Twisted és Python alapú eseményvezérelt programozás

Mielőtt kiszolgáló alkalmazásunkat ezernyi folyamatból álló szörnyeteggé vagy kusza szálak gombolyagává változtatnánk, ismerjünk meg egy letisztult, logikus, eseményvezérelt módszert. Töltsük le a proxykiszolgálót mindössze 600 sorban megvalósító példaprogramot, és próbáljuk magunkévá tenni szellemiségét!

Kezdetekben a kiszolgálók `fork` hívásokkal osztódtak. Utánuk a többszálú kiszolgálók következtek. A több példányban vagy szállal futó kiszolgálók kisebb számú párhuzamos kapcsolatot jól kezelnek, ám amikor a kapcsolatok száma több százra vagy több ezerre nő, akkor túlságosan sok erőforrásigényes folyamatot indítanak ahhoz, hogy hatékonyan működhessenek. Napjainkra kialakult egy jobb megoldás, ezek az aszinkron kiszolgálók. Az eseményvezérelt programozás egykor bonyolult világát mára harmadik generációs programozási nyelvek keretrendszerei teszik barátságossá.

A *Python* közösség egyik fényesen ívelő csillaga a *Twisted*, mely egyszerűvé és elegánssá teszi az aszinkron programozást, miközben eseményvezérelt segédosztályok széles választékát is rendelkezésre bocsátja. Írásomban az aszinkron, eseményvezérelt programozásról és *Twisted* alapú megvalósításáról lesz szó. Ha csak elmélkedünk a kódról, azzal nem jutunk messzire, ezért egy valódi, kifejezetten a cikk céljaira készített *Twisted* alkalmazásból fogok példákat mutatni. Egy egyszerű proxykiszolgálóról van szó, amely letiltja a nem kívánt sütiket, képeket és kapcsolatokat. A teljes forráskód beszerzésével kapcsolatban a források között szerepel részletesebb tájékoztatás.

Mi a Twisted?

A *Twisted Project*, mint hálózati alkalmazások készítésére használható, nagyteljesítményű, egyre üzembiztosabb megoldás növekvő népszerűségnek örvend. Alapjában véve a *Twisted* egy aszinkron hálózatkezelő keretrendszer. A hasonló keretrendszerektől eltérően a *Twisted* a fontosabb protokollok és programozási feladatok kezeléséhez, például a felhasználók hitelesítéséhez vagy éppen a távoli objektumközvetítéshez számos beépített könyvtárral rendelkezik. A *Twisted* mögött álló filozófiák egyike az eszközkészletek közötti hagyományos határok eltörlése; például ugyanaz a kiszolgáló webes tartalom szolgáltatására és *DNS*-lekérdezések feloldására egyaránt alkalmas lehet.

Noha maga a csomag meglehetősen nagy, az alkalmazásoknak nem muszáj a *Twisted* minden összetevőjét beemelniük, vagyis a futtatási többletterhelés elenyésző.

Ahogy a *Python*, úgy a *Twisted* felhasználói bázisa is az oktatási szektor felől terjeszkedett a kereskedelmi és a kormányzati szereplők felé. A *Zotonál* mi a *Twistedet* egy elosztott fényképtároló és -kezelő alkalmazásban használjuk, általa ugyanis rövid idő alatt, közismerten termelékeny nyelven – ez volna a *Python* – tudunk méretezhető hálózati alkalmazásokat készíteni. Napi programozási munkám során újra és újra elismerést vált ki belőlem a *Twisted* lenyűgöző eszközkészlete és a mögötte álló közösség segítőkészsége. A más közösségi, nyílt forrású tervezetekhez hasonlóan *Twisted* üzleti eszközként is biztonságosan alkalmazható, léte ugyanis nem egyetlen vállalat vagy intézet támogatásának függvénye.

Mi az aszinkron programozás?

