Technische Universität Braunschweig

Institute of Operating Systems and Computer Networks

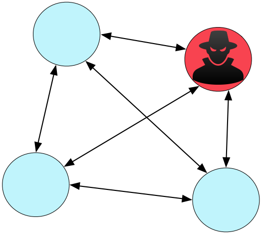
Themis: An Efficient and Memory-Safe BFT Framework in Rust

SERIAL Workshop, December 9, 2019

Signe Rüsch, Kai Bleeke, Rüdiger Kapitza
ruesch@ibr.cs.tu-bs.de
Technische Universität Braunschweig, Germany

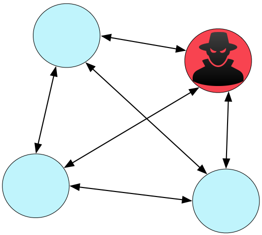
Byzantine Fault Tolerance

- **Consensus** even with participants showing **arbitrarily wrong** behaviour
- E.g. used in permissioned blockchains
- Tolerate f Byzantine faults with $3f+1$ nodes



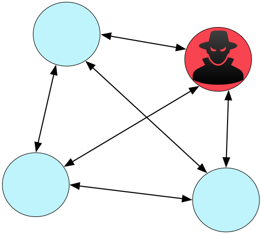
Byzantine Fault Tolerance

- **Consensus** even with participants showing **arbitrarily wrong** behaviour
- E.g. used in permissioned blockchains
- Tolerate f Byzantine faults with $3f + 1$ nodes
- BFT protocols have high **message complexity**
- Frameworks are highly optimised regarding processing time per message
  - Both on protocol and network layer



Byzantine Fault Tolerance

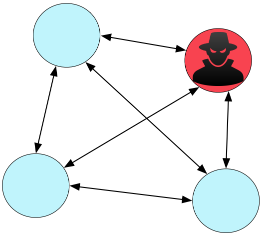
- **Consensus** even with participants showing **arbitrarily wrong** behaviour
- E.g. used in permissioned blockchains
- Tolerate f Byzantine faults with $3f + 1$ nodes
- BFT protocols have high **message complexity**
- Frameworks are highly optimised regarding processing time per message
  - Both on protocol and network layer



➥ BFT frameworks should be **fast**, **efficient**, and **resilient**!

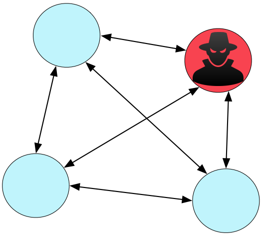
Programming Languages – C

- So far, frameworks mostly written in **C** or **Java**
  - C: PBFT [Castro et al., OSDI'99]
  - Java: Reptor [Behl et al., Middleware'15]



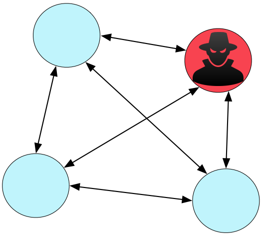
Programming Languages – C

- So far, frameworks mostly written in **C** or **Java**
  - C: PBFT [Castro et al., OSDI'99]
  - Java: Reptor [Behl et al., Middleware'15]
- Low-level programming languages like **C** offer **high performance**
  - Direct access to memory
  - Translation into native instructions



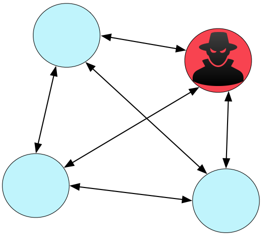
Programming Languages – C

- So far, frameworks mostly written in **C** or **Java**
  - C: PBFT [Castro et al., OSDI'99]
  - Java: Reptor [Behl et al., Middleware'15]
- Low-level programming languages like **C** offer **high performance**
  - Direct access to memory
  - Translation into native instructions
- But **error-prone** due to memory leaks and undefined behaviour, e.g.:
  - Reading uninitialized memory
  - Dereferencing a `NULL` pointer, an invalid pointer
  - Out-of-bounds array access



Programming Languages – C

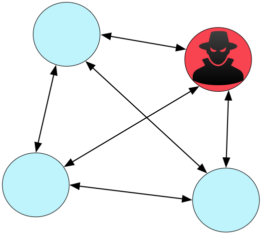
- So far, frameworks mostly written in **C** or **Java**
  - C: PBFT [Castro et al., OSDI'99]
  - Java: Reptor [Behl et al., Middleware'15]
- Low-level programming languages like **C** offer **high performance**
  - Direct access to memory
  - Translation into native instructions
- But **error-prone** due to memory leaks and undefined behaviour, e.g.:
  - Reading uninitialized memory
  - Dereferencing a `NULL` pointer, an invalid pointer
  - Out-of-bounds array access



➥ Eliminate unsafe, unreliable code!

