Reactive Streams — Explained Using a Food Delivery App

A food delivery system is a **perfect analogy** for Reactive Streams because it naturally involves:

- Producers
- Consumers
- Flow control
- Overload protection

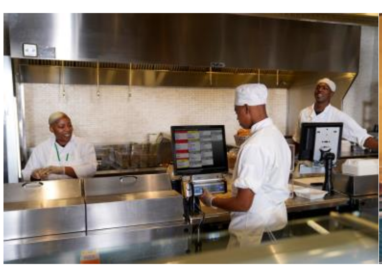
Let's map everything clearly.

Reactive Streams Roles in Food Delivery

Reactive Streams	Food Delivery App
Publisher	Restaurant Kitchen
Subscriber	Delivery Partner
Subscription	Order Control Agreement
Processor	Delivery App System
Backpressure	"Send only what I can handle" rule

---

1 Publisher = Restaurant Kitchen



The **kitchen produces orders**.

Just like a Publisher:

- Generates items (food orders)
- Cannot overwhelm delivery partners
- Should wait for demand

If the kitchen prepares unlimited meals without coordination:

Orders pile up
Food gets cold
Chaos

---

2 Subscriber = Delivery Partner

The delivery partner **consumes orders**.

Like a Subscriber:

- Receives tasks (deliveries)
- Has limited capacity
- Must control workload

A rider cannot carry 25 orders at once.

---

3 Subscription = Flow Control Contract

When a delivery partner connects to the system:

The system gives a **Subscription**

Meaning:

“Tell me how many orders you can handle.”

The rider responds with:

request(2)

Translation:

“I can deliver 2 orders now.”

---

4 Backpressure = Overload Protection

Backpressure is the **core safety mechanism**.

Without it:

- Kitchen floods orders
- Rider overloaded
- Late deliveries
- System failure

With it:

Rider requests orders at their pace
Kitchen obeys demand
Smooth operations

Proper Reactive Streams Flow (Food Delivery Version)

Step 1 — Subscription Created

Delivery partner becomes available.

System:

onSubscribe(subscription)

Meaning:

“Connection established. You control the flow.”

Step 2 — Rider Signals Demand

Rider:

request(3)

Meaning:

“Send me 3 deliveries.”

Step 3 — Kitchen Sends Exactly 3 Orders

Kitchen prepares & dispatches:

onNext(order1)
onNext(order2)
onNext(order3)

NOT more. Never extra.

Step 4 — Rider Finishes & Requests Again

After deliveries:

request(2)

Flow continues safely.

What Happens WITHOUT Backpressure?

Kitchen keeps pushing:

- 10 → 20 → 50 orders

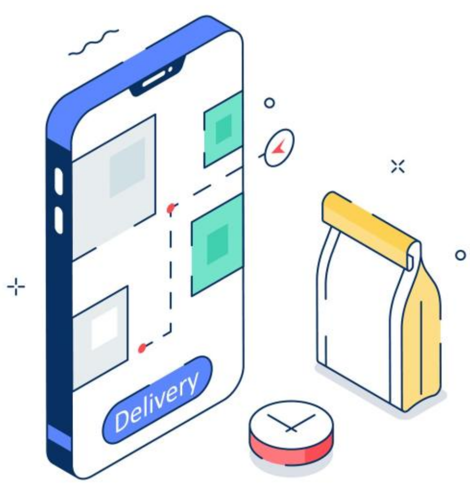
Rider:

Overloaded
Cannot deliver
Food spoils
Customers angry

System collapses.

---

5 Processor = Delivery App System



Real time system Assignment

Unit-1

2 marks

State whether the following statements are TRUE or FALSE. Justify your answer.

1) A hard real-time application consists of only hard real-time tasks.

- A hard real-time application consists of only hard real-time tasks. It is **FALSE.**
- A hard real-time application may also contain several non-real-time tasks such as logging activities etc.

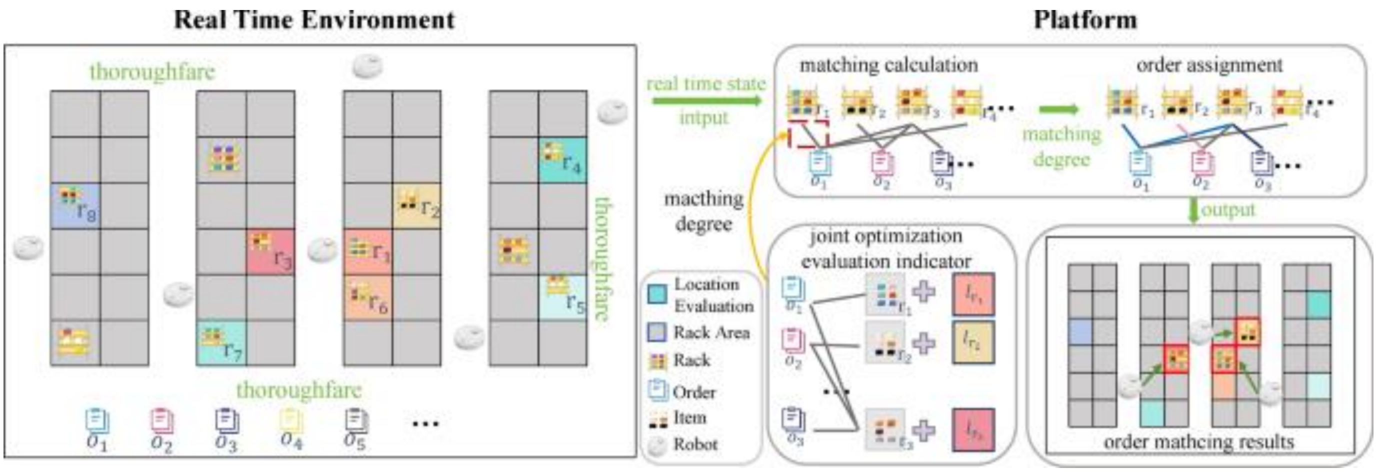
2) Every safety-critical real-time system contains a fail-safe state.

- Every safety-critical real-time system contains a fail-safe state. It is **FALSE.**
- It is false because having fail-safe states in safety-critical real-time systems is meaningless because failure of a safety-critical system can lead to loss of lives, cause damage, etc. E.g.: a navigation system on-board an aircraft.

3) A deadline constraint between two stimuli is a behavioral constraint on the environment of the system.

- A deadline constraint between two stimuli is a behavioral constraint on the environment of the system. The statement is **TRUE.**
- This is because it is a behavioral constraint since the constraint is imposed on the second stimulus event.

4) Hardware fault-tolerance techniques are easily adaptable to provide software fault-tolerance.



The **app acts like a Processor**:

- Receives orders from customers
- Transforms / filters / routes them
- Sends to delivery partners

It is BOTH:

Subscriber (to customer orders)
Publisher (to riders)

Why This Model Is Brilliant

Reactive Streams ensures:

No rider overload
No wasted food

No system crashes
Dynamic scaling

Each consumer controls its capacity.

Key Insight

Backpressure is simply:

"Do not send more work than I requested."

Controlled via:

subscription.request(n)