A kriptikus űrhajó kernel, avagy hogyan áll a FIPER projekt

A hálózaton három típusú **FIPER entitással** találkozunk:
- Kocsi
- Kliens
- Szerver

A cél, hogy a kliens irányítani tudja a kocsit és a kocsi videó streamjét eljuttassuk a klienshez. Ez a kapcsolat jelenleg kétféleképpen valósítható meg:
- Direkt kapcsolat: 1 kliens közvetlenül kapcsolódik 1 kocsihoz. Ez most LAN alapú, lásd észrevételek.
- Indirekt kapcsolat: a fenti kapcsolatot egy közbenső, központi szerver vezérli. Ez inkább internet alapú és eddig csak a szerver-kocsi kapcsolat van kész, a kliens-szerver kapcsolat még készül.

A **kommunikáció** jelenleg három csatornán zajlik, bár ez leredukálható, ha szükséges. A csatornák 1-1 TCP kliens socketként működnek a kocsin és szerver socketként a szerveren és a direkt kliensen.
- Message csatorna: parancsok küldözgetésére használható, kétirányú
- Stream csatorna: frame-ek küldésére használható, egyirányú, a kocsitól jön.
- RC csatorna: fel-le-jobbra-balra, stb. parancsok gyors küldésére használható, egyirányú, a kocsi felé megy.

Az RC és a Message összevonható, ha kell.

Probe:
Ez egy negyedik csatorna, ami egyrészt a fenti három csatorna kiépítéséhez, bootstrap-eléséhez kell a direkt kliens esetében, másrészt, mivel egy embernek több kocsija is lehet, szükséges a még fel nem kapcsolódott kocsik felderítése és azonosítása a hálózaton.

Parancsok:
A kocsi teljesen távirányítású, csak a Message csatornán érkező parancsokkal lehet irányítani, kikapcsolni, streamelést ki-be kapcsolni.
A szervernek van egy konzolja, ahova beírogathatunk egy vagy több szavas parancsokat. Ezek a help paranccsal elérhetőek, illetve az egyes parancsok a „help parancsnév” segítségével kiírják a hozzájuk kapcsolt függvények docstring-jét.

Tehát nagy vonalakban a következőképpen működik a dolog a direkt kliens esetén:
- Elindítjuk a klienst és a kocsit.
- A kocsit hallgatózni kezd a Probe csatornán.
- A kliens végezhet söprést (sweep) a hálón, amire visszakap egy listát azokkal a címekkel, ahol visszaszólt egy kocsi és a kocsi egyedi azonosítójával, nevével.
- A kliens a Probe csatornán küld egy „connect” üzenetet annak kocsinak, amelyikkel kapcsolódni akar.
- A kocsi kapcsolódik a három kommunikációs csatorna socket-jével a kliens szerver socketjeire:
  - Először a Message kapcsolódik. Itt a kocsi és a szerver kölcsönösen validálják magukat és a kocsi elküldi a kamerája felbontását (ez utóbbi nem kell, ha standardizáljuk a webkamerát).
  - Ha minden OK, kapcsolódik a Stream és az RC socket.
- Ezután a kocsi várja a parancsokat a klienstől, jelenleg az alábbiakat érti:
  - stream on/off
  - shutdown
- Az RC kapcsolaton még lehet bármit küldeni.

Kocsi részletezése

A **FIPER.car.carmain.py** script futtatásával indítható el, mely localhoston létrehoz egy kocsit.

A kocsi a következő alrendszerekből (osztályokból) épül fel:
FIPER.car.car.TCPCar csoportosít mindent. A mainloop()-ja fut a fő szálon, kiépíti a kapcsolatokat, majd átadja a futást a **Commander** parancsértelmezőnek.
FIPER.car.probeserver.ProbeServer: a Probe rendszert kezeli, a fő szálon fut, míg ki nem épül a három kommunikációs csatorna. TCP szerver socketet használ!
FIPER.car.channel.TCPSreamer: vezérli a kamera eszköz olvasását és a frame-ek küldését. Külön szálon fut. TCP kliens socketet használ!
FIPER.car.channel.RCReceiver: hallgatózik és várja az RC parancsokat. Külön szálon fut. Az RC parancsok leküldése hiányzik innen. TCP kliens socketet használ!
FIPER.car.component.Commander: a Message csatornán érkező parancsokat értelmezi, a fő szálon fut. Függ a **FIPER.generic.abstract.AbstractCommander**-től, amit a szerver konzolja is használ.
FIPER.generic.messaging.Messaging: a message csatorna implementációja. Generic-ben van, mert a szerver is ugyanezt az implementációt használja. 2 külön szálat indít. TCP kliens socketet használ!

