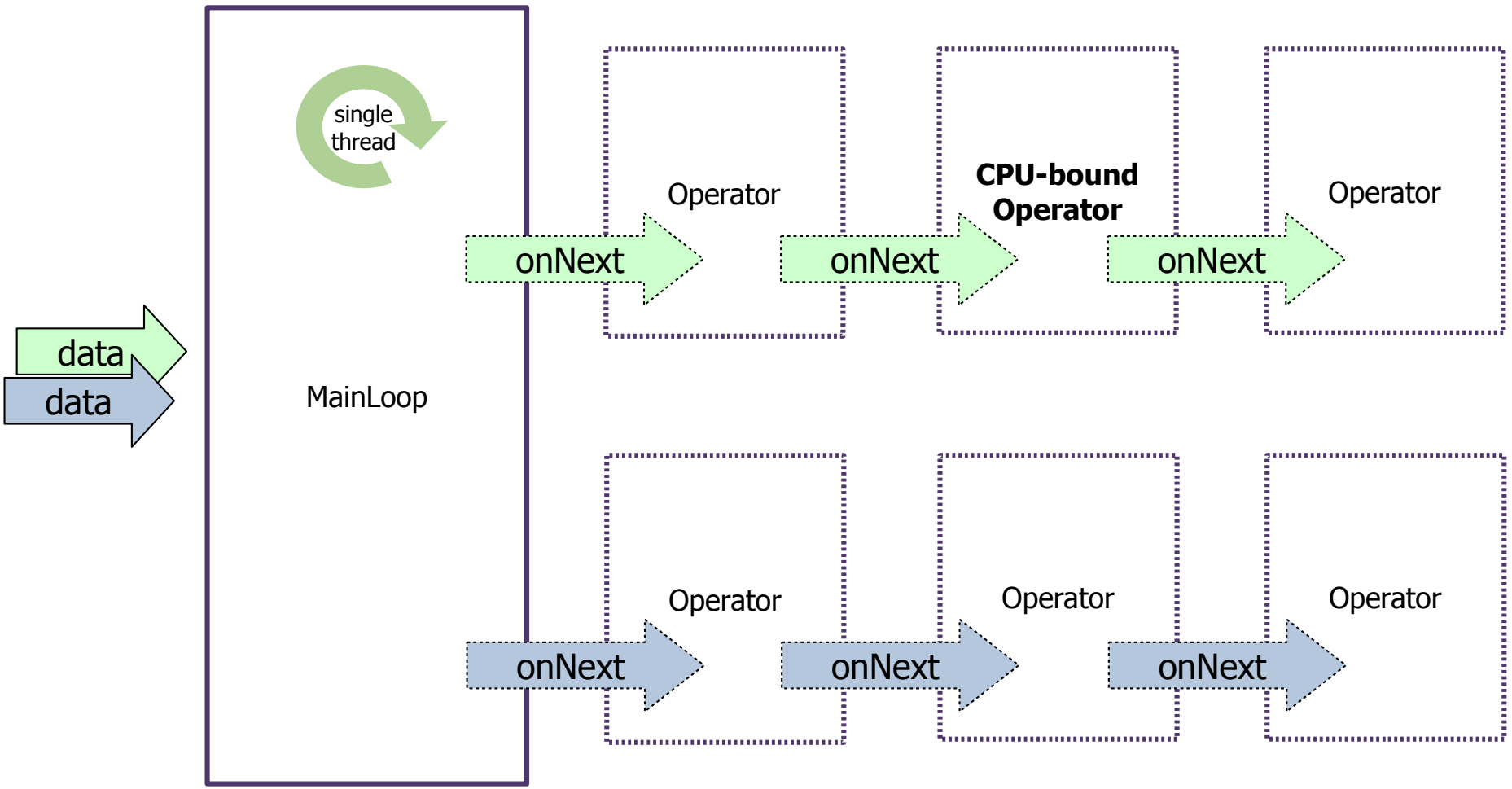
Challenges

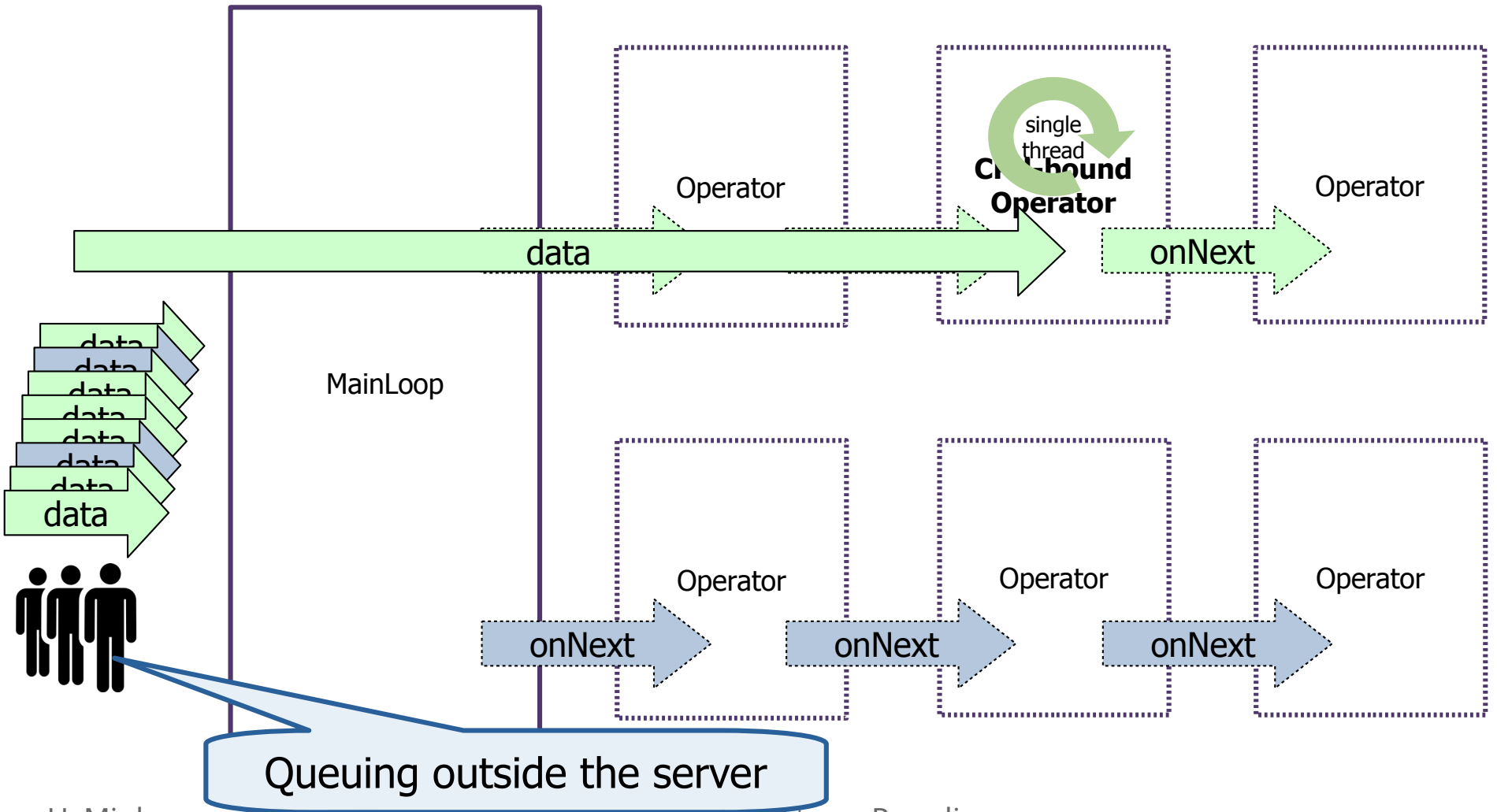
- Using a single thread avoids the complications of concurrent programming (races)

- What if a single CPU core is not enough?
- What if a single stream is using more that its share of resources?

Buffer-based application



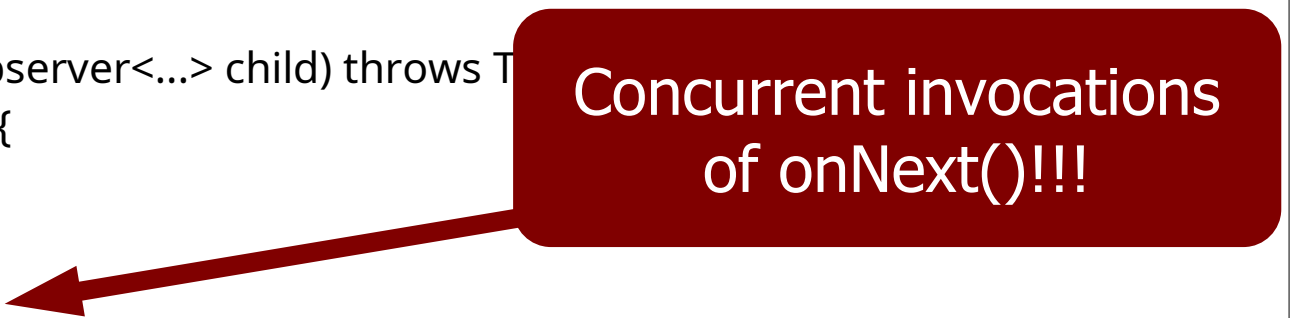
Buffer-based application



Naive parallel operator

```
public class NaiveParallelOperator implements ObservableOperator<T,R> {

   public Observer<...> apply(Observer<...> child) throws T
      return new Observer<...>() {
         public void onNext(...) {
            new Thread(()->{
                  child.onNext(...);
            }).start();
         }
         public void onError(Throwable e) {
            ...
         }
         public void onComplete() {
            ...
         }
    }
}
```



