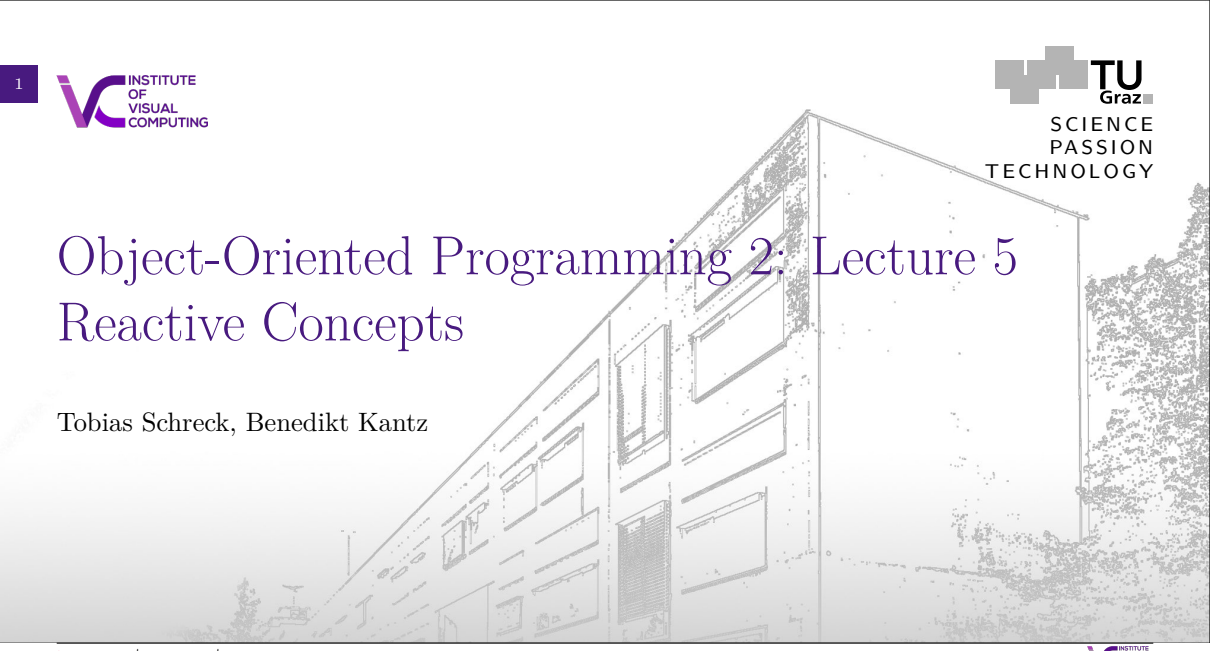
INSTITUTE OF VISUAL COMPUTING

TU Graz
SCIENCE
PASSION
TECHNOLOGY

Object-Oriented Programming 2: Lecture 5 Reactive Concepts

Tobias Schreck, Benedikt Kantz



Have you used `async/await`?

Overview

- Reactive Programming: Concepts
- Indirect Communication: Groups, Message Queues
- Stream Processing
- Virtual Threads
- Futures/Asynchronous Programming

Overview

- Reactive Programming: Concepts
- Indirect Communication: Groups, Message Queues
- Stream Processing
- Virtual Threads
- Futures/Asynchronous Programming

Overview

- Reactive Programming: Concepts
- Indirect Communication: Groups, Message Queues
- Stream Processing
- Virtual Threads
- Futures/Asynchronous Programming

Overview

- Reactive Programming: Concepts
- Indirect Communication: Groups, Message Queues
- Stream Processing
- Virtual Threads
- Futures/Asynchronous Programming

Overview

- Reactive Programming: Concepts
- Indirect Communication: Groups, Message Queues
- Stream Processing
- Virtual Threads
- Futures/Asynchronous Programming

