Számítógépes Hálózatok

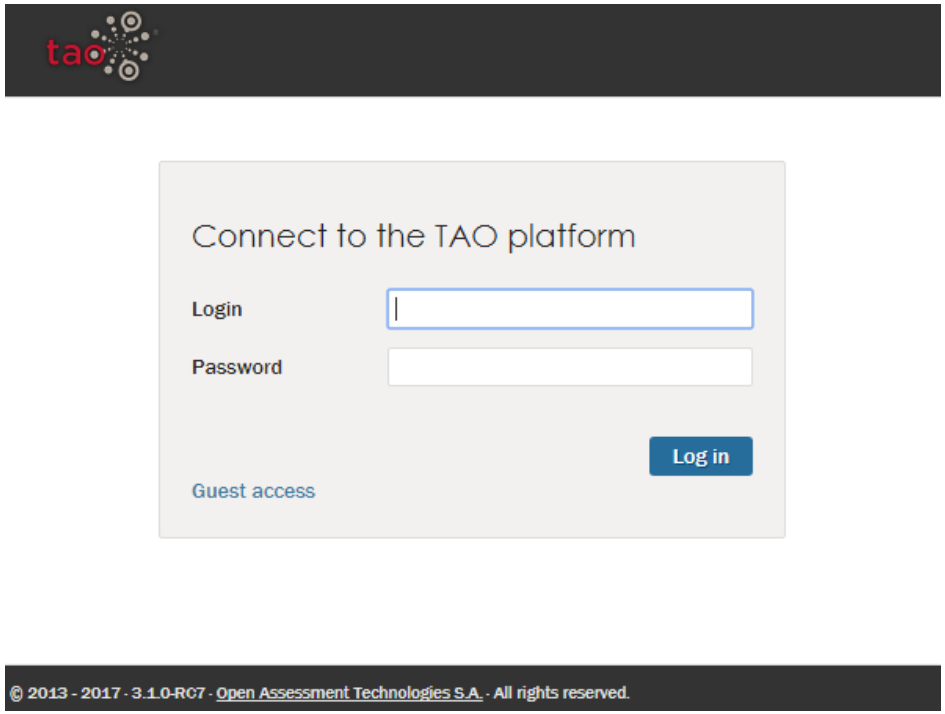
3. gyakorlat

Elérhetőségek

- honlap: http://szalaigj.web.elte.hu/
- email: szalaigindl@inf.elte.hu
- szoba: 2.507 (déli tömb)

Óra eleji kisZH

- Elérés:
  - https://oktnb16.inf.elte.hu



Gyakorlat tematika

- Struktúra küldése binárisan
- UDP
- Multicast
- SELECT

A struct modul

– A struct.**pack**(*fmt*, *v1*, *v2*, …): egy sztringgel tér vissza, amely az adott *fmt* formátumnak megfelelően csomagolja be a bemenetként kapott értéke(ke)t (binárissá alakítja)
  - pl. formátumoknál a 'b' (előjeles) char C-típust (1-bájtos egész),
  - a '4sL' 4 méretű char tömböt és egy (előjel nélküli) long C-típust (4-bájtos egész) jelöl

Struktúraküldése

- Binárissá alakítjuk az adatot
  - A Struct konstruktorában a formátumot adjuk meg, - hasonlóan az előbbihez, - amely alapján írja/olvassa a bináris adatot
  - A *-operátor az alábbi esetben úgy fog viselkedni, mintha ','-vel elválasztva felsoroltuk volna a *values* elemeit

```
import struct
values = (1, 'ab', 2.7)
packer = struct.Struct('I 2s f')                #Int, char[2], float
packed_data = packer.pack(*values)
```

- Visszaalakítjuk a kapott üzenetet

```
import struct
unpacker = struct.Struct('I 2s f')
unpacked_data = unpacker.unpack(data)
```

- megj.: integer 1 – 4 byte, sztringként 1 byte, azaz hatékonyabb sztringként átküldeni.

