Work in progress

A Byzantine Fault-Tolerant Key-Value Store for Safety-Critical Distributed Real-Time Systems

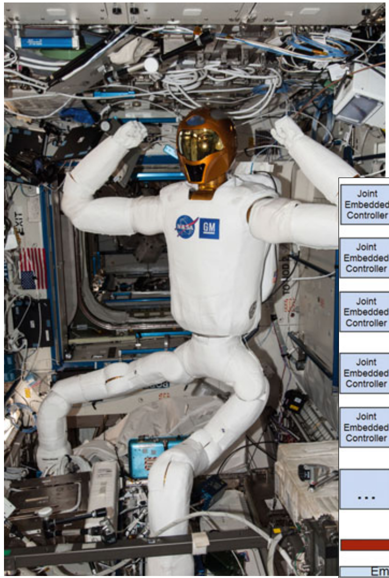
December 5, 2017
CERTS 2017

Malte Appel, Arpan Gujarati and Björn B. Brandenburg

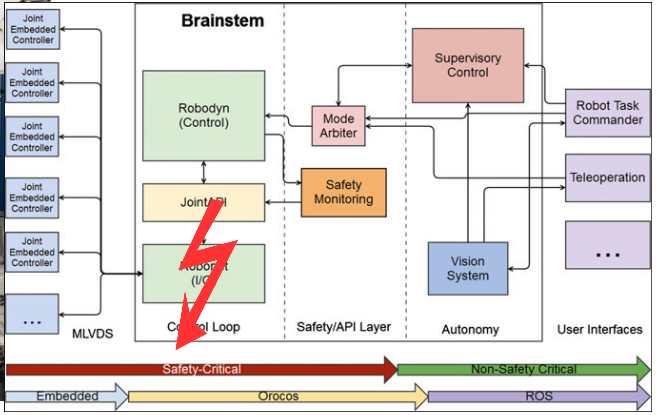
MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

UNIVERSITÄT
DES
SAARLANDES

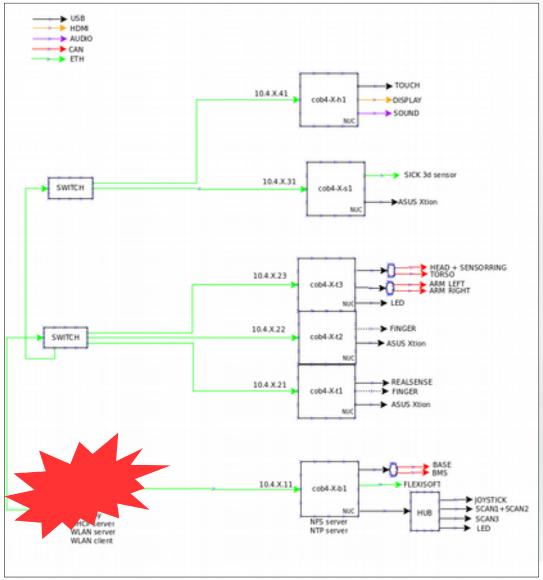
Distributed Real-Time Systems

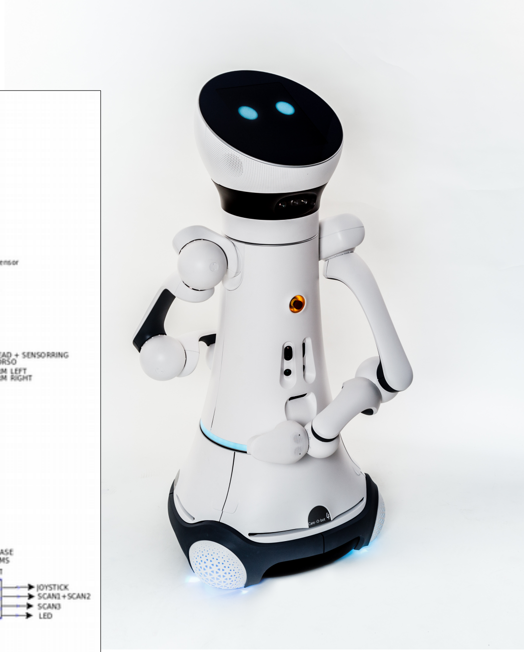


Robonaut 2
[J. Badger et al., 2016]

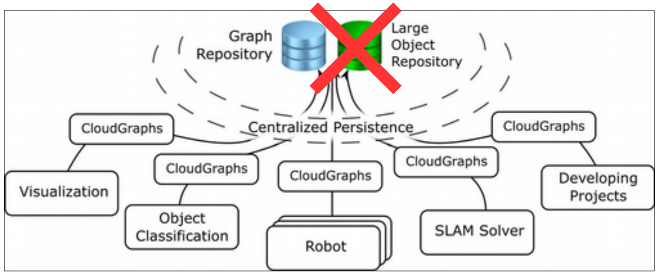


Care-O-bot 4
[Fraunhofer IPA]





Concept using SLAMinDB
[D. Fourie et al., 2017]



Susceptible to faults

- Electromagnetic interference
- Thermal effects
- …

possible consequences

- Bit-flips
- Crashes
- Madness

R. Gaillard, "Single event effects: Mechanisms and classification," in Soft Errors in Modern Electronic Systems, 2011
K. Driscoll et al., "Byzantine fault tolerance, from theory to reality," in SafeComp, 2003

Common Mitigation Techniques

Fail-safe

Assumptions for high-frequency systems

System Repair
- Requires accessible system

Checkpointing
- Tolerates crash (and restart) faults
- But not permanent hardware faults

Fail-operational

- Low latency
- No downtime
- Fail-operational

Passive Replication
- Easy to implement
- Requires additional hardware for replication (typically ≥ 2 replicas)

Active Replication
- Complex replica coordination consumes more bandwidth
- Typically ≥ 3 replicas

Long downtime — **Time for recovery** — **Short downtime**

Problem with Active Replication

- To tolerate Byzantine faults, replica coordination is required
  - Possibly very complex
  - Difficult to analyze
- We want to analyze **worst-case temporal behavior**
  - Aids certification process

Byzantine Fault
A fault presenting different values to different observers.

Prior Work – BFT

- Plenty of Byzantine fault-tolerant protocols exist
  - Chain-based
  - Broadcast-based
  - Probabilistic
  - …
- No strict timing guarantees
- Often significant differences in performance (faulty vs. fault-free)

What about fault tolerance for distributed real-time systems?