Reactive Manifesto I

Development of reactive software

- Need for reactive programming nowadays:
  - Big data
  - Heterogeneous environments from small devices to server farms
  - User expectations: fast responses in milliseconds, always available

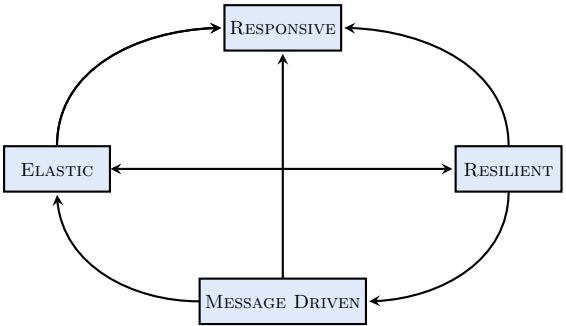
Reactive Manifesto II

Development of reactive software

- Processing done in parallel
- Stream-orientation
- Asynchronous processing
- Respond to data (and events, failures, messages)
- rather than loading all data and processing it completely before coming back with a result
- Coupling of heterogeneous applications/components
- Goals summarized in the **Reactive Manifesto**

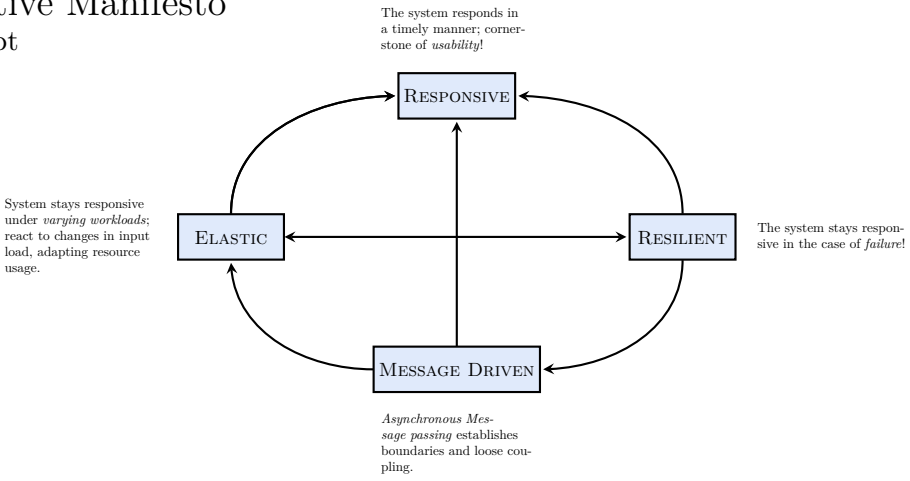
Reactive Manifesto

Concept



Based on the *Reactive Manifesto* [1]

Reactive Manifesto
Concept



Based on the *Reactive Manifesto* [1]

Indirect Communication: Why?

- Basis for distributed systems, interoperability, parallel and reactive applications, redundancy ...
- Temporal Uncoupling
  - Asynchronous communication
  - Sender and receiver do not need to be active (or even exist) at the same time
  - Thread-based tasks!
- Space Uncoupling
  - Identity of sender or receiver does not need to be known to other side
  - Allow exchange, update, or addition of recipients

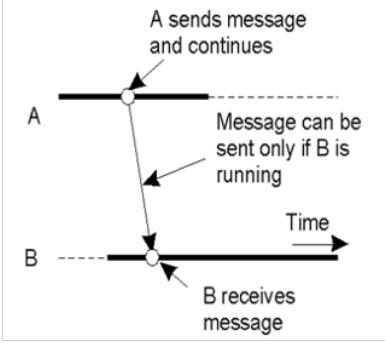
Indirect Communication: Why?

- Basis for distributed systems, interoperability, parallel and reactive applications, redundancy ...
- Temporal Uncoupling
  - Asynchronous communication
  - Sender and receiver do not need to be active (or even exist) at the same time
  - Thread-based tasks!
- Space Uncoupling
  - Identity of sender or receiver does not need to be known to other side
  - Allow exchange, update, or addition of recipients

Indirect Communication: Why?

