Lab 4

Donnacha Oisín Kidney 115702295

Task 1

The client now listens for notifications about the status of services it is interested in. The client can listen for these status messages concurrently with execution of tasks, so currently running tasks can be cancelled if necessary. The server now sends events on the notification manager about its status and errors.

Task 2

```
import threading
import time
from collections import defaultdict


class DuplicateClientNameError(RuntimeError):
    def __init__(self, *args):
        self.args = args


class NotificationManager:
    """A notification manager. Instances of this class will manage publishers
    and subscribers, and send messages from the former to the latter.
    """

    def __init__(self, directory):
        """Initialize a NotificationManager with the directory service.
        """
        self._clients = defaultdict(dict)
        self._topics = set()
        self._name = directory.register("notification_manager", self)
        self._cllock = threading.Lock()
        self._tplock = threading.Lock()

    def register_publisher(self, topic):
        """This method registers a new publisher on the given topic, and
        returns the callback to send out a new event.
        """
        with self._tplock:
            self._topics.add(topic)
```

```
        def publish(event):
            with self._cllock:
                for callback in self._clients[topic].values():
                    callback(event)

        return publish

    def subscribe(self, clientname, callback, *topics):
        """This method takes a client name and callback, and several topics, and
        subscribes that client to those topics. The client name must be unique.
        """
        with self._cllock:
            for topic in topics:
                if clientname in self._clients[topic]:
                    raise DuplicateClientNameError(clientname)
                self._clients[topic][clientname] = callback

    def unsubscribe(self, clientname, *topics):
        """This method unsubscribes the specified client name from the given
        topics.
        """
        with self._cllock:
            for topic in topics:
                del self._clients[topic][clientname]


class Event:
    """An event is what is sent by the notification manager."""

    def __init__(self, topic, contents, time_created=None):
        self.time = time.gmtime() if time_created is None else time_created
        self.topic = topic
        self.contents = contents

    def __str__(self):
        return "New event\nTopic: %s\nTime: %s\n%s" % (
            self.topic, time.strftime("%c", self.time), self.contents)
```

Task 3

Directory:

```
from random import randrange

class ServiceNotFoundError(RuntimeError):
    def __init__(self, *args):
        self.args = args


class Directory:
```

```
    """A directory service."""
    def __init__(self):
        self._services = {}

    def register(self, name, service):
        """Registers a new service, and returns its unique name."""
        if name in self._services:
            newname = name
            while newname in self._services:
                newname = name + str(randrange(2 ** 64))
            name = newname
        self._services[name] = service
        return name

    def lookup_by_name(self, name):
        """Looks up a service given its name. Raises an error if it is not
        found.
        """
        if name in self._services:
            return self._services[name]
        raise ServiceNotFoundError("service of name %s not registered" % name)

    def lookup_by_requirement(self, requirements):
        """Takes a predicate and returns a service which satisfies that
        predicate.
        """
        for name, service in self._services.items():
            if requirements(service):
                return name, service
        raise ServiceNotFoundError(
            "no service found matching specified requirements")


def require_method(method_name):
    """Generate a predicate for requiring a method with a particular name."""
    return lambda instance: callable(getattr(instance, method_name, None))
```

Temperature:

```
import random
import threading
import time

import directory
import notifications


class Temperature(threading.Thread):
    """A class for temperature readings, which sends out the temperature via
    the notification manager.
    """
```

```
    def __init__(self, d, timeout):
        self._pub_callback = d.lookup_by_requirement(
            directory.require_method("register_publisher"))[
                1].register_publisher("temperature")
        threading.Thread.__init__(self)
        self._timeout = timeout

    def run(self):
        temp = random.randrange(100) - 50
        while True:
            newtemp = random.randrange(100) - 50
            if abs(temp - newtemp) > 5:
                self._pub_callback(
                    notifications.Event("temperature",
                                        "the temperature is %i℃" % newtemp))
            temp = newtemp
            time.sleep(self._timeout)
```