Prior Work – FT Distributed RTS

- Protocols for specific components exist…
  - Byzantine fault-tolerant clock synchronization [M. Malekpour, 2006]
  - Omission fault-tolerant CAN bus
  - [J. Rufino et al., 1998]
- … but also general architectures

Fault-tolerant real-time event service for CORBA

[H.-M. Huang and C. Gill, 2006]

- Middleware
- Multiple quality of service levels
- Fault model: **Fail-stop**

System-level Architecture for Failure Evasion in Real-time applications

[K. Junsung et al., 2012]

- Mixed criticality tasks
- Case study: “Boss” autonomous vehicle
- Fault model: **Fail-stop**

Prior Work – FT Distributed RTS

- Protocols for specific components exist...
  - Byzantine fault-tolerant clock synchronization
    [M. Malekpour, 2006]
  - Omission fault-tolerant CAN bus
  - [J. Rufino et al
- ... but also g

How about **Byzantine** fault-tolerant distributed RTS?

Fault-tolerant real-time event service for CORBA

[H.-M. Huang and C. Gill, 2006]

- Middleware
- Multiple quality of service levels
- Fault model: **Fail-stop**

System-level Architecture for Failure Evasion in Real-time applications

[K. Junsung et al., 2012]

- Mixed criticality tasks
- Case study: "Boss" autonomous

This Work

Byzantine Fault Tolerance
- Replication
- Coordination
- → Fail-operational

Real-time Application
- Strict timing requirements
- Low latency
- Scheduleability

This Work

Key-value store

Provides:
- Byzantine fault tolerance
- Effortless replication

Supports:
- Timely termination
  - Inspired by logical execution time [T. A. Henziger et al., 2001]
  - Strong timing semantics
- Configurability
- Analyzability

Outline

- System model
  - Fault types
  - Protocol description
- Implementation
  - Overview
  - Interfaces
- Initial experiments
- Discussion
- Next steps

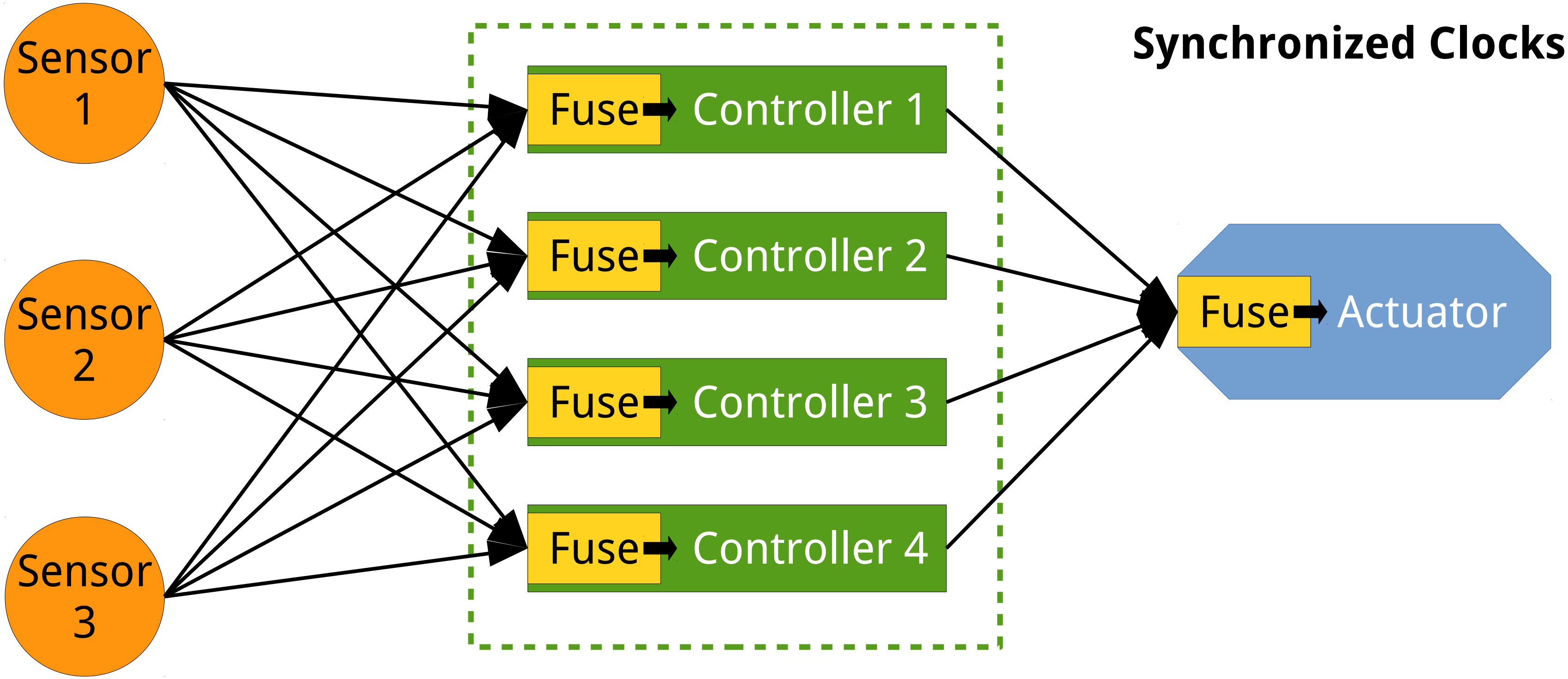
System Model

Multiple Sensors
- Same sensor type
- (Slightly) different outputs

Replicated Controllers
- Multiple (noisy) sensor inputs
- Equal outputs expected

Physical Actuator
- Multiple equal inputs

Synchronized Clocks!