Naive parallel operator

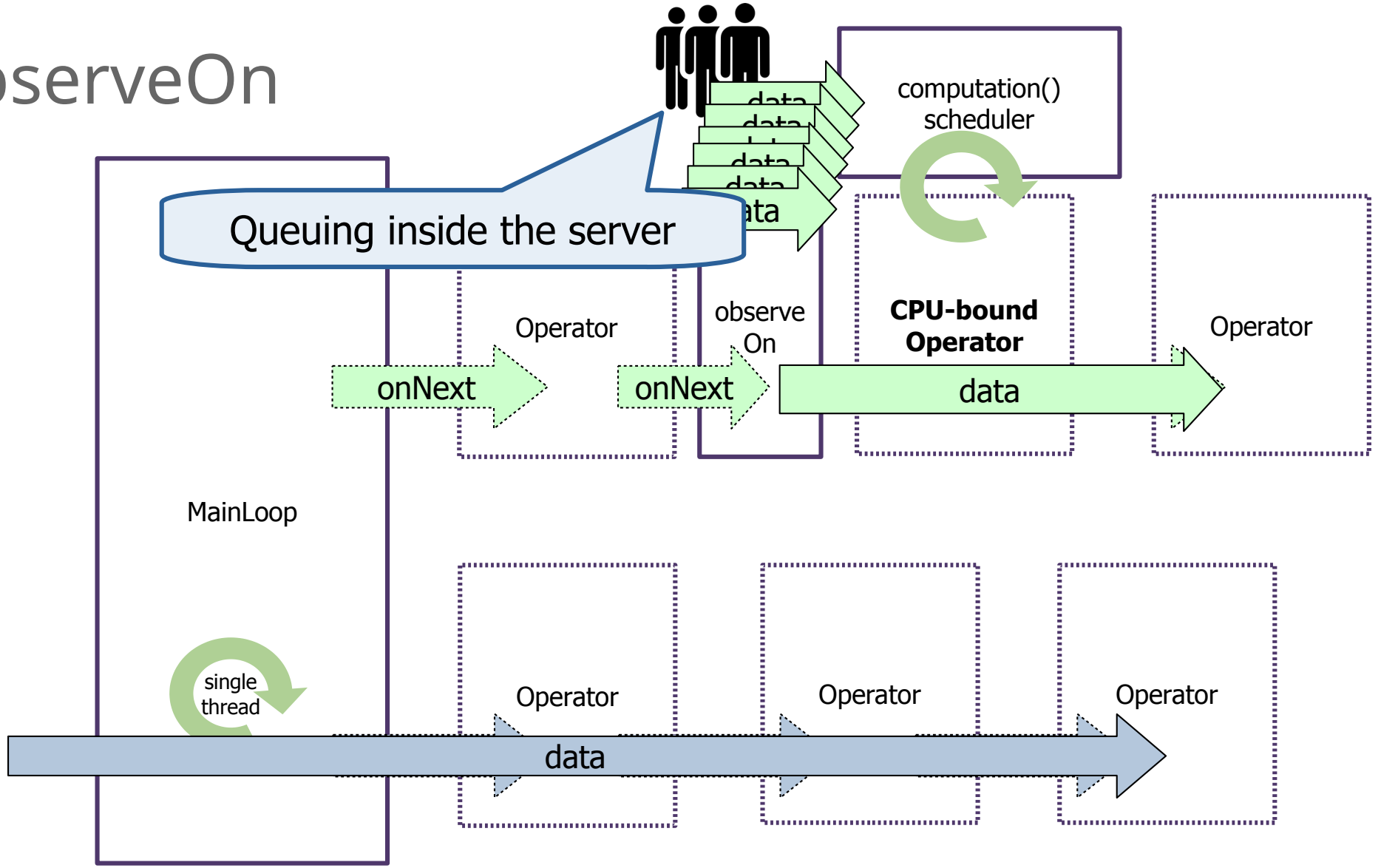
- Frees the calling thread to execute other requests
- Has many issues:
  - Concurrent invocations of onNext are not allowed
    - Races, out of order processing, …
  - Synchronization of onNext with onComplete/onError
  - Does not limit the maximum number of threads
    - #threads > #cores is inefficient
    - potential resource exhaustion

observeOn

- observeOn(...) operator redirects execution to a different thread, but ensures synchronization
- Threads are provided by a scheduler that manages threads according to a policy:
  - computation(), io(), ...

```
server.subscribe(conn -> {
        conn
                .op1(...)
                .observeOn(computation())
                .op2(...)
                .subscribe(...);
});
```

observeOn

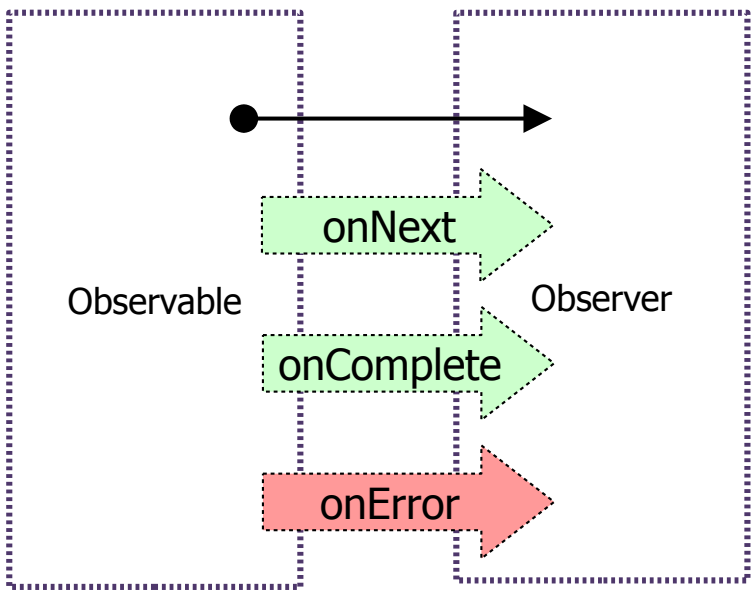

computation()
scheduler
Queuing inside the server
Operator
observe
On
CPU-bound
Operator
Operator
onNext
onNext
data
MainLoop
single
thread
Operator
Operator
Operator
data