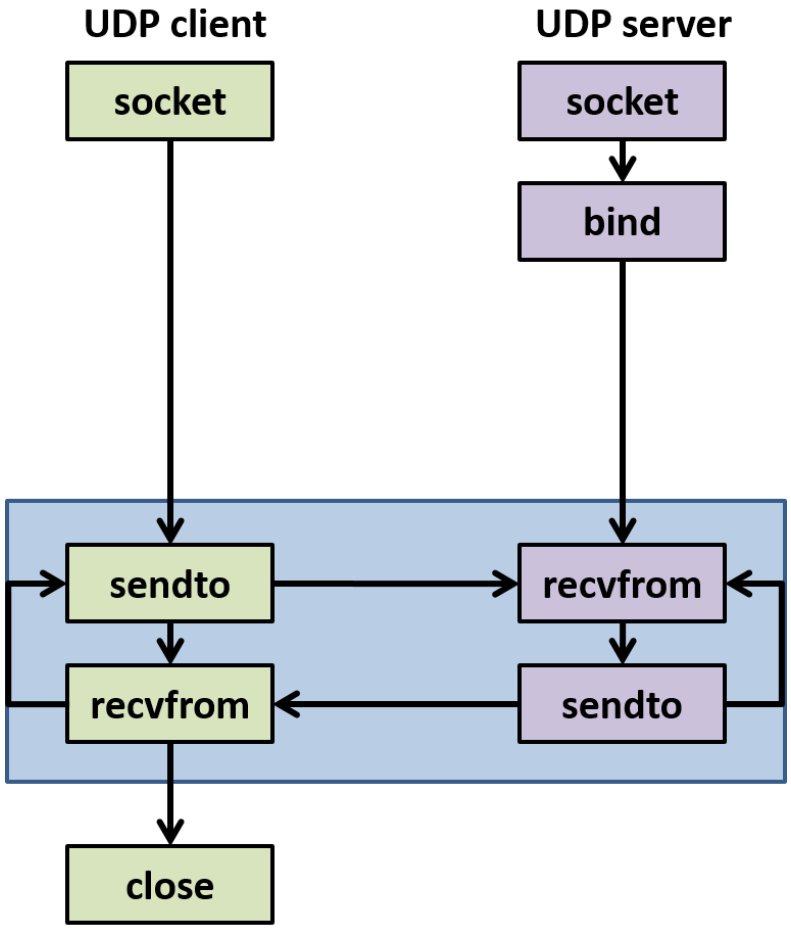
Feladat 1

- Készítsünk egy szerver-kliens alkalmazást, ahol a kliens elküld 2 számot és egy operátort (négy alapművelet közül) a szervernek, amely kiszámolja és visszaküldi az eredményt. A kliens üzenete legyen struktúra.

A kommunikációs csatorna kétféle típusa

- Kapcsolat-orientált modell (analógia: telefonbeszélgetés)
  - csomagok megérkeznek jó sorrendben
  - ilyen protokoll a TCP
  - kapcsolódó típus: stream socket
- **Kapcsolat-nélküli modell (analógia: postai levelezés)**
  - **csomagok nem biztos, hogy sorrend helyesen érkeznek, sőt el is veszhetnek**
  - **előnye a jobb teljesítmény**
  - **ilyen protokoll a UDP**
  - **kapcsolódó típus: datagram socket**

