Dokumentacja serwera MUD - Multi User Dungeon

projekt z przedmiotu ZPI

Kajetan Rzepecki

19 listopada 2013

Spis treści

1 Kompilacja

Niniejsza sekcja dokumentacji opisuje kompilację serwera gry MUD. Serwer został napisany w 100% w języku Erlang i wykorzystuje automatyczne narzędzie do budowania projektu - **Rebar**. W związku z tym faktem, do skompilowania projektu niezbędne są dwa programy:

- **Git** - narzędzie do kontroli wersji wykorzystywane do klonowania bibliotek wykorzystanych w implementacji serwera,
- **Erl** - środowisko uruchomieniowe/środowisko kompilacji języka Erlang (dołączone do projektu).

Poniższe instrukcje zakładają obecność i dostępność obu tych technologii na maszynie docelowej.

1.1 Platforma *nix

Zakładając, że na systemie zainstalowano niezbędne zależności (**Git** oraz **środowisko Erlang**) można przejść do kompilacji projektu. W tym celu należy rozpakować archiwum zawierające źródła serwera i przejść do odpowiedniego katalogu:

```
$ cd path/to/projekt
$ tar -xf mudserverproject.tar.gz
$ cd mudserverproject
```

Następnie wystarczy już tylko posłużyć się dołączonym skryptem **Makefile** wykonując poniższą komendę (`build` jest opcjonalne):

```
$ make [build]
```

Skrypt ten wykorzystuje inne narzędzie (typowe dla programów napisanych w języku Erlang) - **Rebar**. Rebar automatycznie pobierze niezbędne zależności serwera korzystając z publicznie dostępnych repozytoriów **Git**. Następnie, Rebar automatycznie skompiluje wszystkie zależności oraz kod samego serwera i jeśli wszystko pójdzie dobrze, działanie komendy zakończy się następującym logiem (obcięty z oczywistych powodów):

```
==> mudserverproject (get-deps)
Pulling lager from {git,"git://github.com/basho/lager.git",{tag,"2.0.0"}}
Cloning into lager...
...
==> Entering directory ..
==> goldrush (compile)
Compiled src/glc_ops.erl
...
==> mudserverproject (compile)
...
Compiled src/mud_game.erl
```

Serwer został skompilowany i jest gotowy do użytku.

1.2 Platforma Windows

Podobnie jak w przypadku kompilacji na platformie *nix, zakładając że wszystkie niezbędne programy są obecne na docelowej maszynie, należy rozpakować archiwum z kodem źródłowym i nastepnie przejść do niego:

```
...Graficzne narzędzie do obsługi pliku mudserver.zip...
> cd C:\path\to\mudserverproject
```

Podobnie jak w przypadku kompilacji na platformie **nix, wykorzystano automatyczne narzędzie do budowania projektów - *Rebar**. Niestety ze względu na ograniczenia czasowe zostało ono przetestowane jedynie w dwóch *nixo-podobnych środowiskach dostępnych na platformę Windows:

- **MinGW**

- **GitBash**

Drugie z nich powinno być dostępne razem z dystrybucją narzędzia **git**. Nic nie stoi jednak na przeszkodzie, by projekt kompilował się bez dostępu do tych środowisk (dzięki generycznej naturze narzędzia Rebar). Aby skompilować projekt należy uruchomić następujące komendy:

```
> rebar get-deps compile
```

W efekcie narzędzie automatycznie ściągnie zależności projektu wykorzystując program Git, a następnie skompiluje projekt wykorzystując środowisko języka Erlang. W efekcie otrzymamy następujący log:

```
==> mudserverproject (get-deps)
Pulling lager from {git,"git://github.com/basho/lager.git",{tag,"2.0.0"}}
Cloning into lager...
...
==> Entering directory ..
==> goldrush (compile)
Compiled src/glc_ops.erl
...
==> mudserverproject (compile)
...
Compiled src/mud_game.erl
```

Serwer jest gotowy do użytkowania.

2 Instalacja

Poniższe sekcje zawierają instrukcje instalacji oprogramowania serwera gry MUD dołączonego do projektu.
W celu użytkowania serwera gry niezbędne jest jedynie środowisko języka Erlang w wersji **R16B** (lub późniejszej). Do projektu dołączono zatem odpowiednie paczki instalacyjne (na platformę Debian Wheezy 64 bit oraz Windows 32 bit) a poniższe instrukcje zakładają, że zostały one zainstalowanie na maszynie docelowej.
Niestety, z powodu ograniczeń czasowych nie udało się stworzyć automatycznych instalatorów instalujących oprogramowanie "jednym kliknięciem".

