**Comparison Table**

| **Feature** | **Optimistic Locking** | **Pessimistic Locking** |
| --- | --- | --- |
| **Locking** | No initial lock, uses version checking. | Locks resources during the transaction. |
| **Performance** | Better in low-contention scenarios. | Better in high-contention scenarios. |
| **Conflict Handling** | Requires transaction retries. | Prevents conflicts upfront. |
| **Concurrency** | High concurrency. | Limited concurrency due to locks. |
| **Overhead** | Low overhead for reads. | High overhead due to locks. |

**Advantages and Disadvantages**

| **Aspect** | **Serialization** | **Deserialization** |
| --- | --- | --- |
| **Advantages** | Enables data persistence and transmission. | Restores object state seamlessly. |
| **Disadvantages** | Can introduce overhead and versioning issues. | Requires proper class compatibility. |

**Serialization Formats**

1. **Binary Serialization:** Formats like Java's native serialization and Python's pickle.
2. **Text-based Formats:**
   * JSON: Lightweight and widely used.
   * XML: Structured but verbose.
   * YAML: Human-readable.
3. **Protobuf/Avro:** Compact and efficient for large data sets.
4. **Comparison: Fault Tolerance vs. Resilience**

| **Aspect** | **Fault Tolerance** | **Resilience** |
| --- | --- | --- |
| **Goal** | Maintain seamless operation despite faults. | Recover and adapt after failures or stress. |
| **Approach** | Proactively prevents faults from affecting users. | Reactively recovers and adapts. |
| **Focus** | Redundancy and masking failures. | Recovery, adaptability, and stress handling. |
| **Techniques** | Failover, replication, error correction. | Circuit breakers, bulkheads, chaos engineering. |
| **Example** | Backup server automatically taking over. | Service degrading video quality under load. |

**Integration of Fault Tolerance and Resilience**

Modern systems often combine fault tolerance and resilience to achieve high reliability:

1. **Fault Tolerance** ensures immediate continuity during failures.
2. **Resilience** prepares the system to recover and adapt for long-term reliability.

**Cyclomatic Complexity** is a software metric used to measure the complexity of a program's control flow. It quantifies the number of linearly independent paths through a program’s source code, which in turn provides insights into the code's maintainability, testability, and potential risks.

**Formula:**

Cyclomatic complexity (M) is calculated using the formula:

M=E−N+2PM = E - N + 2PM=E−N+2P

Where:

* **E** = Number of edges in the flow graph.
* **N** = Number of nodes in the flow graph.
* **P** = Number of connected components (for a single program, P is typically 1).

**Interpretation:**

* **Low cyclomatic complexity (M ≤ 10):** Easier to understand, test, and maintain.
* **Medium cyclomatic complexity (10 < M ≤ 20):** More complex and requires more testing.
* **High cyclomatic complexity (M > 20):** Difficult to maintain, prone to errors, and requires substantial effort for testing.