observeOn

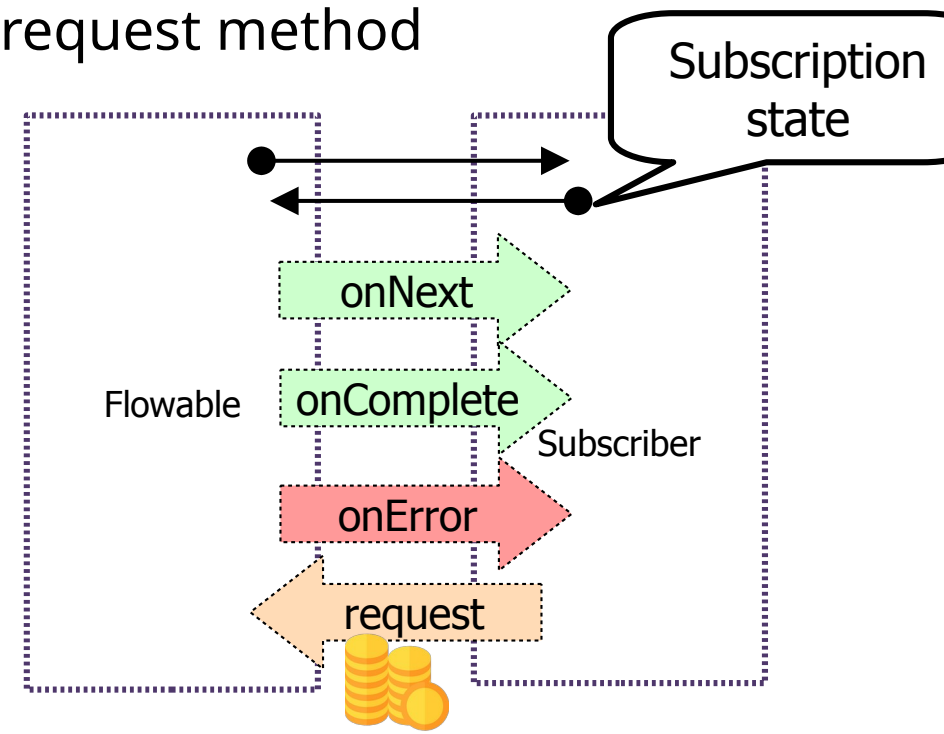
- Frees the main loop thread to process other requests
- Can be used to implement the parallel pipeline pattern:
  - Use observeOn multiple times in the same stream to delimit stages
  - Multiple stages execute in parallel
- Main problem: the waiting queue is now inside the server, consuming memory
- Limiting the queue blocks the main loop again

