Vector Clock and Singhal-Kshemkalyani Optimization: Project Report

---

1. Introduction

This report presents the implementation and comparative analysis of two distributed clock synchronization algorithms: the basic Vector Clock algorithm and its optimized variant, the Singhal-Kshemkalyani optimization. The goal is to demonstrate the strong consistency property of vector clocks and measure the performance improvements achieved through optimization.

1.1 Objectives

- Implement basic Vector Clock algorithm
- Implement Singhal-Kshemkalyani optimization
- Demonstrate strong consistency properties
- Compare performance overheads and message communication savings
- Analyze scalability with varying process counts

1.2 Problem Statement

In a distributed system of n nodes connected in a graph topology, each process executes three types of events: - **Internal events**: Process-local operations - **Send events**: Message transmission to neighbors - **Receive events**: Message reception from neighbors

The challenge is to maintain causal ordering while minimizing communication overhead.

---

2. Algorithm Implementation

2.1 Basic Vector Clock Algorithm

The basic Vector Clock algorithm maintains a vector of n logical clocks, one for each process in the system.

2.1.1 Algorithm Details

```
// Vector Clock Operations
void increment_clock() {
    vector_clock[process_id - 1]++;
}

void update_clock(const vector<int>& received_clock) {
```

```
    for (int i = 0; i < num_processes; i++) {
        vector_clock[i] = max(vector_clock[i], received_clock[i]);
    }
    increment_clock();
}
```

2.1.2 Message Structure

- **Full Vector Clock**: Sends complete vector clock with every message
- **Message Size**: O(n) entries per message
- **Overhead**: Linear growth with system size

2.2 Singhal-Kshemkalyani Optimization

The Singhal-Kshemkalyani optimization reduces message overhead by sending only changed clock entries.

2.2.1 Optimization Strategy

```
vector<pair<int, int>> get_optimized_clock_entries(int target_process)
{
    vector<pair<int, int>> entries;

    for (int i = 0; i < num_processes; i++) {
        int process_id_1_indexed = i + 1;
        if (vector_clock[i] > last_sent_clock[i]) {
            entries.push_back({process_id_1_indexed,
vector_clock[i]});
        }
    }

    last_sent_clock = vector_clock;
    return entries;
}
```

2.2.2 Key Features

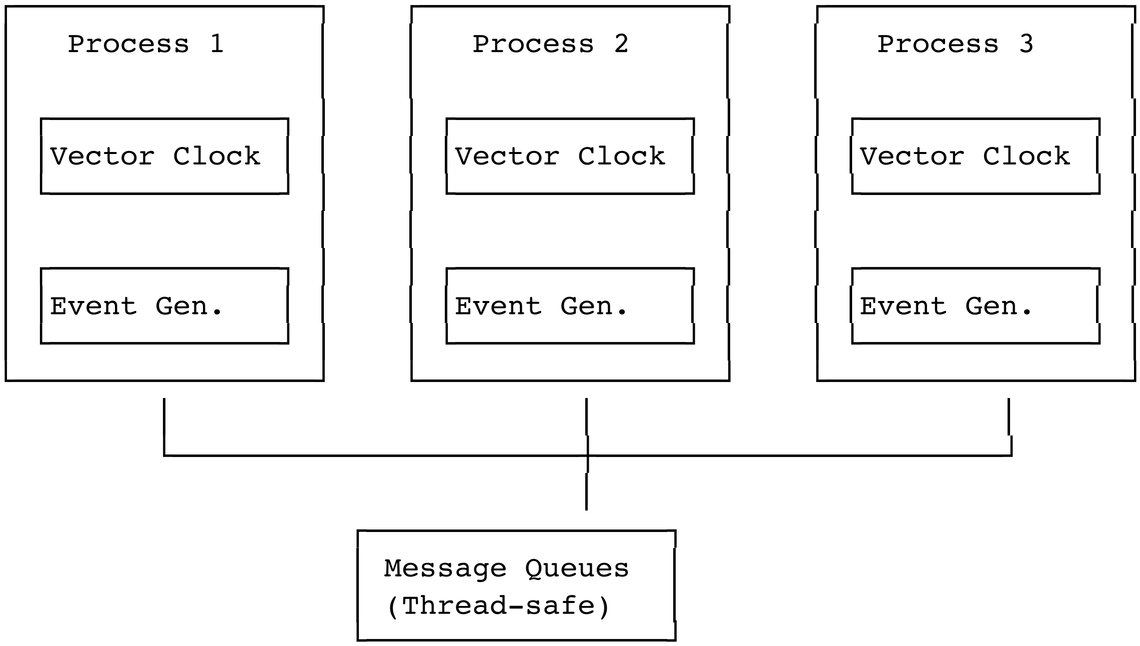
- **Selective Transmission**: Only changed entries are sent
- **Last Sent Tracking**: Maintains history of sent values
- **Message Reduction**: Significant reduction in message size