2.1 Platforma *nix

Do projektu zostało dołączone archiwum ze skompilowaną wersją serwera. Wystarczy je jedynie odpakować by serwer był gotowy do użytkowania:

```
$ cd path/to/projekt
$ tar -xf mudserver.tar.gz
```

2.2 Platforma Windows

Podobnie, jak w przypadku platformy *nix, na platformie Windows wystarczy jedynie rozpakować archiwum serwera (to samo archiwum jest wykorzystywane na obu platformach) by było on gotowy do użytkowania:

```
...Wykorzystanie graficznego narzędzia do rozpakowania dołączonego archiwum mudserver.zip
```

3 Wskazówki użytkowania

Niniejsza sekcja opisuje najważniejsze aspekty użytkowania serwera i dostarczanych przezeń funkcjonalności.

3.1 Konfiguracja

Serwer gry, w związku z zastosowaną architekturą wykorzystuje dwa główne pliki konfiguracyjne. Niestety, z powodu ograniczeń czasowych nie udało się połączyć konfiguracji w jeden, wygodny plik.
Pierwszy plik konfiguracyjny (`./config/config.json`) jest wykorzystywany przez mid-end serwera, który zajmuje się obsługą sesji graczy. Ponieważ plik ten zawiera bardzo dużo opcji konfiguracji poniżej zostały zawarte jedynie najważniejsze jej elementy, a o pozostałych opcjach można przeczytać w zewnętrznej dokumentacji.
Najważniejsze elementy konfiguracji:

- `hive.port` - port, na którym użytkownicy będą mogli łączyć się do gry.
- `hive.accepted_origins` - lista domen, które mogą połączyć się do serwera gry.
- `api.port` - port, na którym mid-end udostępnia RESTowe API służące do zarządzania sesjami klientów.
- `connectors.pools.backend.port` - port, na którym będzie dostępne API serwera odpowiedzialnego za logikę gry.

Drugi plik konfiguracyjny (`./mud.config`) jest plikiem konigurującym działanie back-endu serwera - jego części odpowiedzialnej za obsługę logiki gry. Najważniejsze opcje:

- `port` - port, na którym serwer gry będzie udostępniał swoje API (musi zgadzać się z `connectors.pools.backend.por`
- `(min|max)_allowed_nick_len` - (maksy|mini)malna długość nazwy gracza.
- `stat_(min|max)` - (maksy|mini)malne zakresy losowych statystyk graczy generowane przy tworzeniu postaci.
- `save_timeout` - interwał zapisów stanu gry.
- `*_file` - nazwy plików świata gdy dla lokacji, graczy, przedmiotów i haseł graczy.
- `hive_url` - URL pod którym będzie dostępne API mid-endu (przeważnie `localhost`, ale możliwe jest uruchomienie serwerów na osobnych maszynach).

Do serwera została dostarczona przykładowa konfiguracja, która pozwala na jego użytkowanie.

3.2 Uruchamianie

Po zainstalowaniu/skompilowaniu i opcjonalnym przekonfigurowaniu serwera można przystąpić do jego uruchomienia. Procedura jest bardzo prosta - wystarczy z katalogu, w którym znajduje się serwer posłużyć się dołączonymi skryptami, w zależności od platformy - `start.sh` oraz `start.bat` - podając ścieżkę, w której znajdują się pliki świata gry (pliki `*.json`):

- Windows:

```
> cd C:\path\to\projekt
> start.bat path/to/resources/
```

- *nix:

```
$ cd path/to/projekt
$ ./start.sh path/to/resources/
```

Do projektu dołączono przykładowe pliki świata (dostępne w archiwum `sampleworld.tar.gz` i `sampleworld.zip`).

