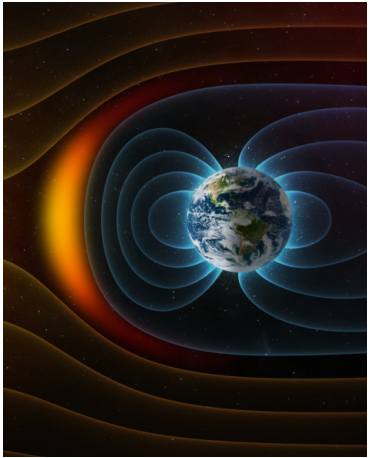
OptSCORE

Self-Optimized Communication of Replicated Services

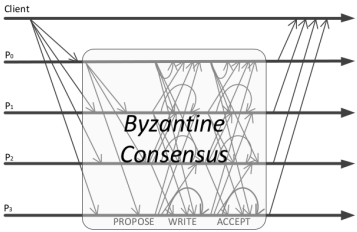
Johannes Köstler

March 2, 2017

1. Motivation
2. OptSCORE
3. Architecture
4. Optimizations
5. Weighted Voting



- Despite all the progress in system design and engineering, critical services/systems are still exposed to faults that are hard to detect and prevent
  - Amazon S3 outage caused by corrupted messages (bit flip) in the administration infrastructure (2008)
  - Candidate gets extra 4096 votes in a Belgium election because of cosmic radiation (2003)
- Additionally those critical services/service are exposed to attacks and intrusions

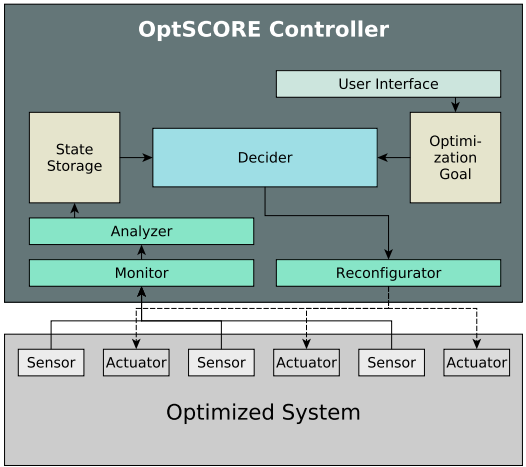


- State Machine Replication (SMR) / Active Replication can tolerate such faults to a certain degree
- SMR Systems are modeled as finite state automata and need to agree on a order of the client requests
- But:
  - Distributed Consensus causes a huge communication overhead
  - Configuration of SMR protocols is complex

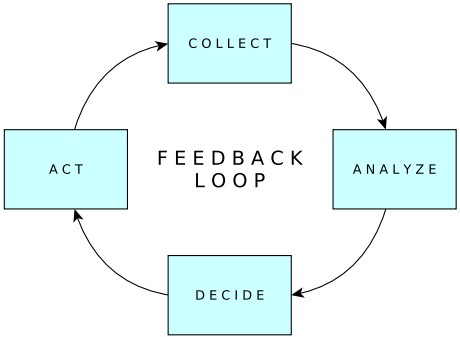
OptSCORE

OptSCORE: Self-Optimized and Self-Configured Communication and Scheduling of Replicated Services

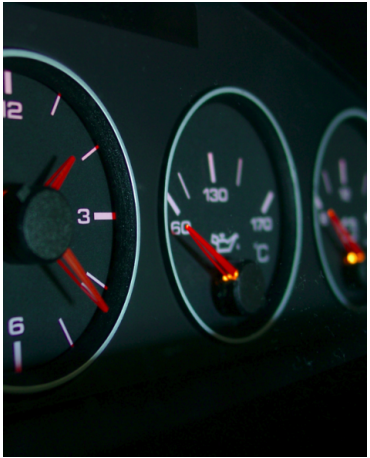
- Joint research project (DFG) between University of Ulm and University of Passau
- Provide a middleware solution that...
  - Efficiently offers fault and intrusion tolerance by withstanding crash and Byzantine faults using active state machine replication
  - Autonomously monitors itself as well the execution environment
  - Dynamically adapts its configuration to approximate an optimal execution (high throughput, low latency, reasonable costs)



