AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Architecture and Design of Distributed Dependable Systems

Journal on Exercises Ex4-Ex5

Group 12

Authors:
Søren Outze Sørensen - 20091917
Daniela Popovici - 201503243
Troels Knudsen - 201270263

Supervisor:
Finn Overgaard Hansen

1 Introduction
1.1 Intro to requirements for the exercises
1.2 Patterns used in the solution
2 Solution
2.1 Discussion of architecture decisions
2.2 Deployment diagram
2.3 Design and implementation of server and client
2.3.1 Intro to design of server and client
2.3.2 Class diagram
2.3.3 Sequence diagrams
2.3.4 Implementation details
3 Discussion of results
4 Conclusion

Revision History

Revision	Date/Authors	Description
1	07/10/2015 - Troels	Document setup
2	09/10/2015 - Daniela	1.1, 1.2, 2.1, 2.2, 2.3.1, 2.3.4, 3, 4
3		
4		

1 Introduction

1.1 Intro to requirements for the exercises

The paper includes 2 exercises.
Exercise 4 requirements:

1. Design and implement a PC client application using the Connector pattern and the Reactor pattern (from exercise 3). Optional: implement also the asynchronous connect possibility in the Connector.
2. Design and implement a PC server application using the Acceptor pattern and the Reactor pattern (from exercise 3).
3. Implement the use case “Request Patient Information" in the client and server application so that a client application requests a patient information record from a Patient Information System (the PC server).
    a. The client application opens a socket connection to the server
    b. A user inputs a CPR number in the client application.
    c. The client application sends a “GetPatientInfo(CPR number)” request to the server via TCP/IP
    d. The PC Server application search for the patient info in a file and responds with the corresponding PatientInfo record (name, address etc.)
    e. The client application receives and displays the received record
    f. Repeat step 2-6. until exit
    g. The client application closes the connection
4. Measure the time for requesting the patient info.
5. Try to open and close a connection for each patient info request. Measure the time for this request and compare it with the time measured in 3.
6. A UML class diagram showing the classes in the application and one or more sequence diagrams and the source code.

Exercise 5 requirements:
The PC server application from exercise 4 should be modified from a single-threaded to a multi-threaded server

1. Select and use one of the following Concurrency patterns:
    a. Half-Sync/Half-Async Pattern
    b. Leader/Followers Pattern
    c. Proactor Pattern
2. Integrate the selected pattern with the Acceptor/Connector and Reactor based server application from the previous exercises.
3. Implement and test the PC server application with minimum two PC client applications requesting different services concurrently from the server.

4. An example scenario could be clients requesting patient information, sending log records and patient value records concurrently.

1.2 Patterns used in the solution

The following patterns are used in the solution:

1. Acceptor Connector
2. Singleton (centralized dispatching)
3. Bridge (pointer to Proactor Implementation in Proactor)
4. Reactor (completion dispatcher implementation)
5. Proactor (asynchronous event demultiplexer)
6. Async Completion Token (track completion information)

