Chapter 11 - Networking

CS 172 - Computer Programming 2
Lanzhou University

These slides use many elements provided in the main bibliographic reference for these lectures:

Programming in Python 3
A Complete Introduction to the Python Language,
2nd Edition,
Mark Summerfield

Outline

Networking

- Allows computer programs to communicate with each other
  - even if they are running on different machines
- Is fundamental for programs such as browsers
- Can add other dimensions to the functionality of programs
  - remote operation or logging
  - retrieving or supplying data to other machines

- Most networking programs work on either
  1. a peer-to-peer basis (e.g. torrents)
     - the same program runs on different machines
  2. a client/server basis (more common)
     - client programs send requests to a server (e.g. web page/server)

Networking

In this chapter

- We will create a basic client/server application

Client/server applications

Networking

Client/server applications

- are normally implemented as two separate programs
  1. a server that waits for and responds to requests
  2. one or more clients that send requests to the server
     - and read back the server's response
- For this to work,
  - clients must know where to connect to the server
    - the server's IP address and port number
  - both clients and server must send and receive data using an agreed-upon protocol
    - i.e., using data formats that they both understand

Networking

Client/server applications

- Python's low-level `socket` module supports
  1. IP addresses
  2. the most commonly used networking protocols, including
     - user datagram protocol (UDP), lightweight but unreliable; data is sent as discrete packets but with no guarantee that they will arrive
     - transmission control protocol (TCP), reliable connection- and stream-oriented protocol; any amount of data can be sent - the socket must break the data into chunks that are small enough to send, and for reconstructing the data at the other end
- In this chapter, we will develop a client/server program
  - so we use TCP

Networking

Client/server applications

- We also need to decide whether to send/receive data as text or binary data
  - And, in the latter case, in what form
- In this chapter, we use blocks of binary data, where
  - the first 4 bytes are the length of the following data
  - the following data is a binary pickle
- The advantage:
  - we can use the same sending and receiving code for *any* application since we can store almost any arbitrary data in a pickle
- The disadvantage:
  - both client and server must understand pickles, so they must be written in Python

Networking

Our running example

- Is a car registration program
- The server holds details of car registrations
  - license plate, seats, mileage and owner
- The client is used to retrieve and/or change car details
  - to change a car's mileage or owner
  - to create a new car registration
- Any number of clients can be used and they won't block each other
  - even if two access the server at the same time
  - because the server hands off each client's request to a separate thread

Networking

Our running example

- For simplicity, we will run the server and the clients on the same machine
  - we can use *localhost* as the IP address
    - although if the server is on another machine the client can be given its IP address on the command line
  - we have chosen an arbitrary port number of 9653
    - the port number should be greater that 1023 and is normally between 5001 and 32767, although port numbers up to 65535 are normally valid
- The server can accept 5 kinds of requests:
  - `GET_CAR_DETAILS`, `CHANGE_MILEAGE`, `CHANGE_OWNER`, `NEW_REGISTRATION` and `SHUTDOWN`
  - with a corresponding response for each
    - The response is the requested data or confirmation of the requested action, or an indication of an error

Outline

The client program

- The code is available in file `car_registration.py`
- An example of interaction is as follows
  - It assumes the server is already running

```
(C)ar (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]:
License: _024 hyr_
License: 024 HYR
Seats: 2
Mileage: 97543
Owner: Jack Lemon
(C)ar (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]: _m_
License [024 HYR]:
Mileage [97543]: _103491_
Mileage successfully changed
```

- The data entered by the user is shown between `_ _`
- Where there is no visible input it means that the user pressed `Enter`
  - to accept the default option (which is between `[ ]`)
- Note each option is written using the character in between parenthesis (can also be lower case)

The client program

- The code is available in file `car_registration.py`
- An example of interaction is as follows
  - It assumes the server is already running

```
(C)ar (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]:
License: _024 hyr_
License: 024 HYR
Seats: 2
Mileage: 97543
Owner: Jack Lemon
(C)ar (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]: _m_
License [024 HYR]:
Mileage [97543]: _103491_
Mileage successfully changed
```

- The user:
  1. asked to see the details of a particular car (option c is the default), and then
  2. updated its mileage