Ugye ismerős a helyzet, hogy sorban állunk egy vegyesbolt gyorspénztáránál, egyetlen üveg ásványvízzel, az előttünk lévő viszont megkérdőjelezi valamelyik tétel árát, és a mögötte állók mindannyian hosszú percekig kénytelenek várakozni az ár ellenőrzésére? Az aszinkron programozást számtalan módon szokták magyarázni, ám szerintem a legszemléletesebb példa az, hogy sorban állunk egy tétlen pénztárosnál. Ha a pénztáros aszinkron módon dolgozna, akkor megtehetné, hogy félreállítja az előttünk lévő személyt, és kiszolgál bennünket, amíg az ár ellenőrzése be nem fejeződik. Sajnos a pénztárosok elég ritkán működnek aszinkron módban. A szoftverek világában ugyanakkor az eseményvezérelt kiszolgálók gazdálkodnak a legjobban a rendelkezésre álló erőforrásokkal, ugyanis esetükben nincsenek értékes memóriát foglaló, a foglalatok forgalmára várakozó szálak. Maradva a vegyesbolt példájánál, a többszálú kiszolgáló további pénztárosok munkába állításával próbálja csökkenteni a sorban állás idejét, az aszinkron megoldás viszont lehetővé teszi, hogy egy-egy pénztáros egyszerre több vevőnek is segítsen.

1. kódrészlet A Twisted példakiszolgáló elküldi az időt, majd lezárja a foglalatot

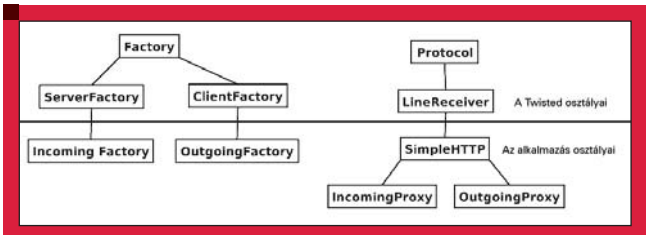
```
import time
from twisted.internet import protocol, reactor
class TimeProtocol(protocol.Protocol):
    def connectionMade(self):
        self.transport.write(
            'Szia! A pontos idő: %s' %
            ➥time.time())
        self.transport.loseConnection()

class TimeFactory(protocol.ServerFactory):
    protocol = TimeProtocol
reactor.listenTCP(1100, TimeFactory())
reactor.run()
```

Természetesen szó sincs arról, hogy a többszálú modellnek ne lennének előnyei. Például mikroszálak alkalmazásával az egyes szálak által lefoglalt erőforrások mennyisége lényegesen csökkenthető. Az aszinkron programozás szükségszerűen bonyolult, különösen, ha egymást követő blokkoló műveleteket kell végeznünk. ***Python*** alatt a szálakra osztásból fakadó előnyöket csökkenti a ***Python Global Interpreter Lock (globális értelmező zár, GIL)***.
A többszálú programozás ***Python*** alatt üdítően egyszerű, ugyanis a ***Python*** belső műveletei kivétel nélkül „szálbiztosak". Ha hozzá akarunk adni egy elemet egy listához, vagy be akarunk állítani egy szótárkulcsot, akkor semmilyen zárra nincs szükségünk ahhoz, hogy el tudjuk kerülni a szálak közötti versenyhelyzeteket. Mindezt egy a teljes értelmezőre kiterjedő zár teszi lehetővé, melyet sajnos a ***Python*** értelmezője meglehetősen nagylelkűen használ. Vagyis egyszerre két szál ugyanahhoz a listához biztonságosan tud elemet hozzáfűzni, ám ugyanez a zár lép érvénybe akkor is, ha két különböző listával dolgoznak. Mivel a többszálú ***Python*** alkalmazások teljesítménye emiatt romlik, az aszinkron, egyszálú programozás a ***Pythonhoz*** hasonló nyelvek esetében hangsúlyozottan előnyös lehet.