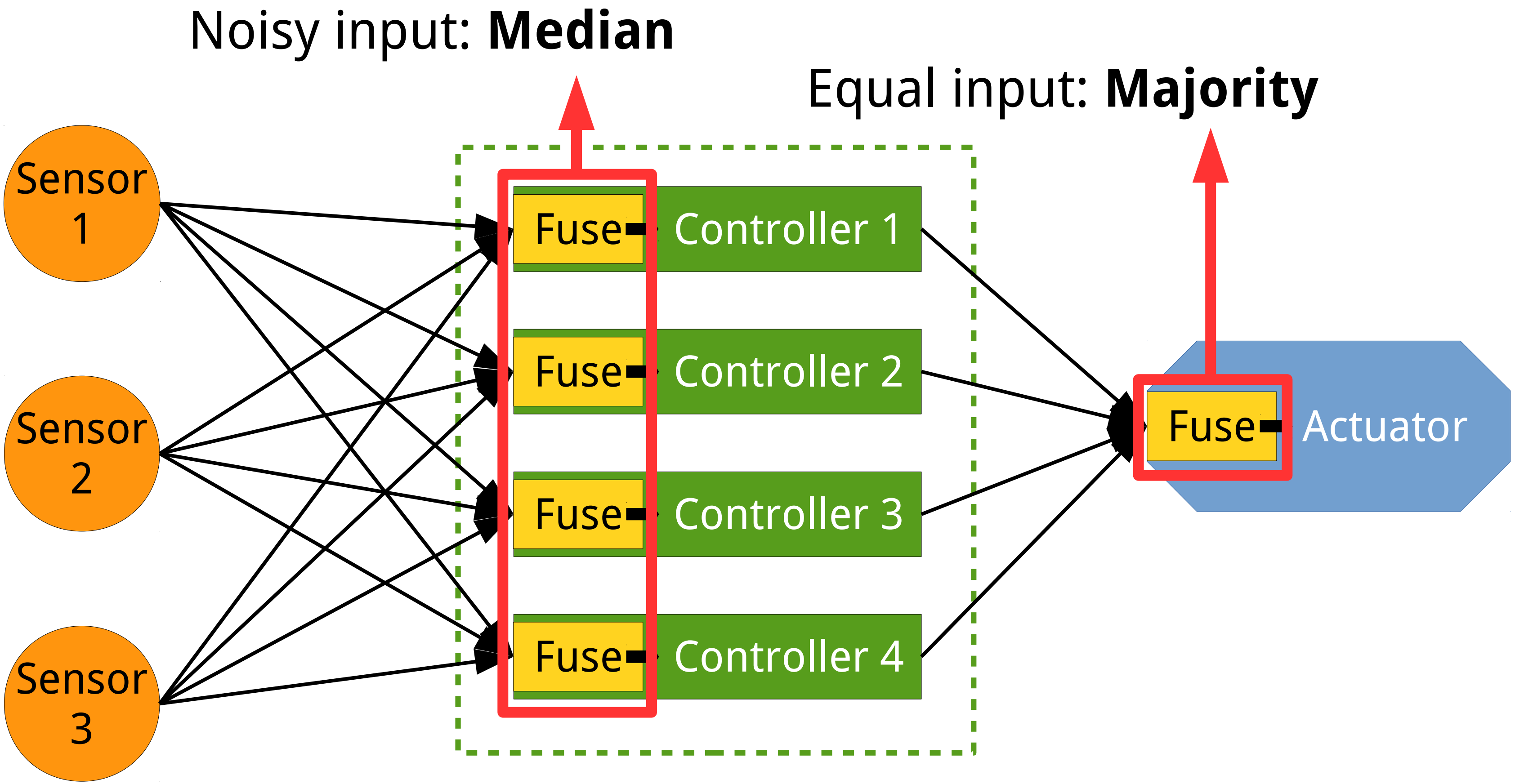


Fuse

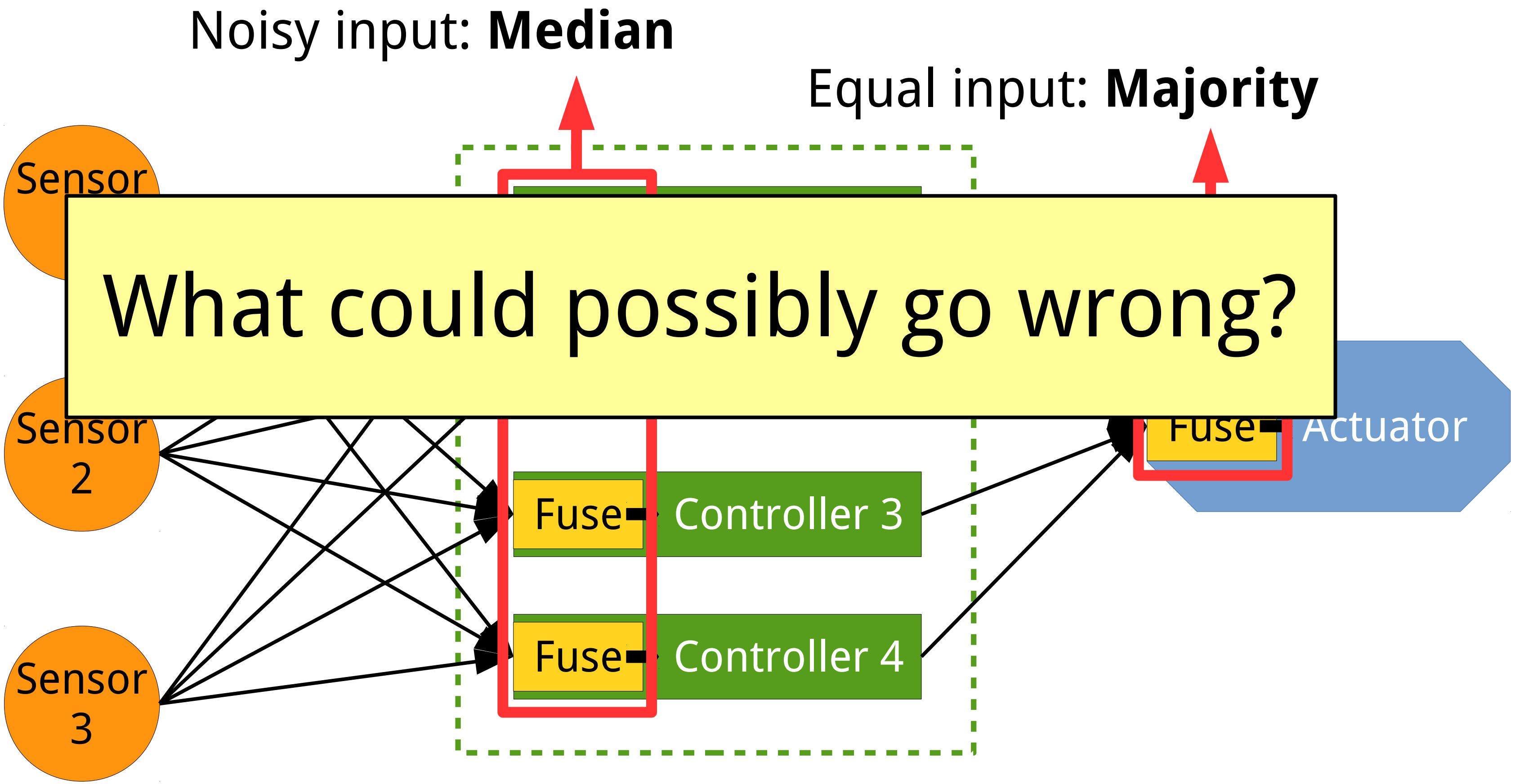
A user-defined function to fuse multiple values into one

- Different definitions possible
  - Average
  - Median
  - Majority
  - …

Fuse


Noisy input: **Median**
Equal input: **Majority**
Sensor 1
Sensor 2
Sensor 3
Fuse
Controller 1
Fuse
Controller 2
Fuse
Controller 3
Fuse
Controller 4
Fuse
Actuator