Back pressure

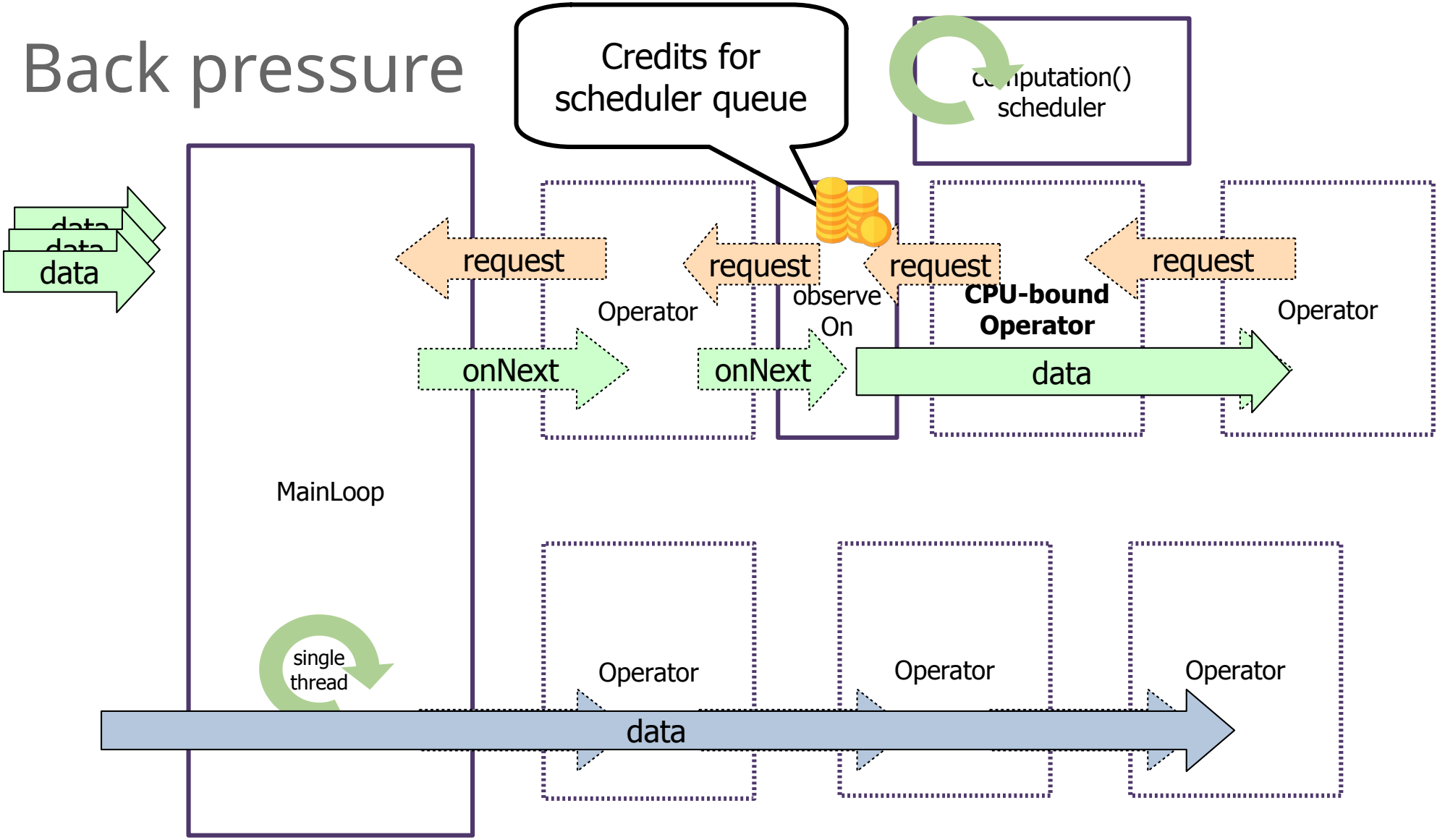
- The observer cannot influence the observable
  - No reference back and no interface



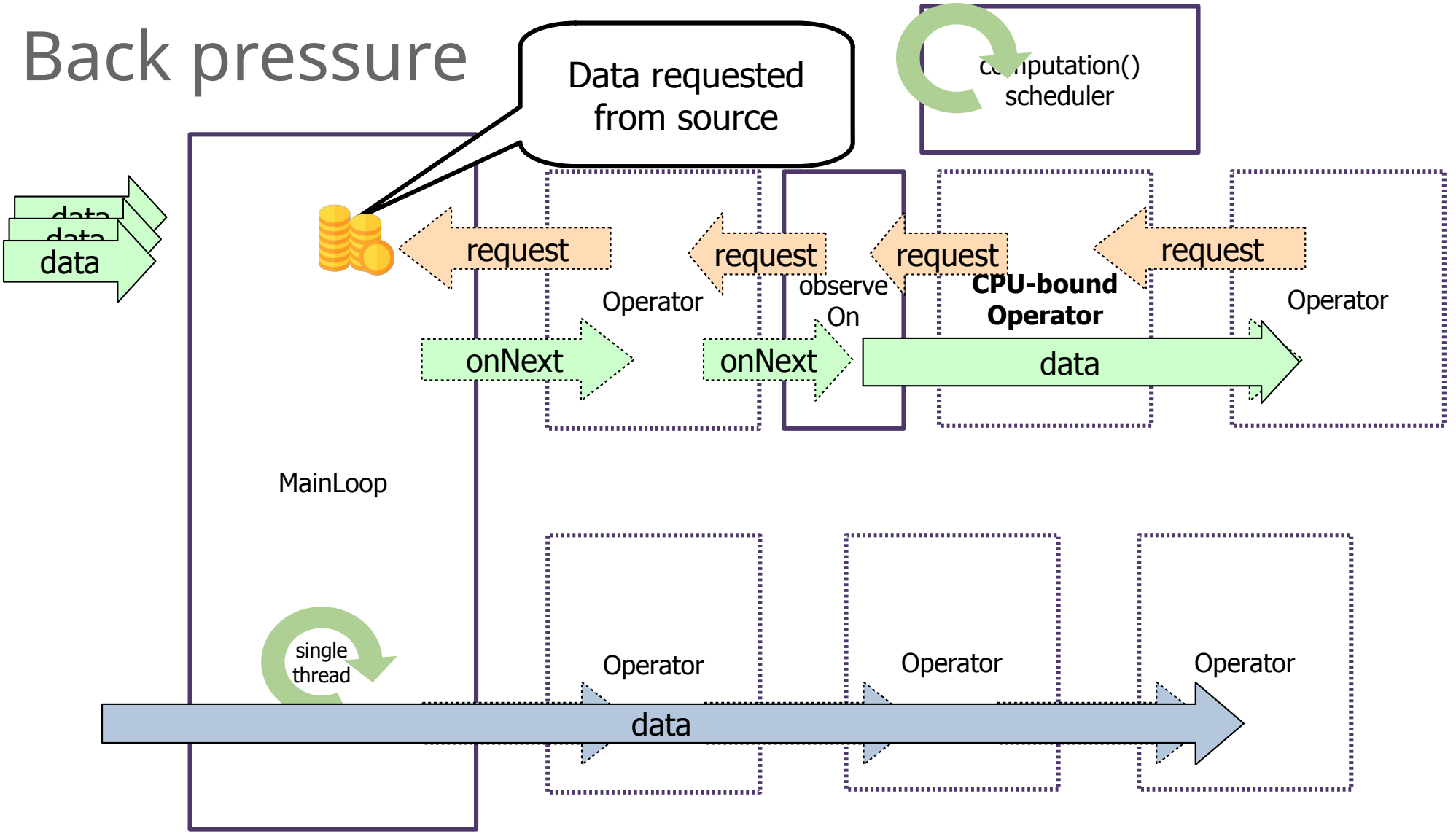
- Flowable streams can only invoke onNext when there is credit left
- Credit is provided by invoking request method