Csatlakozások fogadása és válaszok küldése

Kezdésként nézzünk egy egyszerű, a kapcsolatokat az 1100-as kapun fogadó példakiszolgálót. A kiszolgáló minden csatlakozásnál elküldi a ***UNIX*** időt, majd lezárja a foglalatot. Több kapcsolatot egyetlen szállal kezelni rendkívül bonyolult feladat – éppen ezt a kérdést célozzák meg a ***Twistedhez*** hasonló keretrendszerek. A hálózati kapcsolatokat a `twisted.internet.protocol.Protocol` osztály alosztályai képviselik, minden `Protocol` példány egy-egy hálózati kapcsolatot jelenít meg. Ezeket az objektumokat a `twisted.internet.protocol.Factory`-ból származó `Factory` objektumok hívják életre.
A `twisted.internet.reactor` egymaga végzi a piszkos munkát, vagyis a foglalatok lekérdezését és az események meghívását. ***Twistedben*** a `reactor.run()` meghívásával indul el az eseményhurok. A `run()` akkor lép ki, amikor az alkalmazás futása befejeződik, hasonlóan a ***GTK*** vagy a ***Qt*** eseményhurkaihoz.



1. ábra Egy proxykiszolgáló osztálydiagramja. A Protocol osztályok kezelik az egyes kapcsolatokat, és a Factory osztályok hozzák létre őket.

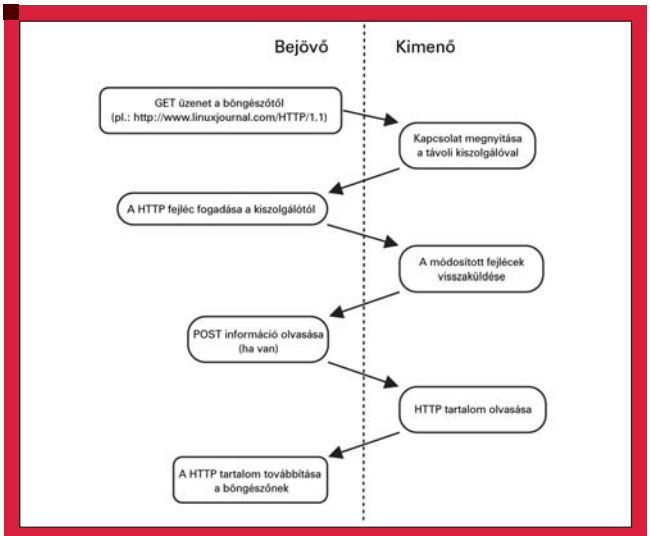
A proxykiszolgálós példa

Proxykiszolgálónk kétféle hálózati csevegőkapcsolatot ismer: bejövő ***HTTP***-kéréseket és a velük párosuló kimenő válaszokat. Mivel a ***HTTP*** egy csevegés jellegű protokoll, `protocol` osztályunkat a ***Twisted LineReceiveréből*** származtatjuk. Ez a `Protocol` alosztálya, a csevegés jellegű – például ***HTTP*** – kapcsolatokhoz külön szolgáltatásokat is biztosít. A ***Twisted*** valójában rendelkezik kifejezetten ***HTTP***-kérések indítására és kezelésére alkalmas osztályokkal is. Azért írunk mégis sajátot, mert a ***Twisted*** beépített osztályai proxyszolgáltatást nem biztosítanak, valamint a feladat programozási gyakorlatnak sem utolsó.
Az *1. ábrán* látható az az osztályszerkezet, amit alkalmazni fogunk. A ***Twisted*** a `Factory` osztályok példányait használja az egyes létrejövő kapcsolatokhoz tartozó `Protocol` példányok létrehozására. Létrehozunk egy `SimpleHTTP` osztályt, majd belőle származtatjuk a bejövő és a kimenő forgalmat kezelő osztályokat. Mivel a ***HTTP*** az ügyfél és a kiszolgáló oldalon szinte azonos, a tennivalók túlnyomó részét rábízhatjuk egy szuperosztályra, a maradékot pedig alosztályokra hagyjuk – pontosan így működnek a ***Twisted*** saját ***HTTP*** osztályai is.