The client program

- As many clients as we like can be running
- When a user quits their particular client, the server is unaffected
- But if the server is stopped, this affects
  - not only the client that stopped it as well as all other clients,
  - which will get a *Connection refused* error, and terminate when they next attempt to access the server
- In a real application, only some clients can shut down the server
  - we have included it in the client to show how it is done

The client program

- We will review its code, starting with the `main()` function
  - and the handling of the user interface

```
def main():
    if len(sys.argv) > 1:
        Address[0] = sys.argv[1]
    call = dict(c=get_car_details, m=change_mileage, o=change_owner,
                n=new_registration, s=stop_server, q=quit)
    menu = ("(C)ar  Edit (M)ileage  Edit (O)wner  (N)ew car  "
            "(S)top server  (Q)uit")
    valid = frozenset("cmonsq")
    ...
```

- The `Address` list is a global variable that holds the IP address and port number:
  - By default it is set to: `Address = ["localhost", 9653]`
  - And it can be overridden if specified on the command line (i.e. if `len(sys.argv) > 1`)
- The `call` dictionary maps menu options to functions
- `menu` contains a description for the user of the options
- `valid` contains the set (unchangeable) of the input options

The client program

```
def main():
    if len(sys.argv) > 1:
        Address[0] = sys.argv[1]
    call = dict(c=get_car_details, m=change_mileage, o=change_owner,
                n=new_registration, s=stop_server, q=quit)
    menu = ("(C)ar  Edit (M)ileage  Edit (O)wner  (N)ew car  "
            "(S)top server  (Q)uit")
    valid = frozenset("cmonsq")
    previous_license = None
    while True:
        action = Console.get_menu_choice(menu, valid, "c", True)
        previous_license = call[action](previous_license)
```

- We keep track of the last entered license that is used as the default
  - most commands start by asking for the license of a car
- We have an infinite loop to get user's options
- The `Console` is supplied with our main bibliographic reference
  - It contains functions for getting values from the user at the console
  - such as `Console.get_string()` and `Console.get_integer()`
- Once the user makes a choice we call the corresponding function
  - passing it the previous license
  - and expecting each function to return the license it used

The client program

- We now review the `get_car_details()` function

```
def get_car_details(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is not None:
        print("License: {0}\nSeats:   {seats}\nMileage: {mileage}\n"
              "Owner:   {owner}".format(license, **car._asdict()))
    return license
```

- This function is used to get information about a particular car
- Since most functions need to request a license from the user and need some car-related data to work on
    - we have factored out this functionality into `retrieve_car_details()`

The client program

```
def retrieve_car_details(previous_license):
    license = Console.get_string("License", "license", previous_license)
    if not license:
        return previous_license, None
    license = license.upper()
    ok, *data = handle_request("GET_CAR_DETAILS", license)
    if not ok:
        print(data[0])
        return previous_license, None
    return license, CarTuple(*data)
```

- This is the first function to make use of networking
- It calls the `handle_request()` function
  - which takes whatever data it is given as argument
  - sends it to the server
  - and returns whatever the server replies
- `handle_request()` does not treat the data it sends/returns
  - It purely provides the networking service

The client program

- In our program, we have a protocol where we always send
  - the name of the action we want the server to perform
    - as the first argument
  - followed by any relevant parameters
    - in this case, just the license
- The protocol for the reply is that the server always returns a tuple
  - whose first item is a Boolean flag; if it is
    - `False`, we have a 2-tuple and the second item is an error message
    - `True`, the tuple is either: i) a 2-tuple whose second item is a confirmation message, or ii) an n-tuple with the second and subsequent items holding the requested data
- So, in `retrieve_car_details()`, if the license is unrecognized
  - `ok` is `False`, we print the error message in `data[0]` and
  - return the previous license unchanged
- Otherwise, we
  - return the license (which will now become the previous license) and
  - a `CarTuple` made from the `data` list

The client program

- We now review the change_mileage() function

```
def change_mileage(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is None:
        return previous_license
    mileage = Console.get_integer("Mileage", "mileage", car.mileage, 0)
    if mileage == 0:
        return license
    ok, *data = handle_request("CHANGE_MILEAGE", license, mileage)
    if not ok:
        print(data[0])
    else:
        print("Mileage successfully changed")
    return license
```