UDP

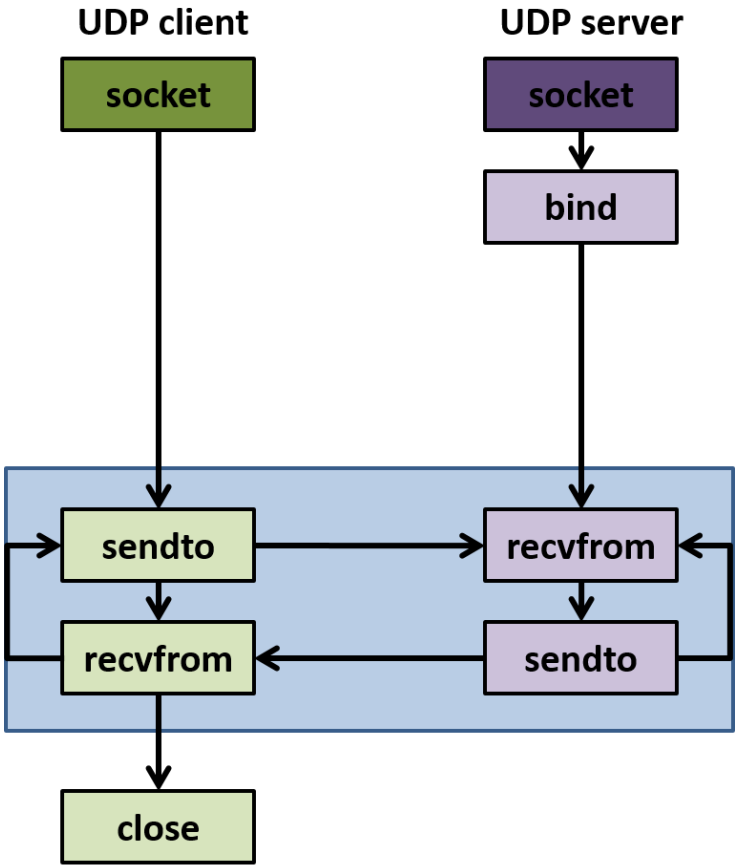

UDP client
UDP server
socket
socket
bind
sendto
recvfrom
recvfrom
sendto
close

Socket leíró beállítása

- socket.**socket**([*family*
  [, *type*
  [, *proto*]]])
- *family* : socket.AF_INET → IPv4 (AF_INET6 → IPv6)
- ***type* : socket.SOCK_DGRAM → UDP**
- *proto* : 0

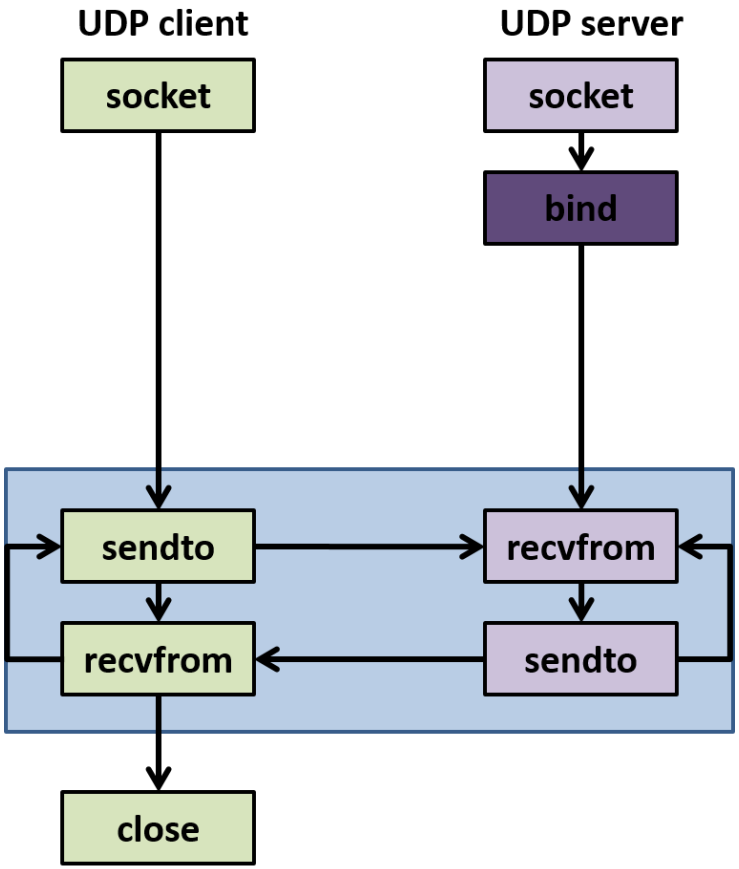
(alapértelmezett protokoll lesz)

- visszatérési érték: egy socket objektum, amelynek a metódusai a különböző socket rendszer hívásokat implementálják



