Considerations for integrating virtual threads in a Java framework: a Quarkus example in a resource-constrained environment

A. Navarro[1,2] J. Ponge[1,2] F. Le Mouël[2] C. Escoffier[1]

[1]Red Hat [2]CITI Laboratory-INSA Lyon

2023

The end of the monolith

- difficult to maintain
- dependency hell
- reboot for every change
- sub-optimal deployment
- limit scalability
- technology lock-in

from

Microservices: yesterday, today, and tomorrow - Dragoni & al

Microservices

- *difficult to maintain*
- split in smaller entities
- *dependency hell*
- less dependencies per service
- *reboot for every change*
- reboot only impacted service
- *sub-optimal deployment*
- per-service deployment
- *limits scalability*
- scale each service
- *technology lock-in*
- per service technology

Remark

Transitioning to cloud becomes a thing

Make microservices cheap

resource-efficiency

Make microservices cheap

resource-efficiency

resource-efficiency

Make microservices cheap

resource-efficiency

resource-efficiency

resource-efficiency

Make microservices cheap

resource-efficiency

resource-efficiency

resource-efficiency

resource-efficiency

resource-efficiency

Make microservices cheap

resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency

resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency
resource-efficiency

Make microservices cheap

resource-efficiency resource-efficiency resource-efficiency
resource-efficiency resource-efficiency resource-efficiency
resource-efficiency resource-efficiency resource-efficiency
resource-efficiency resource-efficiency resource-efficiency
resource-efficiency resource-efficiency resource-efficiency
resource-efficiency resource-efficiency resource-efficiency
resource-efficiency resource-efficiency resource-efficiency
resource-efficiency resource-efficiency resource-efficiency
resource-efficiency resource-efficiency resource-efficiency
resource-efficiency resource-efficiency resource-efficiency
resource-efficiency resource-efficiency resource-efficiency
resource-efficiency resource-efficiency resource-efficiency
resource-efficiency resource-efficiency resource-efficiency
resource-efficiency resource-efficiency resource-efficien

Scaling and reactive programming

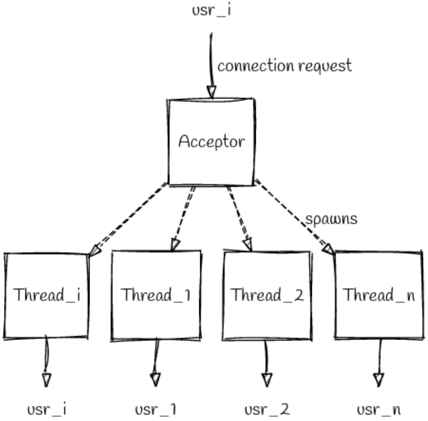
Problem

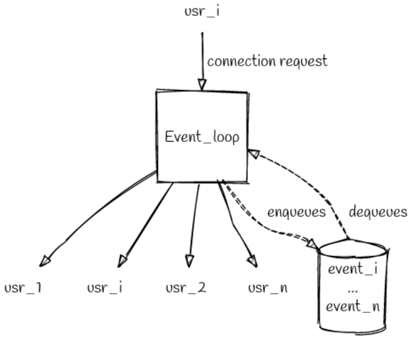
Communications over the network are unreliable and slow.

Goal

Achieving efficient and timely communication between services.

Blocking and Non-blocking programming





Non-blocking abstractions

- asynchronous callbacks
- promises and futures
- reactive streams
- coroutines
- async functions
- light threads

Code comparison - blocking

```
var names = getAll();
var quotes = getQuotes(names.size());
for(int i=0; i < names.size();i ++){
   names.get(i).surname+= ”- ”+quotes.get(i);
}
return names;
```

Code comparison - asynchronous callbacks

```
getAll( names => {
   getQuotes(names.size(), quotes => {
      for(int i=0; i < names.size();i ++){
         names.get(i).surname+= ”- ”+quotes.get(i);
      }
      //continuation
   })
});
```

Code comparison - reactive streams

```
var names = getAll().memoize().indefinitely();
var quotes = names.onItem().transformToUni(list ->
         getQuotes(list.size()));
return Uni.combine().all()
   .unis(names,quotes).asTuple()
   .onItem().transform(tuple -> {
      var nList=tuple.getItem1();
      //can await it since it is already resolved
      var qList = tuple.getItem2();
      for(int i=0; i < namesList.size();i ++){
         nList.get(i).surname += " - "+qList.get(i);
      }
      return namesList;
   });
```

Asynchronous programming abstractions

> *We should do (as wise programmers aware of our limitations) our utmost best to shorten the conceptual gap between the static program and the dynamic process, to* ***make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible****.*