- This function follows a similar pattern to get_car_details()
  - except that once we have the details we update one aspect of them
- There are 2 networking calls
  - retrieve_car_details() calls handle_request() to get details

The client program

- We now review the `change_mileage()` function

```
def change_mileage(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is None:
        return previous_license
    mileage = Console.get_integer("Mileage", "mileage", car.mileage, 0)
    if mileage == 0:
        return license
    ok, *data = handle_request("CHANGE_MILEAGE", license, mileage)
    if not ok:
        print(data[0])
    else:
        print("Mileage successfully changed")
    return license
```

- The reply is always a 2-tuple, with the second item being
    - an error message or
    - None

The client program

- We won't review the change_owner() function
  - it is structurally the same as change_mileage()
- We will also not review new_registration()
  - it differs only in not retrieving car details at the start
  - and asking the user for all the details of the new car
    - rather than just changing one detail
- If the user chooses to quit the program
  - we do a clean termination by calling sys.exit()

```
def quit(*ignore):
    sys.exit()
```

- If you check the main() function
  - you may see that every menu function is called with the previous license
  - so we are specifying a parameter *ignore on quit()
    - which can take any number of positional arguments;
    - The name ignore has no significance to Python

The client program

- If the user chooses to stop the server,
  - we use `handle_request()` to inform the server
  - and specify that we do not want a reply

```
def stop_server(*ignore):
    handle_request("SHUTDOWN", wait_for_reply=False)
    sys.exit()
```

- Once the data is sent, `handle_request()` returns
  - and we do a clean termination using `sys.exit()`

Handling requests (client's side)

- `handle_request()` then provides all the networking handling

```
def handle_request(*items, wait_for_reply=True):
    SizeStruct = struct.Struct("!I")
    data = pickle.dumps(items, 3)
    try:
        with SocketManager(tuple(Address)) as sock:
            sock.sendall(SizeStruct.pack(len(data)))
            sock.sendall(data)
            if not wait_for_reply:
                return
            size_data = sock.recv(SizeStruct.size)
            size = SizeStruct.unpack(size_data)[0]
            result = bytearray()
            while True:
                data = sock.recv(4000)
                if not data:
                    break
                result.extend(data)
                if len(result) >= size:
                    break
        return pickle.loads(result)
    except socket.error as err:
        print("{0}: is the server running?".format(err))
        sys.exit(1)
```

Handling requests (client's side)

The handle_request() function:

```
SizeStruct = struct.Struct("!I")
data = pickle.dumps(items, 3)
```

- starts by creating a `struct.Struct` holding one unsigned integer (more about these format strings here `https://docs.python.org/3/library/struct.html#format-strings`)
  - in a structure that guarantees the sent data will be received coherently
- then creates a pickle of whatever items it is passed
  - the function does not know (or care) what the items are
  - we have set pickle protocol version to 3
    - to ensure that both clients and server use the same pickle version
    - even if a client or server is upgraded to run a different version of Python

Handling requests (client's side)

```
with SocketManager(tuple(Address)) as sock:
        sock.sendall(SizeStruct.pack(len(data)))
        sock.sendall(data)
        if not wait_for_reply:
            return
```

- `SocketManager` is a custom context manager that gives us a socket
  - which we will review later
  - A socket is an endpoint for sending/receiving data across a network
- `sock` will contain the socket where we can send/receive data
- the `socket.socket.sendall()` method sends all the data it is given
  - We always send two items:
    1. the length of the pickle
    2. the pickle itself
- If the `wait_for_reply` argument is `False`
  - we don't wait for a reply and return immediately;
  - The context manager will ensure that the socket is closed before the function actually returns

Handling requests (client's side)

```
size_data = sock.recv(SizeStruct.size)
size = SizeStruct.unpack(size_data)[0]
```

- after sending the data, and when we want a reply,
  - we call `socket.socket.recv()` to get the reply;
  - This method blocks until it receives data;
  - For the first call, we request four bytes
    - the size of the integer that holds the size of the reply pickle;
  - We use the `struct.Struct` to unpack the bytes into the `size` integer

Handling requests (client's side)

```
        result = bytearray()
        while True:
            data = sock.recv(4000)
            if not data:
                break
            result.extend(data)
            if len(result) >= size:
                break
    return pickle.loads(result)
```