A visszahívások kezelése

Eseményvezérelt programozásnál azok a műveletek, amelyeket egyébként egy vagy két eljárással el tudnánk végezni, több visszahívást is szükségessé tesznek. Ökölszabály, hogy minden olyan blokkoló művelet, amelyre várnunk kell, saját kódunkon kívül, azaz két eljárásunk között folyik. Proxykiszolgálónk esetében a kérések kezelésének fázisait számos további lépésre bontjuk. A proxykiszolgáló tevékenysége jelentős részben a böngészőtől érkező adatok beolvasásából, az adatok kismértékű módosításából, majd a távoli webkiszolgálónak való továbbküldéséből áll. A ***HTTP/1.1*** megjelenése óta egyetlen hálózati kapcsolaton keresztül több webes kapcsolatot is lehet kezelni. A *2. ábrán* látható, hogy mi történik az egyes kérésekkel. Ne feledjük, hogy minden ***HTTP***-kapcsolaton belül több kérést is lehet indítani. A négyszögeket összekötő nyilak az események sorrendjét jelzik.
Egy blokkoló jellegű programban szinte természetes, hogy ha meg kell nyitnunk egy távoli kapcsolatot, majd el kell küldenünk rajta egy sornyi szöveget, akkor ezt az alábbi módszerrel végezzük el:

```
connection = socket.open(remote_server,
➥remote_port)
connection.write(get_string)
response = connection.readline()
```



2. ábra A proxykérések feldolgozásának általános lépései

2. kódrészlet A műveletek halasztása Twistedben olyan, mintha tartásba helyeznénk őket, amíg a blokkoló utasítás végrehajtása be nem fejeződik.

```
d = self.outgoing_proxy_cache.getOutgoing
    (host, int(port))
d.addCallback(self.outgoingConnectionMade, uri)
d.addErrback(self.outgoingProxyError, uri)
```

3. kódrészlet Hagyományos hibakezelés Pythonban

```
try:
    (kérdéses kódrész)
except ValueError:
    (hibakezelő kód)
except MyError:
    (hibakezelő kód)
```


Mindannyian láttunk már ilyet, de vajon ***Twistedben*** mi az, ami másképp történik? Az eseményvezérelt programokban nem várjuk meg a kapcsolat felépülését, egyszerűen megadjuk, hogy milyen kódot szeretnénk futtatni, amikor a távoli kiszolgáló végre foglalkozni tud velünk. ***Twistedben*** az ilyen jellegű ütemezést, halasztást a `twisted.internet.defer.Deferred` osztály egy példányával intézzük, mely helyőrzőként szolgál ahhoz az eredményhez, amelyet egy blokkoló művelettől egyébként várnánk. Proxykiszolgálónkban például akkor alkalmazunk `Deferred` objektumot, amikor távoli kapcsolatot kezdeményezünk. (*2. kódrészlet*)
A `self.outgoing_proxy_cache.getOutgoing` eljárás egy kimenő proxykapcsolatot kezdeményez. A kapcsolat létrejöttét azonban nem várja meg, azonnal visszatér a hívóhoz. Minden hívás a lehető leghamarabb visszatér; ez az a megközelítés, mely lehetővé teszi az egyszálú kiszolgáló működését. Az eljárások a lekötött processzoridőt tehát valóban érdemi műveletekre fordítják, és nem arra, hogy külső események bekövetkeztére várjanak.
Érdemes megfigyelni, hogy a kapcsolati objektumot helyettesítő `Deferred` objektum visszaadása hogyan történik.
A `Deferred` objektumon az `addCallback` és az `addErrback` eljárást meghívva utasítások jövőbeli végrehajtását írjuk elő. Például a kimenő kapcsolat felépülése után a `self.outgoingConnectionMade` eljárás kerül meghívásra. Az `addCallback` második átadott értéke az uri, amivel azt közöljük a ***Twisteddel***, hogy a `self.outgoingConnectionMade` eljárást úgyszintén az urival, mint átadott értékkel kell meghívni.

Hibakezelés

Hiba esetén a hibakezelést biztosító `self.outgoingProxyError` eljárás kerül meghívásra egy `Failure` objektummal. A ***Python*** hagyományos hibakezelése kivételekkel dolgozik; a fogalom más magas szintű nyelvekből, például a Javából is ismerős lehet. (*3. kódrészlet*)
Bár a ***Python*** kivételkezelése szinkron programoknál – játsszunk kicsit a szavakkal – kivételesen jól működik, aszinkron programok kezelésére nincs felkészítve.