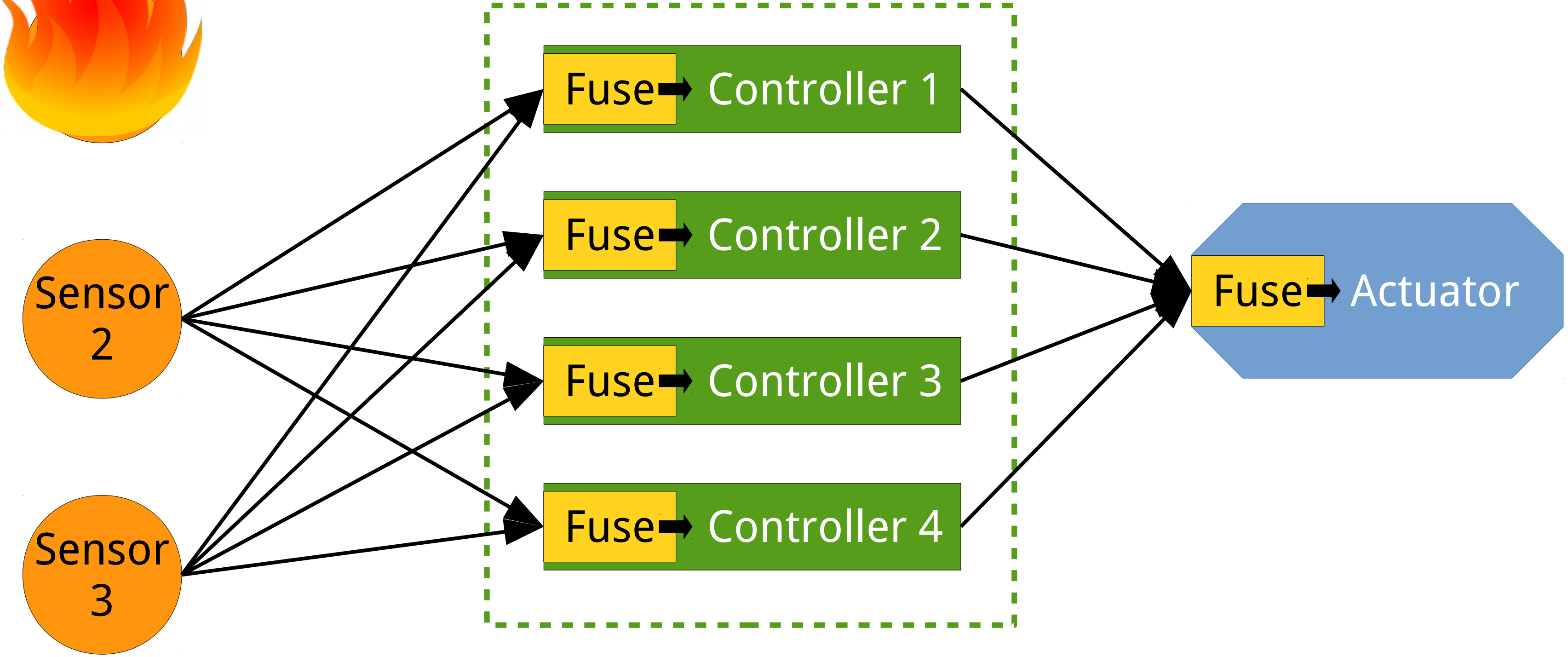
Fuse



Noisy input: **Median**
Equal input: **Majority**
Sensor
Sensor
2
Sensor
3
Fuse
Controller 3
Fuse
Controller 4
Fuse
Actuator
What could possibly go wrong?

Fault Types – Crash

Component **crashes** → Replication provides tolerance (in absence of other faults)