- OptSCORE Controller
  - Each instance needs to make a deterministic decision
  - Data needs to be distributed and synchronized
- Strategy
  - Consist of
    - a set of sensors
    - a set of actuators
    - optimization goals



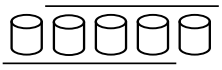
- OptSCORE Controller
  - Implemented as feedback loop
  - Livecycle
    - COLLECT - Monitors gathers sensor data
    - ANALYZE - Analyzers reduces data and detects trends
    - DECIDE - Decider checks if configuration needs to be adapted
    - ACT - Reconfigurator executes decided actions



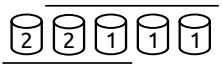
Optimizations

- System Reconfiguration
  - Dynamic adaption of timeout, buffer and cache sizes
- Selection of Algorithms
  - Substitution of different consensus algorithms at runtime
- Horizontal Scaling
  - Adding/Removing additional replicas at runtime
- Vertical Scaling
  - Adding/Removing computational resources at runtime
- Protocol Variants

$n = 4,\ f = 1,$
quorum size $= \lceil \frac{n+f+1}{2} \rceil = 3$



$n = 5,\ f = 1,$
quorum size $= \lceil \frac{n+f+1}{2} \rceil = 4$

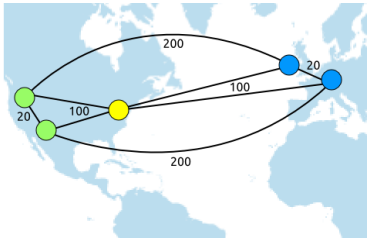


$n = 5,\ f = 1,$
quorum size $= 2f + 1 = 3$

WHEAT: Weight-Enabled Active Replication (Sousa et al., 2015)

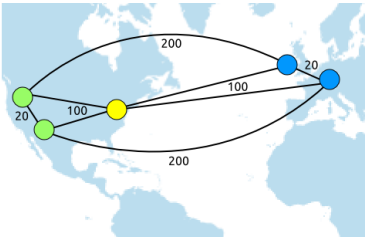
- Add additional replicas to the system
- Assign weights to all replicas
- Build quorums based on weighted votes

- No advantage?
  - Still the same quorum size
  - More replicas produce higher costs
  - But: More variety in the quorum formation!



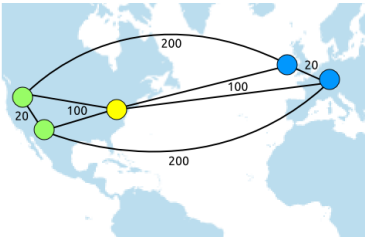
Dynamic Weights in Planetary-Scale SMR Systems

- General idea
  - Add additional replica for additional variance
  - Monitor the request latencies
  - Assign higher weights to better connected replicas
  - Assign leader role to replica that is best connected to the majority of clients
- Advantages
  - System will find a latency-optimal weight assignment
  - System can adapt to varying environmental conditions
  - System respects the client connectivity



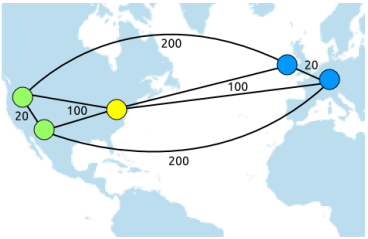
Dynamic Weights in Planetary-Scale SMR Systems

- Implementation (in our BFT-SMaRt prototype)
  - Measure the server-server and client-server latencies on each replica
  - Distribute the local measurements to the other replicas
  - Periodically recalculate the expected request latencies for all possible weight and leader role assignments
  - Reconfigure the system if the reconfiguration cost can be compensated by the expected latency reduction



