Introduction to Reactive Streams

Current & Future

Simone Bordet

Who Am I

- **Simone Bordet**
  - sbordet@webtide.com - @simonebordet

- **Lead Architect at Webtide**
  - Jetty's HTTP/2, SPDY, FastCGI and HTTP client maintainer

- **Open Source Contributor**
  - Jetty, CometD, MX4J, Foxtrot, LiveTribe, JBoss, Larex

- **CometD project leader**

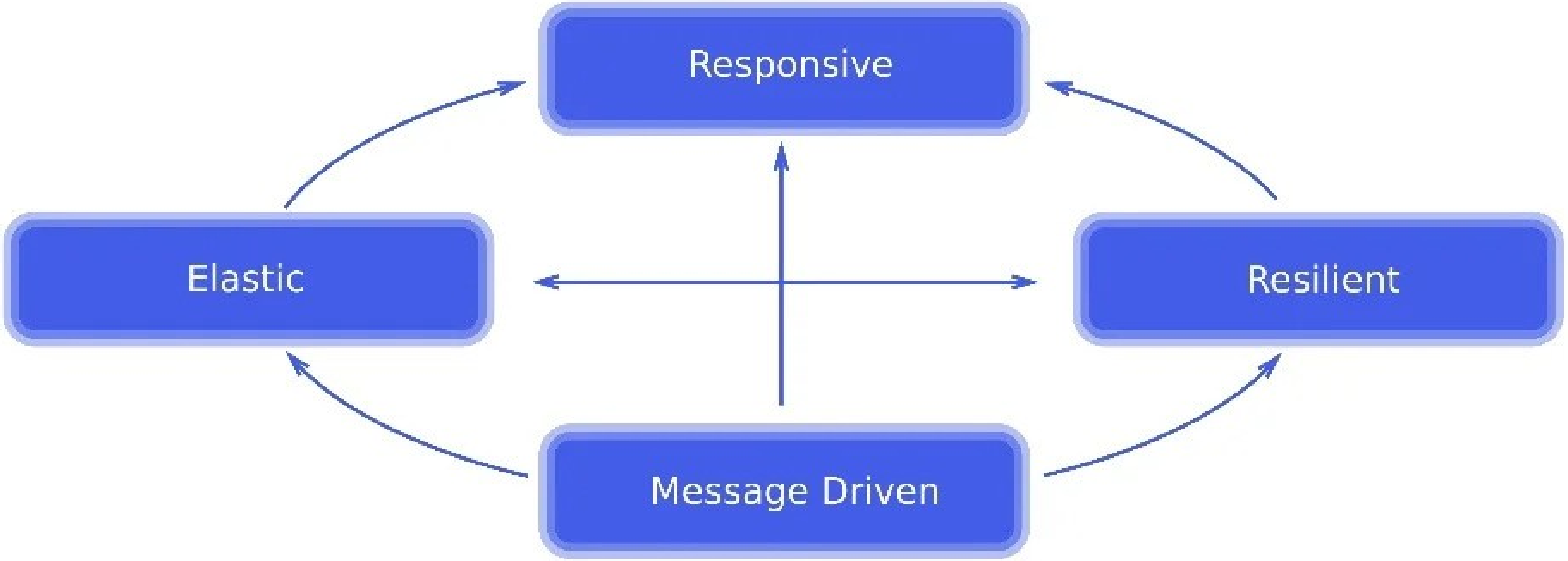
Reactive Systems

Reactive Systems

- **Dramatic changes for IT organizations**
  - From tens of servers to the cloud
  - From latencies in the seconds to milliseconds
  - From gigabytes of data to petabytes
  - From desktop apps to web and mobile apps

- **Software architectures of 10 years ago ?**
  - Not suitable anymore

Reactive Systems



Reactive Systems

- **Message-Driven**
  - The system relies on asynchronous message passing
- **Responsive**
  - The system responds in a timely manner
- **Elastic**
  - The system remains responsive under varying load

