Distributed Systems

Replicated, Highly-Available Data Store

POLITECNICO
MILANO 1863

Prof. Gianpaolo Cugola

Students: *Angelo Prete, Matteo Rossi*

Academic Year - 2024/25

May 2025

Project Overview

Simulated in **OmNet++**

Implement a *highly-available*, *replicated* key value store.

The store offers two primitives: *read(key)* and *write(key, value)*.

The store is implemented by N servers.
Each of them contains a copy of the entire store and cooperate to keep a consistent view of this replicated store.
In particular, the system must provide a **Causal Consistency** model.

The system must be **Highly Available**, meaning that a client can continue to fully operate as soon as it remains connected to its server, even if that server is disconnected from other servers in the system.

Assumptions

Servers and Channels may fail - ***Omission Failures* Only**

Clients may connect to any of the servers, but they are **Sticky Clients**.

Our Additional Hypotheses

Data Types Key: *String* Value: *Integer*

Messages are discarded only during reception from modules, with an uniform distribution probability

Messages cannot be partially processed by the Data Store and suddenly the operation is interrupted, leaving it in an inconsistent state

Protocol

Message Types - Network Message Hierarchy

Client To Server

ReadRequestMsg

WriteRequestMsg

Server To Server

WritePropagationMsg

MissingWritesRequestMsg

HeartbeatMsg

Server To Client

ReadResponseMsg

WriteResponseMsg

Protocol

Ahamad, M., Neiger, G., Burns, J.E. et al.
Causal memory: definitions, implementation, and programming. Distrib Comput 9, 37–49 (1995).
https://doi.org/10.1007/BF01784241

Causal Consistency Implementation

Each node maintains a **Vector Clock** mapping Data Store IDs to Number of Performed Writes

An update (WritePropagationMsg) is applied only if both conditions are satisfied:

- The Receiver has seen at least as many updates from all other nodes as the sender had
- The received update is the next in order from that node
  SenderVectorClock[SourceId] = ReceiverVectorClock[SourceId] + 1

Protocol

Storing Pending Updates

When an update cannot be applied because it doesn't satisfy causal dependencies, it is stored in a *PendingWrites* vector

Each time another update is applied, the content of this data structure is checked in order to verify if any previous message is now applicable

Protocol

Caching All Writes & Seen Updates

Each time a write or an update is processed by a Data Store,
It is also stored inside a Map

When another Data Store realizes it has not seen an Update
It asks to others if they have seen it,
which will eventually reply with all Missing Updates

Garbage Collection Mechanism
When a Data Store acknowledges that everybody has seen that update, it is removed from the Map

Causal Consistency Verification

Log Parser and Checker in Python

i. Produces a **clean log** from the raw .elog file

ii. Parses it to **extract read** and **write operations** with **vector clocks**

iii. Checks if each read value is causally consistent with prior writes

iv. **Reports violations** where no valid causal write exists for a read

✅ `No causal violations found`

❌ `Causal violation: READ at SERVER-1 from CLIENT-2 sees value '162' with VC [2, 7, 3], but no consistent write exists.`

Network Partitions

```
double createNetworkPartitionProbability = default(0.1);
double terminateNetworkPartitionProbability = default(0.5);
double networkPartitionLinkProbability = default(0.5);
double networkPartitionEventInterval @unit(s) = default(uniform(5s,10s));
```

Periodically, Data Stores can be affected by a Network Partition, which involves a random set of links (at least one)

