Handling Multiple Connections

Python

```
# echo-client.py

# ...

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b"Hello, world")
    data = s.recv(1024)

print(f"Received {data!r}")
```

Η `.send()` μέθοδος συμπεριφέρεται τον εξής τρόπο. Επιστρέφει τον αριθμό των byte που αποστέλλονται, ο οποίος μπορεί να είναι μικρότερος από το μέγεθος των δεδομένων που διαβιβάζονται. Είστε υπεύθυνοι να το ελέγξετε και να καλέσετε `.send()` όσες φορές χρειάζεται για να στείλετε όλα τα δεδομένα:

> «Οι εφαρμογές είναι υπεύθυνες για τον έλεγχο της αποστολής όλων των δεδομένων. εάν μεταδόθηκαν μόνο ορισμένα από τα δεδομένα, η εφαρμογή πρέπει να επιχειρήσει την παράδοση των υπόλοιπων δεδομένων."

Στο παραπάνω παράδειγμα, αποφύγατε να το κάνετε αυτό χρησιμοποιώντας `.sendall()`:

> «Σε αντίθεση με τη send(), αυτή η μέθοδος συνεχίζει να στέλνει δεδομένα από byte μέχρι είτε να σταλούν όλα τα δεδομένα είτε να παρουσιαστεί σφάλμα. `None` επιστρέφεται με επιτυχία».

Έχετε δύο προβλήματα σε αυτό το σημείο:

- Πώς χειρίζεστε πολλές συνδέσεις ταυτόχρονα;
- Πρέπει να καλέσετε `.send()` και `.recv()` μέχρι να σταλούν ή να ληφθούν όλα τα δεδομένα.

Τι μπορείς να κάνεις? Υπάρχουν πολλές προσεγγίσεις στον συγχρονισμό. Μια δημοφιλής προσέγγιση είναι η χρήση asynchronous I/O . Η `asyncio` εισήχθη στην τυπική βιβλιοθήκη στην Python 3.4. Η παραδοσιακή επιλογή είναι η χρήση νημάτων.

Το πρόβλημα με τον συγχρονισμό είναι ότι είναι δύσκολο να γίνει σωστός. Υπάρχουν πολλές λεπτές αποχρώσεις που πρέπει να λάβετε υπόψη και να αποφύγετε. Το μόνο που χρειάζεται είναι ένα από αυτά να εκδηλωθεί και η αίτησή σας μπορεί ξαφνικά να αποτύχει με όχι και τόσο λεπτούς τρόπους.

Αυτό δεν έχει σκοπό να σας τρομάξει μακριά από την εκμάθηση και τη χρήση ταυτόχρονου προγραμματισμού. Εάν η εφαρμογή σας χρειάζεται κλίμακα, είναι απαραίτητο εάν θέλετε να χρησιμοποιήσετε περισσότερους από έναν επεξεργαστές ή έναν πυρήνα. Ωστόσο, για αυτό το σεμινάριο, θα χρησιμοποιήσετε κάτι που είναι ακόμα πιο παραδοσιακό από τα νήματα και είναι πιο εύκολο να το αιτιολογήσετε. Θα χρησιμοποιήσετε το granddaddy των κλήσεων συστήματος: `.select()`.

Η `.select()` μέθοδος σάς επιτρέπει να ελέγχετε για ολοκλήρωση εισόδου/εξόδου σε περισσότερες από μία υποδοχές. Έτσι, μπορείτε να καλέσετε `.select()` για να δείτε ποιες sockets έχουν έτοιμες εισόδους/εξόδους για ανάγνωση ή/και εγγραφή. Θα χρησιμοποιήσετε τη μονάδα επιλογέων στην τυπική βιβλιοθήκη, έτσι ώστε να χρησιμοποιείται η πιο αποτελεσματική υλοποίηση, ανεξάρτητα από το λειτουργικό σύστημα στο οποίο τυχαίνει να εκτελείτε:

> «Αυτή η ενότητα επιτρέπει υψηλού επιπέδου και αποτελεσματική πολυπλεξία I/O, βασισμένη στα επιλεγμένα πρωτόγονα modules. Οι χρήστες ενθαρρύνονται να χρησιμοποιούν αυτήν την ενότητα, εκτός εάν θέλουν ακριβή έλεγχο των πρωτόγονων σε επίπεδο λειτουργικού συστήματος που χρησιμοποιούνται."