Edgar. J. Dijkstra, 1968

Code comparison - virtual threads

```
var names = getAll();
var quotes = getQuotes(names.size());
for(int i=0; i < names.size();i ++){
   names.get(i).surname+= ”- ”+quotes.get(i);
}
return names;
```

Virtual threads

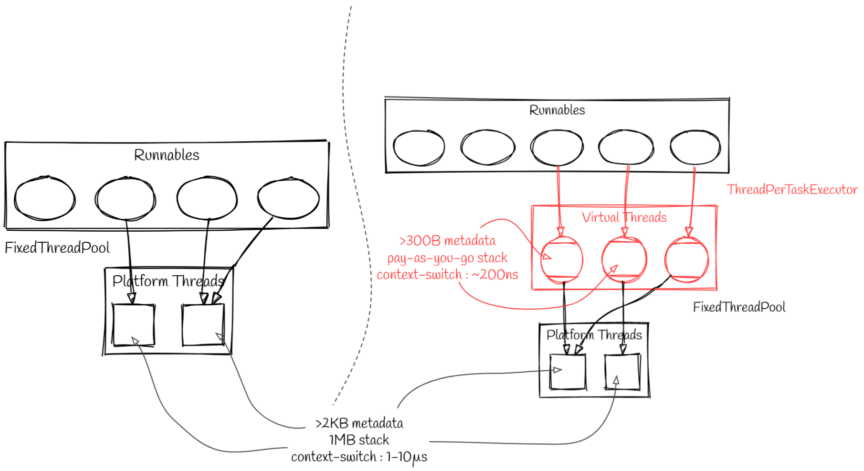


Table of Contents

A QUICK OVERVIEW OF QUARKUS

```
@GET
@Path("/reactive")
Uni<Fortune> reactive() {
   return repo
      .findAllAsync()
      .map(this::pickOne);
}
```

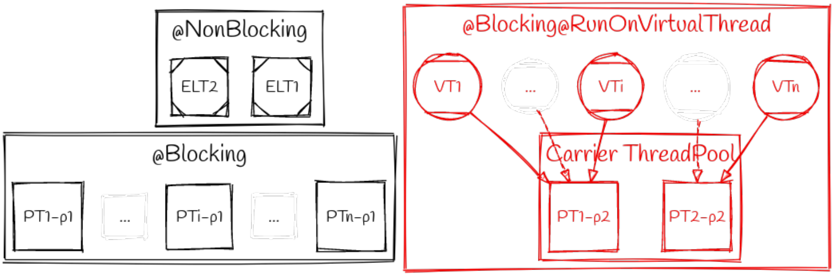
```
@GET
@Path("/virtual-threads")
@RunOnVirtualThread
Fortune loomWithJdbc() {
   var list = repo.findAll();
   return pickOne(list);
}
```

Three integration strategies

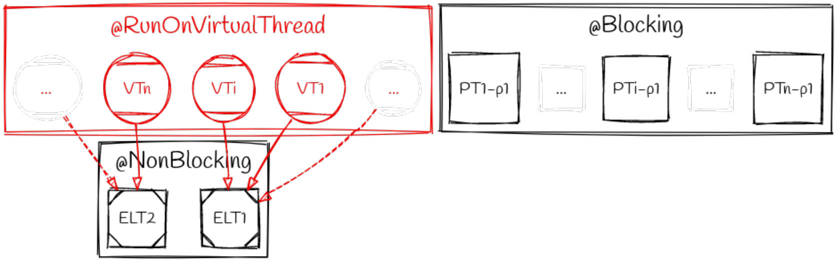
Strategy	Pros	Cons
Forking worker model	Simple, fits virtual threads model	Context switches
Using event-loop as carrier	No context-switch, Fewer threads overall	Potential deadlocks
Modifying Netty event-loops to be virtual threads	Integration done at the Netty level, Netty-based frameworks would benefit from it	Can't modify Netty up-stream, unpredictable effects