Reactive Systems

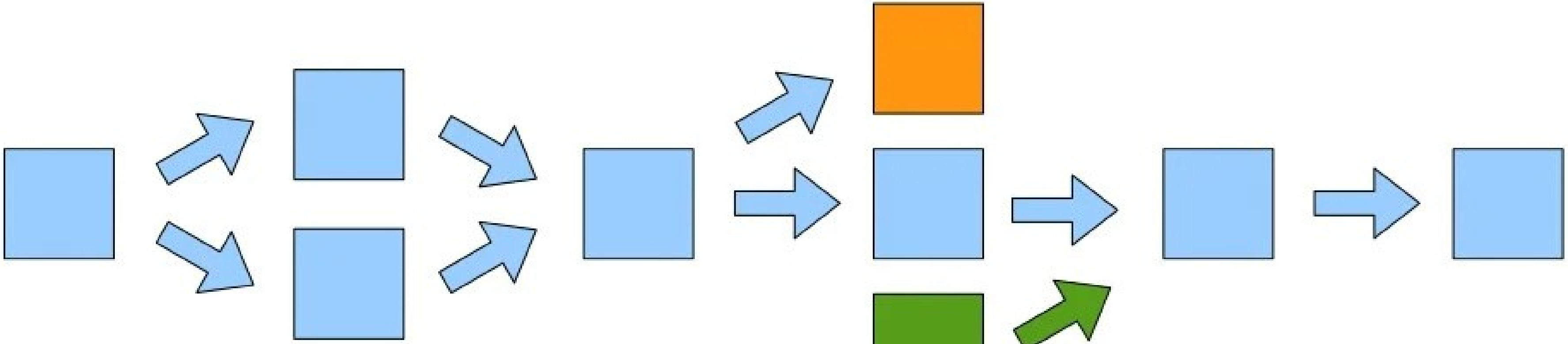
- **How does writing software change ?**
- **Must embrace asynchronous programming**
  - Asynchronous message passing
  - Asynchronous I/O, backpressure
- **BUT ! Asynchronous programming is HARD !**
- **Reactive Streams**

Reactive Streams

Reactive Streams

- **STREAM**
  - An ephemeral flow of data to transform

- **Multiple transformations, multiple systems**



Reactive Streams

- **Example: a typical REST request processing**

System	Message
HTTP IN	(Request + InputStream) →
REST IN	(URI + String) →
JDBC IN	(String) →
JDBC OUT	(ResultSet) →
REST OUT	(String) →
HTTP OUT	(Response + OutputStream)

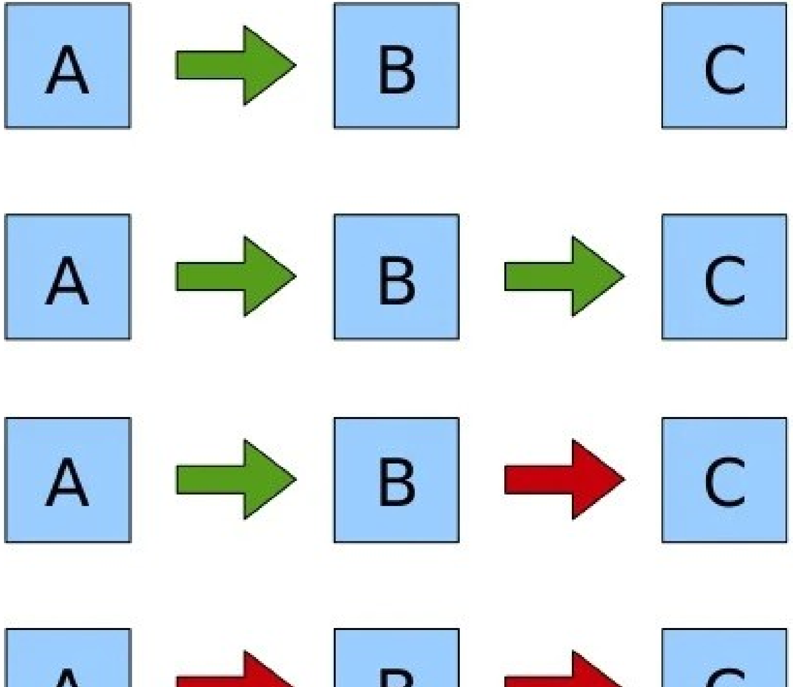
Blocking Messaging

Blocking Messaging

- **Systems that use blocking messaging**
  - Blocking socket I/O
  - Method calls with return values
  - Blocking Queues
- **Producer blocks while consumer processes**
- **System throughput is self-limited**
  - Slowest consumer drives the throughput

Blocking Messaging

- **C is slow, blocks B which blocks A**



Blocking Messaging

- The information “BLOCKED” travels the stream in the opposite direction
- Blocking messaging provides flow control
- BACKPRESSURE == Flow Control