Ωστόσο, χρησιμοποιώντας το `.select()`, δεν μπορείτε να εκτελείτε ταυτόχρονα. Τούτου λεχθέντος, ανάλογα με τον φόρτο εργασίας σας, αυτή η προσέγγιση μπορεί να είναι ακόμα αρκετά γρήγορη. Εξαρτάται από το τι πρέπει να κάνει η εφαρμογή σας όταν εξυπηρετεί ένα αίτημα και από τον αριθμό των πελατών που χρειάζεται να υποστηρίξει.

Η `asyncio` χρησιμοποιεί τη συνεργασία πολλαπλών εργασιών με ένα νήμα και έναν βρόχο συμβάντων για τη διαχείριση εργασιών. Με το `.select()`, θα γράφετε τη δική σας έκδοση ενός βρόχου συμβάντων, αν και πιο απλά και συγχρονισμένα.

Η χρήση `.select()` μπορεί να είναι μια τέλεια επιλογή. Μην αισθάνεστε ότι πρέπει να χρησιμοποιήσετε `asyncio`, νήματα ή την πιο πρόσφατη ασύγχρονη βιβλιοθήκη. Συνήθως, σε μια εφαρμογή δικτύου, η εφαρμογή σας είναι ούτως ή άλλως δεσμευμένη σε I/O: θα μπορούσε να περιμένει στο τοπικό δίκτυο, για τελικά σημεία στην άλλη πλευρά του δικτύου, για εγγραφή δίσκου κ.λπ.

Εάν λαμβάνετε αιτήματα από πελάτες που ξεκινούν εργασία συνδεδεμένη με CPU, δείτε τη λειτουργική μονάδα concurrent.futures . Περιέχει την κλάση ProcessPoolExecutor, η οποία χρησιμοποιεί μια ομάδα διεργασιών για να εκτελεί κλήσεις ασύγχρονα.

Εάν χρησιμοποιείτε πολλές διεργασίες, το λειτουργικό σύστημα μπορεί να προγραμματίσει τον κώδικα Python να εκτελείται παράλληλα σε πολλούς επεξεργαστές ή πυρήνες,

Στην επόμενη ενότητα, θα δείτε παραδείγματα διακομιστή και πελάτη που αντιμετωπίζουν αυτά τα προβλήματα. Χρησιμοποιούνται `.select()` για να χειρίζονται πολλές συνδέσεις ταυτόχρονα και να καλούν `.send()` και `.recv()` όσες φορές χρειάζεται.

Πελάτης και διακομιστής πολλαπλών συνδέσεων

Στις επόμενες δύο ενότητες, θα δημιουργήσετε έναν διακομιστή και έναν πελάτη που χειρίζεται πολλαπλές συνδέσεις χρησιμοποιώντας ένα `selector`αντικείμενο που δημιουργήθηκε από τη λειτουργική μονάδα επιλογέων .

Διακομιστής πολλαπλών συνδέσεων

Πρώτα, στρέψτε την προσοχή σας στον διακομιστή πολλαπλών συνδέσεων. Το πρώτο μέρος ρυθμίζει την υποδοχή ακρόασης:

Πύθων

```
# multiconn-server.py

import sys
```

```
import socket
import selectors
import types

sel = selectors.DefaultSelector()

# ...

host, port = sys.argv[1], int(sys.argv[2])
lsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
lsock.bind((host, port))
lsock.listen()
print(f"Listening on {(host, port)}")
lsock.setblocking(False)
sel.register(lsock, selectors.EVENT_READ, data=None)
```

Η μεγαλύτερη διαφορά μεταξύ αυτού του διακομιστή και του διακομιστή echo είναι η κλήση προς `lsock.setblocking(False)` ρύθμιση παραμέτρων της υποδοχής σε λειτουργία μη αποκλεισμού. Οι κλήσεις που γίνονται σε αυτήν την σοψκετ δεν θα αποκλείονται πλέον . Όταν χρησιμοποιείται με `sel.select()`, όπως θα δείτε παρακάτω, μπορείτε να περιμένετε συμβάντα σε μία ή περισσότερες υποδοχές και στη συνέχεια να διαβάσετε και να γράψετε δεδομένα όταν είναι έτοιμα.