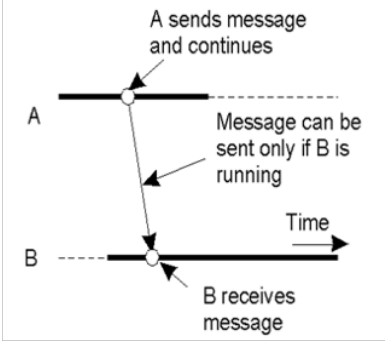
- Basis for distributed systems, interoperability, parallel and reactive applications, redundancy ...
- Temporal Uncoupling
  - Asynchronous communication
  - Sender and receiver do not need to be active (or even exist) at the same time
  - Thread-based tasks!
- Space Uncoupling
  - Identity of sender or receiver does not need to be known to other side
  - Allow exchange, update, or addition of recipients

Transient Communication

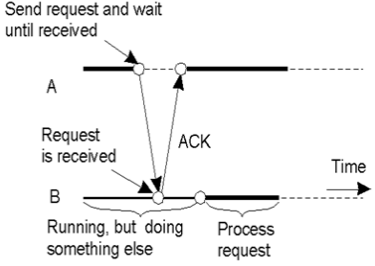


Transient asynchronous
only if receiver is running

Transient Communication

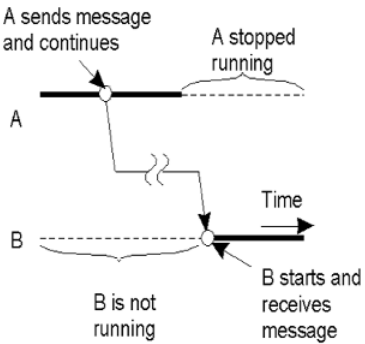


Transient asynchronous
only if receiver is running



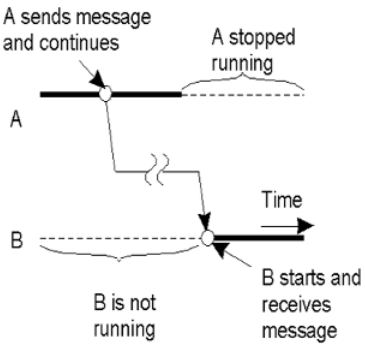
Receiver-oriented transient synchronous
Store on receiver side, sender blocks until acknowledgment of receipt, issued immediately

Persistent Communication

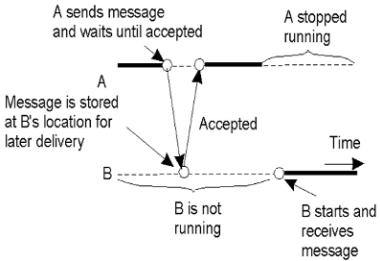


Persistent asynchronous
store on sender side

Persistent Communication



Persistent asynchronous
store on sender side



Persistent synchronous
store on receiver side

Push vs. Pull

- Pull Notification
  - Information sent only after request (Polling)
- Push Notifications
  - Distributed systems often need updates on local object changes (events)
  - Useful to connect heterogeneous systems
  - Efficient notification via multicast
  - Registering of callbacks (reactive programming)
- Reliable or unreliable notification possible

Push vs. Pull

- Pull Notification
  - Information sent only after request (Polling)
- Push Notifications
  - Distributed systems often need updates on local object changes (events)
  - Useful to connect heterogeneous systems
  - Efficient notification via multicast
  - Registering of callbacks (reactive programming)
- Reliable or unreliable notification possible

Push vs. Pull

- Pull Notification
  - Information sent only after request (Polling)
- Push Notifications
  - Distributed systems often need updates on local object changes (events)
  - Useful to connect heterogeneous systems
  - Efficient notification via multicast
  - Registering of callbacks (reactive programming)
- Reliable or unreliable notification possible