Dikert kliens részletezése

Tesztelhető a **FIPER.client.direct.py** script futtatásával. Futáskor kiépíti a kapcsolatot a localhost-on lévő kocsival. Először streamet szed tőle, amit meg is jelenít. Enter leütésével ez megszakítható, utána random RC parancsokat küld neki. Ctrl-C-re szépen leépíti a kapcsolatot és lezárja a socketeket.

A direkt kliens a következő alrendszerekből (osztályokból) épül fel:
FIPER.client.direct.DirectConnection csoportosít mindent. A kliens/GUI példányosíthatja majd.
A konstruktora a kliens saját IP címét várja paraméterként. A következő metódusokon keresztül interfészelhető:

- **probe(ip)**: kideríti, hogy a megadott címen van-e kocsi és ha igen, annak mi az azonosítója.
- **sweep(*ips)**: ugyanaz, mint a probe, csak tetszőleges számú ip cím megadható neki. IP tartományokat is tud kezelni, pl. „192.168.1.1-100”.
- **connect(ip)**: „connect” üzenetet küld a kocsinak a Probe csatornán, mire a kocsi felkapcsolódik a kliensre.
- **get_stream(bytestream: *bool=False*)**: generátor függvény. Ha a bytestream paraméter igaz, akkor a videóframe-ek nyers bájtjait adja vissza, különben magukat a frame-eket 3 dimenziós numpy tömbökként.
- **display_stream()**: kijelzi a streamet opencv-vel. Kezdetleges, külön szálon fut.
- **stop_stream()**: leállítja a streamet.
- **teardown(sleep: *int=0*)**: ha van stream, leállítja. Ha van kocsi kapcsolat, azt leépíti. Felkészíti az osztályt a bezárásra. <sleep> másodpercet altatja a szálat (ha kell).
- **rc_command(*commands)**: parancsokat fogad, összefűzi és elküldi őket a kocsinak.

FIPER.generic.abstract.AbstractListener: végzi a kliens oldalán a kliens-kocsi kapcsolat felépítését.
FIPER.generic.probeclient.Probe: a probe csatorna használatához kell.
FIPER.generic.subsystem.StreamDisplayer: a kocsi stream kijelzését végzi opencv-vel.
FIPER.generic.interface: ez a modul külön is részletezve lesz.

A szerver részletezése:

A szerver a **FIPER.host.servermain.py** futtatásával indítható, mely létrehoz egy szervert localhoston és elindítja a szerver konzolt. A kocsival való kapcsolódást a szerver kezdeményezi a connect <IP > paranccsal (lásd a szerver konzolból elérhető parancsoknál).

A szerver feladata, hogy fogadja a kliens és kocsi kapcsolatokat és összekösse a megfelelő kocsival a megfelelő klienst, utána pedig továbbítsa a kocsitól a videó feedet a kliensnek és a klienstől továbbítsa az RC parancsokat a kocsinak.
A szerver még nincs kész, a kliens-szerver kapcsolat még nincs tesztelve. A teszteléséhez még meg kell írni a kliens indirekt kapcsolatot.

- **FIPER.host.bridge.FleetHandler**: a fő szerver osztály, konstruktora a saját IP címét várja. Példányosítás után meg kell hívni a mainloop() metódusát, ami bedob minket a szerver konzolba (lásd alább). A **FleetHandler** következő osztályokat használja:
- **FIPER.host.component.Listener**: hallgatózik egy Message szerver socketen és bejövő kapcsolatokat vár. A bejövő kapcsolatokat átadja egy InterfaceFactory-nak. mely FIPER entitás típus szerint külön szótárba pakolja őket (lásd az **Interface részletezésénél).** Leszármazik a **FIPER.generic.abstract.AbstractListener** osztályból, amin a direkt kapcsolattal osztozik. Külön szálon fut.
- **FIPER.generic.probeclient.Probe**: a probe protokoll használatához kell, de szerver esetén nem biztos, hogy szükség lesz rá. Mindenesetre most bent van.
- **FIPER.generic.interface.InterfaceFactory**: példányosítja a megfelelő interfészt bejövő kapcsolat esetén (_ClientInterface vagy _CarInterface).
- **FIPER.host.component.Console**: a szerver elindításakor egy konzolba kerülünk, amin keresztül parancsokat adhatunk meg. A parancsok a help beírásával érhetőek el, az egyes parancsok helpjei pedig a help parancsnév segítségével.