Programming Languages – Java

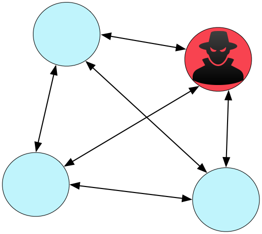
- Strong type system offers **safety**
- Runtime offers platform independence
- No manual **memory management:** Garbage Collector (GC)



Programming Languages – Java

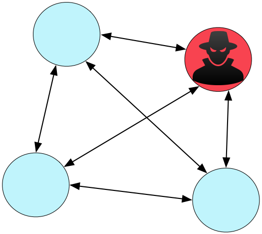
- Strong type system offers **safety**
- Runtime offers platform independence
- No manual **memory management:** Garbage Collector (GC)

- Interpreting bytecode less performant
- JIT and GC add **uncertainty** to performance
- Not resource-efficient: JVM's high memory consumption



Programming Languages – Java

- Strong type system offers **safety**
- Runtime offers platform independence
- No manual **memory management**: Garbage Collector (GC)



- Interpreting bytecode less performant
- JIT and GC add **uncertainty** to performance
- Not resource-efficient: JVM's high memory consumption

↳ **Tradeoff**: performance vs. safety!
How can we combine **both**?

The Rust Programming Language

- **Combines** performance and safety
- Young language: 1.0 release in 2015
- Initiated by Mozilla
- Completely open source

- **Performance:** no runtime or garbage collector
- **Reliability:** strong type system
- **Safety:** memory safety enforced at compile time

Ownership: Safe Memory

- Every value has an **owner**
- Values are **dropped** when owner goes out of scope
- Values are moved to a new owner

```
// heap allocate
let x = Box::new(1000);
// move into y,
// x no longer accessible
let y = x;
println!("{}", x);
//error[E0382]:
// use of moved value: `x`
```

Borrowing and Lifetimes: Safe References

- **Borrow** value to get shared and mutable references
- Either single mutable reference or multiple shared references
- References have **lifetimes**
  - No reference to invalid memory
- Enforced at compile time by the borrow checker

```
let mut x = 1000;
//mutable reference
let c = &mut x;
let d = &x;
//error[E0502]: cannot borrow `x`
// as immutable because it is
// also borrowed as mutable
```

Borrowing and Lifetimes: Safe References

- **Borrow** value to get shared and mutable references
- Either single mutable reference or multiple shared references
- References have **lifetimes**
  - No reference to invalid memory
- Enforced at compile time by the borrow checker

```rust
let mut x = 1000;
//mutable reference
let c = &mut x;
let d = &x;
//error[E0502]: cannot borrow `x`
// as immutable because it is
// also borrowed as mutable
```

➥ Rust eliminates a whole **class of errors** that potentially lead to Byzantine behaviour!

Themis Framework

Requirements for efficient BFT frameworks:

- Concurrency
  - Multiple small requests
  - Asynchronous execution
- Event-driven, non-blocking I/O
  - Often realized with Java NIO

➥ Rust: `Async/Await`, `Futures`, `Tokio` libraries

➥ Recently stabilized language features!

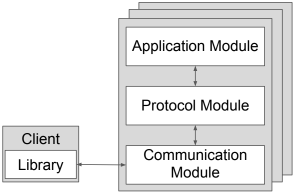
Themis Framework

Requirements for efficient BFT frameworks:

- Concurrency
  - Multiple small requests
  - Asynchronous execution
- Event-driven, non-blocking I/O
  - Often realized with Java NIO

➥ Rust: `Async/Await`, `Futures`, `Tokio` libraries

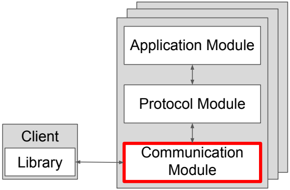
➥ Recently stabilized language features!