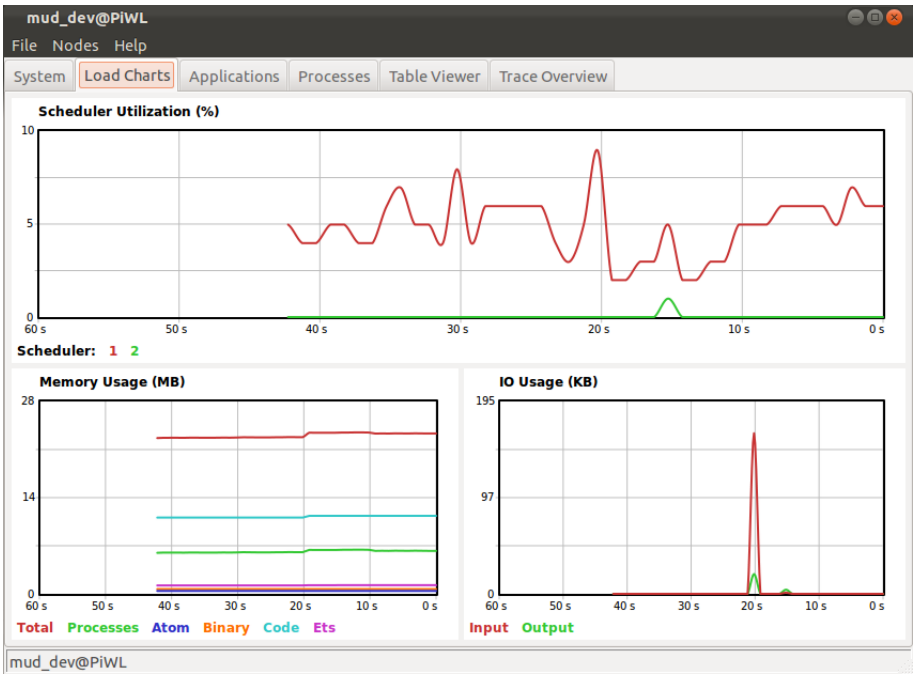
3.3 Wykorzystanie wbudowanego nadzorcy systemu

Ponieważ do implementacji wykorzystano środowisko Erlang, dziedziczy on wszystkie udostępniane przez nie narzędzia do zarządzania aplikacją, czyli tak zwanego **nadzorcę systemu** vel **obserwatora**. Aby go uruchomić należy uruchomić następującą komendę w **konsoli serwera** (jest to wywołanie funkcji języka Erlang):

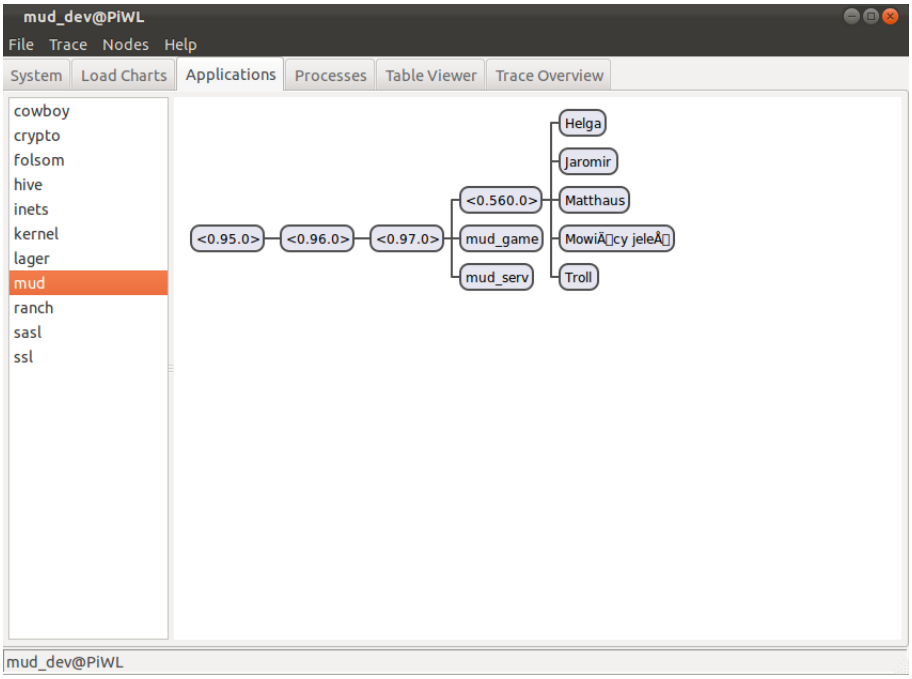
```
(mud_dev@host)1> observer:start().
```

W efekcie otwarte zostanie okno obserwatora, w którym szczególnie interesujące są dwie zakładki:

- **Load Charts** - zawierająca statystyki z działania serwera:



- **Applications** - pozwalająca przeglądać strukturę drzewa procesów, oraz zarządzać nim:



3.4 Logi i troubleshooting

Serwer generuje dużą ilość przydatnych logów. Są one dostępne bezpośrednio w konsoli serwera, ale także w katalogu `./log/mud/`, gdzie:

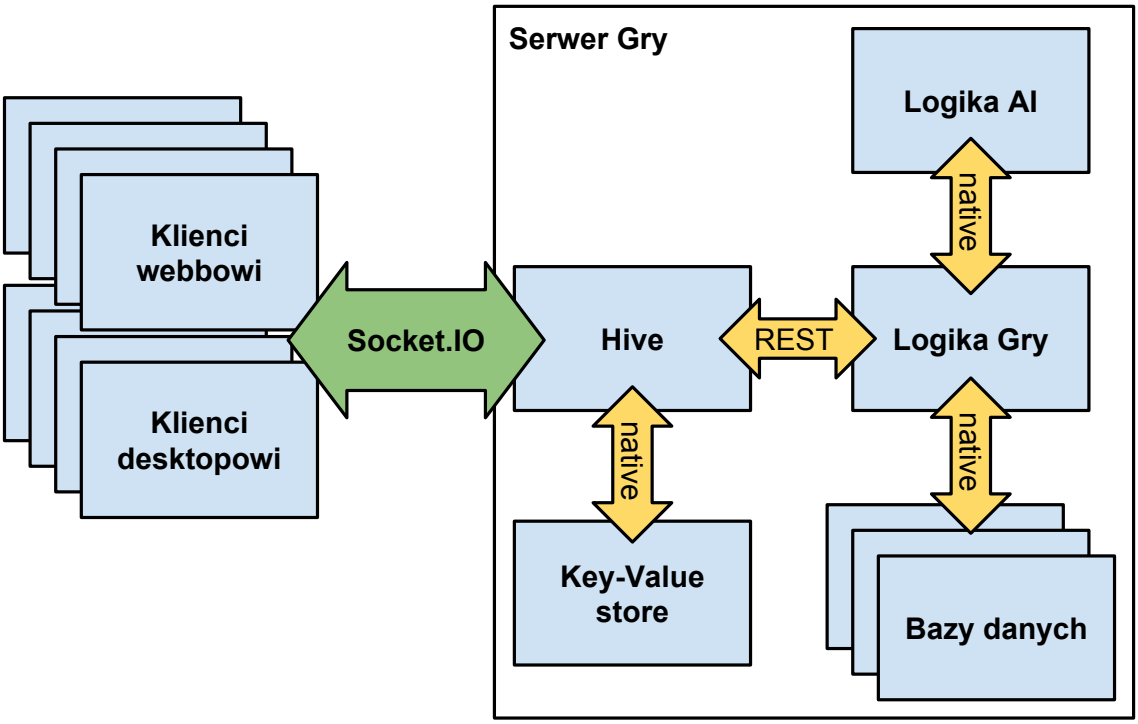
- **console.log** - log konsoli zawierający wszystkie logi ważniejsze lub równoważne poziomowi `info`,
- **error.log** - log konsoli zawierający wszystkie logi ważniejsze lub równaważne poziomomi `error`,
- **crash.log** - log "crash'y" serwera, który miejmy na dzieje nigdy nie powstanie ;-)

4 Implementacja

Ostatnia sekcja tego dokumentu zawiera informacje o implementacji serwera gry MUD.

4.1 Architektura serwera gry

Poniższy schemat pokazuje architekturę serwera gry wyszczególniając poszczególne jego elementy i drogi komunikacji w nim występujące:



4.1.1 Komunikacja z klientami

Komunikacja Klient-Serwer przebiega za pośrednictwem protokołu **Socket.IO** oraz subprotokołów **WebSocket** i **XHR-polling**, dzięki czemu zapewniona jest szybka komunikacja na zasadzie aplikacji **Comet'owej**. Klienci są odpowiedzialni za połączenie się z serwerem oraz przeprowadzenia handshake'u Socket.IO. Następnie cała komunikacja odbywa się na zasadzie asynchronicznego przekazywania **event'ów**. Szczegóły zastosowanego protokołu wymiany danych zostały zawarte w następnej sekcji.

4.1.2 Mid-end - Hive

Głównym punktem wejściowym aplikacji jest szybki serwer Socket.IO (napisany przez autora tego dokumentu i wykorzystywany z powodzeniem w dużo więszych projektach `</shameless-selfplug>`) nazwany **Hive**. Hive zajmuje się zarządzaniem sesjami klientów oraz pośredniczeniem w komunikacji między logiką biznesową aplikacji oraz klientami.
Serwer Hive udostępnia liczne usługi dla obu stron komunikacji:

- **kanały Publisher-Subscriber** wykorzystywane w porojekcie do łatwego propagowania zdarzeń.
- **Key-Value store** na tymczasowe dane dotyczące sesji klienta umożliwiające wykorzystanie bazy danych **Redis** lub przechowywanie ich lokalnie w pamięci.