Applications

- Indirect communication used for distributed systems in which ...
  - Changes are happening frequently
    - Nodes connect and disconnect frequently
  - Nodes are unknown (anonymous nodes)
- Examples
  - Mobile environments
  - Peer-to-peer communication

Applications

- Indirect communication used for distributed systems in which ...
  - Changes are happening frequently
    - Nodes connect and disconnect frequently
  - Nodes are unknown (anonymous nodes)
- Examples
  - Mobile environments
  - Peer-to-peer communication

Motivation

- Why indirect communication?
  - "All problems in computer science can be solved by another level of indirection." (David Wheeler, father of the subroutine)
- But: indirect communication can be expensive!
  - "There is no performance problem that cannot be solved by eliminating a level of indirection." (Jim Gray, Pioneer of Database Research, Turing Award Winner 1998)

Motivation

- Why indirect communication?
  - "All problems in computer science can be solved by another level of indirection." (David Wheeler, father of the subroutine)
- But: indirect communication can be expensive!
  - "There is no performance problem that cannot be solved by eliminating a level of indirection." (Jim Gray, Pioneer of Database Research, Turing Award Winner 1998)

Space and Time Coupling in Distributed Systems

Properties

	Time-Coupled	Time-Uncoupled
Space-Coupling	Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time	Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes
Space-Uncoupling	Sender does not need toknow the identity of the receiver(s); receiver(s) must exist at that moment in time	Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes

Based on Ananda and Poo [2].

Space and Time Coupling in Distributed Systems

Examples

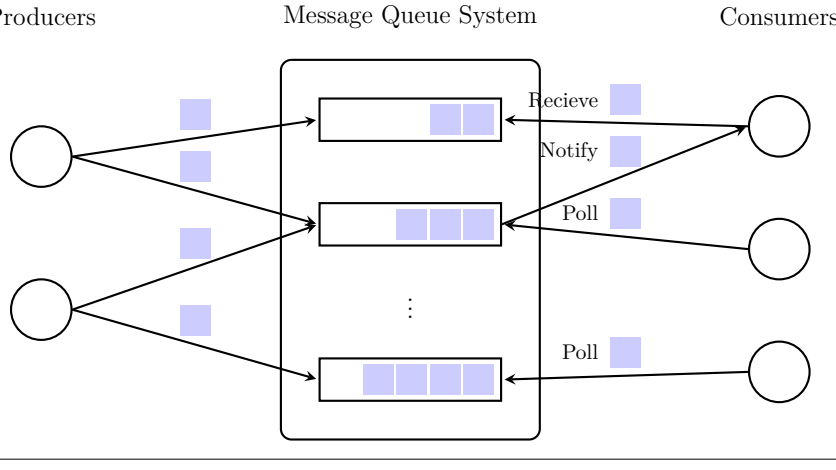
	Time-Coupled	Time-Uncoupled
Space-Coupling	Message passing, remote invocation	Mail
Space-Uncoupling	IP Multicast	Message Queues

Based on Ananda and Poo [2].

Group Communication

- Example (network): IP multicast
  - Infrastructure support to deliver IP packets to multiple destinations
- Example (application): Mailing list
  - List server expands to list names
- Commonalities
  - Clients: need operations to join, leave groups
  - Multicast vs broadcast

Message Queues
Overview



Message Queues
Examples: DIY

- Message Passing
  - Implement queues on sender and recipient side
  - Marshalling
- Message Queues
  - Queues decoupled from sender and receiver
  - First In First Out (FIFO) or priority

Message Queues
Examples: DIY

- Message Passing
  - Implement queues on sender and recipient side
  - Marshalling
- Message Queues
  - Queues decoupled from sender and receiver
  - FIFO or priority

Message Queues
Examples: DIY

- Message Passing
  - Implement queues on sender and recipient side
  - Marshalling
- Message Queues
  - Queues decoupled from sender and receiver
  - FIFO or priority