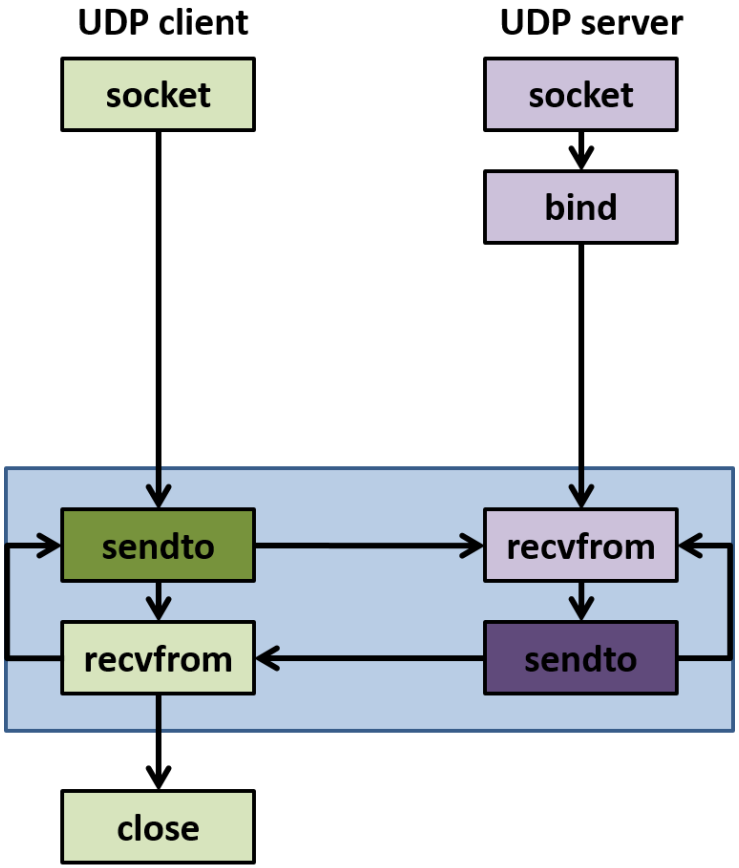
Bindolás – ismétlés

- socket.socket.**bind**(*address*)
- A socket objektum metódusa
- *address* : egy tuple, amelynek az első eleme egy hosztnév vagy IP cím (sztring reprezentációval), második eleme a portszám



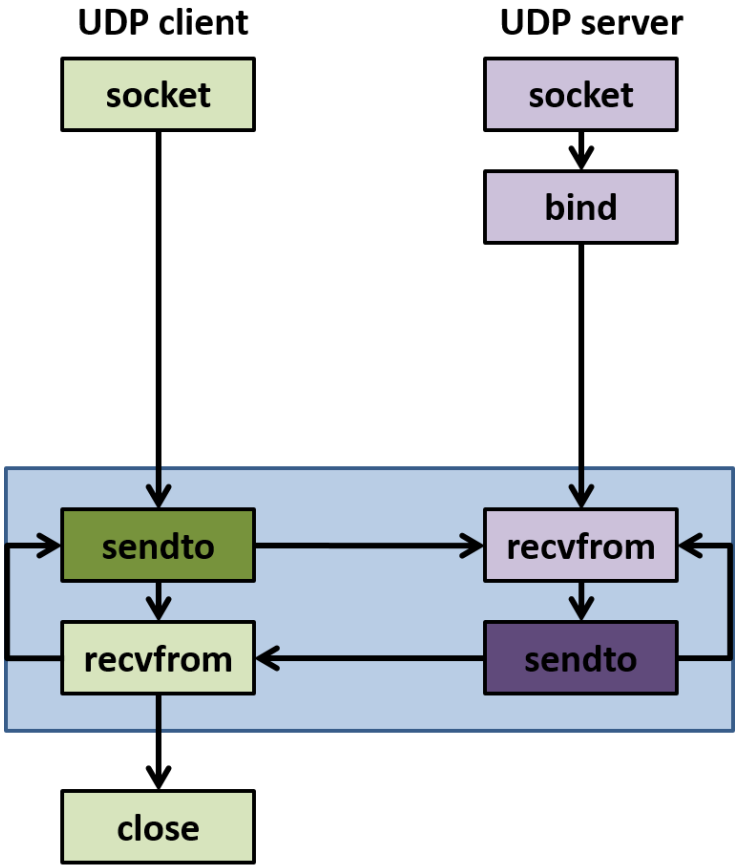
sendto

- socket.socket.**sendto**(*string* , ***address***)
- socket.socket.**sendto**(*string* , *flags* , ***address***)
- A socket objektum metódusai
- Adatküldés (*string*) a socketnek
- *flags* : 0 (nincs flag meghatározva)
- **A socketnek előtte nem kell csatlakozni a távoli sockethez, mivel azt az *address* meghatározza**