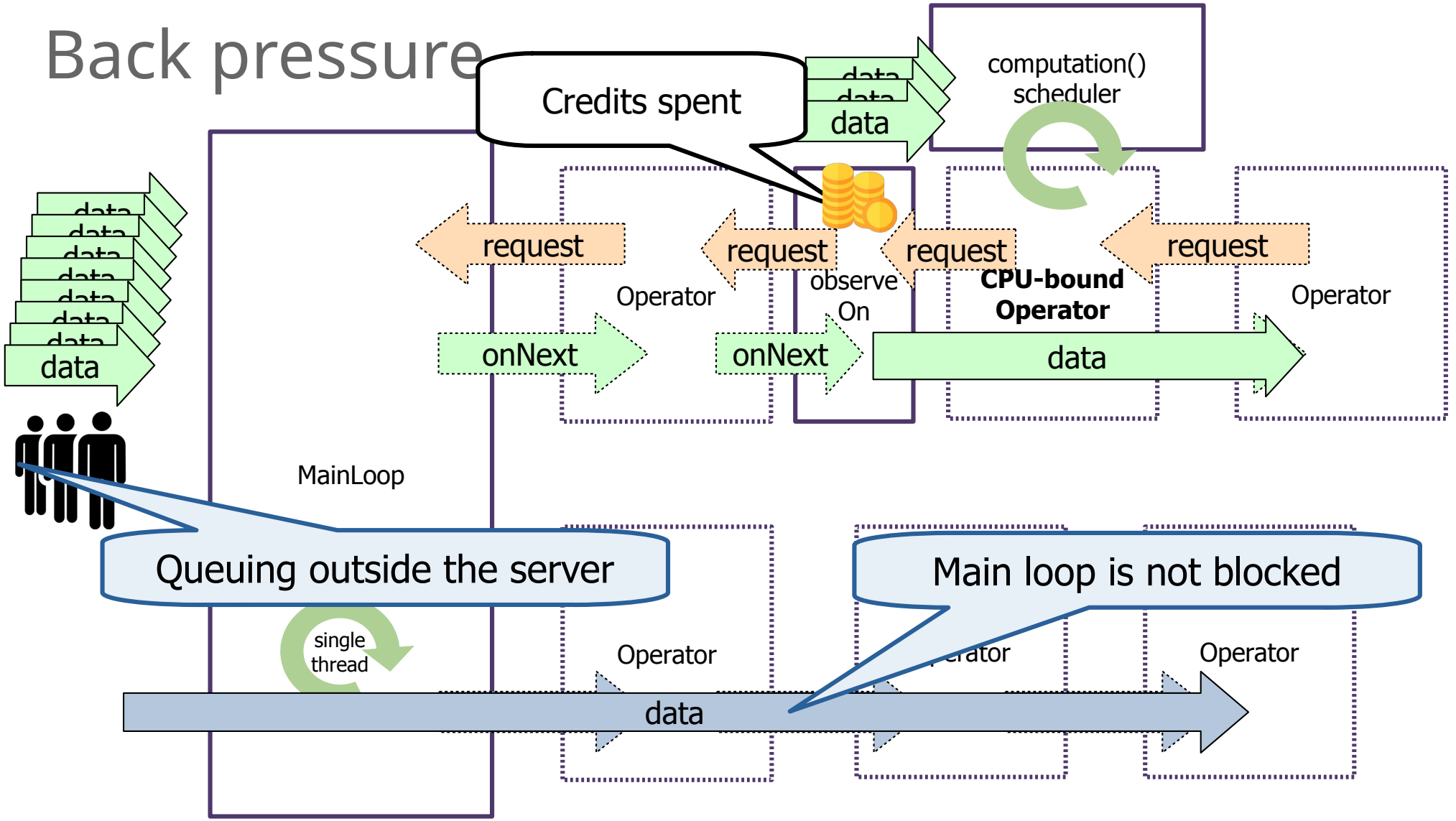
Back pressure



Back pressure



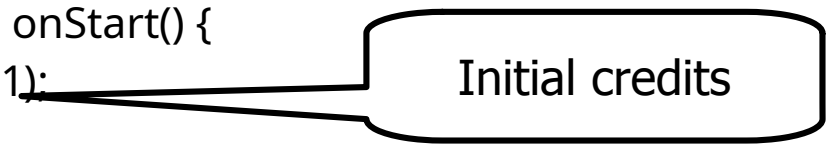
Back pressure



Subscribing flowables

- Sample subscription:

```
f.subscribe(new DefaultSubscriber<...>() {
      public void onStart() {
         request(1);
      }
      public void onNext(...) {
         ...
         request(1);
      }
      ...
   });
```