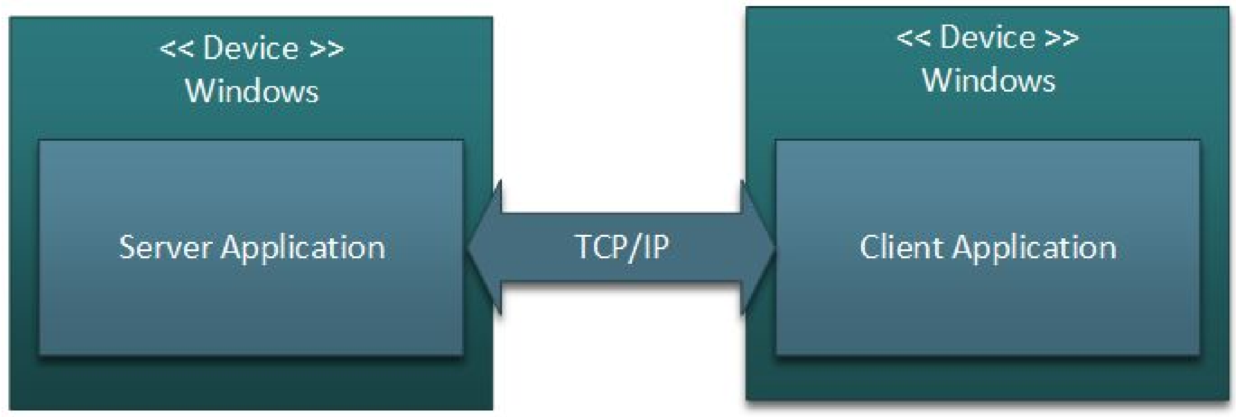
2 Solution

2.1 Discussion of architecture decisions

We made a new project for the Acceptor Connector pattern. It contains 3 .h files: Acceptor, Connector and Service_Handler. All these free implement the Event_Handler interface in order to be able to register them with the Reactor. The Service_Handler defines a generic interface customized by SOCK_Stream from the Wrapper Facade project. The acceptor and the connector are generic interfaces that can be customized using a concrete service handler that implements Service_Handler interface and a acceptor / connector from the Wrapper Facade project. In the Application layer we define the concrete service handler and create an implementation for the open() method.
Another project was created for the Proactor pattern. Its components are: Async_Stream, Async_Result, Async_Stream_Read_Result, Async_Stream_Write_Result, Completion_Handler, Proactor, Proactor_Implementation, Win32_Proactor_Implementation. The proactor holds a pointer to a concrete proactor implementation, which is in this case, Win32_Proactor_Implementation. This latter one implements the abstract class Proactor_Implementation. Async_Stream inherits from OVERLAPPED structure in order to use information in asynchronous input and output. Async_Stream_Write_Result inherits from Async_Result and implements the complete() method that handles a specific event type after the processing finishes. The application layer includes a concrete completion handler which implements the Completion_Handler and Service_Handler interfaces in order to be registrable with the Proactor and able to be used in the acceptor customization.

2.2 Deployment diagram

A deployment diagram would show what hardware components exist (e.g., a web server, an application server, and a database server), what software components ("artifacts") run on each node (e.g., web application, database), and how the different pieces are connected (e.g. JDBC, REST, RMI).
The server and client app can be used as stand alone apps on a windows based operating system device and also, the devices should be in the same local area network in order to be able to test use them.