- We then create an empty `bytearray`
  - and try to retrieve the incoming pickle in blocks of up to 4000 bytes
- If we run out of data
  - we break out the loop
- Otherwise, we append the data to the `result` variable
- Once we have read `size` bytes
  - we break out the loop
- And finally unpickle the data using `pickle.loads()`

Handling requests (client's side)

```
except socket.error as err:
    print("{0}: is the server running?".format(err))
    sys.exit(1)
```

- If something goes wrong with the network connection
  - e.g., the server isn't running or the connection fails,
  - a `socket.error` exception is raised

The client program

- We finally review the custom context manager class:

```
class SocketManager:
    def __init__(self, address):
        self.address = address

    def __enter__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect(self.address)
        return self.sock

    def __exit__(self, *ignore):
        self.sock.close()
```

- The address is a 2-tuple (IP address, port number)
  - which is set when the context manager is created
- Once the context manager is used in a `with` statement (`__enter__`)
  - it creates a socket and tries to make a connection
  - it blocks until a connection is established or an exception raised
  - The 1st argument to `socket.socket()` initializer is the address family
    - we used `socket.AF_INET` (IPv4); alternatively: `socket.AF_INET6` (IPv6)
  - The 2nd is either `socket.SOCK_STREAM` (TCP) or `socket.SOCK_STREAM` (UDP)

The client program

- We finally review the custom context Manager class:

```
class SocketManager:
    def __init__(self, address):
        self.address = address

    def __enter__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect(self.address)
        return self.sock

    def __exit__(self, *ignore):
        self.sock.close()
```

- When the flow of control leaves the `with` statement's scope
  - the context object's `__exit__()` method is called
- We don't care whether an exception was raised or not
  - for simplicity, so we ignore the exception arguments
  - and we just close the socket
  - Since the method returns `None`, any exception is propagated
    - this works well since we put an except block in `handle_request()` to process any socket exceptions that occur

Outline

The server program

- The code for creating servers often follows the same design
- We can use the high-level `socketserver` module
  - which takes care of all the housekeeping for us
- All we have to do is provide a request handler class
  - with a `handle()` method,
    - which is used to read requests and write replies
- The `socketserver` module handles the communication for us,
  - servicing each connection request
  - in a transparent way so that we are insulated from the low-level details

- The code for the server is given in `car_registration_server.py`
  - which contains a very simple `Car` class
    - that holds seats, mileage and owner information as properties
    - seats is read only
  - The class does not hold car licenses: the cars are stored in a dictionary
    - where the licenses are the keys

The server program

- We start by looking at the `main()` function

```
def main():
    filename = os.path.join(os.path.dirname(__file__), "car_registrations.dat"
    cars = load(filename)
    print("Loaded {0} car registrations".format(len(cars)))
    ...
```

- We are storing the car registration data in the running directory
- The `cars` object is set to a dictionary
  - whose keys are license strings
  - and whose values are `Car` objects
- `load()` generates sample data if the file does not yet exist

The server program

```
def main():
    filename = os.path.join(os.path.dirname(__file__), "car_registrations.dat")
    cars = load(filename)
    print("Loaded {0} car registrations".format(len(cars)))
    RequestHandler.Cars = cars
    ...
```

- Our request handler class needs to access the cars dictionary
  - but we cannot pass the dictionary to an instance because the server creates an instance for each request
  - So we set the dictionary to the `RequestHandler.Cars` class variable where it is accessible to all instances.

The server program

```
def main():
    filename = os.path.join(os.path.dirname(__file__), "car_registrations.dat")
    cars = load(filename)
    print("Loaded {0} car registrations".format(len(cars)))
    RequestHandler.Cars = cars
    server = None
    try:
        server = CarRegistrationServer(("", 9653), RequestHandler)
        server.serve_forever()
    except Exception as err:
        print("ERROR", err)
    finally:
        if server is not None:
            server.shutdown()
            save(filename, cars)
            print("Saved {0} car registrations".format(len(cars)))
```

- We create an instance of the server passing it:
  - the address, the port number, and the `RequestHandler` class object
  - `""` as address indicates any accessible IPv4 address (e.g., localhost)
- We then tell the server to serve requests forever
  - When the server shuts down, we save the (changed) cars dictionary

