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
Distributed Real-Time Systems

Susceptible to faults
- Electromagnetic interference
- Thermal effects
- ...

possible consequences
- Bit-flips
- Crashes
- Madness

Common Mitigation Techniques

- **System Repair**
  - Requires accessible system

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

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

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

Assumptions for high-frequency systems

- Low latency
- No downtime
- Fail-operational

Fail-safe

Fail-operational

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
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 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: “Booo” autonomous vehicle

- How about **Byzantine** fault-tolerant distributed RTS?
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 ➔ Fuse ➔ Controller 1 ➔ Fuse ➔ Controller 2 ➔ Fuse ➔ Controller 3 ➔ Fuse ➔ Controller 4 ➔ Actuator
What could possibly go wrong?

Noisy input: Median

Equal input: Majority
Fault Types – Crash

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

- **Sensor 1**
- **Sensor 2**
- **Sensor 3**
- **Controller 1**
- **Controller 2**
- **Controller 3**
- **Controller 4**
- **Fuse**
- **Actuator**
Fault Types – Consistent Wrong Value

Faulty component sends **wrong** values but 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 “Synchronisation”
- Comparatively high cost and latency
  → Only periodically executed
- Frequency depends on the application
Implementation – Overview

- 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
Implementation – Write

\[
\text{write}(k, v, t)
\]

KVS.write("temp", 23.5, t)

New value is visible to incoming read

New value is present on all correct hosts

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 \textbf{guaranteed} to have finished (or be ignored).
- Rationale: Writes that take \textbf{too long} are of \textbf{no use} anyways
- Actual execution and coordination is decoupled from logical execution ← \textbf{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", t_{0.5})

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

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Publishing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp</td>
<td>22.0</td>
<td>t_0</td>
</tr>
<tr>
<td>temp</td>
<td>23.5</td>
<td>t_1</td>
</tr>
<tr>
<td>temp</td>
<td>24.0</td>
<td>t_2</td>
</tr>
</tbody>
</table>

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

---

**Diagram:**

- `read(k,t)` from `KVS` to `KVS`
- `k?` from `KVS` to `KVS`
- `read(k,t)` from `KVS` to `KVS`
- `k?` from `KVS` to `KVS`
- `read(k,t)` from `KVS` to `KVS`
- `read(k,t)` from `KVS` to `KVS`
- `read(k,t)` from `KVS` to `KVS`
- “Sorry” from `KVS` to `KVS`
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?

Example
1000 writes in 2s (=500Hz)
99th percentile of write latencies: 1500 microseconds
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
- 99\textsuperscript{th} percentile plotted

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

- Timed Byzantine fault-tolerant key-value store
- Guarantees
  - 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: \( f+1 \)
  \( \rightarrow \) **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?