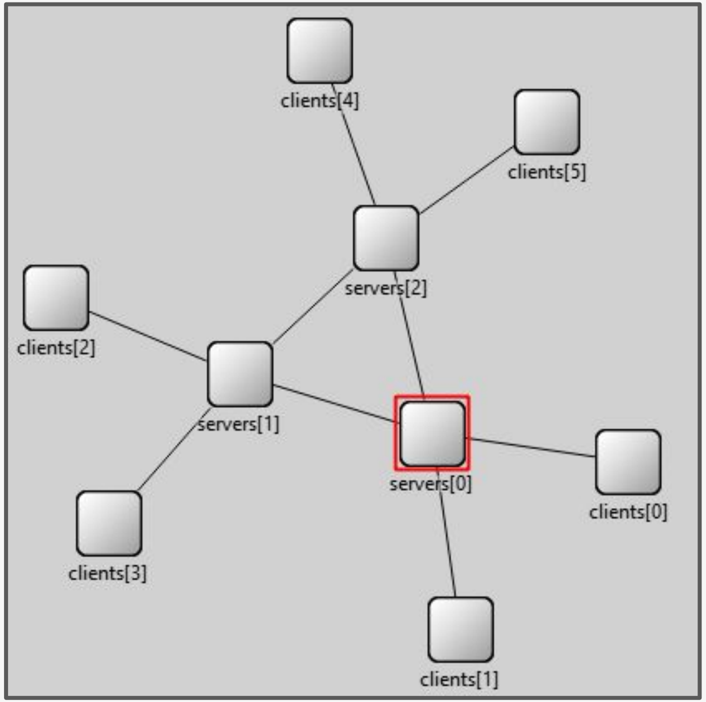
LOG: [SERVER-0] Network Partition Created | Affected Links: 1 2

All the messages exchanged on the affected links during that time interval are discarded

LOG: [SERVER-0] Incoming Packet Lost | Cause: Network Partition | From: [SERVER-1] | Type: WritePropagationMsg

Statistical Analysis

Changing the Parameters | Servers Topology: Fully Connected Network



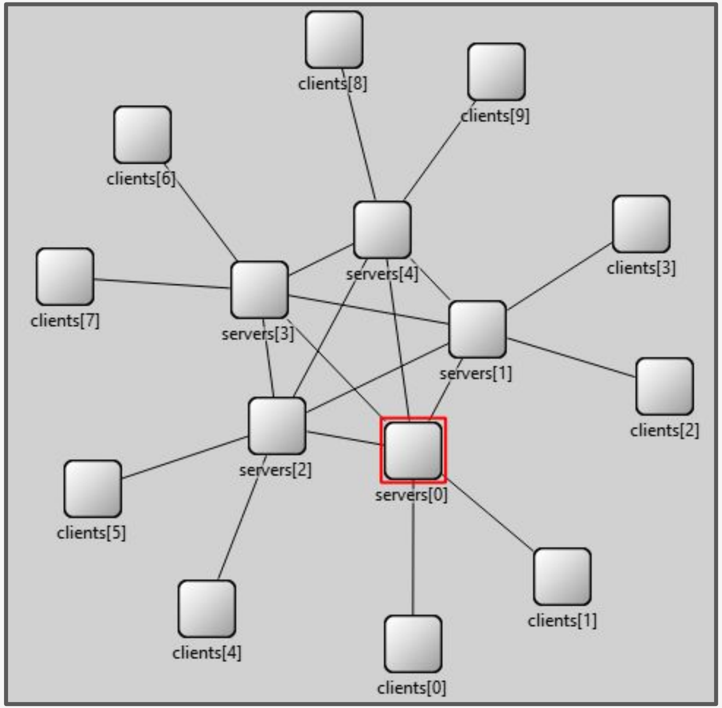
Simulation Time: **960s = 15 min**

3 Data Stores - 2 Clients on each

1. No Omissions & No Network Partitions
2. Omissions & Network Partitions - Linear
3. Omissions & Network Partitions - Flooding

Statistical Analysis

Changing the Parameters | Servers Topology: Fully Connected Network



Simulation Time: **960s = 15 min**

5 Data Stores - 2 Clients on each

4. Omissions & Network Partitions - *Linear*

Strategy *linear* means

$$\left\lceil \frac{\log(\#\text{Data Stores})}{\log(2)} \right\rceil = 3$$

Statistical Analysis

Convergence of Vector Clocks by Increasing #Data Stores

N = 3

	VC[0]	VC[1]	VC[2]	Partitions Started
Server[0]	**295**	273	300	9
Server[1]	287	**287**	307	9
Server[2]	291	282	**311**	7