Storage:

```
import directory
import notifications
import time


class DuplicateKeyError(RuntimeError):
    def __init__(self, *args):
        self.args = args


class StorageKilledError(RuntimeError):
    def __init__(self, *args):
        self.args = args


class Storage:
    """A service for key-value storage. Any errors are published on its status
    topic.
    """
    def __init__(self, d):
        self._name = d.register("storage", self)
        self._pub_callback = d.lookup_by_requirement(
            directory.require_method("register_publisher"))[
                1].register_publisher("storage_status")
        self._store = {}

    def store(self, key, value):
        """Store a value for the given key. Errors are sent via the notification
        manager.
        """
        if key in self._store:
```

```
            self._pub_callback(
                notifications.Event("storage_status", DuplicateKeyError(key)))
        else:
            self._store[key] = value

    def retrieve(self, key):
        """Retrieve a value for a given key. Errors are sent via the
        notification manager.
        """
        try:
            return self._store[key]
        except KeyError as k:
            self._pub_callback(notifications.Event("storage_status", k))

    def modify(self, key, fn):
        """Modify a value at a given key. Errors are sent via the notification
        manager.
        """
        try:
            self._store[key] = fn(self._store[key])
        except KeyError as k:
            self._pub_callback(notifications.Event("storage_status", k))

    def kill(self):
        """Kill the service. A killed notification is sent via the notification
        manager.
        """
        del self._store
        self._pub_callback(
            notifications.Event("storage_status", StorageKilledError("storage kille
d")))
```

Main test:

```
import threading
import time
import temperature
import directory
import notifications
import storage

d = directory.Directory()
m = notifications.NotificationManager(d)
t = temperature.Temperature(d, 1)
t.start()
s = storage.Storage(d)


class Client:
    """A basic client which stores temperature readings."""
    def __init__(self, d):
```

```
        self._notif_manager = d.lookup_by_requirement(directory.require_method("su
bscribe"))[1]
        self._notif_manager.subscribe("client", self.handle_event, "storage_status
", "temperature")
        self._storage = d.lookup_by_requirement(directory.require_method("store"))
[1]
        self._storage.store("temperatures", {})
        self._continue = True

    def handle_event(self, event):
        if event.topic == "temperature":
            threading.Thread(target=self.log_temperature, args=(event,)).start()
        elif event.topic == "storage_status":
            threading.Thread(target=self.handle_storage_error, args=(event,)).star
t()

    def log_temperature(self, event):
        if self._continue:
            def assgn(d):
                d[time.strftime("%c", event.time)] = event.contents
                return d
            self._storage.modify("temperatures", assgn)
        if self._continue:
            print(self._storage.retrieve("temperatures"))

    def handle_storage_error(self, event):
        self._continue = False
        print(event)
        self._notif_manager.unsubscribe("client", "temperature", "storage_status")


c = Client(d)
time.sleep(5)
s.kill()
```

Output:

```
{'Tue Nov  7 13:08:17 2017': 'the temperature is -26℃'}
{'Tue Nov  7 13:08:18 2017': 'the temperature is 18℃', 'Tue Nov  7 13:08:17 2017':
 'the temperature is -26℃'}
{'Tue Nov  7 13:08:19 2017': 'the temperature is 7℃', 'Tue Nov  7 13:08:18 2017':
'the temperature is 18℃', 'Tue Nov  7 13:08:17 2017': 'the temperature is -26℃'}
{'Tue Nov  7 13:08:19 2017': 'the temperature is 7℃', 'Tue Nov  7 13:08:18 2017':
'the temperature is 18℃', 'Tue Nov  7 13:08:17 2017': 'the temperature is -26℃',
'Tue Nov  7 13:08:20 2017': 'the temperature is 49℃'}
New event
Topic: storage_status
Time: Tue Nov  7 13:08:21 2017
storage killed
```