

CONSIDERATIONS FOR INTEGRATING VIRTUAL THREADS IN A JAVA FRAMEWORK: A QUARKUS EXAMPLE IN A RESOURCE-CONSTRAINED ENVI- RONMENT

A. NAVARRO^{1,2} J. PONGE^{1,2}
F. LE MOUËL² C. ESCOFFIER¹



¹RED HAT ²CITI LABORATORY-INSA LYON

17TH ACM INTERNATIONAL CONFERENCE ON DISTRIBUTED AND
EVENT-BASED SYSTEMS 2023

1 Context

2 Integration

3 The experiment

4 Results

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 cost efficient if done well.

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-efficiency

SCALING AND REACTIVE PROGRAMMING

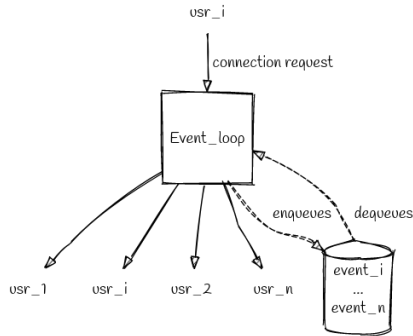
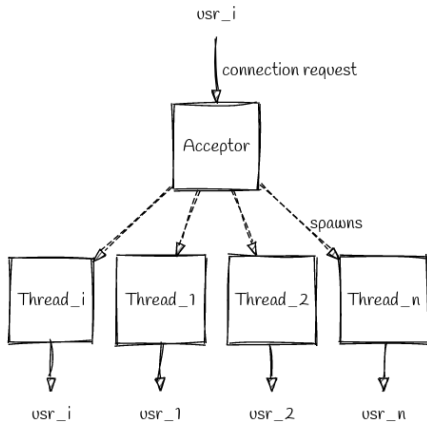
Problem

Communications over the network are unreliable and slow.

Goal

Achieving efficient and timely communication between services.

BLOCKING AND NON-BLOCKING PROGRAMMING



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;
    });
```

*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.***

Go To Statement considered harmful. - 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

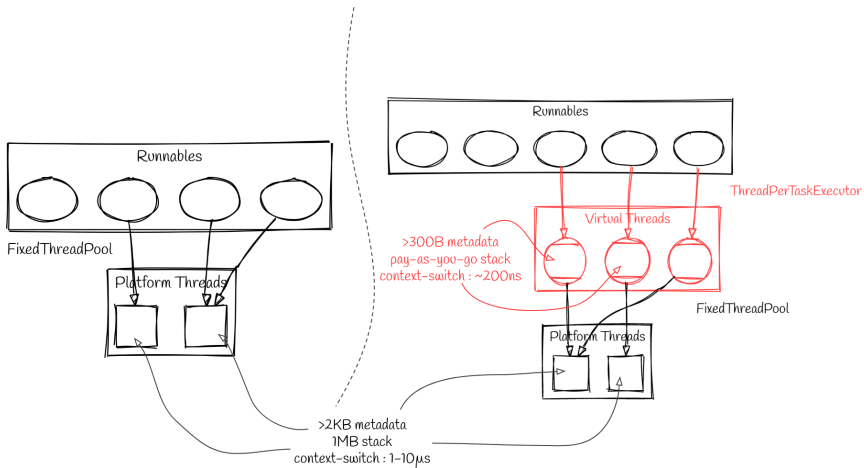


TABLE OF CONTENTS

1 Context

2 Integration

3 The experiment

4 Results

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);
}
```

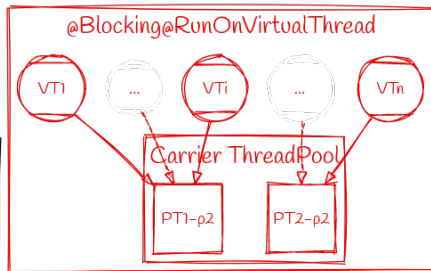
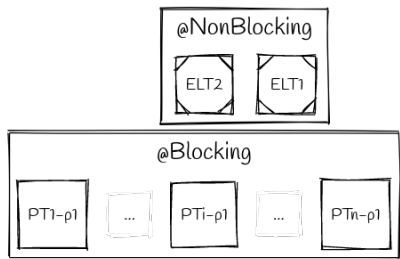
THREE INTEGRATION STRATEGIES

Forking the worker model.

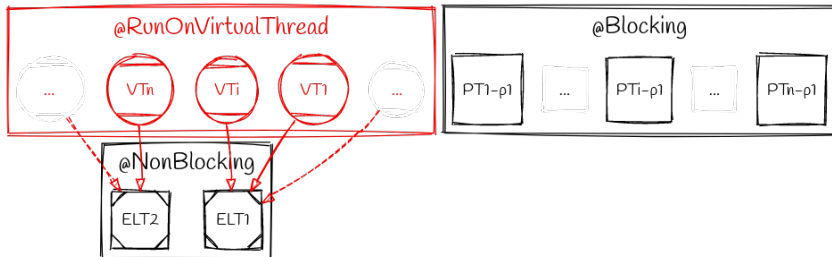
Using event-loops as carriers.

“Virtualizing” Netty event-loops.