Komunikacja w obrębie mid-endu przebiega **natywnie** poprzez wysyłanie wiadomości Erlanga. Więcej o serwerze Hive można dowiedzieć się z zewnętrznego źródła.

4.1.3 Back-end - Logika

Logika gry została zaimplementowana jako dwa oddzielne serwery (a właściwie serwer główny i drobne, autonomiczne maszyny stanów - **FSM** - implementujące logikę sztucznej inteligencji gry). Serwer wykorzystuje JSONową bazę danych **NoSQL** - filesystem maszyny, na której jest uruchomiony ;-)
Serwer logiki odpowiada za wykonywanie akcji graczy i zarządzanie stanem świata gry. Logiką autonomicznych jednostek obecnych w świecie gry, a nie będących graczami zajmują sie małe maszyny stanów, które są w pełni skryptowalne w języku Erlang. Umożliwiają one odciążenie serwera głównego oraz lepszą organizację przepływu danych w całym systemie.
Komunikacja w obrębie back-endu przebiega drogą natywną - za pośrednictwem przesyłania wiadomości Erlanga. Dodatkowo warto jest zauważyć, że maszyny stanów sztucznej inteligencji spoof'ują procesy użytkowników systemu podłączając się bezpośrednio do usług mid-endu.

4.2 Protokół komunikacji

4.2.1 Socket.IO

Serwer korzysta z protokołu Socket.IO do komunikacji:

- specyfikacja - https://github.com/LearnBoost/socket.io-spec
- referencyjna implementacja klienta - https://github.com/LearnBoost/socket.io-client

. . . oraz dwóch protokołów transportujących:

- WebSocket - http://en.wikipedia.org/wiki/WebSocket
- XHR-polling - http://en.wikipedia.org/wiki/Comet_(programming)#XMLHttpRequest_long_polling
- Klient przeglądarkowy
  Implementacja klienta przeglądarkowego może wykorzystać gotowego klienta Socket.IO wymienionego powyżej.
- Klient desktopowy
  Implementacja klienta desktopowego może wykorzystać dowolną bibliotekę kliencką Socket.IO:
  - https://github.com/benkay/java-socket.io.client (Java)
  - https://github.com/Gottox/socket.io-java-client (Java)
  - https://pypi.python.org/pypi/socketIO-client (Python)
  - http://socketio4net.codeplex.com/ (.NET)

  . . . lub wykorzystać gołe połączenie WebSocket:
  - http://docs.oracle.com/javaee/7/tutorial/doc/websocket.htm (Java)
  - http://java-websocket.org/ (Java)
  - https://pypi.python.org/pypi/websocket-client/0.4 (Python)

  . . . oraz prosty parser wiadomości Socket.IO zakładający, że przyjmowane będą następujące wiadomości:

  ```
  "1:::" - po nawiązaniu połączenia z serwerem,
  "5:::JSON" - przy każdym evencie, gdzie "JSON" to zakodowany w JSONie event gry (więcej poniżej),
  "8:::" - po okresie bez żadnej aktywności,
  "0:::" - po rozłączeniu z serwerem,
  ```

  Ostatnia opcja będzie wymagała samodzielnego przeprowadzenia połączenia z serwerem poprzez HTTP oraz następnie połączenia WebSocket pod odpowiedni przydzielony przez serwer URL.

4.2.2 Event'y gry

Komunikacja z serwerem odbywa się tylko i wyłącznie przez event'y zakodowane jako krótkie JSON'y. Każdy event wysyłady do/przychodzący z serwera musi być następującej postaci:

```
{
    "name" : nazwa_eventu,
    "args" : argumenty_eventu
}
```

Konkretny format argumentów zależy od typu event'u i będzie opisany poniżej.