Message Queues
Examples: Middlewares

- RabbitMQ [1]
- MQTT: IoT Applications [2]
- Apache Kafka [3]
  - Widely used by many big companies
  - Netflix, Oracle, ...
- All of them: APIs for all major languages

[1] https://www.rabbitmq.com/tutorials
[2] https://mqtt.org/
[3] https://kafka.apache.org/powered-by

Message Queues

Examples: Middlewares

- RabbitMQ [1]
- MQTT: IoT Applications [2]
- Apache Kafka [3]
  - Widely used by many big companies
  - Netflix, Oracle, ...
- All of them: APIs for all major languages

[1] https://www.rabbitmq.com/tutorials
[2] https://mqtt.org/
[3] https://kafka.apache.org/powered-by

Message Queues

Examples: Middlewares

- RabbitMQ [1]
- MQTT: IoT Applications [2]
- Apache Kafka [3]
  - Widely used by many big companies
  - Netflix, Oracle, ...
- All of them: APIs for all major languages

[1] https://www.rabbitmq.com/tutorials
[2] https://mqtt.org/
[3] https://kafka.apache.org/powered-by

Message Queues

Examples: Middlewares

- RabbitMQ [1]
- MQTT: IoT Applications [2]
- Apache Kafka [3]
  - Widely used by many big companies
  - Netflix, Oracle, ...
- All of them: APIs for all major languages

[1] https://www.rabbitmq.com/tutorials
[2] https://mqtt.org/
[3] https://kafka.apache.org/powered-by

Message Queues

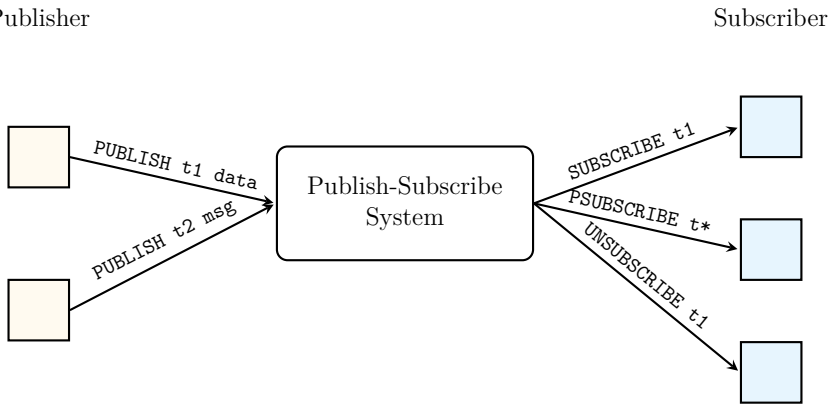
Examples: Middlewares

- RabbitMQ [1]
- MQTT: IoT Applications [2]
- Apache Kafka [3]
  - Widely used by many big companies
  - Netflix, Oracle, ...
- All of them: APIs for all major languages

[1] https://www.rabbitmq.com/tutorials
[2] https://mqtt.org/
[3] https://kafka.apache.org/powered-by

Publish-Subscribe

Overview



Publish-Subscribe based on on Ananda and Poo [2] and Redis [3].

Publish-Subscribe
Definition

Publish-Subscribe

- Participants do not know each other (Spatial uncoupling)
- Broker manages participants, distributed brokers (i.e. Redis)
- Delivery type: Direct (multicast), MQ (temporal uncoupling)

Publish-Subscribe
Definition

Publish-Subscribe

- Participants do not know each other (Spatial uncoupling)
- Broker manages participants, distributed brokers (i.e. Redis)
- Delivery type: Direct (multicast), MQ (temporal uncoupling)

Publish-Subscribe
Definition

Publish-Subscribe

- Participants do not know each other (Spatial uncoupling)
- Broker manages participants, distributed brokers (i.e. Redis)
- Delivery type: Direct (multicast), MQ (temporal uncoupling)