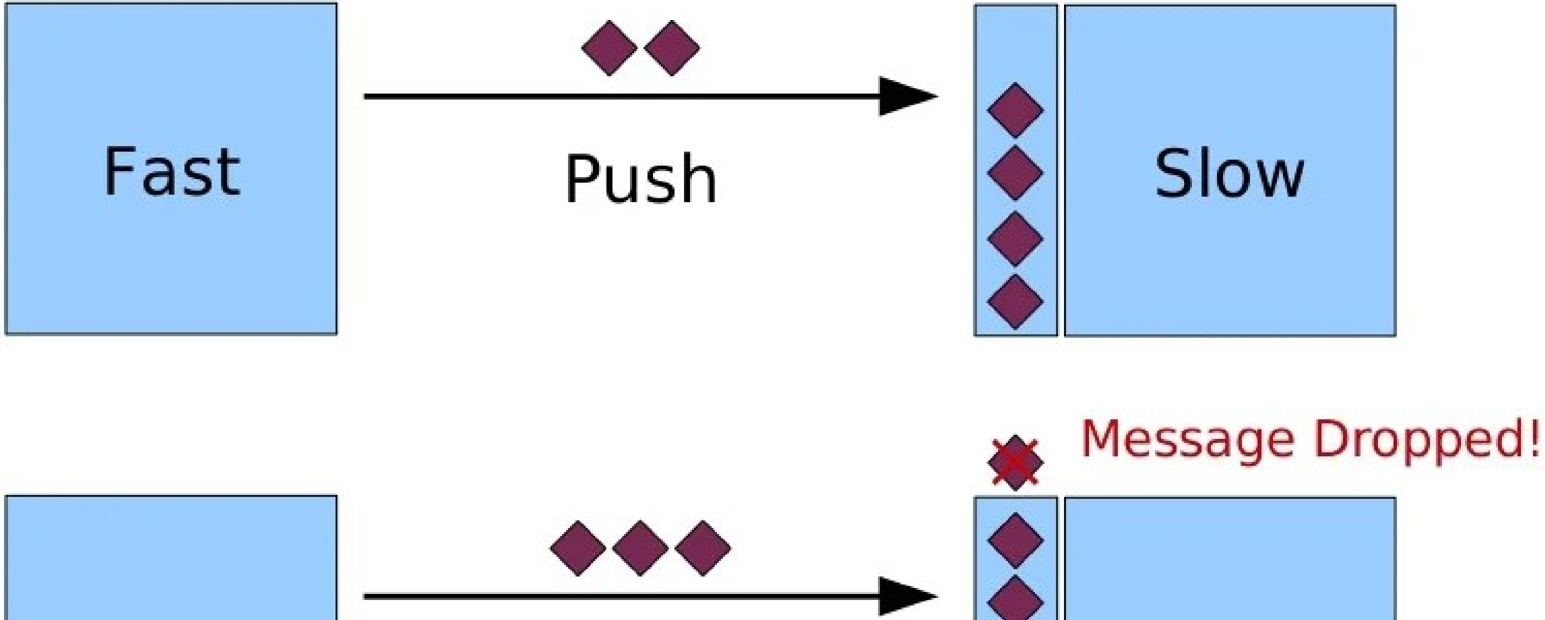
Asynchronous Messaging

Asynchronous Messaging

- **Blocking messaging consumes resources**
  - Memory, threads, CPU, etc.
  - 100 HTTP requests for a long JDBC query → 100 threads
- **Asynchronous messaging reduces resource consumption**
  - Less threads, less memory, etc.
- **Asynchronous messaging introduces new problems**