- Themis has three modules:
  - Communication
  - Protocol
  - Application

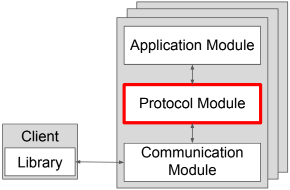
Communication Module

- Handles connection management
- Spawn tasks:
  - **Listener** for new connections
  - **Sender and receiver** for each connection
- Communication between tasks with **asynchronous** channels
- Messages are verified and batched before entering protocol stage



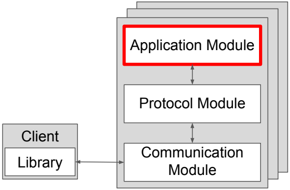
Protocol Module

- Protocol implementation as interface (**trait**)
- **Easy implementation** of new protocols
- Handles incoming and outgoing messages
- Currently includes:
  - PBFT: ordering, checkpointing, view change
  - Hybster [Behl et al., EuroSys'17]: hybrid protocol with trusted subsystem based on Intel SGX



Application Module

- Application implementation as interface
- **Asynchronous** for higher flexibility:
  - `execute()` method takes request
  - Returns a `Future` of a response
- Creates **snapshots** for checkpointing and failure recovery
- Does not have to be implemented in Rust



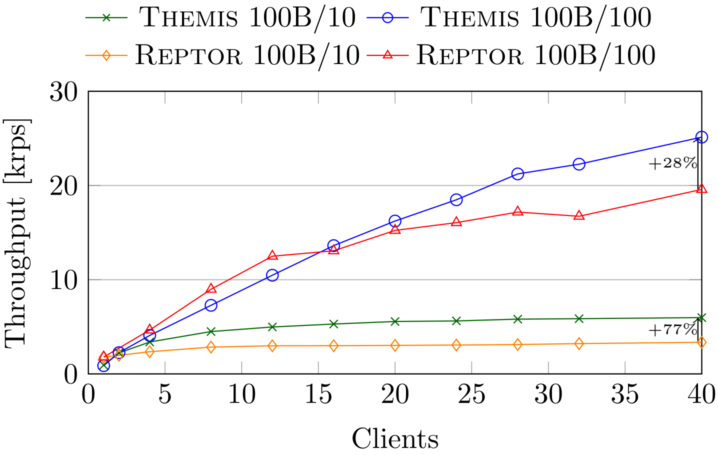
Evaluation

- Themis implementation with PBFT
  - 8.6 kLoC

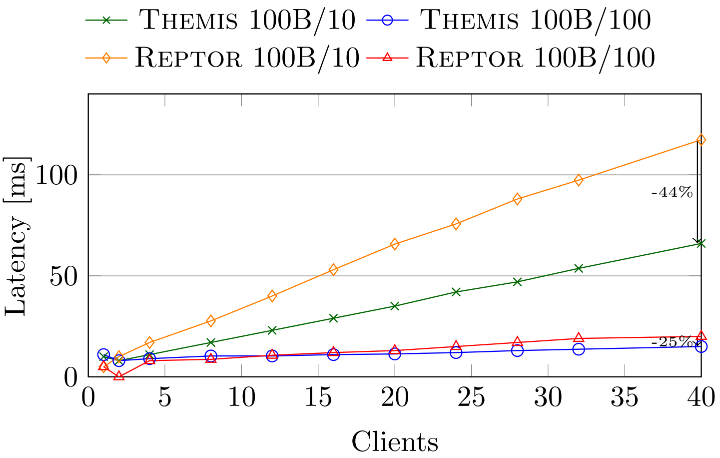
- Compare to Reptor framework: Java-based PBFT
- **Single-threaded** execution
- **RSA** for message authentication
- Checkpoint creation at every 1000 requests
- Four replicas and one client machine
  - Intel Core i7-6700 @ 3.40GHz, 24GB RAM
  - Intel Xeon E5645 @ 2.40GHz, 24GB RAM

- **Research Questions:**
  - How does Rust's **throughput** and **latency** compare to Java?
  - How is the **memory consumption** of the frameworks?