2.3 Design and implementation of server and client

2.3.1 Intro to design of server and client

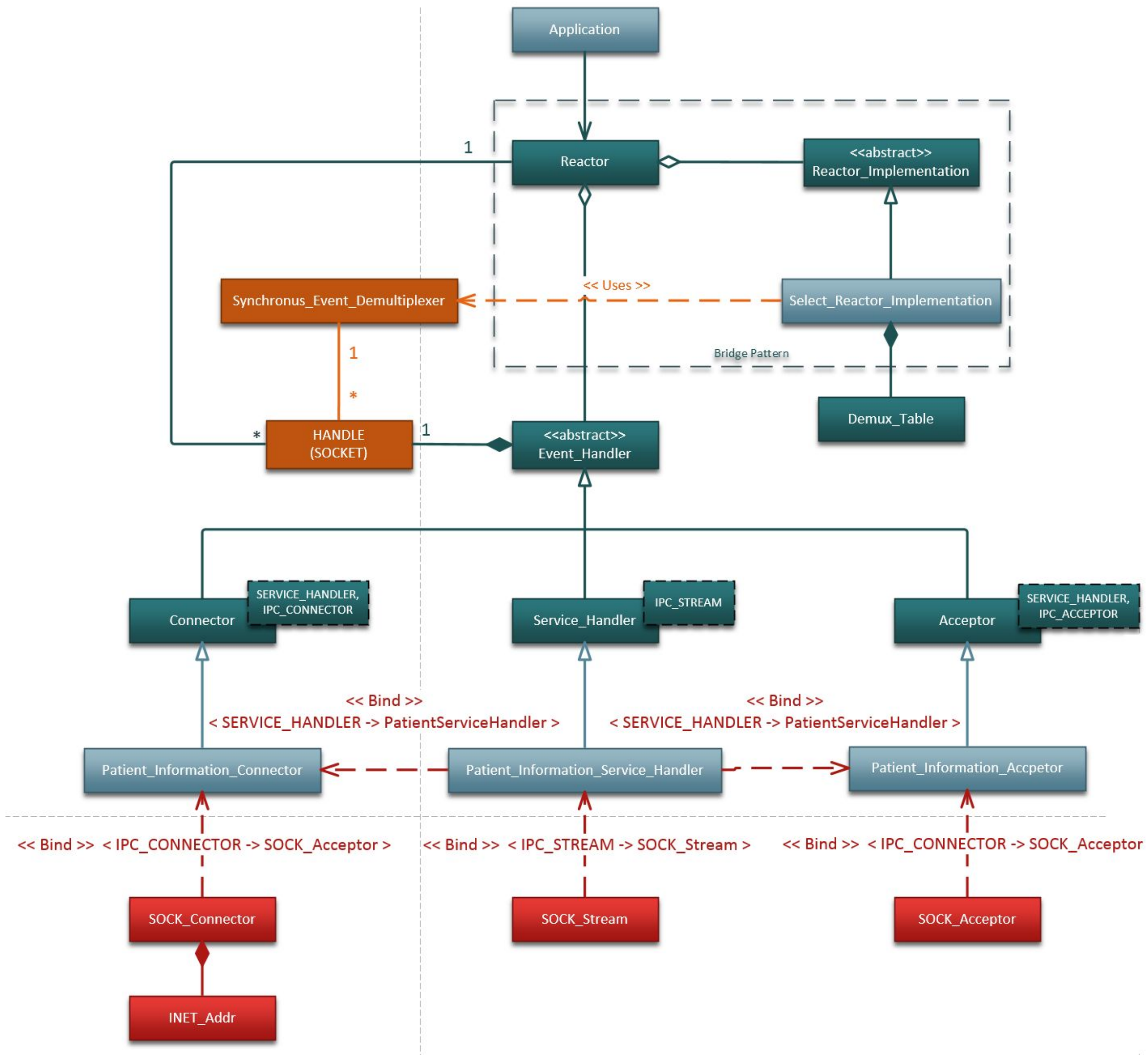
Acceptor Connector:
The application layer contains the concrete service handlers implementation of the open() method. In the server side, the server application creates a concrete service handler using the SOCK_Stream to customize it. This concrete service handler is then used to customize the acceptor along with SOCK_Acceptor. The main method creates an acceptor of that type, passes the INET_Addr and the reactor instance, and handles events on the reactor instance. The client side creates a connector customized with the concrete service handler and the SOCK_Connector. In its main method the client creates a connector of that type, passes the address and the reactor instance to it and handles events on the reactor instance.

Proactor:
On the server a Proactor is implemented to demultiplex the results from asynchronous operations. Events from these asynchronous operations are stored in a completion event queue managed by the operation system which in this case is windows. The server creates an

acceptor customized with a concrete completion handler and SOCK_Acceptor. The acceptor accepts connection and handles them with the reactor instance, but then it handles the completion events with the Proactor. There are 2 clients. One client is sending a request about a patient's information by his CPR and the other client is sending log events. The first client has the implementation from the exercise 4 and the second client has the implementation from the exercise 3.