Például, amikor kimenő ***HTTP***-kapcsolatot kezdeményezünk, a ***Twisted*** a kapcsolat felépüléséig más események feldolgozásával foglalkozik. Nyilván azt szeretnénk, ha ettől függetlenül a kapcsolat felépítése közben esetlegesen előforduló hibák bármelyikét kezelni tudnánk. Szerencsére a ***Twisted*** készítői – áldjuk a nevüket – erre is gondoltak. Kódot nemcsak úgy lehet időzíteni, hogy blokkoló jellegű művelet véget érésekor fusson, hanem úgy is, hogy hiba felmerülésekor kapjon szerepet.
A ***Twisted*** az eseményhurkon belül felmerülő hibákat is kezeli, a kivételek kézben tartásában és naplózásában csatlakozási pontokkal segíti a fejlesztőket. Mindennek van egy járulékos előnye is: egy-egy kivétel ugyan meggátolhatja egy-egy művelet befejezését, a teljes kiszolgáló leállását viszont nem okozza, még akkor sem, ha egyetlen betűnyi kivételkezelő kódot sem írtunk.

A Twisted osztályai és a kivételkezelés

A ***Twisted*** osztályok egy részének használatakor, ilyen a most alkalmazott `LineReceiver` osztály is, számos eseményt úgy tudunk kezelni, hogy egyszerűen hozzáadjuk a megfelelő nevű eljárásokat az osztályokhoz. Minden alkalommal, amikor a protokoll fogad egy sort, a `lineReceived` eljárás kerül meghívásra, átadott értékként a sor szövegével. `SimpleHTTP` osztályunk, melyet a ***HTTP***-kapcsolatok alapszintű kezelésére készítettünk, a következő eljárásokkal rendelkezik:

- `startNewRequest`: Az egyes kérések kezdetekor kerül meghívásra.
- `lineReceived`: Célja a csevegés jellegű protokollok használatának segítése. Minden alkalommal, amikor újabb sornyi szöveg érkezik a foglalaton keresztül, ez az eljárás hívódik meg.
- `rawDataReceived`: Bináris fájl vagy nyers adatfolyam küldésekor nincs értelme új sor karakterekkel elválasztott részek feldolgozásáról beszélni. Éppen ezért a `LineReceiver` lehetővé teszi, hogy nyers átviteli módba váltsunk, ilyenkor a `lineReceived` helyett a `rawDataReceived` kerül meghívásra.

4. lista Események ütemezése Twistedben

```
def outgoingConnectionMade(self, outgoing_proxy,
                           uri):
    """
    Akkor kerül rá a vezérlés, amikor egy kimenő
    ➥proxykapcsolat
    felépült. Ez is egy Twisted visszahívó
    ➥eljárás.
    """
    assert(outgoing_proxy, OutgoingProxy)
    self.outgoing_proxy = outgoing_proxy
    outgoing_proxy.incoming_proxy = self

    # HTTP parancs küldése és a válasz visszaadása
    outgoing_proxy.write('%s %s %s' % \
        (self.http_command,
         uri,
         self.http_version) \
         + self.delimiter)

    outgoing_proxy.firstline_sent_def.addCallback(
        self.outgoingFirstlineReceived)

    # A sorba állított adatok elküldése
    self.flushOutgoingBuffer()

    # Visszahívók hozzáadása a fejlécek
    # megérkezésének idejére

outgoing_proxy.headers_finished_def.addCallback(
        self.outgoingHeadersReceived)

outgoing_proxy.request_finished_def.addCallback(
        self.handleOutgoingRequestFinished)
```

- `handleFirstLine`: A *HTTP* úgy működik, hogy minden kérést egy különálló sorral indít. Általában az ügyfél egy *GET* vagy *POST* kérést küld el egy *URI*-val, a kiszolgáló pedig egy állapotkóddal válaszol. A `handleFirstLine` mindegyik eset kezelésére alkalmas.
- `handleHeadersFinished`: A *HTTP* fejrészek teljes elküldése után kerül meghívásra.
- `handleRequestFinished`: Magának a *HTTP*-kérésnek a befejezésekor kerül meghívásra.