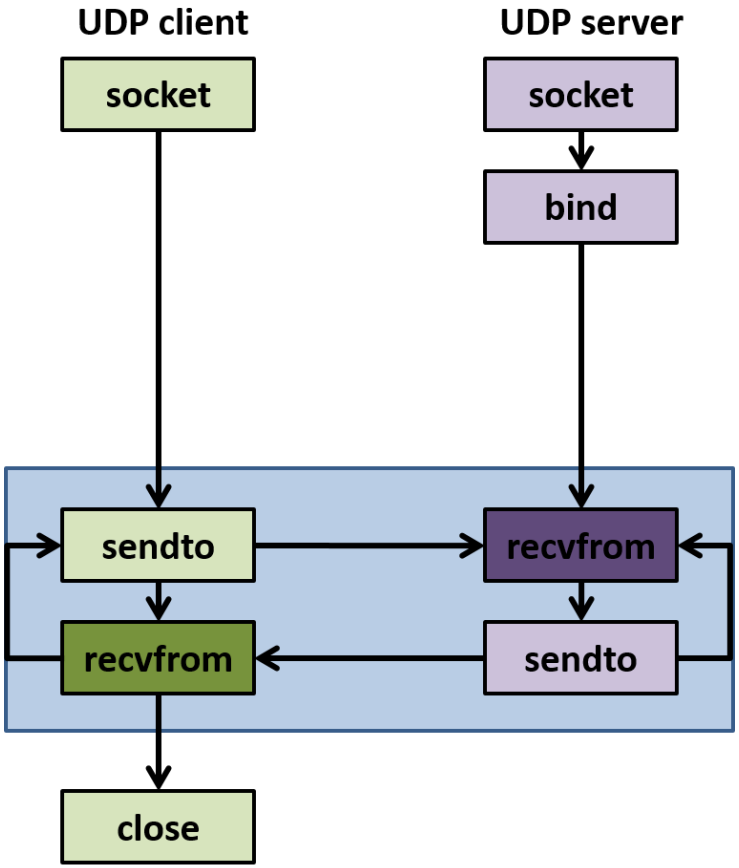
sendto

- **Fontos, hogy egy UDP üzenetnek bele kell férni egy egyszerű csomagba (ez IPv4 esetén kb. 65 KB-ot jelent)**
- visszatérési érték: az átküldött bájtok száma
  - az alkalmazásnak kell ellenőrizni, hogy minden adat átment-e
  - ha csak egy része ment át: újra kell küldeni a maradékot



recvfrom

- socket.socket.**recvfrom**(*bufsize [, flags]*)
- A socket objektum metódusa
- Üzenet fogadása
- *bufsize* : a max. adatmennyiség, amelyet egyszerre fogadni fog
- *flags* : 0 (nincs flag meghatározva)
- **visszatérési érték: egy (*string*, *address*) tuple, ahol a fogadott adat sztring reprezentációja és az adatküldő socket címe szerepel**