2.3.2 Class diagram

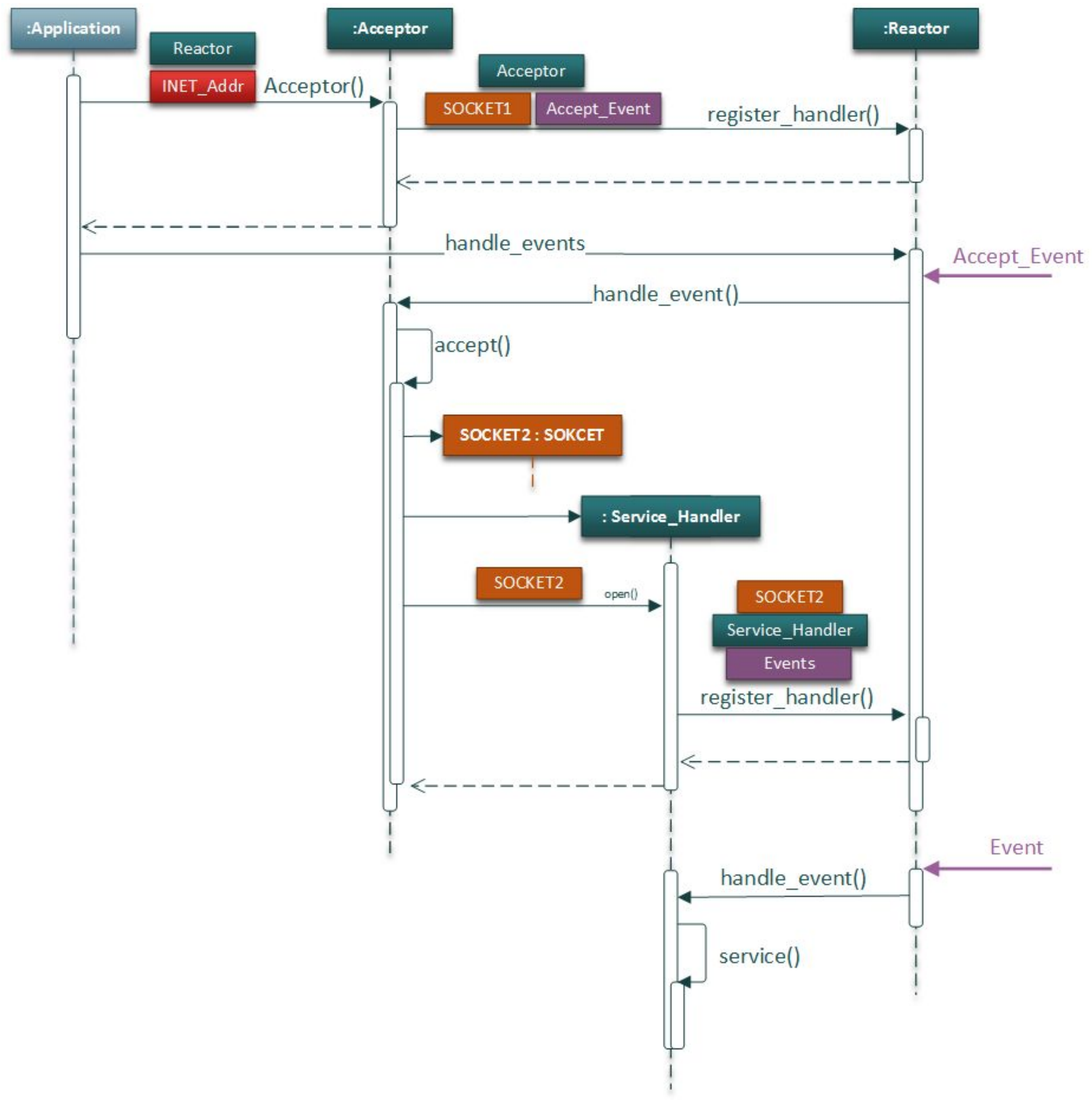
The client and server both use the same base framework for their respective operations. The client makes use of the connector pattern, while the server uses the acceptor pattern. Both client and Server must uses concrete classes for specific domain use, which are made using inheritance from base classes provided by the framework.