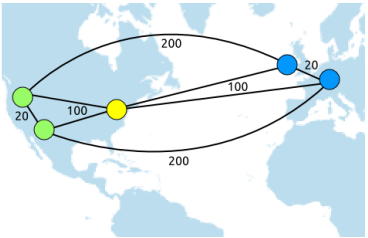
Dynamic Weights in Planetary-Scale SMR Systems

- Challenges
  - Data measurement and synchronization
    - What to do with data that is not send because processes fail or are too slow?
    - How to reach a deterministic decision?
  - Attacks
    - How to deal with tampered data from malicious node?
    - How to prevent that the leader role is constantly assigned to a fast but malicious replica?



Tampered Data (BFT)

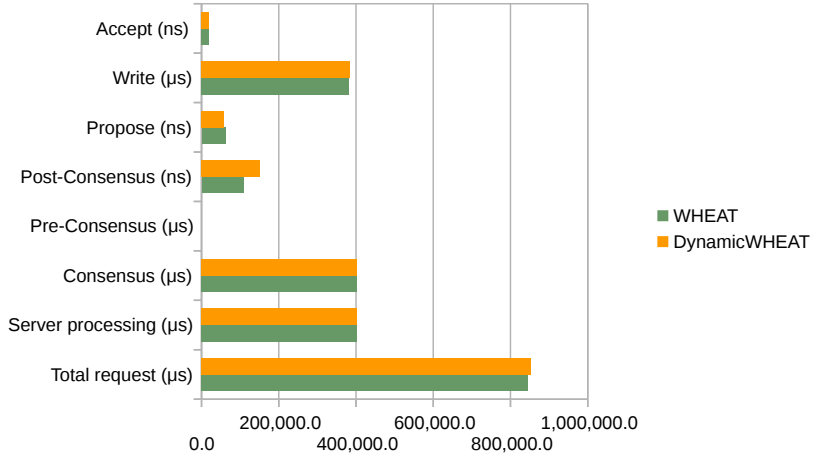
- Request latencies are measured in both directions
- Utilize request symmetry in a pessimistic way
  - Use the larger delay for calculations
  - Replica cannot make itself faster



Malicious leader (BFT)

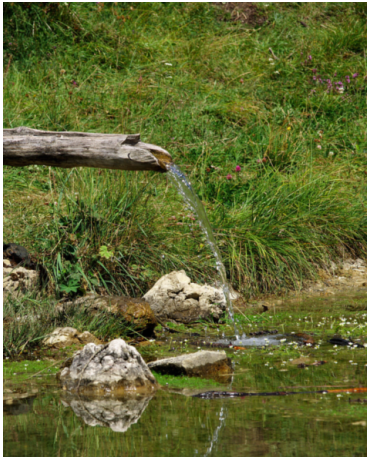
- Fast, but malicious replica gets the leader
- Leader does not propose any requests
  - Reelection, reconfiguration, reelection, ...
- Blacklist prevents reconfiguration decisions
- Each reelection doubles the number of blocked reconfiguration rounds

Performance Evaluation (average sample of 1000 client requests after some warm up)



Performance Evaluation (average sample of 1000 client requests after some warm up)

	WHEAT	DynamicWHEAT	Overhead
Total request	844,796 μs	851,999 μs	0.85%
Server processing	401,660.1 μs	403,643.2 μs	0.49%
Consensus	401,598.1 μs	403,639.7 μs	0.51%
Pre-Consensus	0.0 μs	0.0 μs	0.00%
Post-Consensus	108.7 μs	151.5 μs	39.37%
Propose	62.0 μs	59.0 μs	-4.84%
Write	381,411.8 μs	383,256.2 μs	0.48%
Accept	20,123.0 μs	20,223.3 μs	0.50%



- Optimization causes just a reasonable overhead
- The solution offers some performance potential
- But: This potential can only be activated in dynamic environments