4.2.3 Błędy sewera

Błędy serwera są przekazywane jako specjalny event `hive_error`, więc mogą być obsułgiwane w taki sam sposób, jak pozostałe event'y.

```
{
    "name" : "hive_error",
    "args" : {
        "error" : kod_bledu,
        "description" : opis_bledu
    }
}
```

4.2.4 Autoryzacja

Przed rozpoczęciem gry gracz musi się autoryzować na swoje konto wysyłając następujący event:

```
{
    "name" : "authorize",
    "args" : [
        {
            "nick" : login,
            "password" : hash_hasla
        }
    ]
}
```

... gdzie `login` to wybrany Nick gracza, a `hash_hasla` to hash **SHA1** otrzymany z wybranego przez gracza hasła, posolonego jego nazwą użytkownika:

```
nickname = "Nickname";
password = "Password"
// (nickname + password) == "NicknamePassword"
hash = sha1(nickname + password);
// hash == "ca805ddc46b39fc3cb1099ec5442b9c7aae49e47"
```

W odpowiedzi otrzymamy:

```
{
    "name" : "authorize",
    "args" : [
        {
            "permission" : wynik_autoryzacji
        }
    ]
}
```

... gdzie `wynik_autoryzacji` to string `granted` lub wartość `null` odpowiednio dla powodzenia i niepowodzenia autoryzacji.

4.2.5 Tworzenie postaci

Tworzenie nowej postaci przebiega bardzo prosto - przeprowadzamy autoryzację do serwera podając nowy nick i nowe hasło. Jeśli postać o takim nicku nie istnieje konto zostanie utworzone, a serwer w odpowiedzi zwróci:

```
{
    "name" : "authorize",
    "args" : [
        {
            "permission" : wynik_autoryzacji
        }
    ]
}
```

. . . gdzie `wynik_autoryzacji` to string `granted` lub wartość `null` (odpowiadająca sytuacji, gdy nick został już przez kogoś zajęty).
Obecnie nie mam w planach dodawania zmiany hasła itd, więc będzie to jedyny sposób tworzenia nowych kont graczy.

4.2.6 "Wejście" do gry

Bezpośrednio po wejściu do gry otrzymamy kilka event'ów opisujących świat gry, w którym się znajdujemy i wydarzenia w nim się odbywające:

- `location_info` - opisane przy okazji komendy examine,
- `character_info` - opisane przy okazji komendy examine,
- `player_enters` - opisane przy okazji komendy move

4.2.7 Rozmowa

Rozmowa odbywa się przez wysłanie eventu `say` zawierającego typ wypowiedzi oraz jej tekst:

```
{
    "name" : "say",
    "args" : [
        {
            "text" : wiadomosc,
            "type" : typ_wiadomosci
        }
    ]
}
```

`wiadomosc` zawiera tekst wysyłanej wiadomości. `typ_wiadomosci` zawiera krótki string prezentujący typ wypowiedzi (na przykład `says`, `whispers`, `yells`, etc) dla potrzeb kosmetycznych. W efekcie otrzymamy event:

```
{
    "name" : "msg",
    "args" : [
        {
            "nick" : nazwa_gracza,
            "type" : typ_wypowiedzi,
            "text" : tekst_wypowiedzi
        }
    ]
}
```

Taki sam event dostaniemy przy każdej wypowiedzi innych graczy.

4.2.8 Komendy gracza

Interakcję ze światem gry umożliwiają graczowi komendy, które są przesyłane poprzez event `do`:

```
{
    "name" : "do",
    "args" : [komenda]
}
```

W przypadku podania błędnych argumentów dla komendy otrzymamy następujący event zawierający `opis` problemu:

```
{
    "name" : "bad_command",
    "description" : opis
}
```

Więcej o dostępnych komendach tutaj.

4.3 Dostępne komendy

4.3.1 `examine`

Przykład:

```
{
    "action" : "examine",
    "args" : id_obiektu
}
```

`id_obiektu` może być nazwą gracza/NPC/przeciwnika, identyfikatorem lokacji lub identyfikatorem przedmiotu osiągalnego z lokacji, w które aktualnie znajduje się gracz. W zależności od typu obiektu w odpowiedzi otrzymamy:

```
{
    "name" : "character_info",
    "args" : [opis_gracza]
}
// ...lub:
{
    "name" : "location_info",
    "args" : [opis_lokacji]
}
// ...lub:
{
    "name" : "item_info",
    "args" : [opis_przedmiotu]
}
```

Więcej o `opisie_gracza` tutaj, więcej o `opisie_lokacji` tutaj, więcej o `opisie_przedmiotu` tutaj.