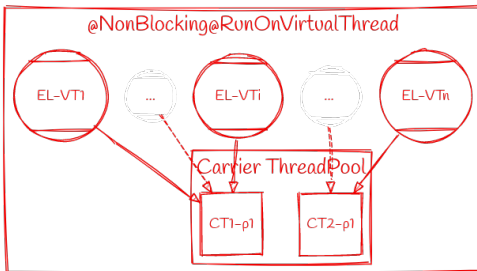
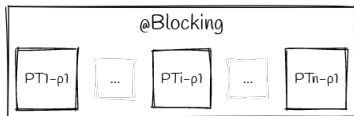
FORKING THE WORKER MODEL



REUSING EVENT-LOOPS



"VIRTUALIZING" NETTY EVENT-LOOPS



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 upstream, unpredictable effects

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

FINAL CHOICE: FORKING THE WORKER MODEL

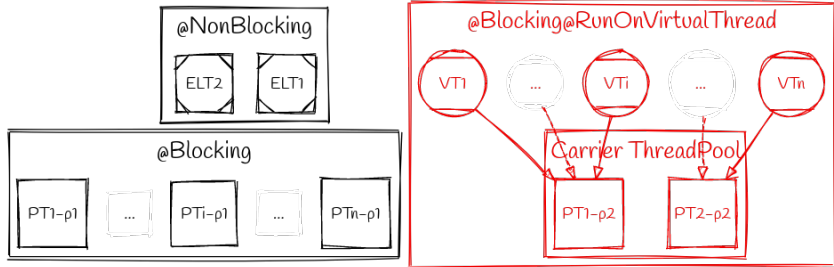


TABLE OF CONTENTS

1 Context

2 Integration

3 The experiment

4 Results

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**

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

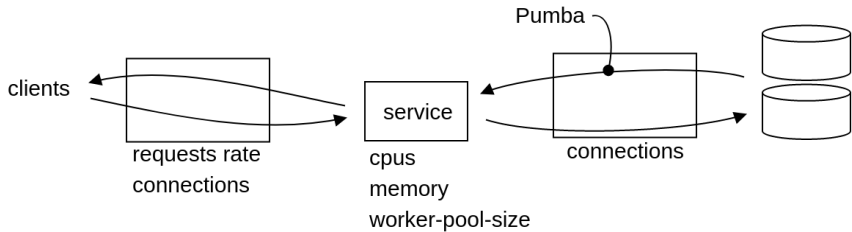


TABLE OF CONTENTS

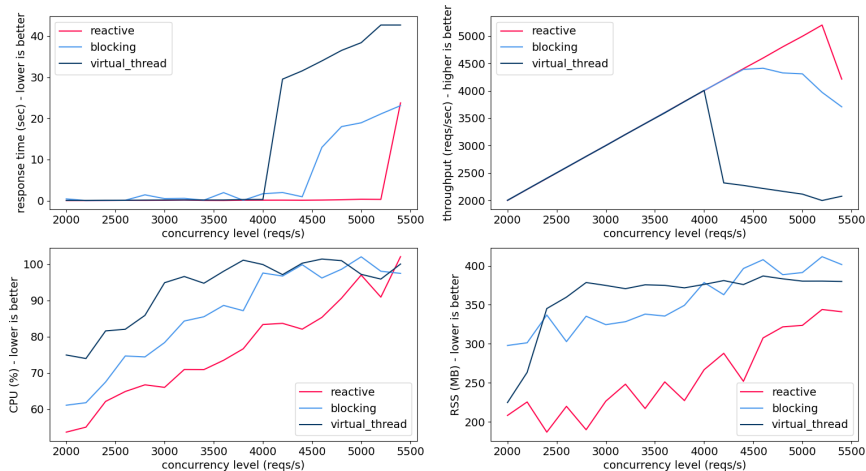
1 Context

2 Integration

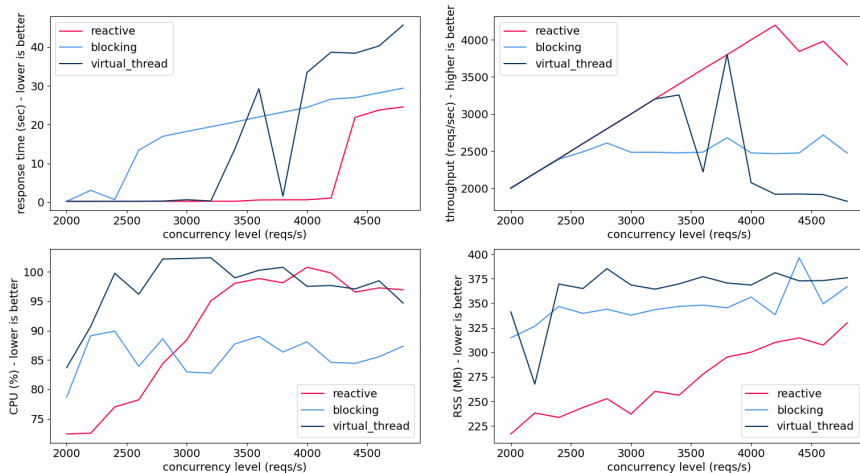
3 The experiment

4 Results

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

- *Quarkus-blocking* < *Quarkus-virtual-threads* < *Quarkus-reactive*.
- High Memory consumption.
- Virtual threads are not responsible for the high memory consumption.
- Data structures generated by *Quarkus/Netty* are the main memory consumers.

GENERALIZING THE RESULTS

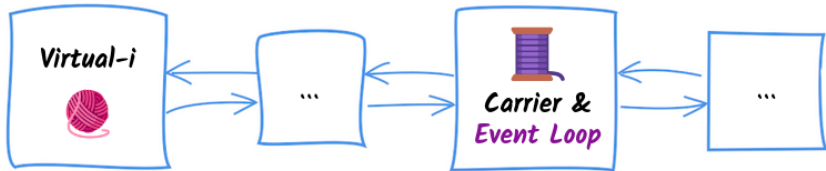
The integration of virtual threads in frameworks built on top of Netty or that rely heavily on ThreadLocals, presuming a small thread count, is likely to experience significant garbage collection (GC) pressure.

SUMMARY

DEADLOCK SITUATION

```
@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



Conclusion

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