jest również stworzenie zdalnych konsoli sterowniczych opartych na innych niż Java technologiach, jak chociażby środowiska QT, GTK czy FLTK.

Literatura

[1] T. Winiarski C. Zieliński, W. Szynkiewicz and T. Kornuta. MRROC++ based system description. Technical Report nr 06-9, IAIS, May Warsaw, 2006.

[2] QNX Realtime operating system (RTOS), www.qnx.com.

[3] Kręglewska U. Sobczyk J. Śluzek A. Zielińska T. Zieliński C., Grodecki A. Sterownik robotów przeznaczony do celów badawczych. Technical report, IAIS, December Warsaw, 1992.

[4] Kierzenkowski K. Zielińska T. Grodecki A. Gosiewski A. Szynkiewicz W., Zieliński C. Środowisko programowe do tworzenia sterowników wielorobotowych dla złożonych zastosowań. Technical report, IAIS, May Warsaw, 1997.

[5] W. Szynkiewicz M.Staniak W. Czajewski C. Zieliński, T. Winiarski and T. Kornuta. MRROC++ based controller of a dual arm robot system manipulating a rubik's cube. Technical Report 06-10, IAIS, April Warsaw, 2006.

[6] QNX Photon microGUI, www.qnx.com/products/middleware/graphics/photon.html.

[7] Java SE at a Glance, java.sun.com.

[8] Creating a GUI with JFC/Swing, java.sun.com.

[9] T. Socolofsky and C. Kale. A TCP/IP Tutorial (Request for comments 1180), http://tools.ietf.org/html/rfc1180 January 1991

[10] P. Szufladowicz. Wizualizacja pracy robotów w systemie MRROC++. Master's thesis, WEiTI, Warsaw, 2008.

Dodatek A - Dodanie obsługi nowego robota

Zrealizowana przeze mnie aplikacja pisana była z myślą o przyszłym rozszerzeniu gamy obsługiwanych przez system MRROC++ robotów. Zarówno serwer, jak i graficzna konsola, napisane zostały tak, by dodanie nowego robota wymagało jak najmniej zmian w kodzie aplikacji. W poniższym rozdziale zaprezentuję czynności wystarczające do zapewnienia obsługi robota kartezjańskiego. Oferował on będzie jedynie podstawowe funkcje w celu możliwie czytelnego przedstawienia realizacji tego zadania.

Klient

Pierwszym krokiem po stronie klienckiej jest stworzenie klasy reprezentującej robota - nazwijmy ją *cRobot* - dziedziczącej po klasie *Robot*. Powinna ona zawierać implementacje odziedziczonych z klasy *Robot* abstrakcyjnych metod, dodatkowo ruchy ręczne tym robotem możliwe będą przy użyciu okna dialogowego *MoveDialog*, dziedziczącego po *ReadSetDialog*.