Publish-Subscribe
Definition

Publish-Subscribe

- Participants do not know each other (Spatial uncoupling)
- Broker manages participants, distributed brokers (i.e. Redis)
- Delivery type: Direct (multicast), MQ (temporal uncoupling)

Publish-Subscribe

Delivery Guarantees

- **At most once**: not resilient, usually ephemeral messages (*Redis*)
- **At least once**: resilient, but might double deliver messages
- **Exactly once**: guarantees exactly one delivery of each message (*Kafka*)

Publish-Subscribe

Delivery Guarantees

- **At most once**: not resilient, usually ephemeral messages (*Redis*)
- **At least once**: resilient, but might double deliver messages
- **Exactly once**: guarantees exactly one delivery of each message (*Kafka*)

Publish-Subscribe

Delivery Guarantees

- **At most once**: not resilient, usually ephemeral messages (*Redis*)
- **At least once**: resilient, but might double deliver messages
- **Exactly once**: guarantees exactly one delivery of each message (*Kafka*)

Publish-Subscribe

Delivery Guarantees

- **At most once**: not resilient, usually ephemeral messages (*Redis*)
- **At least once**: resilient, but might double deliver messages
- **Exactly once**: guarantees exactly one delivery of each message (*Kafka*)

Publish-Subscribe
Subscription modes

- By Channel: e.g. name
- By Topic: flag on channel
- By Type: super-topic
- By Content: Based on structure of content/attributes

Publish-Subscribe

Subscription modes

- By Channel: e.g. name
- By Topic: flag on channel
- By Type: super-topic
- By Content: Based on structure of content/attributes

Publish-Subscribe

Subscription modes

- By Channel: e.g. name
- By Topic: flag on channel
- By Type: super-topic
- By Content: Based on structure of content/attributes

Publish-Subscribe I

Example: Redis[a]

[a] https://redis.io/docs/latest/develop/pubsub/

```
SUBSCRIBE alerts.user
```

- Subscribe client to channel
- Delivery type: *At-most-once* (Compare Kafka: *Exactly Once*)
- Selection: by channel

Publish-Subscribe II

Example: Redis[a]

[a] https://redis.io/docs/latest/develop/pubsub/

```
PSUBSCRIBE alerts.*
```

- Subscribe client to channel
- Delivery type: *At-most-once* (Compare Kafka: *Exactly Once*)
- Selection: by channel

Publish-Subscribe III

Example: Redis[a]

[a] https://redis.io/docs/latest/develop/pubsub/

```
PUBLISH alert.user singout
```

- Publish the data in channel
- Data can be arbitrary string (e.g. Base64/JSON encoded data!)

Stream Processing

- Streams provide a high-level abstraction for processing sequences of elements
- Can be processed sequentially or in parallel
- Functional-style operations (map, filter, reduce, etc.)

Stream Processing

- Streams provide a high-level abstraction for processing sequences of elements
- Can be processed sequentially or in parallel
- Functional-style operations (map, filter, reduce, etc.)

```
List<Float> numbers =
↪  FloatStream.rangeClosed(1,100_000);
numbers.stream()
        .map(Math::exp)
        .forEach(System.out::println);
```

Stream Processing

- Streams provide a high-level abstraction for processing sequences of elements
- Can be processed sequentially or in parallel
- Functional-style operations (map, filter, reduce, etc.)

```
List<Float> numbers =
↪  FloatStream.rangeClosed(1,100_000);
numbers.stream()
        .parallel()
        .map(Math::exp)
        .forEach(System.out::println);
```

Streams
Examples

- See a detailed example in the repository!

Executors for Virtual Threads (Java 21+)

- Java 21 introduces virtual threads for lightweight concurrency
- Virtual creates an executor that assigns each task to a new virtual thread
- Great for high-concurrency workloads: blocking I/O threads!