- Subscription with a simple callback provides infinite credits when subscribing
  - Useful when the bottleneck is in the middle of the pipeline

Subscribing flowables

```
public class Mainloop {
        public void write(Flowable<ByteBuffer> flow, SocketChannel s) {
                flow.subscribe(new DefaultSubscriber<ByteBuffer>() {
                        public void onStart() {
                           request(1);
                        }
                        public void onNext(ByteBuffer bb) {
                                s.write(bb);
                                if (bb.hasRemaining()) {
                                        key.interestOpsOr(OP_WRITE);
                                } else {
                                        request(1);
                                }
                        }
                });
        }
```

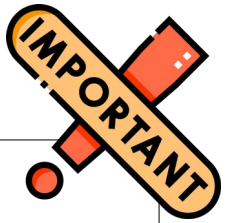
(*) Implement "Publisher" for compatibility with other frameworks.

Subscribing flowables

```
public class Mainloop {

        public run() {
                ...
                if (key.isWritable()) {
                        if (there is a pending write) {
                                s.write(bb);
                        } else {
                                flow.request(1);
                                key.interestOpsAnd(~OP_WRITE);
                        }
                }
        }
}
```

Complete client-server application

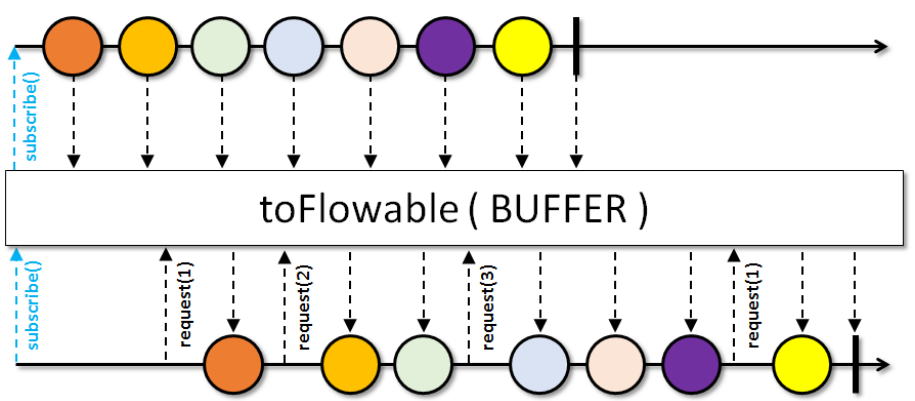


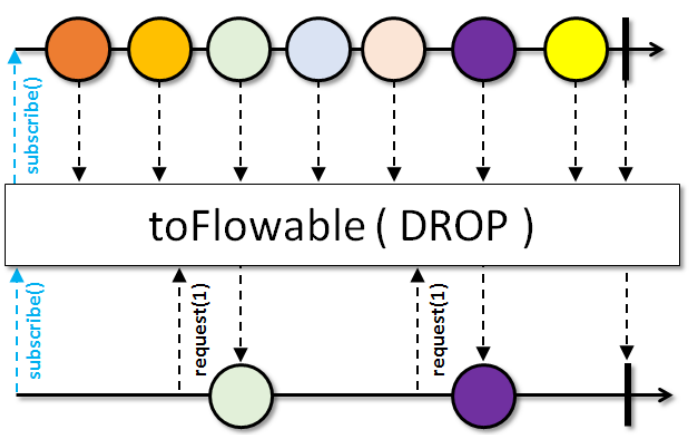

```
public class Server {
	public static void main(String[] args) throws Exception {
		var ss = ServerSocketChannel.open(new InetSocketAddress(12345));
		var loop = new MainLoop();
		var ss_obs = loop.accept(ss);
		ss_obs.subscribe(s -> {
			var s_flow = loop.read(conn)
				.observeOn(computation())
				.lift(new LineSplitOperator())
				.map(bb -> StandardCharsets.UTF_8.decode(bb))
				.map(String::toUpperCase)
				.map(s -> StandardCharsets.UTF_8.encode(s));
			loop.write(s_flow, conn);
		});
	}
}
```

Business logic

Combining flowable and observable

- Convert flowable to observable:
  - f.toObservable().op(...).subscribe();
  - subscribes with infinite credit
- Observable to flowable needs to deal with back pressure:
  - o.toFlowable(strategy).op(...).subscribe(...);
  - multiple strategies available, for example:





Implementing flowables

```
public class Mainloop {
        public Flowable<ByteBuffer> read(SocketChannel s) {
                return subscriber -> {
                        s.configureBlocking(false);
                        var key = s.register(sel, 0);
                        var sub = new ReadSubscription(key, subscriber)
                        key.attach(sub);
                        subscriber.onSubscribe(subscription);
                }
        }
        public run() {
                …
                if (key.isReadable()) {
                        var sub = (MySubscription) k.attachment();
                        …
                        subscription.subscriber.onNext(bb);
                        if (sub.credits == 0) key.interestOpsAnd(~OP_READ);
                …
        }
}
```