Η `sel.register()` καταχωρεί την υποδοχή για παρακολούθηση `sel.select()` για τα συμβάντα που σας ενδιαφέρουν. Για την υποδοχή ακρόασης, θέλετε να διαβάσετε συμβάντα: `selectors.EVENT_READ`.

Για να αποθηκεύσετε όσα αυθαίρετα δεδομένα θέλετε μαζί με την υποδοχή, θα χρησιμοποιήσετε `data`. Επιστρέφεται όταν η `.select()` επιστρέφει. Θα χρησιμοποιήσετε `data` για να παρακολουθείτε τι έχει αποσταλεί και ληφθεί στην socket.

Ακολουθεί ο βρόχος του συμβάντος:

```
# multiconn-server.py

# ...

try:
    while True:
        events = sel.select(timeout=None)
        for key, mask in events:
            if key.data is None:
                accept_wrapper(key.fileobj)
            else:
                service_connection(key, mask)
except KeyboardInterrupt:
    print("Caught keyboard interrupt, exiting")
finally:
    sel.close()
```

Η `sel.select(timeout=None)` κάνει μπλοκ μέχρι να υπάρχουν υποδοχές έτοιμες για I/O. Επιστρέφει μια λίστα με πλειάδες, μία για κάθε υποδοχή. Κάθε πλειάδα περιέχει ένα `key` και ένα `mask`. Το `key`είναι ένα SelectorKey `namedtuple` που περιέχει ένα `fileobj` χαρακτηριστικό. Η `key.fileobj` είναι το αντικείμενο υποδοχής και η `mask` είναι μια μάσκα συμβάντος των λειτουργιών που είναι έτοιμες.

Εάν η `key.data` είναι `None`, τότε ξέρετε ότι προέρχεται από την υποδοχή ακρόασης και πρέπει να αποδεχτείτε τη σύνδεση. Θα καλέσετε τη δική σας `accept_wrapper()` συνάρτηση για να λάβετε το νέο αντικείμενο υποδοχής και να το καταχωρήσετε στον επιλογέα.

Εάν η `key.data` δεν είναι `None`, τότε ξέρετε ότι είναι μια υποδοχή πελάτη που έχει ήδη γίνει αποδεκτή και πρέπει να την επισκευάσετε. Η `service_connection()` στη συνέχεια καλείται με `key` και `mask` ως ορίσματα, και αυτό είναι ό,τι χρειάζεστε για να λειτουργήσετε στην socket.

Δείτε τι κάνει η `accept_wrapper()` στη λειτουργία σας:

```
# multiconn-server.py

# ...

def accept_wrapper(sock):
    conn, addr = sock.accept()  # Should be ready to read
    print(f"Accepted connection from {addr}")
    conn.setblocking(False)
    data = types.SimpleNamespace(addr=addr, inb=b"", outb=b"")
    events = selectors.EVENT_READ | selectors.EVENT_WRITE
    sel.register(conn, events, data=data)

# …
```

Επειδή η υποδοχή ακρόασης έχει καταχωρηθεί για το συμβάν `selectors.EVENT_READ`, θα πρέπει να είναι έτοιμη για ανάγνωση. Καλείτε `sock.accept()` και μετά καλείτε `conn.setblocking(False)` για να θέσετε την socket σε λειτουργία *μη μπλοκαρίσματος*.

Θυμηθείτε, αυτός είναι ο κύριος στόχος σε αυτήν την έκδοση του διακομιστή, επειδή δεν θέλετε να αποκλείεται. Εάν μπλοκάρει, τότε ολόκληρος ο διακομιστής είναι αδρανής μέχρι να επιστρέψει. Αυτό σημαίνει ότι άλλες υποδοχές μένουν σε αναμονή παρόλο που ο διακομιστής δεν λειτουργεί ενεργά. Αυτή είναι η επίφοβη κατάσταση "κολλήματος" στην οποία δεν θέλετε να βρίσκεται ο διακομιστής σας.