Table: Comparison of the different Quarkus-virtual-threads options

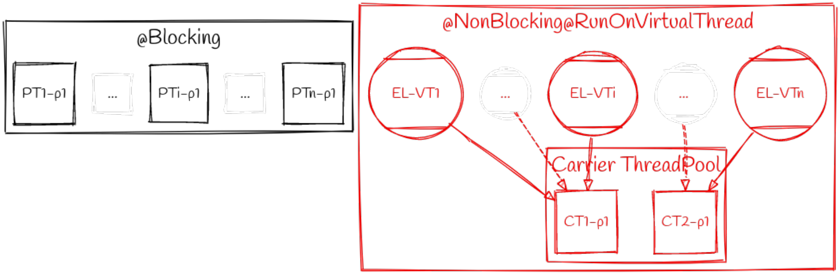
Forking the worker model



Reusing event-loops



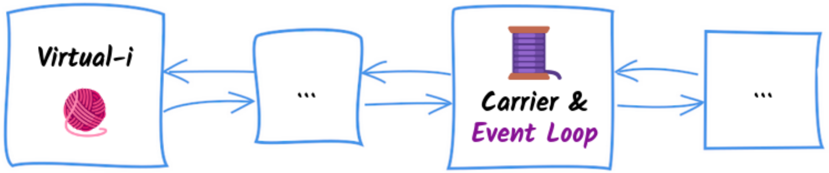
”Virtualizing” Netty event-loops



Deadlock situation

```java
@GET
@RunOnVirtualThread
@NonBlocking
@Path("/print-both")
public List<Fortune> getAll() {
   System.out.println("outer - " +Thread.currentThread());
   var list =
     db.getPool().preparedQuery(SELECT_ALL)
        .execute()
        .map(item -> {
            System.out.println("inner - " + Thread.currentThread());
            return createListOfFortunes(item);
        });
   return list.await().indefinitely();
}
```

Deadlock explanation



Deadlock: sharing locks

Conclusion

The event-loop can’t reuse locks *as a carrier*

Integration strategies summary

Strategy	Pros	Cons
Forking worker model	Simple, fits virtual threads model	Context switches
Using event-loop as carrier	No context-switch, Fewer threads overall	Potential deadlocks
Modifying Netty event-loops to be virtual threads	Integration done at the Netty level, Netty-based frameworks would benefit from it	Can’t modify Netty up-stream, unpredictable effects

Table: Comparison of the different Quarkus-virtual-threads options

Final Choice: Forking the worker model

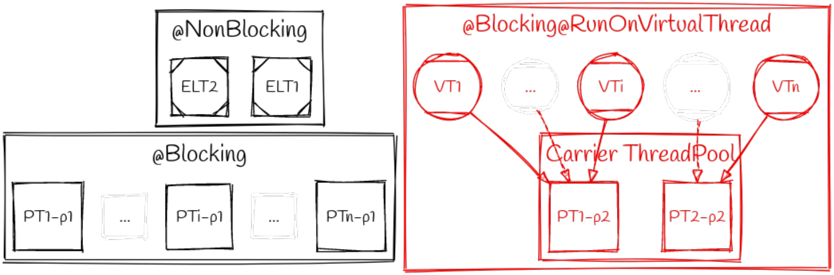


Table of Contents

The experiment

Goal

Measure how performance of the application is affected by replacing reactive endpoints with virtual-threads offloading

Hypothesis

Quarkus-virtual-threads should perform better than **Quarkus-blocking** but not as well as **Quarkus-reactive**

The environment

Limited resources

- 512MB memory
- 0.5 vCPU
- 256MB heap

Fault-inducing settings

- 200ms of delay between the DB and the server
- Hyperfoil to avoid Coordinated Omission

Elements of the experiment

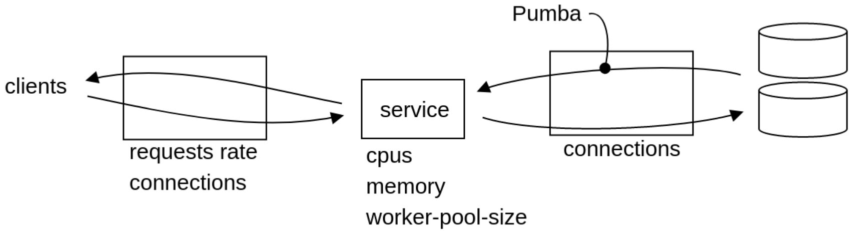
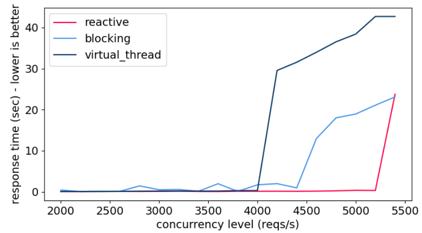
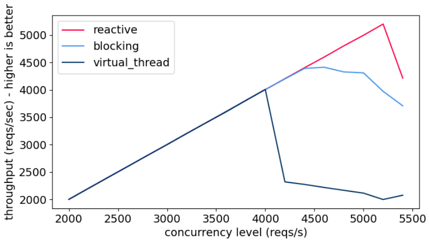
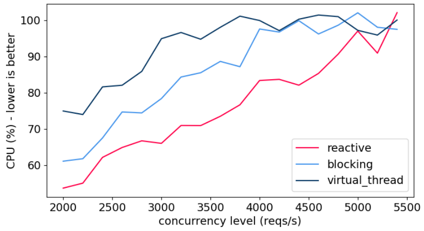