UDP

- recvfrom()

```
data, address = sock.recvfrom(4096)
```

- sendto()

```
sent = sock.sendto(data, address)
```

Feladat 2

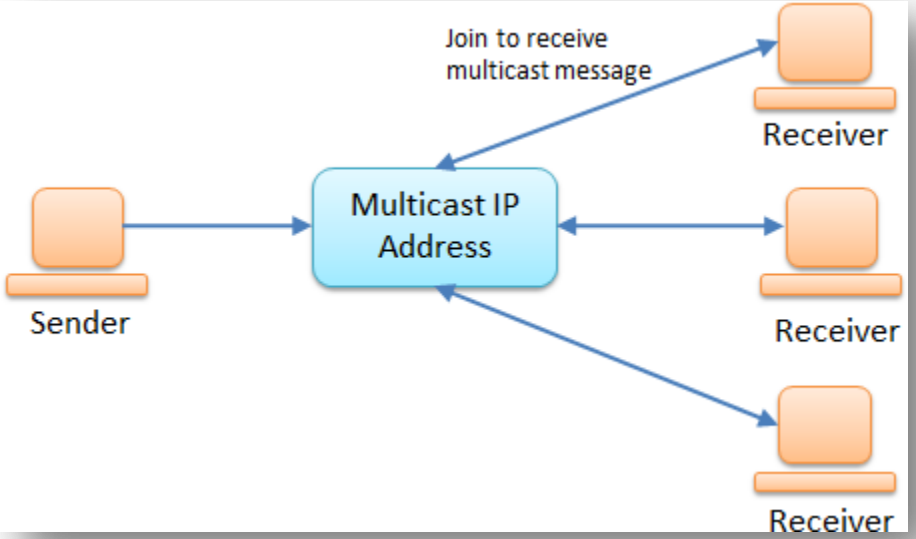
- Készítsünk egy kliens-szerver alkalmazást, amely UDP protokollt használ. A kliens küldje a ‚Hello Server' üzenetet a szervernek, amely válaszolja a ‚Hello Kliens' üzenetet.

Socket beállítása

- socket.**setsockopt**(*level*, *optname*, *value*): az adott socket opciót állítja be
- Általunk használt *level* értékek az alábbiak lesznek:
  - socket.**IPPROTO_IP**: jelzi, hogy IP szintű beállítás
  - socket.**SOL_SOCKET**: jelzi, hogy socket API szintű beállítás
- Az *optname* a beállítandó paraméter neve, pl.:
  - socket.**SO_REUSEADDR**: a kapcsolat bontása után a port újrahasznosítása
- A *value* lehet sztring vagy egész szám:
  - Az előbbi esetén biztosítani kell a hívónak, hogy a megfelelő biteket tartalmazza (a struct segítségével)
  - A socket.**SO_REUSEADDR** esetén ha 0, akkor lesz hamis a „tulajdonság”, egyébként igaz

Multicast

- A pont-pont összeköttetés sokféle kommunikációs igényt ki tud szolgálni



- Ugyanazt az infót külön-külön elküldeni a társaknak nem optimális az erőforrás kihasználtság szempontjából
- A multicast egy időben több végpontnak is tudja szállítani az üzenetet → jobb hatékonyság

Multicast

- A multicast üzenetek küldésénél **UDP-t** használunk
  - (a TCP végpontok közötti kommunikációs csatornát igényel)
- Egy IPv4 címtartomány van lefoglalva a multicast forgalomra
  - (224.0.0.0-230.255.255.255)
- Ezeket a címeket speciálisan kezelik a routerek és switchek

Multicast üzenet küldése

- Ha a multicast üzenet küldője választ is vár, nem fogja tudni, hogy hány db. válasz lesz
- → időtúllépési értéket állítunk be, hogy elkerüljük a blokkolást a válaszra történő határozatlan idejű várakozás miatt:

```
sock.settimeout(0.2) # 0.2 sec.
```

Multicast üzenet küldése

- Továbbá életidő (Time To Live (TTL)) értéket is szükséges beállítani a csomagon:
  - A TTL kontrollálja, hogy hány db. hálózat kaphatja meg a csomagot
    - „Hop count”: a routerek csökkentik az értékét, ha 0 lesz → eldobják a csomagot
  - A **setsockopt** függvény segítségével majd a socket.**IP_MULTICAST_TTL**-t kell beállítani

Multicast üzenet fogadása

- A fogadó oldalon szükség van arra, hogy a socket-et hozzáadjuk a multicast csoporthoz:
  - A **setsockopt** segítségével az **IP_ADD_MEMBERSHIP** opciót kell beállítani
  - A socket.**inet_aton**(*ip_string*): az IPv4 cím sztring reprezentációjából készít 32-bitbe csomagolt bináris formátumot
  - Meg lehet adni azt is, hogy a fogadó milyen hálózati interfészen figyeljen, esetünkben most az összesen figyelni fog: socket.**INADDR_ANY**

Multicast üzenet fogadása

- socket.**INADDR_ANY** a bind hívásnál is lehet használni
  - Ott az '' (üres) sztring reprezentálja → a socket az összes lokális interfészhez kötve lesz
- Nem mindenhol tudunk kötni egy multicast címre
  - Nem minden platform támogatja, a Windows az egyik ilyen
  - Ilyenkor: „socket.error: [Errno 10049] The requested address is not valid in its context” hiba jön
  - Kénytelenek vagyunk ebben az esetben az **INADDR_ANY**-t használni, viszont az fontos, hogy a portnak **a szerver által használt portot adjuk meg**
  - (localhost-tal nem működne, mert akkor a multicast hálózatot nem tudjuk elérni)