Στη συνέχεια, δημιουργείτε ένα αντικείμενο για να διατηρεί τα δεδομένα που θέλετε να συμπεριληφθούν μαζί με την υποδοχή χρησιμοποιώντας ένα `SimpleNamespace`. Επειδή θέλετε να μάθετε πότε η σύνδεση πελάτη είναι έτοιμη για ανάγνωση και εγγραφή, και τα δύο αυτά συμβάντα ορίζονται με τον τελεστή bitwise OR:

```
# multiconn-server.py

# ...

def accept_wrapper(sock):
    conn, addr = sock.accept()  # Should be ready to read
    print(f"Accepted connection from {addr}")
    conn.setblocking(False)
    data = types.SimpleNamespace(addr=addr, inb=b"", outb=b"")
    events = selectors.EVENT_READ | selectors.EVENT_WRITE
    sel.register(conn, events, data=data)

# ...
```

Στη συνέχεια, η `events` μάσκα, η υποδοχή και τα αντικείμενα δεδομένων μεταβιβάζονται στο `sel.register()`.

Τώρα ρίξτε μια ματιά στην `service_connection()` για να δείτε πώς γίνεται ο χειρισμός μιας σύνδεσης πελάτη όταν είναι έτοιμη:

Πύθων

```
# multiconn-server.py

# ...

def service_connection(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(1024)  # Should be ready to read
        if recv_data:
            data.outb += recv_data
        else:
            print(f"Closing connection to {data.addr}")
            sel.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if data.outb:
            print(f"Echoing {data.outb!r} to {data.addr}")
            sent = sock.send(data.outb)  # Should be ready to write
            data.outb = data.outb[sent:]

# …
```

Αυτή είναι η καρδιά του απλού διακομιστή πολλαπλών συνδέσεων. Το `key` είναι το `namedtuple` επιστρεφόμενο από την `.select()` αυτό που περιέχει το αντικείμενο υποδοχής (`fileobj`) και το αντικείμενο δεδομένων. Η `mask` περιέχει τα συμβάντα που είναι έτοιμα.

Εάν η υποδοχή είναι έτοιμη για ανάγνωση, τότε η `mask & selectors.EVENT_READ` θα αξιολογηθεί σε `True`, Η `sock.recv()` ονομάζεται έτσι. Όλα τα δεδομένα που διαβάζονται επισυνάπτονται `data.outb` έτσι ώστε να μπορούν να σταλούν αργότερα.

Σημειώστε το `else:` μπλοκ για να ελέγξετε εάν δεν λαμβάνονται δεδομένα:

```
# multiconn-server.py

# ...

def service_connection(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(1024)  # Should be ready to read
        if recv_data:
            data.outb += recv_data
        else:
            print(f"Closing connection to {data.addr}")
            sel.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if data.outb:
```

```
            print(f"Echoing {data.outb!r} to {data.addr}")
            sent = sock.send(data.outb)  # Should be ready to write
            data.outb = data.outb[sent:]

# ...
```

Εάν δεν ληφθούν δεδομένα, αυτό σημαίνει ότι ο υπολογιστής-πελάτης έχει κλείσει την υποδοχή του, οπότε και ο διακομιστής θα πρέπει να το κάνει. Αλλά μην ξεχάσετε να καλέσετε `sel.unregister()` πριν κλείσετε, ώστε να μην παρακολουθείται πλέον από την `.select()`.