Klasa reprezentująca robota

Kod klasy *cRobot* przedstawiono poniżej:

```
//cRobot.java
class Conveyor extends Robot
{
  /**
   * Typ wyliczeniowy określający akcję
   */
  enum ActionId{Read,Set};

  /**
   * Typ wyliczeniowy określający okno dialogowe
   */
  enum DialogId{EDPLoad,EDPUnload,Move};

  /**
   * Tworzy obiekt robota
   *
   * @param MrrocppUI ui referencja do głównego okna aplikacji
   * @param int id identyfikator robota
   * @param String name nazwa robota
   */
  public cRobot(MrrocppUI ui,int id,String name)
  {
    super(ui,id,name);
  }

  /**
   * Blokuje okna dialogowe
   */
  public void disableDialogs()
  {
    moveDialog.getGlassPane().setVisible(true);
```

```
}

/**
 * Odblokowuje okna dialogowe
 */
public void enableDialogs()
{
  moveDialog.getGlassPane().setVisible(false);
}

/**
 * Zwraca menu robota
 */
JMenu getMenu()
{
  menu = new JMenu(name);
  menu.setEnabled(false);
  edpLoadAction = new JMenuItem("EDP Load",'L');
  edpLoadAction.addActionListener(this);
  menu.add(this.edpLoadAction);
  edpUnloadAction = new JMenuItem("EDP Unload",'U');
  edpUnloadAction.addActionListener(this);
  menu.add(this.edpUnloadAction);
  edpUnloadAction.setEnabled(false);
  menu.addSeparator();
  moveAction = new JMenuItem("Move",'M');
  servoAction.setEnabled(false);
  servoAction.addActionListener(this);
  menu.add(this.servoAction);

  return menu;
}

/**
 * Ukrywa okna dialogowoe
 */
public void hideDialogs()
{
  if(moveDialog != null)
  {
    moveDialog.setVisible(false);
  }
}

/**
 * Obsługuje uruchomienie procesu EDP robota
 */
public void EDPLoad()
{
  if(moveDialog == null)
  {
    moveDialog = new MoveDialog(ui,robotId);
  }
  moveAction.setEnabled(true);
  edpLoadAction.setEnabled(false);
  edpUnloadAction.setEnabled(true);
}

/**
```

```
 * Obsługuje zabicie procesu EDP robota
 */
public void EDPUnload()
{
  moveAction.setEnabled(false);
  edpLoadAction.setEnabled(true);
  edpUnloadAction.setEnabled(false);
  hideDialogs();
}

/**
 * Obsługuje zdarzenia wygenerowane przez komponenty związane z robotem
 *
 * @param ActionEvent e obiekt reprezentujący zdarzenie
 */
public void actionPerformed(ActionEvent e)
{
  if(e.getSource() == edpLoadAction)
  {
    ui.SendMessage(robotId,DialogId.EDPLoad.ordinal(),0,"",
        MessageType.FatalError);
  }
  else if(e.getSource() == edpUnloadAction)
  {
    ui.SendMessage(robotId,DialogId.EDPUnload.ordinal(),0,"",
        MessageType.FatalError);
  }
  else if(e.getSource() == moveAction)
  {
    MoveDialog.showDialog();
    ui.SendMessage(robotId,DialogId.Move.ordinal(),
        ActionId.Read.ordinal(),"",MessageType.FatalError);
  }
}

/**
 * Obsługuje komunikaty skierowane do robota
 *
 * @param int dialogId identyfikator okna dialogowego
 * @param int actionId e identyfikator akcji
 * @param double[] values przeslane wartosci
 */
public void handleRequest(int dialogId,int actionId,double[] values)
{
  switch (dialogId)
  {
    case DialogId.Move.ordinal():
    {
      moveDialog.UpdateDialog(values);
      break;
    }
    case DialogId.EDPLoad.ordinal():
    {
      EDPLoad();
      break;
    }
    case DialogId.EDPUnload.ordinal():
    {
      EDPUnload();
```

```
        break;
      }
    }
  }

  private JMenu menu;
  private JMenuItem edpLoadAction,edpUnloadAction,moveAction;
  private MoveDialog moveDialog;
}
```

Okno ruchów ręcznych

Możliwości zdefiniowanej klasy *cRobot* ograniczają się do uruchomienia i zatrzymania procesu EDP robota. Umożliwiać ma ona jeszcze ruchy ręczne robotem, dlatego też należy zdefiniować w klasie *cRobot* zagnieżdżoną klasę *MoveDialog*, dziedziczącą po *ReadSetDialog* i reprezentującą okno ruchów ręcznych.

```
//cRobot.java
class MoveDialog extends ReadSetDialog
{
  public MoveDialog(MrrocppUI ui,int id)
  {
    super(ui,name + " - Move");
    robotId = id;
    dialogId = DialogId.Move.ordinal();

    //Współrzędne robota
    variableLabelText = "Axis";
    variableNumber = 3;
    variableLabels = new String[]{"X","Y","Z"};

    //Włączenie importowania pozycji i ruchów krokowych
    enableImportExport = true;
    enableIncremental = true;

   //Parametry kroku
    stepMin = 0;
    stepMax = 10;
    stepDefault = 0;
    stepStep = 0.01;

    //Zakresy ruchu robota
    desiredPositionParams =
     new double[][]{
       {0,-100,100,0.5},
       {0,-100,100,0.5},
       {0,-100,100,0.5},
     };
```

```
    //Rysowanie okna dialogowego - odziedziczone z ReadSetDialog
    drawDialog();
  }
}
```