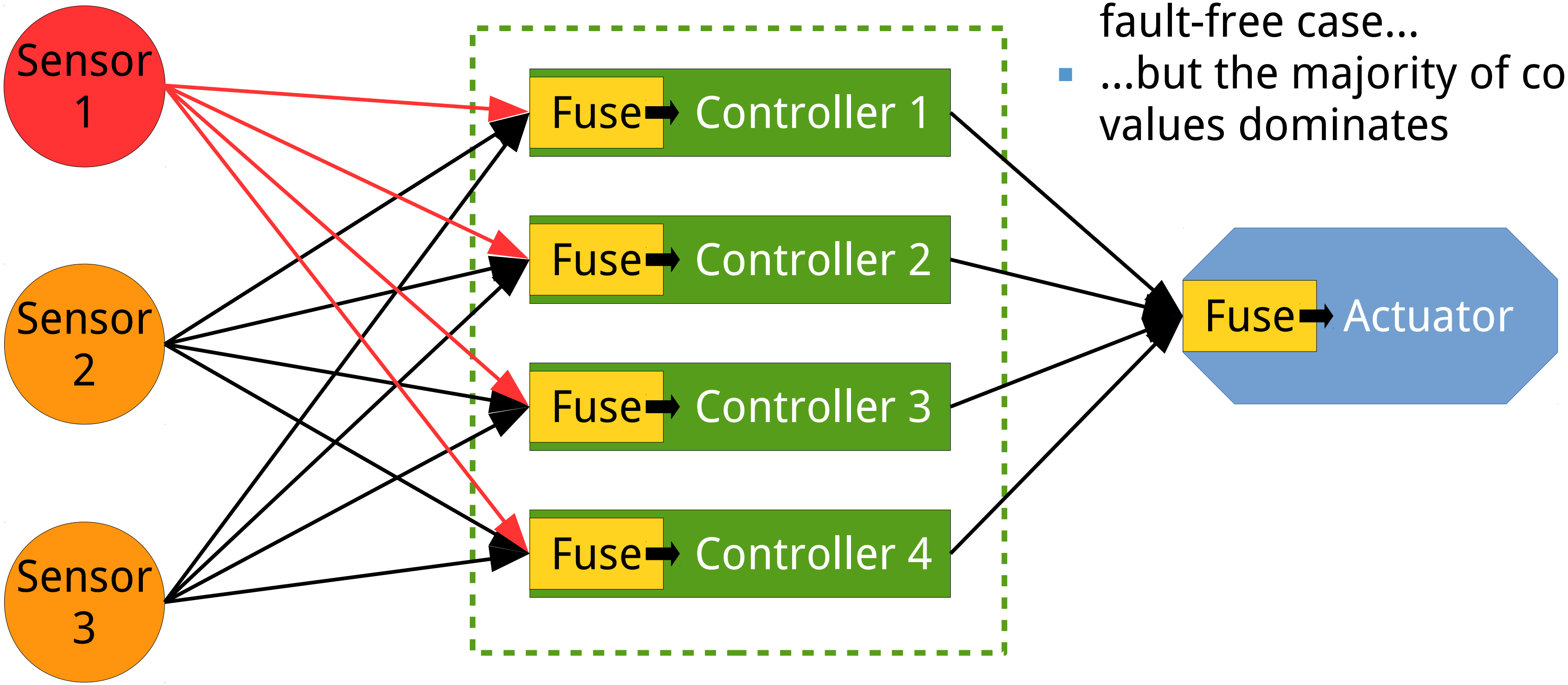


Fault Types – Consistent Wrong Value

Faulty component sends **wrong** values **<u>but</u>** values are **consistent**

→ Output of fuse is still **equal** on **all** replicas

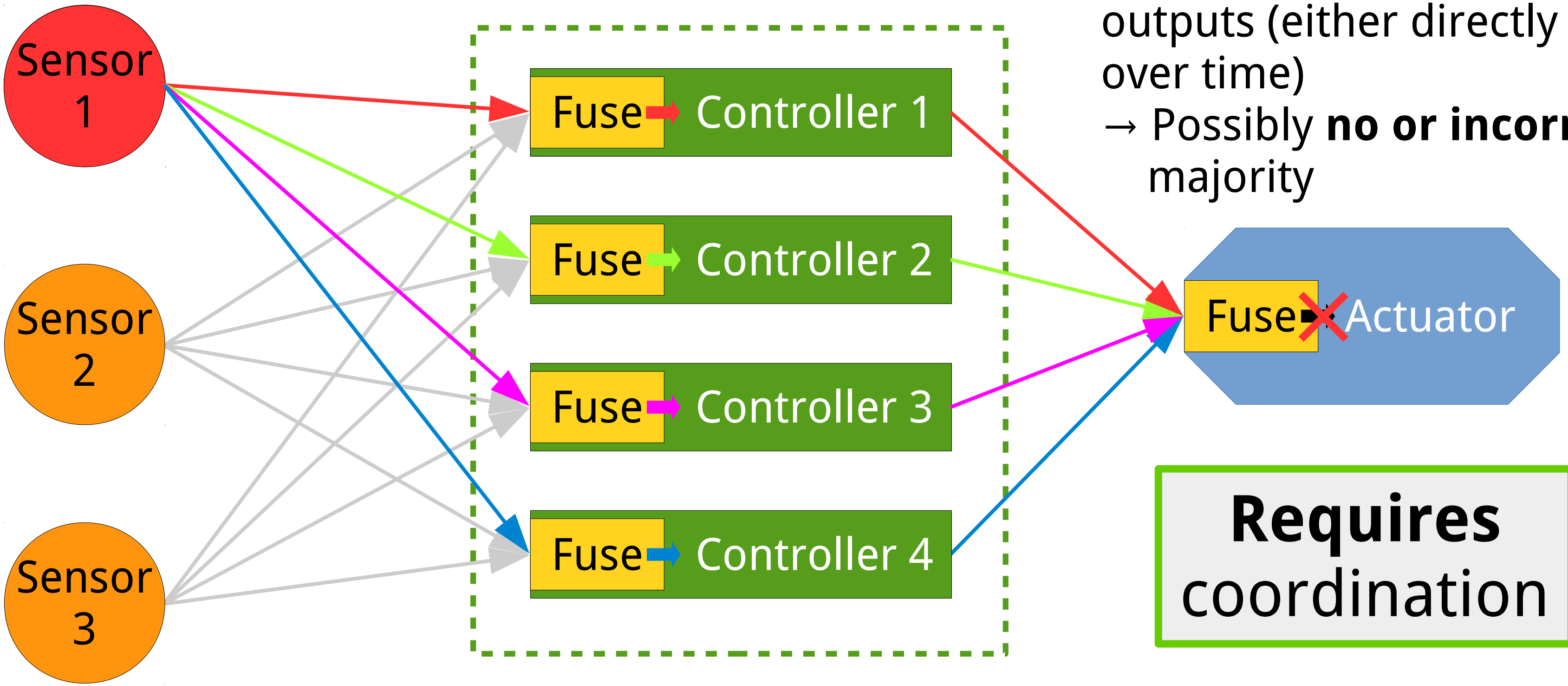
- Different, if compared to the fault-free case...
- ...but the majority of correct values dominates



Fault Types – Inconsistent Values

Faulty component sends **wrong** values **and** values are **inconsistent** → Output of fuse **differs** on **all** replicas

- Might lead to different outputs (either directly or over time)
  → Possibly **no or incorrect** majority

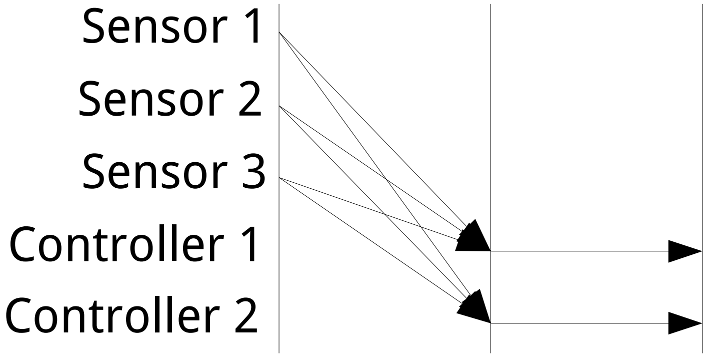


Requires coordination

Proposed Protocol

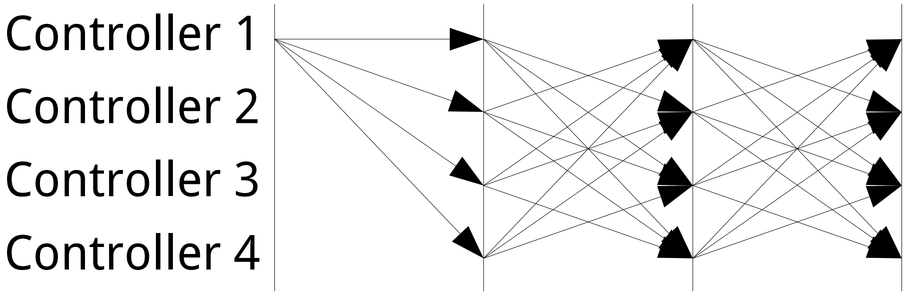
Simple broadcast + fuse

- For main operation
- Tolerates simple faults



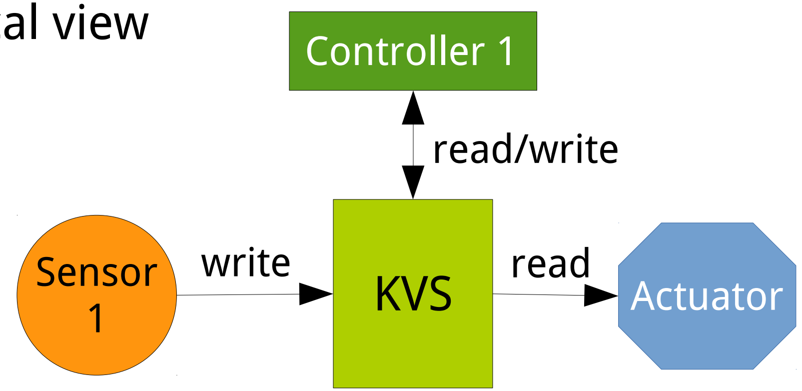
Periodical "Synchronization"