Statistical Analysis

Convergence of Vector Clocks by Increasing #Data Stores

N = 5

	VC[0]	VC[1]	VC[2]	VC[3]	VC[4]	Partitions Started
Server[0]	**299**	325	316	346	289	4
Server[1]	299	**326**	316	346	289	4
Server[2]	297	324	**318**	342	287	9
Server[3]	299	326	316	**346**	289	12
Server[4]	299	326	316	346	**289**	9

Statistical Analysis

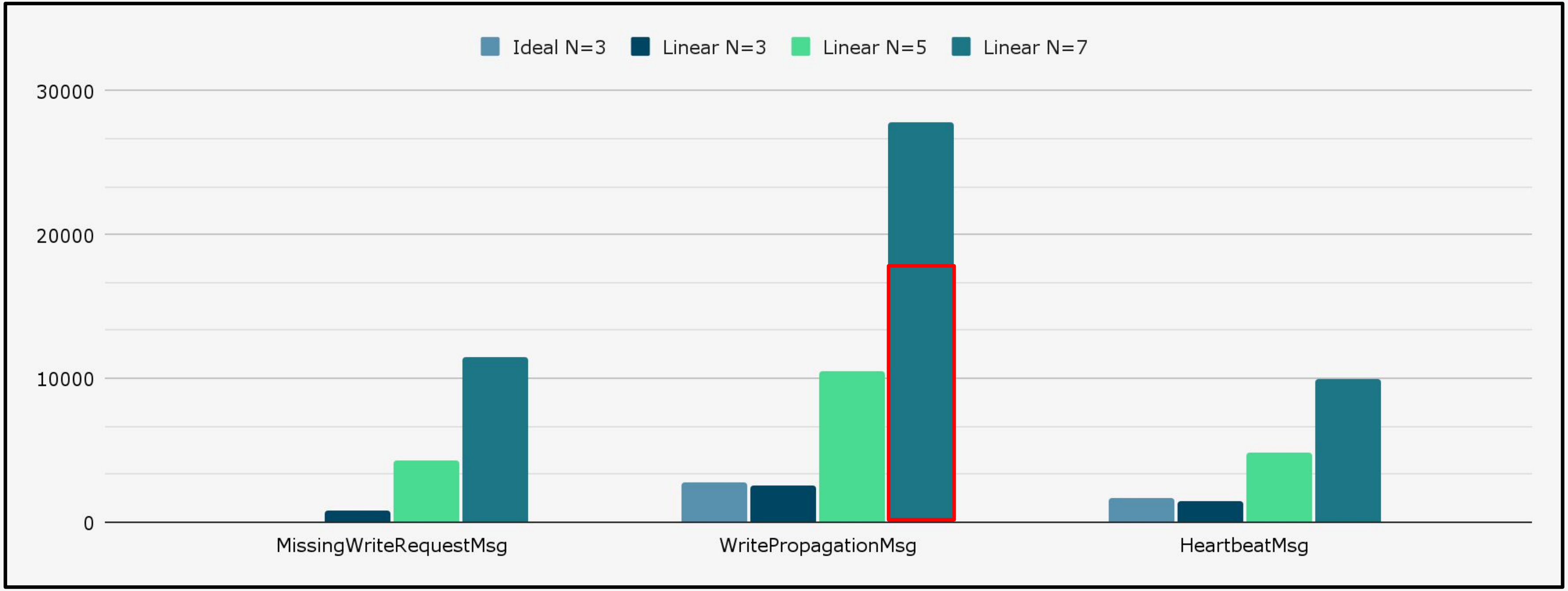
Convergence of Vector Clocks by Increasing #Data Stores

N = 7

	VC[0]	VC[1]	VC[2]	VC[3]	VC[4]	VC[5]	VC[6]	Partitions Started
Server[0]	**338**	291	354	342	367	364	287	8
Server[1]	338	**292**	354	343	368	364	285	11
Server[2]	338	291	**354**	342	368	364	286	10
Server[3]	338	291	354	**344**	368	364	285	6
Server[4]	338	292	354	343	**368**	364	285	5
Server[5]	338	290	354	342	368	**364**	285	8
Server[6]	337	290	350	340	363	361	**288**	9