Application
Reactor
<<abstract>> Reactor_Implementation
1
Synchronus_Event_Demultiplexer
<< Uses >>
Select_Reactor_Implementation
Bridge Pattern
Demux_Table
1
*
*
HANDLE (SOCKET)
1
<<abstract>> Event_Handler
Connector
SERVICE_HANDLER, IPC_CONNECTOR
Service_Handler
IPC_STREAM
Acceptor
SERVICE_HANDLER, IPC_ACCEPTOR
<< Bind >> < SERVICE_HANDLER -> PatientServiceHandler >
<< Bind >> < SERVICE_HANDLER -> PatientServiceHandler >
Patient_Information_Connector
Patient_Information_Service_Handler
Patient_Information_Accpetor
<< Bind >> < IPC_CONNECTOR -> SOCK_Acceptor >
<< Bind >> < IPC_STREAM -> SOCK_Stream >
<< Bind >> < IPC_CONNECTOR -> SOCK_Acceptor
SOCK_Connector
SOCK_Stream
SOCK_Acceptor
INET_Addr

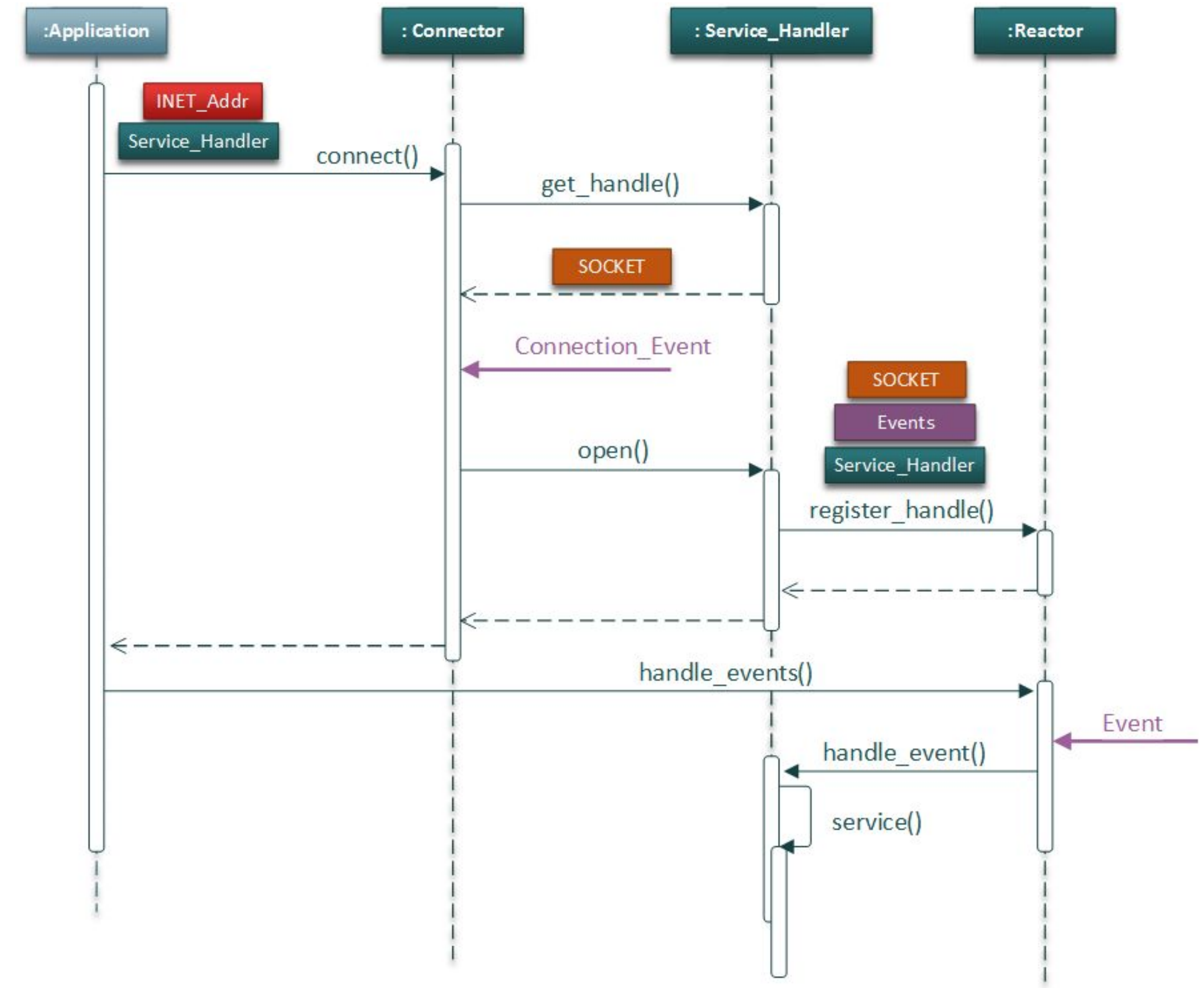
- The red boxes are the wrapper facade classes.
- The orange boxes represent operation system facilities.
- The blue boxes are the application specific classes. They should be implemented for each application use of the framework.
- The green boxes are the reactor and acceptor/connector patterns linked together in one framework.