Klasa MrrocppUI

Dodanie okna dialogowego uzupełnia brakujące funkcje w klasie *cRobot*. Jedyne, co pozostało jeszcze do zrobienia po stronie klienckiej to dodanie obsługu zdefiniowanego wcześniej robota do graficznej konsoli sterowniczej. Sprowadza się to do dodania jednej linijki do kostruktora klasy *MrrocppUI*.

```
//MrrocppUI.java
/**
 * Inicjalizuje wektor obiektów reprezentujących obsługiwane roboty
 */
public MrrocppUI()
{
  robots = new Vector<Robot>();
  addRobot(new Irp6OnTrack(this,1,"Irp6 on Track"));
  addRobot(new Irp6Postument(this,2,"Irp6 Postument"));
  addRobot(new Conveyor(this,3,"Conveyor"));
  addRobot(new Speaker(this,4,"Speaker"));
  addRobot(new Irp6Mechatronika(this,5,"Irp6 Mechatronika"));

  //Dodanie robota kartezjańskiego
  addRobot(new Irp6Mechatronika(this,6,"Kartezjanski"));

  createGUI();
}
```

Skutkiem dodania tej linijki będzie rozszerzenie graficznej konsoli sterowniczej o obsługę nowego robota klasy *cRobot* o identyfikatorze równym 6. Od tej chwili menu robota pojawiać się będzie w menu górnym *Robot*, co umożliwi dostęp do funkcji robota. Przekazywane mu również będą skierowane do niego komunikaty otrzymane z serwera. W oknie dialogowym *Process Control*, służącym do sterowania przebiegiem wykonania zadania, zostanie dodany rónież nowy wiersz przycisków, odpowiadających za sterowanie wykonaniem zadania na nowym manipulatorze.

Serwer

W porównaniu ze stroną kliencką, po stronie serwera potrzeba zdecydowanie mniej modyfikacji, aby umożliwić obsługę kolejnego robota. Obsługa ta wymaga oczywiście dodatkowo zaimplementowania w systemie MRROC++ funkcji tegoż robota, nie jest to jednak przedmiotem tej pracy, a dodanie obsługi robota do serwera odbywa się przy założeniu istniejącej już implementacji robota w systemie.

Synchronizacja dostępu do manipulatora

W celu zapobieżenia jednoczesnego wykonania więcej niż jednego rozkazu przez pojedynczy manipulator, niezbędne jest zdefiniowanie dwóch zmiennych:

sem_t sem_cRobot semafor binarny, synchronizujący dostęp do flagi block_cRobot

int block_cRobot flaga określająca dostępność robota

Definicje tych dwóch zmiennych należy umieścić w pliku *ui_init.cc*.

```
sem_t sem_cRobot;
int block_cRobot = 0;
```

Dodatkowo w funkcji *init()* należy zainicjalizować nowy semafor:

```
//ui_init.cc
int init()
{
    if(sem_init(&sem,0,1) == -1
    || sem_init(&sem_conveyor,0,1) == -1
    || sem_init(&sem_irp6_on_track,0,1) == -1
    || sem_init(&sem_irp6_postument,0,1) == -1
    || sem_init(&sem_irp6_mechatronika,0,1) == -1
    || sem_init(&sem_speaker,0,1) == -1
    //semafor synchronizujący dostęp do nowego robota
    || sem_init(&sem_cRobot,0,1) == -1
    || sem_init(&sem_ui,0,1) == -1
    || sem_init(&sem_mp,0,1) == -1)
    {
     perror("Unable to initialize semaphore\n");
     return NULL;
    }
    //reszta funkcji init()...
}
```