4.3.2 `move`

Przykład:

```
{
    "action" : "move",
    "args" : id_lokacji
}
```

`id_lokacji` musi być prawidłowym ID lokacji osiągalnej z lokacji, w której aktualnie znajduje się gracz. W odpowiedzi gracz zostanie przeniesiony do nowej lokacji i otrzyma następujący event:

```
{
    "name" : "location_info",
    "args" : [opis_lokacji]
}
```

Dodatkowo zostaną wygenerowane dwa event'y propagowane do wszystkich graczy obecnych w starej i nowej lokacji gracza:

```
{
    "name" : "player_leaves",
    "args" : [
        {
            "location" : nazwa_opuszczanej_lokacji,
            "nick" : nick_opuszczajacego_gracza
        }
    ]
}

{
    "name" : "player_enters",
    "args" : [
        {
            "location" : nazwa_nowe_lokacji,
            "nick" : nick_gracza
        }
    ]
}
```

Event'y te istnieją z czysto kosmetycznych względów.
Więcej o `opisie_lokacji` tutaj.

4.3.3 `attack`

Przykład:

```
{
    "action" : "attack",
    "args" : nazwa_gracza
}
```

`nazwa_gracza` musi być prawidłowym ID gracza/przeciwnika/NPC obecnego w lokacji, w której aktualnie znajduje się gracz. W odpowiedzi gracz zaatakuje `nazwa_gracza` i otrzyma następujący event:

```
{
    "name" : "battle",
    "args" : [
        {
            "attacker" : nazwa_gracza_atakujacego,
            "defender" : nazwa_drugiego_gracza,
            "type" : typ_wydarzenia,
            "value" : wartosc_wydarzenia
        }
    ]
}
```

`typ_wydarzenia` zawiera typ zaistniałego wydarzenia (na przykład "hit", "miss", "kill"); jeśli obecne jest pole `wartosc_wydarzenia` zawiera ono wartość liczbową opisującą zdarzenie (na przykład dla typu "hit" `wartosc_wydarzenia` będzie opisywała siłę uderzenia). Podobne event dostaną wszyscy gracze obecni w danej lokacji. Wykonanie tej komendy może rozzłościć NPC lub przeciwnika prowadząc do walki na śmierć i życie (lub ucieczkę do innej lokacji). W przypadku śmierci któregoś z graczy otrzymamy taki sam event ze stosownym opisem natomiast przegrany gracz zostanie usunięty z obecnej lokacji (jego przedmioty w niej zostają).

4.3.4 `take` / `drop`

Przykład:

```
{
    "action" : "take"/"drop",
    "args" : id_przedmiotu
}
```

`id_przedmiotu` musi być prawidłowym ID przedmiotu obecnego w lokacji, w której aktualnie znajduje się gracz (lub w jego inwentarzu). W odpowiedzi przedmiot zostanie przeniesiony do inwentarza gracza (lub do lokacji, w której obecnie się znajduje) i otrzymamy nastepujący event:

```
{
    "name" : "inventory_update",
    "args" : {
        "nick" : nazwa_gracza,
        "type" : typ_aktualizacji,
        "id" : id_przedmiotu,
        "name" : nazwa_przedmiotu
    }
}
```

Event taki otrzymamy także w wyniku akcji innego gracza znajdującego się w tej samej lokalizacji. Więcej o przedmiotach tutaj.

4.4 Reprezentacja świata gry

Poniższe sekcje zawierają opisy różnych obiektów świata gry, które mogą się zmieniać w trakcie gry w reakcji na akcje graczy.
Serwer spodziewa się pojedynczych plików zawierających JSON'owe array'e obiektów opisanych poniżej (przykładowy świat dostępny jest tutaj). Dodatkowo serwer zakłada, że wszelkie identyfikatory (`id` dla lokacji i przedmiotów oraz `nick` dla graczy) są **unikatowe**.

4.4.1 Gracze/NPC/Przeciwnicy

Stan gracza można zrozumieć jako następujący JSON:

```
{
    "nick" : nazwa_gracza,
    "stats" : {
        "health" : zdrowie,
        "strength" : sila,
        "toughness" : odpornosc
    },
    "inventory" : inventarz
}
```

- `nazwa_gracza` jest unikatową nazwą gracza identyfikującą go w świecie gry,
- `zdrowie` jest liczbą całkowitą określającą poziom zdrowia gracza (po osiąginięciu wartości <= 0 gracz ginie),
- `sila` jest liczbą całkowitą określającą siłę gracza, która odpowiada za siłę jego ataków,
- `odpornosc` jest liczbą całkowitą określającą wytrzymałość gracza, która odpowiada za odporność na ataki innych graczy,
- `inventarz` jest obiektem zawierającym ID przedmiotów posiadanych przez gracza:

```
{
    id_przedmiotu : nazwa_przedmiotu,
    ...
}
```

Wszystkie powyższe wartości, poza `nazwa_gracza` mogą ulegać zmianie w trakcie gry.