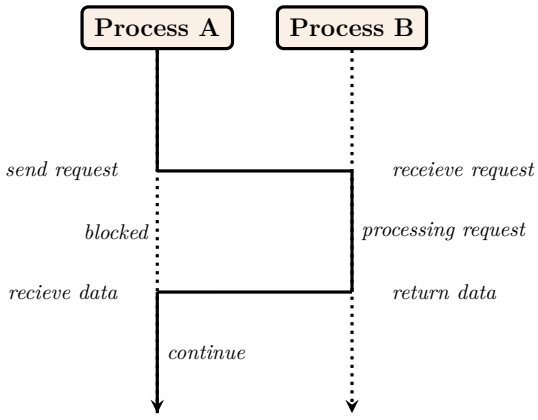
```
try (ExecutorService vexec =
    Executors
        .newVirtualThreadPerTaskExecutor()) {
        vexec.submit(() -> doWork());
}
```

Virtual Threads
Examples

- See a detailed example in the repository!

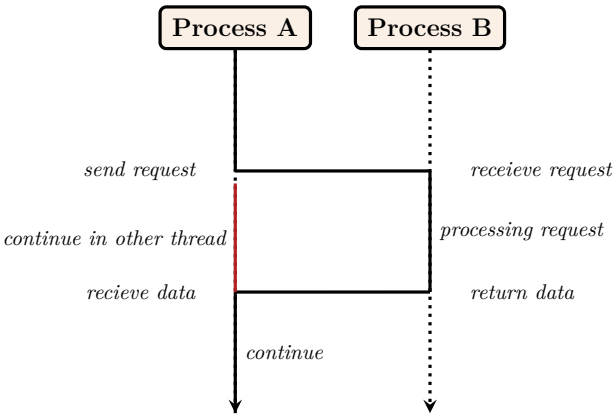
Asynchronous Programming

- Until now: blocking access to I/O (network, files, ...)
- Enables the process to continue running while waiting on blocked tasks



Asynchronous Programming

- Until now: blocking access to I/O (network, files, ...)
- Enables the process to continue running while waiting on blocked tasks



Asynchronous Examples
Java Futures

- Java uses `Future` and `CompletableFuture` for async tasks.
- Submit tasks to an executor and retrieve results later.

```
ExecutorService executor =
↪  Executors.newSingleThreadExecutor();
Future<Integer> future = executor.submit(() -> {
    Thread.sleep(1000);
    return 42;
});
System.out.println("Doing other work...");
Integer result = future.get(); // blocks if not done
System.out.println("Result: " + result);
executor.shutdown();
```

Asynchronous Examples
JavaScript `async/await`

- JavaScript uses `async` functions and `await` for promises.
- Allows writing asynchronous code that looks synchronous.

```
async function fetchData() {
    let response = await
    ↪  fetch('https://api.example.com/data');
    let data = await response.json();
    console.log(data);
}
console.log("Fetching...");
fetchData();
```

Asynchronous Examples
Python `async/await`

- Python uses `async def` and `await` for coroutines.
- Requires an event loop to run async functions.

```python
import asyncio
async def main():
    print("Waiting...")
    await asyncio.sleep(1)
    print("Done!")
asyncio.run(main())
```

Feedback

Bibliography I

References

[1] J. Boner, D. Farley, R. Kuhn, and M. Thompson, Sep. 2014. [Online]. Available: https://www.reactivemanifesto.org/pdf/the-reactive-manifesto-2.0.pdf#page=2.00.

[2] A. Ananda and G. Poo, **Distributed systems: Concepts and design: By george coulouris, jean dollimore and tim kindberg, addison-wesley, 2nd ed, 1994, 644 pp, isbn 0201624338,** eng, *Computer communications*, vol. 18, no. 7, pp. 521–522, 1995, ISSN: 0140-3664.

[3] **Redis Pub/sub,** [Online; accessed 5. Oct. 2025], Oct. 2025. [Online]. Available: https://redis.io/docs/latest/develop/pubsub.