Obsługa rozkazów skierowanych do robota

Do obsługi rozkazów zdefiniować trzeba funkcję, przyjmującą jako parametry dwa identyfikator - okna dialogowego i akcji - oraz tablicę przesłanych od klienta parametrów. Zwyczajowo funkcję tę nazywa się *NAZWAROBOTA_handle_request()* i umieszcza w pliku *fun_r_NAZWAROBOTA.cc*. Kod tej funkcji zależny jest już od tego, jakie rozkazy robot przyjmuje. Powinna ona na podstawie przesłanych identyfikatorów wywołać odpowiednie metody, implementujące działania robota po stronie MRROC++. Metody te powinny znajdować się w pliku *fun_r_NAZWAROBOTA.cc*. Dodatkowo w pliku *gcc_ntox86/proto.h* należy zdefiniować stałą NAZWAROBOTA o wartości równej identyfikatorowi po stronie graficznej konsoli oraz stałe reprezentujące akcje równe pozycjom odpowiadających im po stronie klienckiej. W tym wypadku w pliku *gcc_ntox86/proto.h* powinna znaleźć się linijka:

```
#define CROBOT 6

#define EDPLOAD  0
#define EDPUNLOAD 1
#define MOVE   2
```

Na potrzeby przykładu zakładam, że w pliku *fun_r_NAZWAROBOTA.cc* zdefiniowane zostały następujące funkcje:

EDP_cRobot_create() uruchamia proces EDP robota

EDP_cRobot_slay() zabija proces EDP robota

cRobot_Move() wykonuje ruch do zadanych współrzędnych

UWAGA! Bardzo ważne jest, by każde wywołanie funkcji *NAZWAROBOTA_handle_request()* skutkowało nadaniem komunikatu zwrotnego do serwera, zawierające identyczne identyfikator jak te, przekazane do funkcji. Można to uczynić albo w kodzie funkcji *NAZWAROBOTA_handle_request()*, albo w kodzie funkcji z niej wołanych.
Przykładowy kod funkcji *NAZWAROBOTA_handle_request()* przedstawiono poniżej:

```
//fun_r_cRobot.cc
/**
 * Obsługuje komunikaty skierowane do robota cRobot
 */
int cRobot_handle_request(int DialogId,int ActionId,double[] values)
{
  switch(DialogId)
  {
    case EDPLOAD:
      EDP_cRobot_create();
      break;
    case EDPUNLOAD:
      EDP_cRobot_slay();
      break;
    case MOVE:
      cRobot_Move(values);
      break;
  }

  //Wysłanie potwierdzenia wykonania polecenia
  replySend(new Message(CROBOT,DialogId,ActionId,0,NULL,NULL));
}
```

Obsługa poleceń otrzymanych z konsoli

Aby dodać obsługę rozkazów dla nowego robota otrzymywanych z konsoli klienckiej, należy już tylko rozszerzyć funkcję *callfunc*. Jednym z jej elementów jest wyrażenie

switch, które w zależności od przesłanego identyfikatora robota sprawdza jego dostępność, a następnie wywołuje odpowiednią fukcję obsługi zdarzeń. Kod wyrażenia *case*, które należy dodać, przedstawiono poniżej:

```
//ui_init.cc
case CROBOT:
{
  //Sprawdzenie dostępności robota
  sem_wait(&sem_cRobot);
  if(block_cRobot == 1 || (!ui_robot.cRobot))
  {
    sem_post(&sem_cRobot);
    return (void*)NULL;
  }
  block_cRobot = 1;
  sem_post(&sem_cRobot);

  //Obsługa rozkazu
  cRobot_handle_request(DialogId,ActionId,values);

  //Zwolnienie robota po wykonaniu polecenia
  sem_wait(&sem_cRobot);
  block_cRobot = 0;
  sem_post(&sem_cRobot);
  break;
}
```

Włączenie obsługi nowego manipulatora w funkcję *callfunc* wyczerpuje modyfikacje niezbędne do tego, by rozszerzyć graficzną konsolę sterowniczą o możliwość sterowania nowym robotem.