4.4.2 Lokacje

Stan lokacji przedstawia następujący JSON:

```
{
    "id" : id_lokacji,
    "name" : nazwa_lokacji,
    "description" : opis_lokacji,
    "players" : gracze_w_lokacji,
    "items" : przedmioty_w_lokacji,
    "locations" : drogi_do_innych_lokacji
}
```

- `id_lokacji` jest unikatowym indentyikatorem lokacji,
- `nazwa_lokacji` jest krótkim stringiem będącym nazwą lokacji,
- `opis_lokacji` zawiera krótki opis tego, co znajduje się w danej lokacji,
- `gracze_w_lokacji` jest array'em nazw graczy/NPC/przeciwników znajdujących się w danej lokacji,
- `przedmioty_w_lokacji` jest obiektem zawierającym ID przedmiotów znajdujących się w danej lokacji:

```
{
    id_przedmiotu : nazwa_przedmiotu,
    ...
}
```

- `drogi_do_innych_lokacji` jest obiektem zawierającym ścieżki do innych lokacji:

```
{
    droga_1 : id_lokacji_1,
    droga_2 : id_lokacji_2
}
```

...gdzie każda `droga` jest unikatową nazwą ścieżki a każde `id_lokacji` unikatowym identyfikatorem lokacji, na przykład:

```
{
    "north" : "starting_tavern",
    "south" : "deep_woods"
}
```

4.4.3 Przedmioty

Opis przedmiotów dostępnych w świecie przedstiawia nostępujący JSON:

```
{
    "id" : id_przedmiotu,
    "name" : nazwa_przedmiotu,
    "description" : opis_przedmiotu,
    "modifiers" : {
        "health" : zdrowie,
        "strength" : sila,
        "toughness" : odpornosc
    }
}
```

- `id_przedmiotu` jest unikatowym identyfikatorem przedmiotu,
- `nazwa_przedmiotu` to krótki string reprezentujący nazwę przedmiotu,

- `opis_przedmiotu` to krótki string opisujący przedmiot,
- `zdrowie` jest liczbą całkowitą określającą modyfikator zdrowia gracza,
- `sila` jest liczbą całkowitą określającą modyfikator siły gracza,
- `odpornosc` jest liczbą całkowitą określającą modyfikator wytrzymałości gracza,

4.5 AI - sztuczna inteligencja

Dzięki skryptowej naturze języka implementacji serwera możliwe było stworzenie prostego systemu sztucznej inteligencji opartego o moduły języka Erlang - tworzone są autonomiczne i lekkie maszyny stanów reprezentujące postaci sterowane przez AI.

4.5.1 Skrypty AI

Głównym zadaniem skryptów jest zapewnienie obsługi event'ów gry przy jednoczesnej modyfikacji zachowania w zależności od stanu maszyny. Obecnie wyróżniane są trzy stany - **friendly**, **neutral** oraz **hostile**. Możniwa jest także obsługa innych event'ów, które nie zależą od stanu, gdzięki czemu można osiągnąć bardzo skomplikowane zachowanie w stosunkowo prosty sposób.
Struktura skryptów jest następująca:

```
-module(some_ai_module).
-author('kajtek@idorobots.org').

-export([init/1, on_friendly/3, on_neutral/3, on_hostile/3, on_info/3]).
-include("mud_ai.hrl").

init(State) ->
    %% Inicjalizacja maszyny stanów.
    {ok, neutral, State}.

on_friendly(Name, _Args, State) ->
    %% Obsługa event'u Name w stanie Friendly.
    {ok, friendly, State}.

on_neutral(Name, _Args, State) ->
    %% Obsługa event'u Name w stanie Neutral.
    {ok, neutral, State}.

on_hostile(Name, _Args, State) ->
    %% Obsługa event'u Name w stanie Hostile.
    {ok, hostile, State}.

on_info(Info, StateName, State) ->
    %% Obsługa event'u Info nie zależącego od stanu maszyny.
    {ok, StateName, State}.
```

4.5.2 Wbudowane skrypty

W związku z ograniczeniem czasowym nałożonym na projekt do projektu dołączone jedynie jeden skrypt AI prezentujący jak największą ilość udostępnionych przez system AI funkcjonalności. Skrypt ten, `generic_npc` jest wbudowany w serwer i wykorzystywany jako skrypt domyśly dla każdej postaci nie będącej postacią gracza.