Όταν η υποδοχή είναι έτοιμη για εγγραφή, κάτι που θα πρέπει πάντα να ισχύει για μια υγιή πρίζα, τυχόν λαμβανόμενα δεδομένα που είναι αποθηκευμένα σε αυτήν `data.outb` επαναλαμβάνονται στον πελάτη χρησιμοποιώντας το `sock.send()`. Τα byte που αποστέλλονται αφαιρούνται στη συνέχεια από το buffer αποστολής:

```
# multiconn-server.py

# ...

def service_connection(key, mask):

    # ...

    if mask & selectors.EVENT_WRITE:
        if data.outb:
            print(f"Echoing {data.outb!r} to {data.addr}")
            sent = sock.send(data.outb)  # Should be ready to write
            data.outb = data.outb[sent:]

# ...
```

Η `.send()` μέθοδος επιστρέφει τον αριθμό των byte που αποστέλλονται. Αυτός ο αριθμός μπορεί στη συνέχεια να χρησιμοποιηθεί με συμβολισμό slice στο `.outb` buffer για να απορρίψετε τα byte που αποστέλλονται.

Πελάτης πολλαπλών συνδέσεων

Τώρα ρίξτε μια ματιά στον υπολογιστή-πελάτη πολλαπλών συνδέσεων, `multiconn-client.py`. Είναι πολύ παρόμοιο με τον διακομιστή, αλλά αντί να ακούει για συνδέσεις, ξεκινά με την εκκίνηση των συνδέσεων μέσω `start_connections()`:

```
# multiconn-client.py

import sys
import socket
import selectors
import types

sel = selectors.DefaultSelector()
messages = [b"Message 1 from client.", b"Message 2 from client."]

def start_connections(host, port, num_conns):
    server_addr = (host, port)
    for i in range(0, num_conns):
        connid = i + 1
```

```
        print(f"Starting connection {connid} to {server_addr}")
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.setblocking(False)
        sock.connect_ex(server_addr)
        events = selectors.EVENT_READ | selectors.EVENT_WRITE
        data = types.SimpleNamespace(
            connid=connid,
            msg_total=sum(len(m) for m in messages),
            recv_total=0,
            messages=messages.copy(),
            outb=b"",
        )
        sel.register(sock, events, data=data)

# ...
```

Η `num_conns` διαβάζεται από τη γραμμή εντολών και είναι ο αριθμός των συνδέσεων που πρέπει να δημιουργηθούν στον διακομιστή. Ακριβώς όπως ο διακομιστής, κάθε υποδοχή έχει ρυθμιστεί σε λειτουργία μη αποκλεισμού.

Χρησιμοποιείτε `.connect_ex()` αντί για `.connect()` επειδή η `.connect()` θα δημιουργήσει αμέσως μια `BlockingIOError` εξαίρεση. Η `.connect_ex()` μέθοδος επιστρέφει αρχικά μια *ένδειξη* σφάλματος ,`errno.EINPROGRESS` αντί να δημιουργήσει μια εξαίρεση που θα παρεμπόδιζε τη σύνδεση σε εξέλιξη. Μόλις ολοκληρωθεί η σύνδεση, η υποδοχή είναι έτοιμη για ανάγνωση και εγγραφή και επιστρέφεται από το `.select()`.

Μετά τη ρύθμιση της υποδοχής, τα δεδομένα που θέλετε να αποθηκεύσετε με την υποδοχή δημιουργούνται χρησιμοποιώντας το `SimpleNamespace`. Τα μηνύματα που θα στείλει ο πελάτης στον διακομιστή αντιγράφονται χρησιμοποιώντας `messages.copy()` επειδή κάθε σύνδεση θα καλεί `socket.send()` και θα τροποποιεί τη λίστα. Όλα όσα χρειάζονται για να παρακολουθείτε τι χρειάζεται να στείλει, έχει στείλει και έχει λάβει ο πελάτης, συμπεριλαμβανομένου του συνολικού αριθμού των byte στα μηνύματα, αποθηκεύονται στο αντικείμενο `data`.