Evaluation: Throughput



Evaluation: Latency



Evaluation: Memory Consumption

	100B / 10	100B / 100
THEMIS	12.5 MB	44 MB
Reptor	1.8 GB	2.8 GB

- Reptor: 64–144× higher memory consumption
- Complete memory per process measured at end of benchmark runs
- Lower memory consumption due to lack of runtime

Roadmap

Improvements since submission:

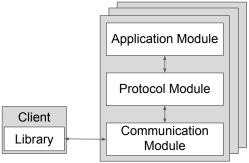
- Bug fixes in evaluation
- Message authentication using elliptic curve cryptography, e.g. **ECDSA**
  - 93 % higher throughput, 53 % lower latency than RSA
- WIP implementation of **Hybster**

Future Work:

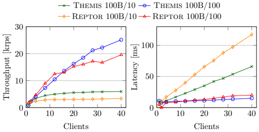
- BFT for **embedded** settings with restricted memory capacity
- Consensus in **embedded blockchains**, e.g. in railway systems

Conclusion

- Rust has high performance **and** memory safety
- New features allow implementation of **safe high-performance BFT frameworks**

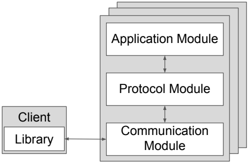


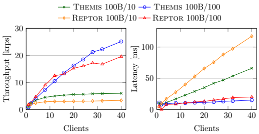
- Themis presents a **first prototype** of PBFT
- Evaluation shows promising results
- Investigation of usage of BFT for blockchains in **embedded** settings



Conclusion

- Rust has high performance **and** memory safety
- New features allow implementation of **safe high-performance BFT frameworks**
- THEMIS presents a **first prototype** of PBFT
- Evaluation shows promising results
- Investigation of usage of BFT for blockchains in **embedded** settings

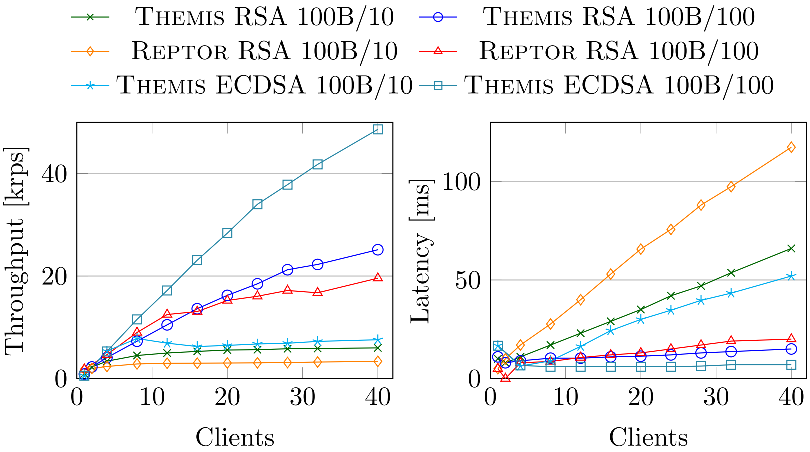




Thank you for your attention! Questions?

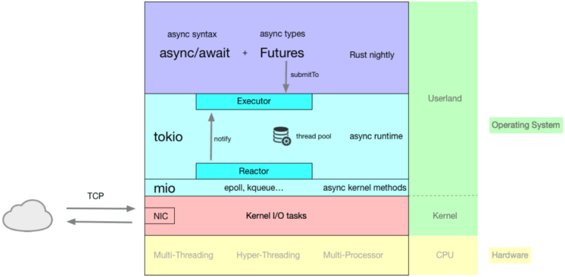
ruesch@ibr.cs.tu-bs.de

Evaluation: ECDSA



Async/Await in Rust

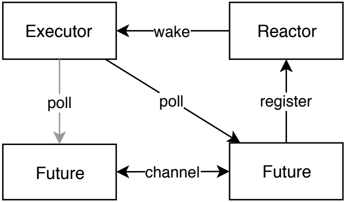
- Event-based architecture
- Reactor: notifies about incoming event
- Executor: takes data and executes async function (`Future`)



https://dev.to/gruberb/explained-how-does-async-work-in-rust-46f8

Executing Futures

- Spawned as tasks on an **Executor**
- Executor polls tasks when **Waker** is called
- I/O objects (sockets) register with **Reactor**
- Reactor waits for socket readiness
- Reactor wakes task when socket is ready



Futures

```
trait Future {
type Output;
 fn poll(&mut self, waker: &Waker) -> Poll<Self::Output>;
}
enum Poll<T> {
  Ready(T),
  Pending,
}

trait Future {
  type Output;
  fn poll(self: Pin<&mut Self>, waker: &Waker) -> Poll<Self::Output>;
}
```

- Future are lazy and have to be polled
- Future resolves to type `Output`, provided by implementer