The server program

- Creating a custom server class is (extremely) easy:

```
class CarRegistrationServer(socketserver.ThreadingMixIn,
                            socketserver.TCPServer): pass
```

- To create a server that used processes rather than threads
  - the only change would be to inherit `socketserver.ForkingMixIn`
  - instead of `socketserver.ThreadingMixIn`
- The `socketserver` module's classes can be used to create a variety of custom servers
  - including UDP servers
  - by inheriting the appropriate pair of base classes

The server program

- The socket server creates a request handler to handle each request
- Our custom `RequestHandler` class provides:
  - a method for each kind of request it can handle, and
  - the `handle()` method that it must have
    - since it is the only method used by the socket server
- We start by looking at the class declaration and its class variables

```
class RequestHandler(socketserver.StreamRequestHandler):

    CarsLock = threading.Lock()
    CallLock = threading.Lock()
    Call = dict(
            GET_CAR_DETAILS=(
                    lambda self, *args: self.get_car_details(*args)),
            CHANGE_MILEAGE=(
                    lambda self, *args: self.change_mileage(*args)),
            CHANGE_OWNER=(
                    lambda self, *args: self.change_owner(*args)),
            NEW_REGISTRATION=(
                    lambda self, *args: self.new_registration(*args)),
            SHUTDOWN=lambda self, *args: self.shutdown(*args))
```

The server program

- We have created a `socketserver.StreamRequestHandler` subclass
  - since we are using a streaming (TCP) server;
  - `socketserver.DatagramRequestHandler` is available for UDP servers

```
class RequestHandler(socketserver.StreamRequestHandler):

    CarsLock = threading.Lock()
    CallLock = threading.Lock()
    Call = dict(
            GET_CAR_DETAILS=(
                    lambda self, *args: self.get_car_details(*args)),
            CHANGE_MILEAGE=(
                    lambda self, *args: self.change_mileage(*args)),
            CHANGE_OWNER=(
                    lambda self, *args: self.change_owner(*args)),
            NEW_REGISTRATION=(
                    lambda self, *args: self.new_registration(*args)),
            SHUTDOWN=lambda self, *args: self.shutdown(*args))
```

- `RequestHandler.Cars` is a dictionary class variable added in `main()`

The server program

- Almost every request-handling method needs to access the `Cars` data
  - it must never be accessed by 2 methods (threads) at the same time;
  - otherwise, the dictionary may become corrupted!
  - We have a lock class variable that we are using to ensure this!

```
class RequestHandler(socketserver.StreamRequestHandler):
    CarsLock = threading.Lock()
    CallLock = threading.Lock()
    Call = dict(
            GET_CAR_DETAILS=(
                    lambda self, *args: self.get_car_details(*args)),
            CHANGE_MILEAGE=(
                    lambda self, *args: self.change_mileage(*args)),
            CHANGE_OWNER=(
                    lambda self, *args: self.change_owner(*args)),
            NEW_REGISTRATION=(
                    lambda self, *args: self.new_registration(*args)),
            SHUTDOWN=lambda self, *args: self.shutdown(*args))
```

- The `Call` class variable is also a dictionary, in which
  - each key is the name of an action, and
  - each value is a function for performing the action;
  - We have also implemented a lock for it!

The server program

- Whenever a client makes a request, a new thread is created
  - with a new instance of the `RequestHandler` class,
  - and then the instance's `handle()` method is called

```
def handle(self):
    SizeStruct = struct.Struct("!I")
    size_data = self.rfile.read(SizeStruct.size)
    size = SizeStruct.unpack(size_data)[0]
    data = pickle.loads(self.rfile.read(size))
    ....
```

- We start by creating a struct as in the client
- We need it for the *length+pickle* format that we use
- The data coming from the client can be read from `self.rfile`
- We begin by reading 4 bytes and unpacking this as the integer `size`
- Then we read `size` bytes and unpickle them into the `data` variable
  - the read will block until the data is read
- We know (by protocol design) the `data` will always be a tuple
  - with the first item being the requested action
  - and the other items being the parameters for the action

