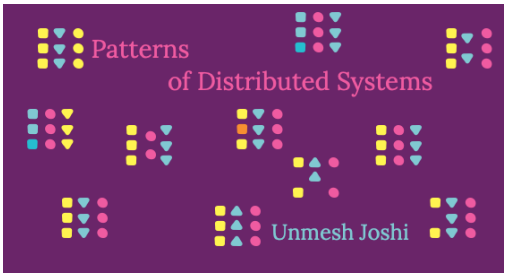


This pattern is part of Patterns of Distributed Systems

Write-Ahead Log

Provide durability guarantee without the storage data structures to be flushed to disk, by persisting every state change as a command to the append only log.

12 August 2020

Unmesh Joshi
aka: Commit Log

Problem

Strong durability guarantee is needed even in the case of the server machines storing data failing. Once a server agrees to perform an action, it should do so even if it fails and restarts losing all of its in-memory state.

Solution

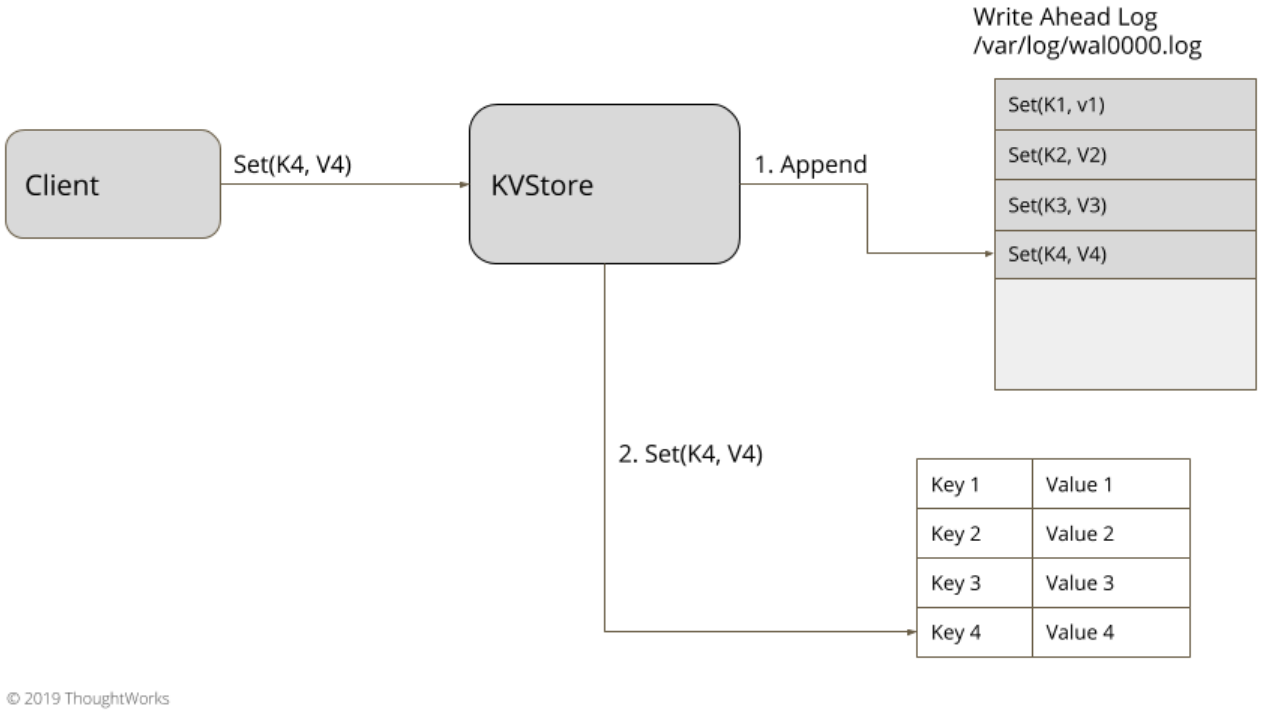


Figure 1: Write Ahead Log

Store each state change as a command in a file on a hard disk. A single log is maintained for each server process which is sequentially appended. A single log which is appended sequentially, simplifies the handling of logs at restart and for subsequent online operations (when the log is appended with new commands). Each log entry is given a unique identifier. The unique log identifier helps in implementing certain other operations on the log like Segmented Log or cleaning the log with Low-Water Mark etc. The log updates can be implemented with Singular Update Queue

The typical log entry structure looks like following

class WALEntry...

```
  private final Long entryId;
  private final byte[] data;
  private final EntryType entryType;
  private long timeStamp;
```

The file can be read on every restart and the state can be recovered by replaying all the log entries.