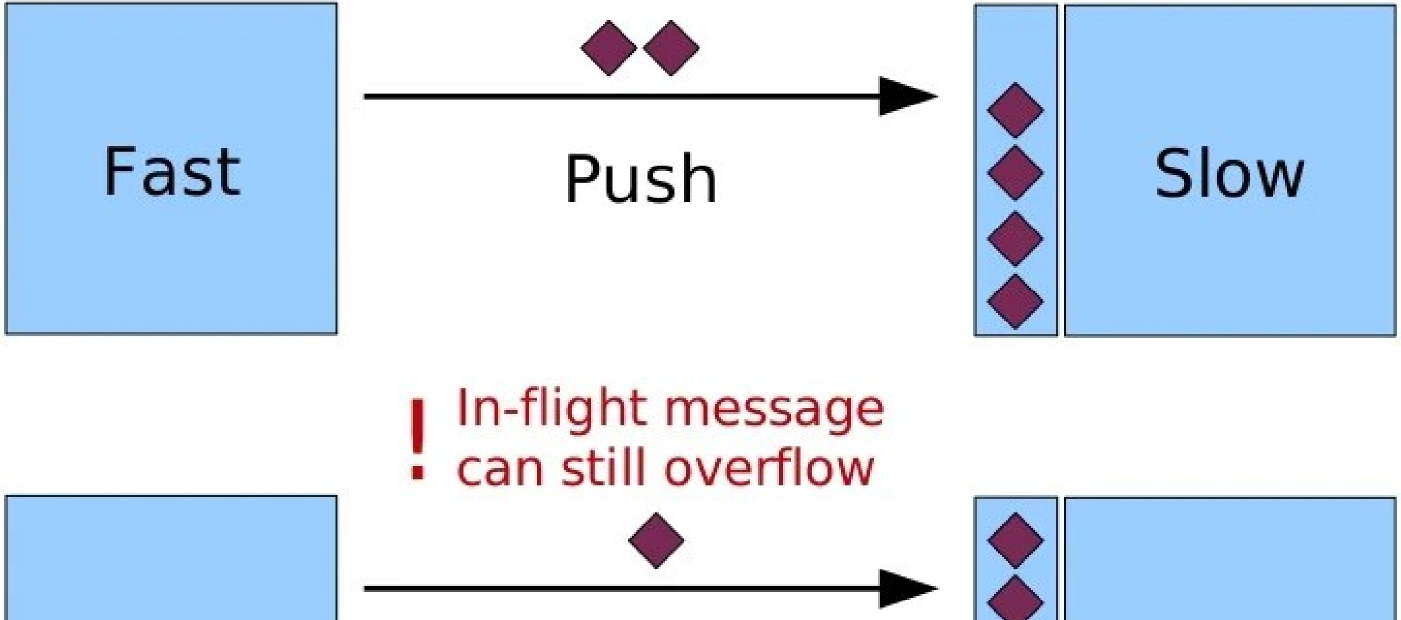
Asynchronous Messaging

- **Fast Producer Problem: Message Dropping**
  - Consumers typically have a queue



Asynchronous Messaging

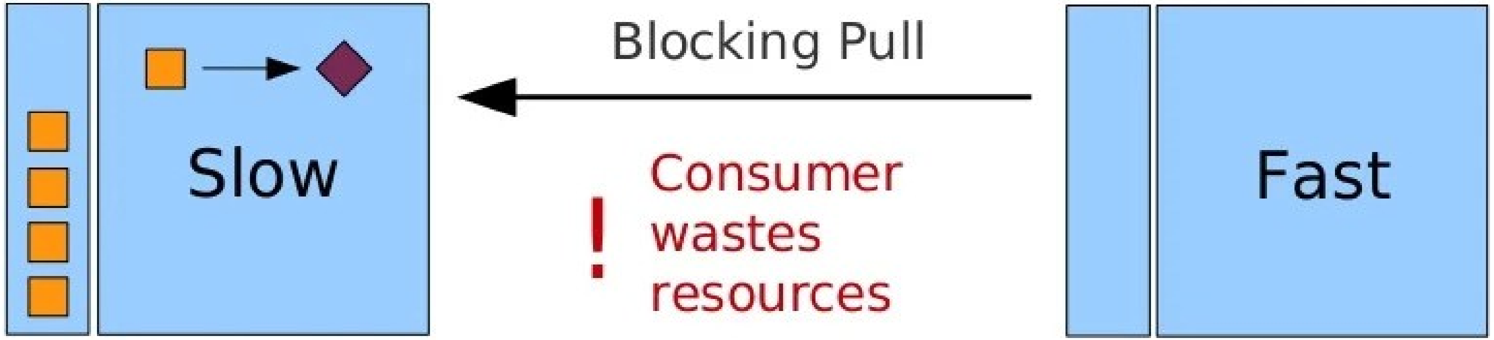
- **Fast Producer Problem: Rejection/Negative ACK**