The server program

```
def handle(self):
    SizeStruct = struct.Struct("!I")
    size_data = self.rfile.read(SizeStruct.size)
    size = SizeStruct.unpack(size_data)[0]
    data = pickle.loads(self.rfile.read(size))
    try:
        with RequestHandler.CallLock:
            function = self.Call[data[0]]
        reply = function(self, *data[1:])
    except Finish:
        return
    data = pickle.dumps(reply, 3)
    self.wfile.write(SizeStruct.pack(len(data)))
    self.wfile.write(data)
```

- Inside the `try` block we get the `lambda` function for the action
    - we use a lock to protect access to the `Call` dictionary
    - as always, we do as little as possible within the scope of the lock
- Once we have the function we call it, passing
    - `self` as the first argument
    - and the rest of the `data` tuple as the other arguments
- The outcome is that the call `self._method_(*data[1:])` is made
    - where _method_ is the method corresponding to the action given in `data[0]`

The server program

```
def handle(self):
    SizeStruct = struct.Struct("!I")
    size_data = self.rfile.read(SizeStruct.size)
    size = SizeStruct.unpack(size_data)[0]
    data = pickle.loads(self.rfile.read(size))

    try:
        with RequestHandler.CallLock:
            function = self.Call[data[0]]
        reply = function(self, *data[1:])
    except Finish:
        return
    data = pickle.dumps(reply, 3)
    self.wfile.write(SizeStruct.pack(len(data)))
    self.wfile.write(data)
```

- If the action is to shut down,
  - a custom `Finish` exception is raised in the `shutdown()` method,
  - no reply is expected by the client, so we just return
- For any other action,
  - we pickle the result of calling the action's corresponding method
  - and write the size of the pickle and then the pickle data itself

The server program

- Going through the actions themselves, we start by

```
def get_car_details(self, license):
    with RequestHandler.CarsLock:
        car = copy.copy(self.Cars.get(license, None))
    if car is not None:
        return (True, car.seats, car.mileage, car.owner)
    return (False, "This license is not registered")
```

- The method tries to acquire the car data lock
  - and blocks until it gets the lock
- It then uses `dict.get()` with a second argument of `None`
  - to get a copy of the car with the given license
  - or to get `None`
- The car is immediately copied and the `with` statement is finished
  - This ensures that the lock is in force for the shortest possible time
- Like all the action-handling methods, we return a tuple
  - whose first item is a Boolean success/failure flag
  - and whose other items vary

The server program

- Now looking at `change_mileage()`

```
def change_mileage(self, license, mileage):
    if mileage < 0:
        return (False, "Cannot set a negative mileage")
    with RequestHandler.CarsLock:
        car = self.Cars.get(license, None)
        if car is not None:
            if car.mileage < mileage:
                car.mileage = mileage
                return (True, None)
            return (False, "Cannot wind the odometer back")
    return (False, "This license is not registered")
```

- Notice that we do one check without acquiring the lock
    - only if the mileage is non-negative
        - we acquire the lock and get the relevant car
        - and if we have a car (i.e., if the license is valid) we must stay within the scope of the lock to change the mileage or to return an error tuple
- If no car has the given license, we drop out of the `with` statement
    - and return an error tuple

The server program

- Now looking at `new_registration()`

```
def new_registration(self, license, seats, mileage, owner):
    if not license:
        return (False, "Cannot set an empty license")
    if seats not in {2, 4, 5, 6, 7, 8, 9}:
        return (False, "Cannot register car with invalid seats")
    if mileage < 0:
        return (False, "Cannot set a negative mileage")
    if not owner:
        return (False, "Cannot set an empty owner")
    with RequestHandler.CarsLock:
        if license not in self.Cars:
            self.Cars[license] = Car(seats, mileage, owner)
            return (True, None)
    return (False, "Cannot register duplicate license")
```

- Again, we can do many sanity checks before accessing registration data
- If the license does not yet exist
  - we create a new Car object and store it in the dictionary;
  - This must all be done within the scope of the same lock
    - we must not allow any other client to add a car with this license in the time between the existence check and adding the new car

The server program

- Finally looking at `shutdown()`

```
def shutdown(self, *ignore):
    self.server.shutdown()
    raise Finish()
```

- This is to be called to shut down the server, which
  - will stop it from accepting further requests, although
  - it will continue running while it is still servicing any existing requests;
- We then raise a custom exception to notify the handler that we are finished
  - this causes the handler to return without sending any reply to the client