Statistical Analysis

Comparison between Number of Packets Exchanged by Servers in 15 mins



Reducing the Traffic

```cpp
// Tracking of Last Request made for a specific update
typedef struct {
    simtime_t lastRequestTime;
    int requestCount;
}RequestInfo;

// Map to track when we last requested each missing update:
// <serverId, <updateId, RequestInfo>>
std::map<int, std::map<int, RequestInfo>> lastRequestTimes;
```

```cpp
// Populate the map with the missing updates
if (vectorClock[i] < senderVectorClock.at(i)) {
    int lowestUpdateId = vectorClock[i];
    for (int j = lowestUpdateId + 1; j <= senderVectorClock.at(i); j++) {
        // Find the Update with the given updateId (vectorClock[sourceId]) and sent by the server
        // with its ID equal to sourceId. This guarantees that the (possibly) found update is the one we were looking for.
        auto it = std::find_if(pendingUpdates.begin(), pendingUpdates.end(),
            [j, i](const Update& u) { return (u.senderVectorClock.at(u.sourceId) == j) && (u.sourceId == i); });

        // If the update has not been found in the pending ones, add its ID to the missing writes to be requested
        if (it == pendingUpdates.end()) {
            allMissingWrites[i].insert(j);

            // Check if we've recently requested this update
            std::vector<std::tuple<int, int, simtime_t, simtime_t>> skippedRequests;

            if (lastRequestTimes.find(i) == lastRequestTimes.end() || lastRequestTimes[i].find(j) == lastRequestTimes[i].end()) {
                // 1st time requesting this update
                missingWritesToRequest[i].insert(j);
                lastRequestTimes[i][j] = {currentTime, 1};
                requestedCount++;
            } else {
                RequestInfo& reqInfo = lastRequestTimes[i][j];

                // Calculate the cooldown period with exponential backoff
                simtime_t cooldown = std::min(INITIAL_COOLDOWN * std::pow(BACKOFF_FACTOR, reqInfo.requestCount - 1), MAX_COOLDOWN);

                // Check if the cooldown period has passed
                if (currentTime - reqInfo.lastRequestTime >= cooldown) {
                    missingWritesToRequest[i].insert(j);
                    reqInfo.lastRequestTime = currentTime;
                    reqInfo.requestCount++;
                    requestedCount++;
                } else {
                    skippedRequests.emplace_back(i, j, currentTime - reqInfo.lastRequestTime, cooldown);
                    skippedCount++;
                }
            }
```

Cooldown Period

\+

Exponential Backoff

Mechanism

`lastRequestTimes`

is periodically cleaned by the received Updates

Reducing the Traffic

MissingWritesRequestMsg [N=5]							
Incoming	Outgoing	Discarded	Fulfilled [Msg]	**Requested [Updates]**	**Skipped [Updates]**	Fulfillment Rate	**Cooldown Effectiveness Rate**
missingWritesRequestStrategy: "Linear", maxNodesToContact: 3							
964	820	215	632	617	1131	65.56%	**64.70%**
857	858	189	530	684	1792	61.84%	**72.37%**
885	892	210	541	641	1121	61.13%	**63.62%**
797	888	202	509	1236	5487	63.86%	**81.61%**
822	867	207	484	960	3979	58.88%	**80.56%**

Reducing the Traffic

MissingWritesRequestMsg [N=7]							
Incoming	Outgoing	Discarded	Fulfilled [Msg]	**Requested [Updates]**	**Skipped [Updates]**	Fulfillment Rate	**Cooldown Effectiveness Rate**
missingWritesRequestStrategy: "Linear", maxNodesToContact: 3							
1523	1526	377	902	2523	20616	59.22%	**89.1%**
1826	1511	522	1083	1547	6684	59.31%	**81.2%**
1610	1483	434	946	1489	5807	58.76%	**79.59%**
1581	1670	345	974	1290	4695	61.61%	**78.45%**
1598	1503	351	965	1129	4508	60.39%	**79.97%**
1607	1805	422	944	1766	12264	58.74%	**87.41%**
1695	1945	382	1070	1551	11807	63.13%	**88.39%**

DEMO!