- Comparatively high cost and latency
  → Only <u>periodically</u> executed
- Frequency depends on the application



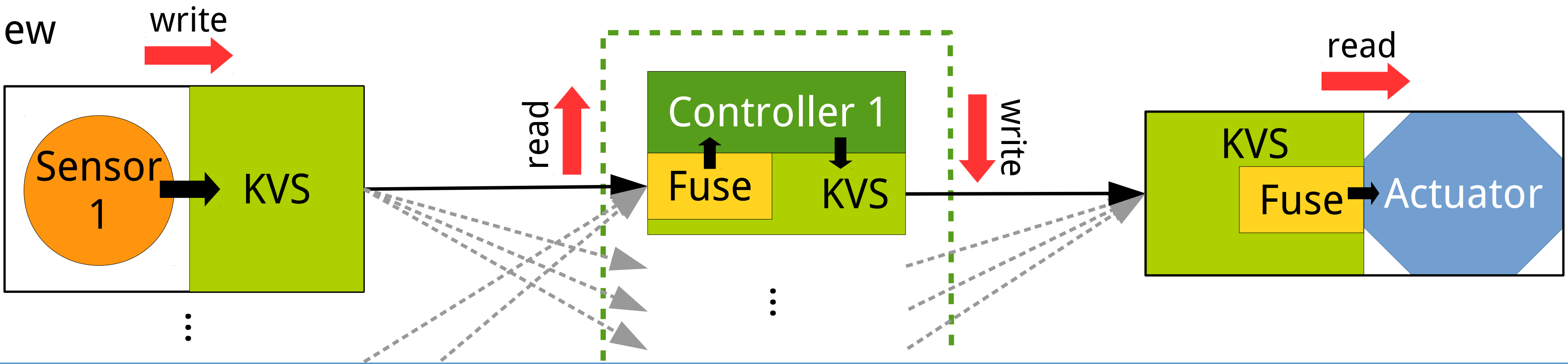
Implementation – Overview

Logical view



- All applications see **one logical** KVS
- Reality: One KVS **per node**
- Multiple applications (e.g., Sensor 1 & Controller 1) can be situated on the same node
- No manual networking or fuse, only **read** and **write**
- Values are accessible on **all** correct nodes

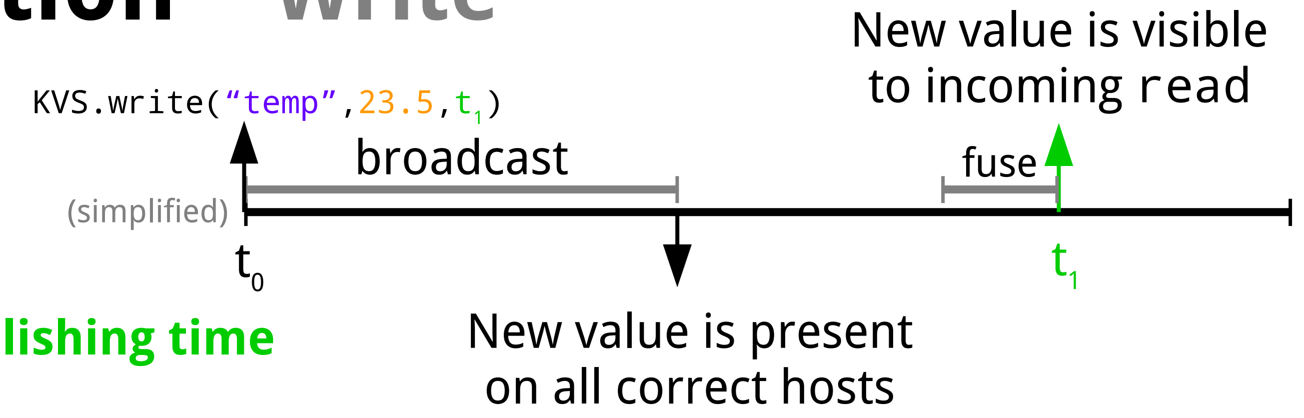
Actual view



Implementation – Write

`write(k,v,t)`

Key   Value   **Publishing time**



Latency of a single write can differ, because of...

- Network congestion
- Node utilization
- Faults
- ...

→ ~~**unpredictable** (and hard to coordinate)~~

Clear semantics allow reasoning about time

- Publishing time provides point in time when a write is **guaranteed** to have finished (or be ignored).
- Rationale: Writes that take **too long** are of **no use** anyways
- Actual execution and coordination is decoupled from logical execution ← **Logical execution time paradigm**
- `t` has to be lower bounded depending on the actual system