Δείτε τις αλλαγές που έγιναν από τον διακομιστή `service_connection()` για την έκδοση του πελάτη:

Αλλαγές αρχείου (διαφορά)

```
_def service_connection(key, mask):
_    sock = key.fileobj
_    data = key.data
_    if mask & selectors.EVENT_READ:
_        recv_data = sock.recv(1024)  # Should be ready to read
_        if recv_data:
-            data.outb += recv_data
+            print(f"Received {recv_data!r} from connection {data.connid}")
+            data.recv_total += len(recv_data)
-        else:
-            print(f"Closing connection {data.connid}")
+        if not recv_data or data.recv_total == data.msg_total:
+            print(f"Closing connection {data.connid}")
_            sel.unregister(sock)
_            sock.close()
_    if mask & selectors.EVENT_WRITE:
+        if not data.outb and data.messages:
+            data.outb = data.messages.pop(0)
_        if data.outb:
```

```
-             print(f"Echoing {data.outb!r} to {data.addr}")
+             print(f"Sending {data.outb!r} to connection {data.connid}")
              sent = sock.send(data.outb)  # Should be ready to write
              data.outb = data.outb[sent:]
```

Είναι ουσιαστικά το ίδιο αλλά για μια σημαντική διαφορά. Ο υπολογιστής-πελάτης παρακολουθεί τον αριθμό των byte που έχει λάβει από τον διακομιστή, ώστε να μπορεί να κλείσει την πλευρά της σύνδεσης. Όταν ο διακομιστής το εντοπίσει αυτό, κλείνει και την πλευρά της σύνδεσης.

Σημειώστε ότι με αυτόν τον τρόπο, ο διακομιστής εξαρτάται από την καλή συμπεριφορά του πελάτη: ο διακομιστής αναμένει από τον πελάτη να *κλείσει την πλευρά της σύνδεσης όταν τελειώσει* η αποστολή μηνυμάτων. Εάν ο πελάτης δεν κλείσει, ο διακομιστής θα αφήσει τη σύνδεση ανοιχτή. Σε μια πραγματική εφαρμογή, μπορεί να θέλετε να προφυλαχθείτε από αυτό στον διακομιστή σας εφαρμόζοντας ένα χρονικό όριο για να αποτρέψετε τη συσσώρευση συνδέσεων πελάτη εάν δεν στείλουν αίτημα μετά από ένα ορισμένο χρονικό διάστημα.

Εκτέλεση του προγράμματος-πελάτη και διακομιστή πολλαπλών συνδέσεων

Τώρα ήρθε η ώρα να τρέξετε το `multiconn-server.py` και το `multiconn-client.py`. Και οι δύο χρησιμοποιούν ορίσματα γραμμής εντολών . Μπορείτε να τα εκτελέσετε χωρίς ορίσματα για να δείτε τις επιλογές.

Για τον διακομιστή, με `host`και `port` αριθμούς:

```
$ python_multiconn-server.py
Usage: multiconn-server.py <host> <port>
```

Για τον πελάτη, περάστε επίσης τον αριθμό των συνδέσεων που θα δημιουργήσετε στον διακομιστή, `num_connections`:

```
$ python_multiconn-client.py
Usage: multiconn-client.py <host> <port> <num_connections>
```

Παρακάτω είναι η έξοδος διακομιστή κατά την ακρόαση στη διεπαφή loopback στη θύρα 65432:

Κέλυφος

```
$ python_multiconn-server.py_127.0.0.1_65432
Listening on ('127.0.0.1', 65432)
Accepted connection from ('127.0.0.1', 61354)
Accepted connection from ('127.0.0.1', 61355)
Echoing b'Message 1 from client.Message 2 from client.' to ('127.0.0.1', 61354)
Echoing b'Message 1 from client.Message 2 from client.' to ('127.0.0.1', 61355)
Closing connection to ('127.0.0.1', 61354)
Closing connection to ('127.0.0.1', 61355)
```

Ακολουθεί η έξοδος πελάτη όταν δημιουργεί δύο συνδέσεις στον παραπάνω διακομιστή:

Κέλυφος

```
$ python_multiconn-client.py_127.0.0.1_65432_2
Starting connection 1 to ('127.0.0.1', 65432)
Starting connection 2 to ('127.0.0.1', 65432)
Sending b'Message 1 from client.' to connection 1
Sending b'Message 2 from client.' to connection 1
Sending b'Message 1 from client.' to connection 2
Sending b'Message 2 from client.' to connection 2
Received b'Message 1 from client.Message 2 from client.' from connection 1
```

```
Closing connection 1
Received b'Message 1 from client.Message 2 from client.' from connection 2
Closing connection 2
```