No OP_READ until we have credits

No OP_READ if we exhausted credits

Implementing flowables

```
public class Mainloop {
        private class ReadSubscription implements Subscription {
                private long credits = 0;

                public void request(long n) {
                        credits += n;
                        if (not currently reading) {
                                key.interestOpsOr(OP_READ);
                                key.selector().wakeup();
                        }
                }
        }
}
```

Restore OP_READ

Make the selector consider new interests

Custom operator

```
public class LineSplitOperator implements FlowableOperator<ByteBuffer,ByteBuffer> {
   public Subscriber<...> apply(Subscriber<...> child) throws Throwable {
      return new Subscriber<ByteBuffer>() {
         public void onSubscribe(Subscription parent) {
               this.parent = parent;
               child.onSubscribe(new Subscription() {
                     public void request(long l) {
                           ...
                     }
               });
          }
         public void onNext(ByteBuffer bb) {
               ...
               if (have downstream credits)
                     child.onNext(...);
               if (can receive more data)
                     parent.request(...);
         }
    }
}
```

Store reference to give credits

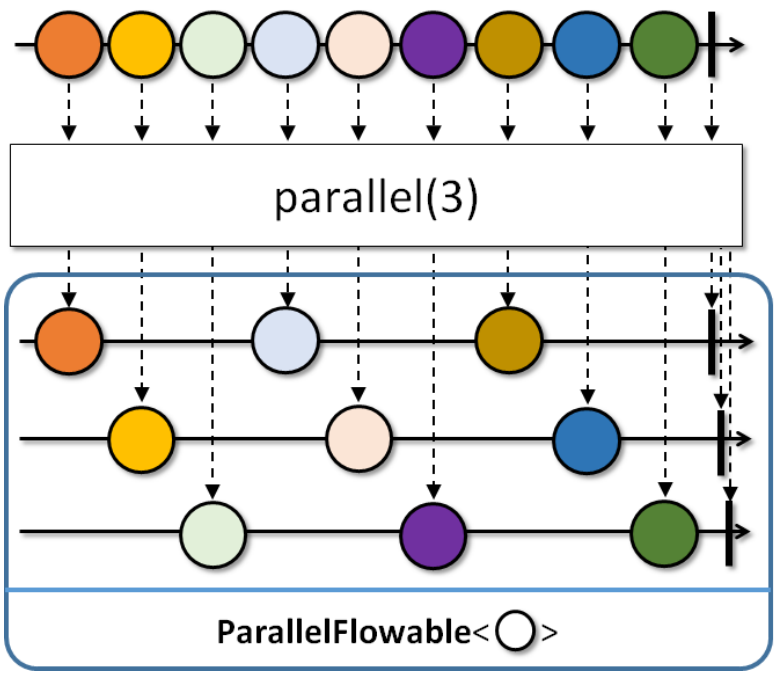
Receive credits from child

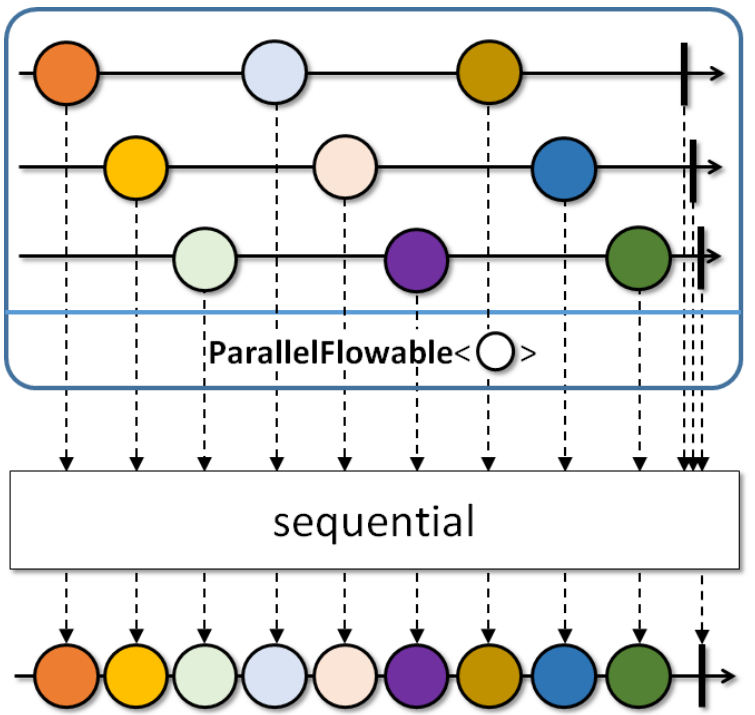
Consume credits from child

Give credits to parent

Parallelism

- In addition to parallel pipeline with observeOn(), back pressure allows parallel processing within one stage:





References

- ReactiveX / RxJava documentation: https://reactivex.io/
- Tomasz Nurkiewicz and Ben Christensen. *Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications.* O'Reilly, 2017.
  - Chap. 6
  - *Warning:* Uses an older version of the RxJava API.

- For Python: https://github.com/MichaelSchneeberger/rxbackpressure