Consider a simple in memory key-value store:

class KVStore...

```
  private Map<String, String> kv = new HashMap<>();

  public String get(String key) {
      return kv.get(key);
  }

  public void put(String key, String value) {
      appendLog(key, value);
      kv.put(key, value);
  }

  private Long appendLog(String key, String value) {
      return wal.writeEntry(new SetValueCommand(key, value).serialize());
  }
```

The put action is represented as Command, which is serialized and stored in the log before updating the in memory hashmap.

class SetValueCommand...

```
  final String key;
  final String value;
  final String attachLease;
  public SetValueCommand(String key, String value) {
      this(key, value, "");
  }
  public SetValueCommand(String key, String value, String attachLease) {
      this.key = key;
      this.value = value;
      this.attachLease = attachLease;
  }

  @Override
  public void serialize(DataOutputStream os) throws IOException {
      os.writeInt(Command.SetValueType);
```

```
        os.writeUTF(key);
        os.writeUTF(value);
        os.writeUTF(attachLease);
    }

    public static SetValueCommand deserialize(InputStream is) {
        try {
            DataInputStream dataInputStream = new DataInputStream(is);
            return new SetValueCommand(dataInputStream.readUTF(), dataInputStream.readU
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
```

This makes sure that once the put method returns successfully, even if the process holding the KVStore crashes, its state can be restored by reading the log file at startup.

class KVStore...

```
    public KVStore(Config config) {
        this.config = config;
        this.wal = WriteAheadLog.openWAL(config);
        this.applyLog();
    }

    public void applyLog() {
        List<WALEntry> walEntries = wal.readAll();
        applyEntries(walEntries);
    }

    private void applyEntries(List<WALEntry> walEntries) {
        for (WALEntry walEntry : walEntries) {
            Command command = deserialize(walEntry);
            if (command instanceof SetValueCommand) {
                SetValueCommand setValueCommand = (SetValueCommand)command;
                kv.put(setValueCommand.key, setValueCommand.value);
            }
        }
    }
```

```
public void initialiseFromSnapshot(SnapShot snapShot) {
    kv.putAll(snapShot.deserializeState());
}
```

Implementation Considerations

There are some important considerations while implementing Log. It's important to make sure that entries written to the log file are actually persisted on the physical media. File handling libraries provided in all programming languages provide a mechanism to force the operating system to 'flush' the file changes to physical media. There is a trade off to be considered while using flushing mechanism.

Flushing every log write to the disk gives a strong durability guarantee (which is the main purpose of having logs in the first place), but this severely limits performance and can quickly become a bottleneck. If flushing is delayed or done asynchronously, it improves performance but there is a risk of losing entries from the log if the server crashes before entries are flushed. Most implementations use techniques like Batching, to limit the impact of the flush operation.

The other consideration is to make sure that corrupted log files are detected while reading the log. To handle this, log entries are generally written with the CRC records, which then can be validated when the files are read.

Single Log files can become difficult to manage and can quickly consume all the storage. To handle this issue, techniques like Segmented Log and Low-Water Mark are used.

The write ahead log is append-only. Because of this behaviour, in case of client communication failure and retries, logs can contain duplicate entries. When the log entries are applied, it needs to make sure that the duplicates are ignored. If the final state is something like a HashMap, where the updates to the same key are idempotent, no special mechanism is needed. If they're not, there needs to be

some mechanism implemented to mark each request with a unique identifier and detect duplicates.

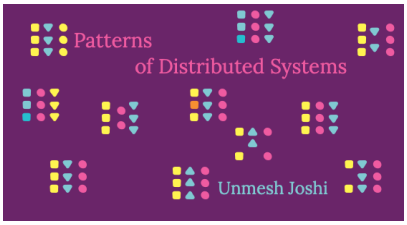
Examples

- The log implementation in all Consensus algorithms like Zookeeper and RAFT is similar to write ahead log
- The storage implementation in Kafka follows similar structure as that of commit logs in databases
- All the databases, including the nosql databases like Cassandra use write ahead log technique to guarantee durability

This page is part of:

Patterns of Distributed Systems

by **Unmesh Joshi**



Main Narrative Article

Patterns

Consistent Core
Fixed Partitions †
Follower Reads
Generation Clock
Gossip Dissemination
HeartBeat
High-Water Mark
Hybrid Clock
Idempotent Receiver
Key And Value †
Lamport Clock

Leader and Followers
Lease
Low-Water Mark
Paxos †
Quorum
Replicated Log †
Request Batch †
Request Pipeline
Segmented Log
Single Socket Channel
Singular Update Queue
State Watch
Two Phase Commit †
Version Vector
Versioned Value
Write-Ahead Log

† pattern in progress

▶ Significant Revisions