! In-flight message
can still overflow

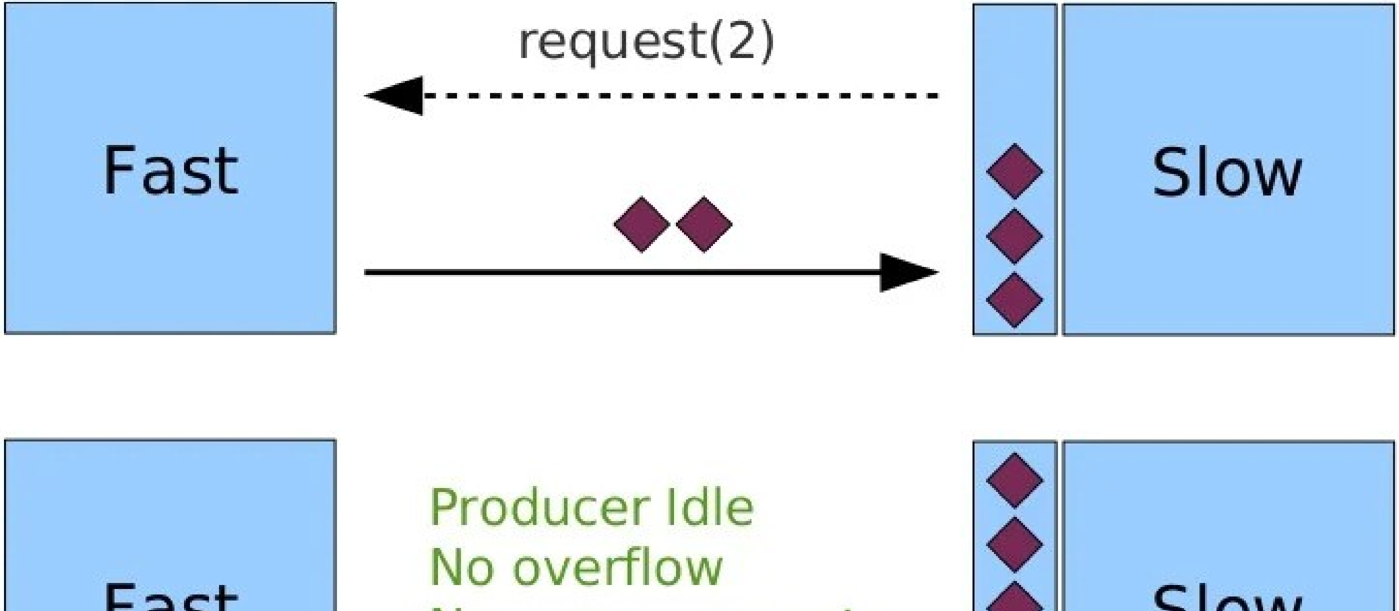
Asynchronous Messaging

Fast Consumer Problem: Blocking Pull



Asynchronous Messaging

- **Reactive Streams**
  - Dynamic Push/Pull



request(2)

Fast

Slow

Fast

Producer Idle
No overflow

Slow

Asynchronous Messaging

- **Reactive Streams**
  - Backpressure obtained with lack of demand
  - Push behavior when consumer is fast
  - Pull behavior when consumer is slow
  - Automatic switch between push & pull
- **Backpressure propagates**
  - C is slow
    - B must slow down
      - A must slow down

Reactive Streams API

Reactive Streams API

```
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> s);
}
public interface Subscriber<T> {
    void onSubscribe(Subscription s);
    void onNext(T t);
    void onError(Throwable t);
    void onComplete();
}
public interface Subscription {
    void request(long n);
    void cancel();
```

Reactive Streams API

- **ReactiveStreams 1.0**
  - http://www.reactive-streams.org/
  - https://github.com/reactive-streams/reactive-streams-jvm
  - Maven: org.reactivestreams:reactive-streams:1.0.0
- **RxJava**
  - https://github.com/ReactiveX/RxJavaReactiveStreams
- **Akka Streams**
  - https://github.com/akka/akka
- **Vert.x**

Reactive Streams API

- **MongoDB Example**

```
MongoCollection<Document> collection = ...
Publisher<Success> publisher = collection.insertOne(document);
publisher.subscribe(new Subscriber<Success>() {
    public void onSubscribe(Subscription s) {
        s.request(1);  // <-- Only now insertion will occur
    }
    public void onNext(Success success) { /* Inserted */ }
    public void onError(Throwable t) { /* Failed */ }
    public void onComplete() {}
});
```

Reactive Streams API

- **MongoDB Example**

```
Publisher<Document> publisher = collection.find();
publisher.subscribe(new Subscriber<Document>() {
    private Subscription _subscription;
    public void onSubscribe(Subscription s) {
        _subscription = s; s.request(1);
    }
    public void onNext(Document document) {
        System.out.println(document);
        _subscription.request(1);
    }
```

Future Work

Future Work

- **JDK 9**
  - `java.util.concurrent.Flow`
  - http://download.java.net/jdk9/docs/api/java/util/concurren

- **Servlet 4.0 considering using Flow**
  - For HTTP processing
  - For asynchronous I/O

- **HTTP clients ? REST clients ? JDBC drivers ?**

Conclusions

Conclusions

- **Look into new technologies**

- **Evaluate the benefits of going Reactive**
  - You may need only some of the 4 features

- **Reactive Streams**
  - Worth using now
  - May simplify your code a lot

Questions
&
Answers