A szerver konzolból elérhető parancsok (a kacsacsőr zárójeleket nem kell beírni):

- **cars**: kiírja a kocsik ID-jét
- **kill <ID>**: szétkapcsolja a megadott ID-jű entitást (egyelőre csak kocsikat kezel)
- **watch <ID>**: a megadott kocsinak bekapcsolja a stream-jét és kijelzi openCV-vel egy külön szálon.
- **unwatch <ID>**: leállítja a streamet, kilövi szálat.
- **status**: kiír egy státusz jelentést a szerverről
- **message <ID>**: tetszőleges üzenetet küldhetünk egy FIPER entitásnak
- **probe <IP>**: probe-olja a megadott IP címet
- **connect <IP>**: kapcsolatot kér a Probe csatornán keresztül adott IP-n lévő kocsitól
- **sweep <IP range>**: adott IP range-et végigpásztáz probe üzenettel. Táblázatosan kiírja, hogy adott IP címről milyen ID jött vissza. Pl. sweep 192.168.1.1-10.
- **shutdown**: szétkapcsol és leállít minden kocsit, minden streamet, leállít minden szálat, bezár minden socketet és kikapcsol.
- **help**: kiírja az elérhető parancsokat.

A konzol lényegében egy szótárat használ, aminek a kulcsa a parancs neve és az értéke egy, a parancshoz tartozó függvény vagy metódus referenciája. Paraméteres parancs beírásakor a **commands[commandname](*args)** utasítás fut le, ahol **commands** az előbb leírt szótár, **commandname** a parancs első szava, ***args** pedig a parancs második, harmadik, stb. szavai.

A szerver sok szálat kezel, entitásonként 2-4 darabot + a Listener is hallgatózik.

Interface részletezése:

A FIPER entitásokkal való kommunikációra interface osztályok készültek. Pl. a kocsival való kommunikációt a **FIPER.generic.interface._CarInterface** osztály végzi, ezt az osztályt példányosítja a direkt kliens és a szerver is. Interface készült még a kliensre is, de az még nincs tesztelve.
A konkrét interface osztályok privát osztályok a modulban, egy InterfaceFactory osztályon keresztül lehet példányosítani őket. Ennek az az oka, hogy előzetesen nem feltétlenül ismert, hogy egy adott FIPER entitás a hálózaton kocsi vagy kliens, illetve kód duplikációt lehetett így elkerülni.
FIPER.generic.interface.InterfaceFactory: konstruktora a következő paramétereket várja:

- **msock**: egy összekapcsolt Message socket. Használja is és tovább is adja a konkrét Interface konstruktornak.
- **dlistener**: egy hallgatózó TCP socket a stream kapcsolat felépítéséhez. Továbbadja a konkrét Interface konstruktornak.
- **rclistener**: egy hallgatózó TCP socket az RC kapcsolat felépítéséhez. Továbbadja a konkrét Interface konstruktornak.

get(): felépíti a kapcsolatot és visszatér a megfelelő interfész példánnyal.
A kapcsolat felépítése a következőképpen zajlik [zárójelben az InterfaceFactory megfelelő privát metódusainak neve, melyeket a get() hív]
A message kapcsolaton a FIPER entitástól fogadjuk a bemutatkozást [**_read_introduction()**], mely validálásra kerül [**_valid_introduction()**]. Ha valid, visszaküldünk egy „HELLO”-t a Message csatornán. Itt lehetőségünk lenne szintén azonosítani magunkat a kocsi felé, egyelőre a „HELLO” egy placeholder. Szétszedjük a bemutatkozást [**_parse_introductory_string()**], innen megtudjuk, hogy a FIPER entitás kocsi-e vagy esetleg kliens, megtudjuk az egyedi azonosítóját és kocsi esetén megtudjuk a videóframe-jeinek a méretét, ami kell majd a kijelzéshez (ez kivehető, ha standardizáljuk a webkamerát).
Ha minden OK, mindent tudunk, akkor példányosítjuk a megfelelő konkrét Interface osztályt és visszatérünk az objektummal.

Messaging részletezése:
A három kommunikációs csatorna közül a messaging külön kifejtést igényel. Bonyolultabb a többinél, mivel ez kétirányú kommunikációt tesz lehetővé. Van egy fogadó és egy küldő puffere a bejövő és kimenő üzeneteknek. A Message socket olvasása és írása egy-egy külön szálon történik.
FIPER.generic.messaging.Messaging: konstruktora a következő paramétereket várja:

- **conn**: egy felépített Message TCP socket.
- **tag**: egy bináris (nem unicode) string, amit minden elküldött üzenet elejére automatikusan odabiggyeszt.
- **sendtick**: float vagy int, a küldési időköz, lást **send()**.