2.3.3 Sequence diagrams

Server Acceptor



Synchronous connector



2.3.4 Implementation details

Acceptor Connector:

1. Implement the demultiplexing / dispatching infrastructure layer components:
    a. Sockets API - Wrapper Facade
    b. Dispatching mechanisms - Reactor
    c. Connector, Acceptor, Service_Handler implements Event_Handler from the Reactor.
2. Implement the connection management layer components:
    a. Generic service handler interface - SOCK_Stream customization

   b. Define the generic acceptor and connector interface - parameterized types with concrete Service_Handler and SOCK_Acceptor and SOCK_Connector customization.

3. Implement the application layer components - concrete service handler

Proactor:
In the implementation of the Proactor the windows specific operation GetQueuedCompletionStatus is used for taking results out of the queue if any exist. Or wait on a result if the queue is empty.
The class “Async_Result” was created to contain the result from any asynchronous operation. This class extends the OVERLAPPED structure which contains information used for asynchronous input and output.

1. Separate application services into asynchronous operations and completion handlers
2. Define the completion handler interface
3. Implement the asynchronous operation processor: Async_Result inherits from OVERLAPPED; single-method completion handler dispatch interface strategy, asynchronous completion tokens (ACTs) in Async_Sream, Async_Result
4. Define the proactor interface - Reactor exercise (pretty similar)
5. Implement the proactor interface - Reactor exercise (pretty similar), GetQueuedCompletionStatus() asynchronous event demultiplexing function to dequeue the next completion event from the completion port.
6. Determine the number of proactors in the application - singleton
7. Implement the concrete completion handlers
8. Implement the initiators (= main application) - the concrete completion handlers are the initiators

3 Discussion of results

Acceptor Connector: The connector has both SYNC and ASYNC implementations. We have accomplished all the requirements except the measuring of the request.
Proactor: The reactor was used to register the acceptor while the proactor handles the completion handler events. The Client apps are available from the previous exercises and are able to run in order to test the way the Proactor works. There is an issue when calling the GetQueuedCompletionStatus because of the OVERLAPPED pointer that is passed to it. For some reason it throws an exception of violation of accessing the memory 00000x0. The issue is going to be fixed soon.
The requirements were almost fully met.

4 Conclusion

Connector pattern decouples connection initiation from service initialization and service processing. When used in conjunction with other patterns like the Reactor and Acceptor, the Connector pattern enables the creation of flexible and efficient communication software.
The Proactor pattern embodies a powerful design paradigm that supports efficient and flexible event dispatching strategies for high-performance concurrent applications. The Proactor pattern provides the performance benefits of executing operations concurrently, without constraining the developer to synchronous multi-threaded or reactive programming.