A *Twisted* programozók úgy illeszthetik sorba az eseményeket, hogy az adott protokoll kezelése folyamán felmerülő műveletekhez és állapotokhoz külön eljárásokat írnak. Adott kérés elindításakor megadhatjuk, hogy a kérés kezelésének egyes lépéseiben milyen eseményekre kerüljön sor. Korábbi példánkban egy kapcsolat létrejötte után a `self.outgoingConnectionMade` eljárást hívtuk meg. Vizsgáljuk meg ezt az eljárást. (*4. kódrészlet*)
Megjegyezném, hogy az `outgoing_proxy` a távoli kiszolgálóval a böngészőprogram nevében létrehozott kapcsolatot képviseli. A *HTTP*-kérést az `outgoing_proxy.write` eljárással küldjük el. A `self.outgoingFirstlineReceived` eljárás meghívását akkorra ütemezzük, amikor választ kapunk a távoli kiszolgálótól, a `self.outgoingHeadersReceived` eljárás pedig akkor jut szerephez, amikor a távoli kiszolgáló az összes *HTTP* fejlécet elküldte. A záró hívás a `self.handleOutgoingRequestFinished`, erre akkor kerül sor, amikor a távoli kiszolgáló teljes egészében elküldte a kimenő *HTTP*-kérésünkre adott válaszát.
Bár az `outgoingConnectionMade` eljárás visszatér, mielőtt bármi is történne, mi sorba állítjuk a jövőbe ütemezett eseményeket. Akár az is előfordulhat, hogy miközben adott kapcsolaton válaszra várunk, ugyanazon a szálon belül további tíz kérést nyitunk meg és zárunk le. A kapcsolatokkal összefüggő adatok mindegyike protokollosztályok példányadataként tárolódik. A `Factoryk` életre hívják a protokollpéldányokat, a protokollpéldányok tárolják a kapcsolatállapotokat, a `deferred` objektumok pedig a jövőbeli adatokat párosítják az eseménykezelőkkel.
A kirakójátékot a `reactor` teszi teljessé, mely elvégzi a foglalatok lekérdezésének piszkos munkáját.
Ez a *Twisted* alapjául szolgáló eszközkészlet.

Összeáll a kép

Barkácsolás céljára bárki letöltheti az eddigiekben tárgyalt proxykiszolgáló egészen pontosan 606 soros kódját. Nem tagadom, a vállalati intranetünket nem tenném mögé, itthon viszont már egy hete használom a nemkívánatos sütik és képek eldobására, illetve egy bizonyos cég oldalának elérését is letiltottam. Amikor elkezdtem a *Twistedet* használni, könnyedén rá tudtam hangolódni az aszinkron programozás gondolatvilágára; az általam kívánt adatáramlás és az események összerendelése már nehezebben ment, a legnehezebb pedig mindezt elmagyarázni más valakinek. Senki ne ijedjen meg! Mi a *Zotonál* gyakorlatilag nulla *Twisted*-ismerettel indultunk, mégis kevesebb mint egy év alatt fel tudtunk építeni egy rendkívül sokoldalú, magas fokon bővíthető, fürtözött fényképkezelő és -tároló alkalmazást. Ráadásul én voltam az egyetlen, aki teljes időben a kiszolgálóval foglalkozott.
Természetes, hogy a *Twisted* nem felel meg mindenki igényeinek. Kiterjedtsége, bár sokoldalúságának alapja, riasztó lehet. Aki egyszerű, *Python* alapú, aszinkron csevegő kiszolgálót keres, ismerkedjen meg a *Medusa*-val. A *Twistedhez* hasonlóan a *Medusa* is *Factorykba* (ezek a *Dispatcherek*) és csevegő osztályokba szervezi az aszinkron programozást.

Linux Journal 2005. március, 131. szám

Ken Kinder jelenleg egy fürtözött Twisted kiszolgálót fejleszt a Zoto számára az Oklahoma állambeli Oklahoma városában. Szeret túrázni, síelni, fényképezni – és persze Linuxot nyúzni. Szülővárosa a Colorado államban található Boulder.

KAPCSOLÓDÓ CÍMEK

A cikkhez tartozó források elérhetősége:
➲ www.linuxjournal.com/article/7963