Implementation – Read

`read(k,t)`

Key — Earliest publish time

`KVS.read("temp",t0.5)` →

Local KVS

Key	Value	Publishing Time
temp	22.0	t_0
temp	23.5	t_1
temp	24.0	t_2

Now

← `23.5`

$t_0 < t_{0.5} < t_1 < t_2$ absolute timestamps

Newest value that is already published is returned

- t_0 too old
- t_2 not yet published

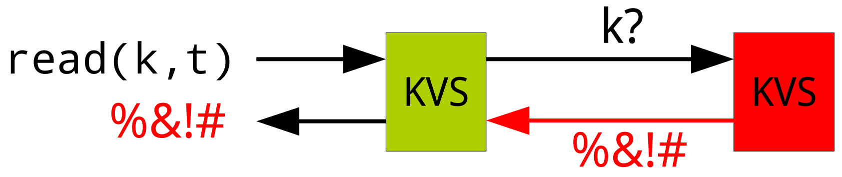
→ Value for t_1 is returned

Reads are always handled by the local KVS
→ Faster response

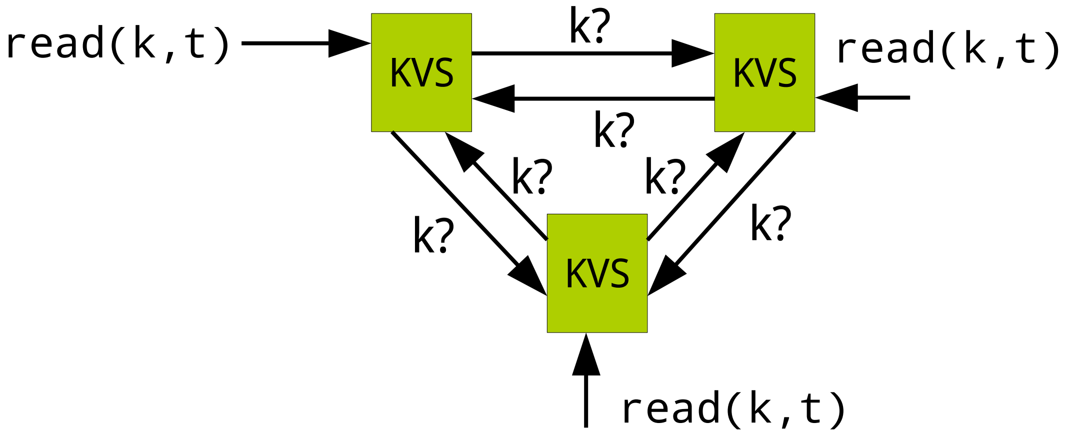
Implementation – Read

But what if there is no (fresh) value present?

- **Query the value from another KVS**
  → Might be faulty



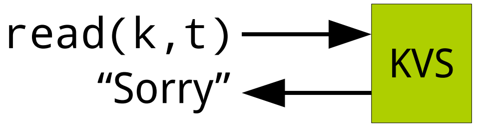
- **Query the value from all KVS**
  → Risk of flooding the network if value is not present in the system



Impossible to distinguish

(without querying everything)

- **Reply with error**
  → If value was missed because of a transient network partition (that is not present anymore), newer writes will be received, so try again later

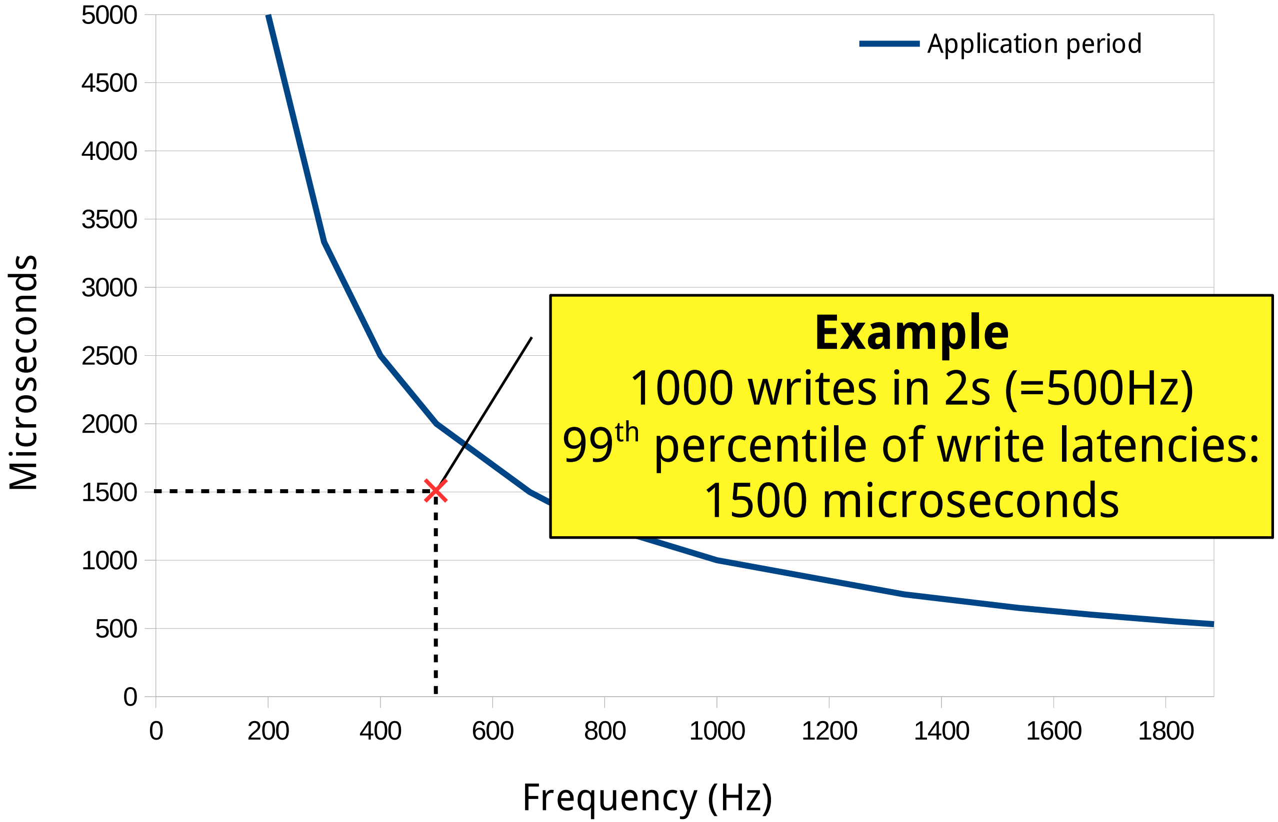


Initial Experiments – Baseline

Setup
- 2 physical nodes
- Ethernet connection
- 1 application
- 4 KVS replicas
- **3-phase commit**
- **No faults**

Measurements
- Performance baseline
- Write latency
- Application issues 1000 writes for each frequency
- 99th percentile plotted

→ When is the write latency higher than the period of the application?