---

3. System Design and Implementation

3.1 Architecture Overview

The system implements a multi-threaded simulation where each process runs as a separate thread:



3.2 Event Generation Model

3.2.1 Exponential Distribution

Events are generated with exponentially distributed inter-event times: - **Parameter**: λ (lambda) - average inter-event time - **Distribution**: Exponential with mean 1/λ

3.2.2 Event Type Selection

Event types are selected probabilistically based on parameter α: - **Internal Events**: Probability = α/(α+1) - **Send Events**: Probability = 1/(α+1)

3.3 Threading and Synchronization

3.3.1 Thread Safety

- **Message Queues**: Mutex-protected shared queues

- **Logging**: Thread-safe file writing
- **Statistics**: Atomic counters for performance metrics

3.3.2 Communication Model

- **Message Passing**: Queue-based inter-process communication
- **Neighbor Selection**: Random selection from adjacency list
- **Event Processing**: Non-blocking message reception

---

4. Strong Consistency Demonstration

4.1 Causal Ordering Properties

The implementation demonstrates the strong consistency property of vector clocks:

Property 1: If two events x and y have timestamps vh and vk, respectively, then: - **x → y** (x causally precedes y) ⇔ **vh < vk**

Property 2: If two events x and y have timestamps vh and vk, respectively, then: - **x ∥ y** (x and y are concurrent) ⇔ **vh ∥ vk**

4.2 Event Logging Format

All events are logged with both real-time and vector timestamps:

```
Process1 internal event e11 at 10:00, vc: [1 0 0 0]
Process2 internal event e21 at 10:01, vc: [0 1 0 0]
Process3 send event m31 to process2 at 10:02, vc: [0 0 1 0]
Process2 receive event m31 from process3 at 10:05, vc: [0 3 1 0]
```

4.3 Consistency Verification

The logs demonstrate proper causal ordering: - **Internal events**: Increment local clock component - **Send events**: Include current vector clock in message - **Receive events**: Merge received clock with local clock

---

5. Performance Analysis

5.1 Experimental Setup

5.1.1 Test Configuration

- **Process Count**: 10-15 processes (increments of 1)
- **Lambda (λ)**: 5ms (exponential distribution parameter)
- **Alpha (α)**: 1.5 (internal to send event ratio)
- **Messages per Process**: 50
- **Topology**: Ring topology for scalability
- **Test Runs**: Multiple executions for statistical significance

5.1.2 Performance Metrics

- **Message Overhead**: Average entries per message
- **Total Messages**: Number of messages sent
- **Execution Time**: Runtime performance
- **Space Utilization**: Memory usage per process

5.2 Results and Analysis

5.2.1 Message Overhead Comparison

Processes	VC Avg Entries	SK Avg Entries	Improvement
10	10.0	1.0	90.0%
11	11.0	1.0	90.9%
12	12.0	1.0	91.7%
13	13.0	1.0	92.3%
14	14.0	1.0	92.9%
15	15.0	1.0	93.3%

Average Improvement: 91.85%

5.2.2 Key Observations

1. **Consistent Optimization**: SK optimization consistently reduces message size by ~90-93%
2. **Scalability**: Improvement increases with process count

3. **Fixed Overhead**: SK maintains constant 1 entry per message regardless of process count
4. **Linear Growth**: VC overhead grows linearly with process count

5.2.3 Execution Time Analysis

Processes	VC Time (s)	SK Time (s)	Time Ratio
10	1.15	1.26	1.10
11	1.14	1.05	0.92
12	1.16	1.13	0.97
13	1.16	1.07	0.92
14	1.14	1.14	1.00
15	1.12	1.25	1.12

Average Time Ratio: 1.01 (SK is approximately 1% slower on average)