Példa hívások multicast-nál

- setsockopt() (sender)

```
ttl = struct.pack('b', 1) # '\x01'
sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, ttl)
```

- socket hozzávétele a multicast grouphoz (recv)

```
multicast_group = '224.3.29.71'
group = socket.inet_aton(multicast_group)                    # '\xe0\x03\x1dG'
mreq = struct.pack('4sL', group, socket.INADDR_ANY)          # '\xe0\x03\x1dG\x00\x00\x00\x00'
sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
```

Feladat 3

- Készítsünk egy multicast fogadó és küldő alkalmazást!
- Először csak a saját gépünkön fusson a szerver és a kliens is (TTL értéke 1)
- Majd a tanári gépen fogjuk futtatni a szervert, és mindenki feliratkozhat rá (TTL értéke 2)

Select

- Több socketet is szeretnénk egy időben figyelni (a bejövő kapcsolódásokra és a meglevő kapcsolatokból való olvasásra is)
- Probléma: accept és a recv függvények blokkolnak
- Hatékonyabb megoldást szeretnénk, mint a socket timeout használatával egy folyamatos lekérdező ciklus
- → A select fv. segítségével a monitorozás az op. rsz. hálózati rétegében történik

Select

- select.**select**(*rlist, wlist, xlist[, timeout]*)
- Az első három argumentum a „várakozó objektumok” listái:
  - *rlist*: a socketek halmaza, amelyek várakoznak, amíg készek nem lesznek az olvasásra
  - *wlist*: … készek nem lesznek az írásra
  - *xlist*: … egy „kivétel” nem jön
- Az opcionális *timeout* argumentum mp.-ben adja meg az időtúllépési értéket
  - (ha ez nincs megadva → addig blokkol, amíg az egyik socket kész nincs)

Select

- select.**select**(*rlist, wlist, xlist[, timeout]*)
- Visszatér három listával:
  1. visszadja a socketek halmazát, amelyek készek az olvasásra (adat jön)
  2. … készek az írásra (szabad hely van a pufferükben, és lehet írni oda)
  3. … amelyeknél egy „kivétel” jön

Select

- Az „olvasható” socketek három lehetséges esetet reprezentálhatnak:
  - Ha a socket a fő „szerver” socket, amelyiket a kapcsolatok figyelésére használunk → az „olvashatósági” feltétel azt jelenti: kész arra, hogy egy másik bejövő kapcsolatot elfogadjon
  - Ha a socket egy meglévő kapcsolat egy klienstől jövő adattal → az adat a recv() fv. segítségével kiolvasható
  - Ha az előző, de nincs adat → a kliens szétkapcsolt, a kapcsolatot le lehet zárni

Példa hívások select-nél

- setblocking() vagy settimeout()

```
connection.setblocking(0)      # or connection.settimeout(0.0)
connection.setblocking(1)      # or connection.settimeout(None)
```

- select()

```
inputs = [ server ]
outputs = [ ]
timeout=1
readable, writable, exceptional = select.select(inputs, outputs, inputs,timeout)
…
for s in readable:
      if s is server:    #new client connect
            ….
      else:
            ….          #handle client
```

Queue – szálbiztos FIFO konténer

- A Queue python modul egy FIFO implementáció, ami megfelelő a többszálúsághoz
- A Queue.**Queue** osztály az alap FIFO konténert valósítja meg
- A sor végére a **put()** függvénnyel helyezzük az elemeket
- Az elejéről a **get()** függvénnyel szedjük le,
- vagy a **get_nowait()**-tel,
  - amely nem blokkol, azaz nem vár elérhető elemre,
  - és kivételt jön, ha üres a sor

Feladat 4

- Készítsünk egy TCP chat alkalmazást, amelyen több kliens képes beszélni egymással, egy közös felületen.

VÉGE
KÖSZÖNÖM A FIGYELMET!