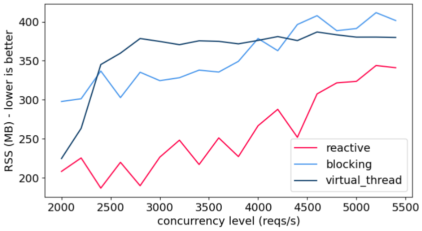
Table of Contents

Results without delay

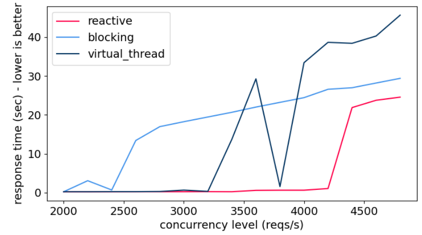


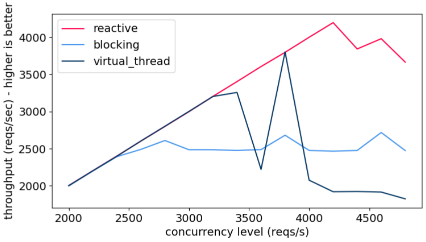


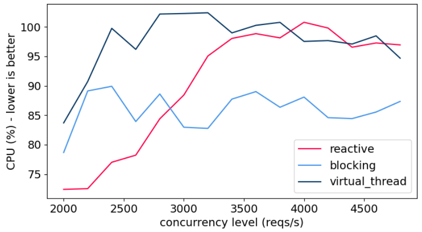


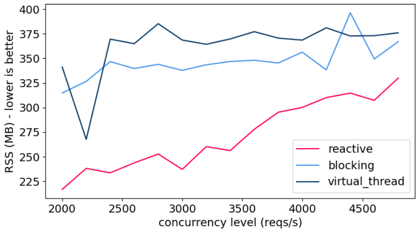


Results with 200ms delay

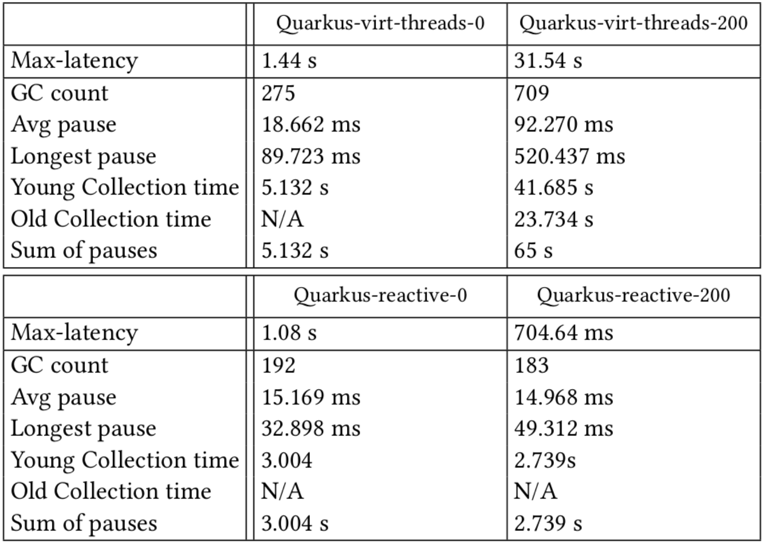








Garbage Collection detail



	Quarkus-virt-threads-0	Quarkus-virt-threads-200
Max-latency	1.44 s	31.54 s
GC count	275	709
Avg pause	18.662 ms	92.270 ms
Longest pause	89.723 ms	520.437 ms
Young Collection time	5.132 s	41.685 s
Old Collection time	N/A	23.734 s
Sum of pauses	5.132 s	65 s

	Quarkus-reactive-0	Quarkus-reactive-200
Max-latency	1.08 s	704.64 ms
GC count	192	183
Avg pause	15.169 ms	14.968 ms
Longest pause	32.898 ms	49.312 ms
Young Collection time	3.004	2.739s
Old Collection time	N/A	N/A
Sum of pauses	3.004 s	2.739 s

Explaining longer Garbage Collection

- Too many virtual threads exist in memory at the same time.
- Data structures generated by Quarkus/Netty pollute memory.

Explaining longer Garbage Collection

- Too many virtual threads exist in memory at the same time.
- Data structures generated by Quarkus/Netty pollute memory.

Conclusions

- Although *Quarkus-virtual-threads* outperforms *Quarkus-blocking* in the context of the experiment, it still does not scale to the concurrency of *Quarkus-reactive*.
- Memory consumption is especially high for *Quarkus-virtual-threads*.
- Virtual threads are not directly responsible for the high memory consumption.
- Data structures generated by *Quarkus*/*Netty* are the main memory consumers.

Generalizing the results

The integration of virtual threads in frameworks built on top of Netty or that rely heavily on ThreadLocals, presuming a small thread count, is likely to experience significant garbage collection (GC) pressure.