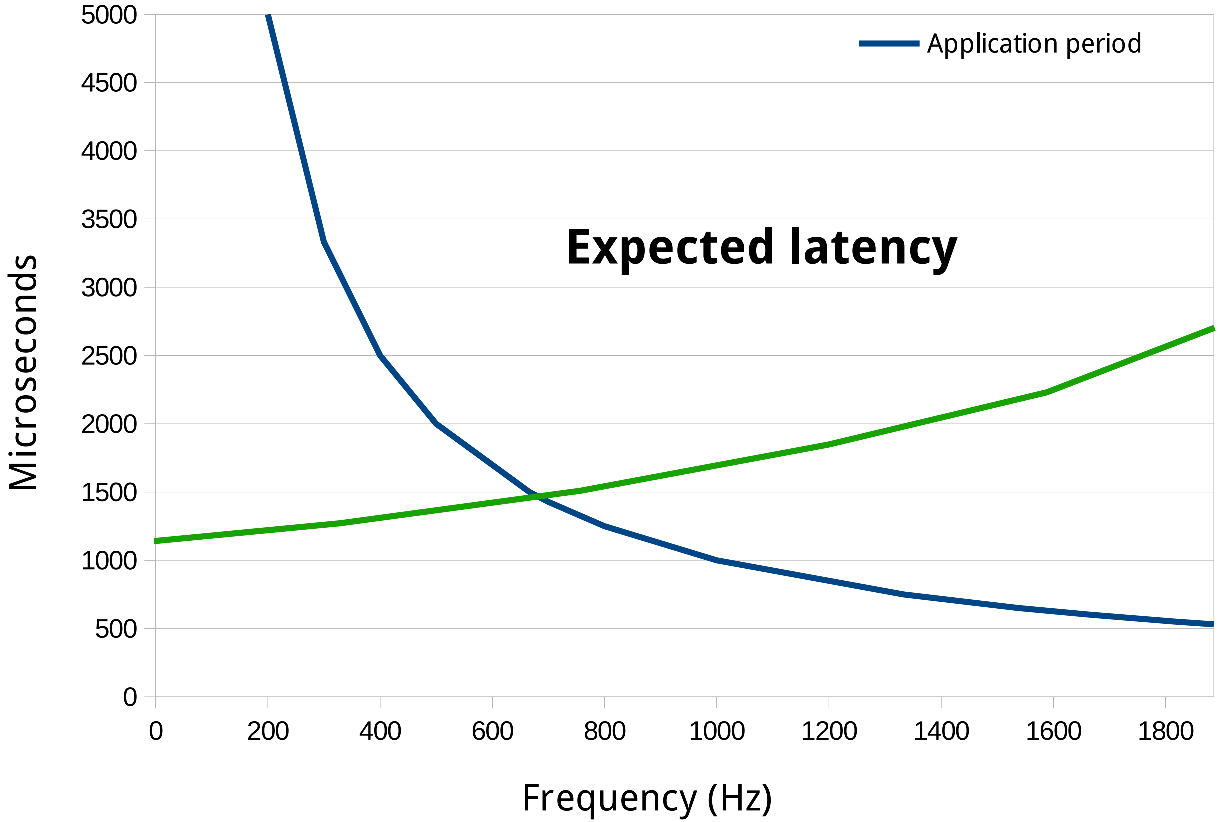


Initial Experiments – Baseline

Setup
- 2 physical nodes
- Ethernet connection
- 1 application
- 4 KVS replicas
- **3-phase commit**
- **No faults**

Measurements
- Performance baseline
- Write latency
- Application issues 1000 writes for each frequency
- 99th percentile plotted

→ When is the write latency higher than the period of the application?

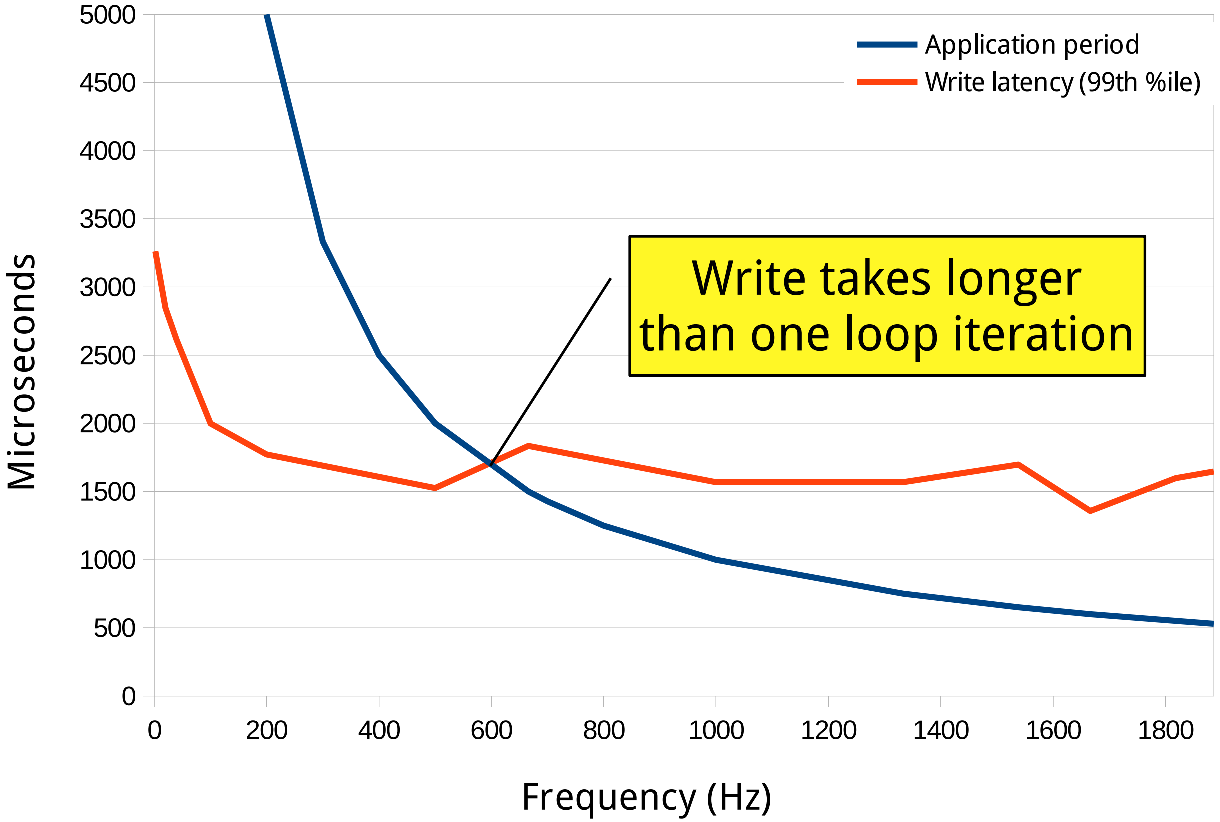


Initial Experiments – Baseline

Setup
- 2 physical nodes
- Ethernet connection
- 1 application
- 4 KVS replicas
- **3-phase commit**
- **No faults**

Measurements
- Performance baseline
- Write latency
- Application issues 1000 writes for each frequency
- 99th percentile plotted

→ When is the write latency higher than the period of the application?



Discussion

- Timed Byzantine fault-tolerant key-value store
- Guarantees

Common for BFT

  - **Validity**
  - **Freshness** (read t parameter)
  - **Agreement**

+

  - **Timely Termination** (write t parameter)

- Usable with fewer replicas if a lower level of fault tolerance is sufficient
  - Byzantine: 3f+1
  - Crash: f+1

  → **Time semantics** stay the same
- This allows for **effortless replication** of an application
  1. Spin up a new replica
  2. Start the application without code changes (same key / timestamp usage)

Next steps

- Implement remaining parts of the system
- Evaluation
  - Fault injection experiments
    - Inject faults into random parts of the implementation: Fuse, KVS, synchronization, ...
    - ... and into physical host memory, to see how the complete system reacts.
    - → Fault injection **not** limited to our binary!
  - Performance
- More functionality?

Thanks! Questions?