Használható interfész:

- **connect_to(IP, tag, timeout)**: alternatív konstruktor, ami a megadott IP címmel felépít egy message kapcsolatot kliensként. Tag ugyanaz, mint a normál konstruktornál, timeout opcionálisan megadható, tovább passzolja a socket.create_connection() függvénynek, ami a python socket library-jében van, lásd a dokumentációját ott.
- **send(*msgs)**: a megadott üzeneteket ellátja tag-gel és a végükre a „ROGER” stringet fűzi, majd hozzáadja a küldő pufferhez. Az üzeneteket a kimenő szál kiküldi adott időközönként (**sendtick**).
- **recv(n=1, timeout=0)**: n darab üzenetet lekér a fogadó pufferből. Ha nincs benne üzenet, timeout másodpercet vár, ha ezután is timeout-ol, akkor None-nal tér vissza.
- **teardown(sleep=0)**: leállítja a szálakat és bezárja a Message socketet, utána **sleep** másodpercet vár.

FIPER.generic.abstract részletezése

AbstractCommander: absztrakt osztály parancsértelmezéshez. Használja a szerver konzol, és a kocsi parancsértelmezője. Egy szótárban tárolja a parancs nevét és a hozzá kapcsolt függvényt/metódust. Paraméteres parancsokat is tud értelmezni.
Implementálni kell a **read_cmd**() metódusát a leszármazottainak, mely visszatér egy
(parancs, (argumentumok...)) beágyazott kettős tuple-lel.
AbstractListener: absztrakt osztály, mely FIPER entitások kapcsolódási kérelmét várja. A leszármazottaknak implementálni kell a **callback(msock)** metódusát, mely a bejövő kapcsolat Message socketjét várja paraméterként.

FIPER.generic.const részletezése

STREAM_SERVER_PORT
MESSAGE_SERVER_PORT
CAR_PROBE_PORT
RC_SERVER_PORT
A három kommunikációs csatorna (Message, Stream, RC) és a Probe csatorna által használt portok (az adott csatorna szerver socketjét ide bind-oljuk)

FPS: mintavételezés gyakorisága a kameráról
DTYPE: a stream adat numpy típusa. Javasolt a uint8, ami 0-255-ig tartalmazza a számokat és az RGB legtömörebb reprezentációja.

FIPER.generic.routine részletezése

Hasznos függvények
white_noise(shape): fehér zaj frame-eket ad vissza, ha nincs elérhető video eszköz.
my_ip(): megbízhatatlan hack, amivel ki lehet deríteni a lokális IP címet.
srvsock(ip, channel, timeout=None): szerver socketet csinál a megadott csatornához, pl. srcsocket(„127.0.0.1”, „rc”) visszaad egy bindolt, hallgatózó socketet.

FIPER.generic.util részletezése:
CaptureDeviceMocker: mock-olja a cv2 device objektumot, ami a webkamerához kapcsolódik, hogy ott is tudjunk tesztelni, ahol nincs cam.
Table: szép ASCII táblázatot lehet vele rajzolni. IP címek sweep-elésekor használj a szerver a találatok kiírására.

Javaslatok, észrevételek:

A szerver konzol prompt eltűnik, „lemarad”, ha a háttérben futó szálak konzolra printelnek. Szóval ha nincs prompt a szerver konzolban, ütni kell egy üres entert és újra megjelenik.

A kocsi kapcsolat most LAN alapon van, de elvileg úgy van faktorálva, hogy ezt át lehet vinni WiFi-re. WiFi alapon valószínűleg bukjuk a platform függetlenséget, bár ez nem biztos, hogy baj, mivel a kocsi fixen Linux, a kliens meg fixen Windows. Ha a szerver is fixen Linux, akkor meg fogjuk tudni oldani.

Ha minden igaz, a kód OS független, de csak Python 2-n fut.
Ha valami nem OS független és elszáll, az bug és/vagy büdös kód.

A kocsi, kliens, szerver eléggé függetlenek egymástól, közös cuccok mind a generic-ben vannak. Érdemes lenni szétszedni ezt a 4 modult 4 repóra és akkor lenne FIPER-car, FIPER-client, FIPER-host, FIPER-generic és külön tervezhetnénk mindegyikre a munkát.

Necces a scriptek futtatása. Pythonban nem lehet egy scriptből „felfelé” importálni, tehát ha van egy futtatható scriptem A/B/script.py elérési úttal és futtatom, akkor nem importálhatok A/asd/-ból, csak A/B-ből (a script mellől) és A/B/C, stb. alkönyvtárakból (lefelé importálás).
Sajnos a generic modult el kell érni mindenkinek, ezért jelenleg a FIPER könyvtárat tartalmazó parent könyvtárat hozzá kell adni a PYTHONPATH környezeti változóhoz és az __init__.py is emiatt van a FIPER gyökérben.
Ez feloldható, ha a FIPER-generic-et külön projektként kiszedjük a repóból és Python könyvtárként fejlesszük tovább.
Feloldható úgy is, ha a car, client, host könyvtárakba bemásolunk egy-egy példányt a genericből.