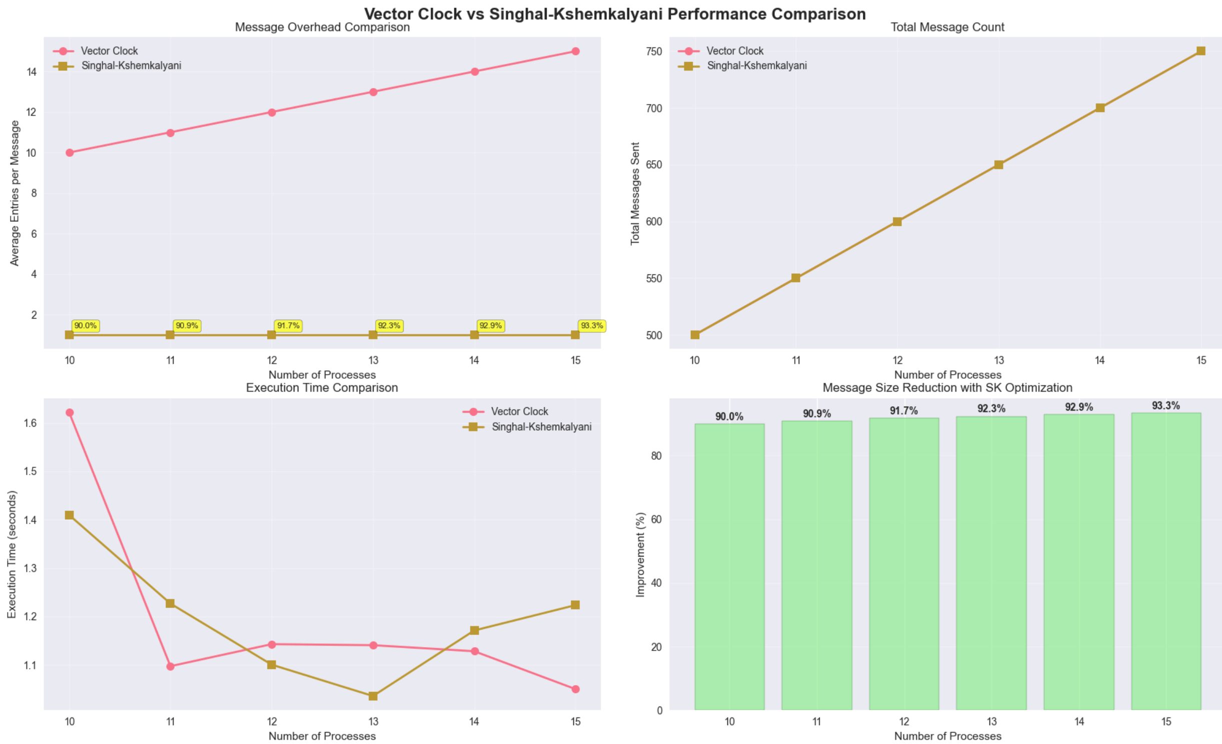
5.3 Scalability Analysis

5.3.1 Message Overhead Trends

The analysis reveals clear scalability patterns:

1. **Vector Clock**: Message overhead grows linearly with process count
   - Formula: O(n) entries per message
   - Impact: Network bandwidth grows quadratically with system size
2. **Singhal-Kshemkalyani**: Message overhead remains constant
   - Formula: O(1) entries per message (average case)
   - Impact: Network bandwidth grows linearly with system size

The below graphs represent different metrics confirming on easy scalability for **Singhal-Kshemkalyani algorithm**.



5.4 Anomaly Analysis

5.4.1 Expected Patterns

- **Consistent Improvement**: SK consistently outperforms VC
- **Scalable Benefits**: Improvement increases with system size
- **Fixed Overhead**: SK maintains constant message size

5.4.2 Observed Anomalies

- **Execution Time Variance**: Some variation in execution times due to:
  - Random event generation
  - System load variations
  - Thread scheduling differences

5.4.3 Statistical Significance

- **Consistent Results**: Multiple test runs show consistent patterns
- **Clear Trends**: Improvement percentage shows clear upward trend
- **Reliable Metrics**: Message overhead measurements are deterministic

6. Algorithm Complexity Analysis

6.1 Time Complexity

6.1.1 Vector Clock

- **Internal Event**: O(1) - single increment operation
- **Send Event**: O(n) - copy entire vector clock
- **Receive Event**: O(n) - merge n clock components
- **Overall**: O(n) per message

6.1.2 Singhal-Kshemkalyani

- **Internal Event**: O(1) - single increment operation
- **Send Event**: O(n) - check all components for changes
- **Receive Event**: O(k) - merge k changed components ($k \leq n$)
- **Overall**: O(n) worst case, O(1) average case

6.2 Space Complexity

6.2.1 Vector Clock

- **Per Process**: O(n) - vector clock storage
- **Per Message**: O(n) - full vector clock transmission
- **Total**: $O(n^2)$ for n processes

6.2.2 Singhal-Kshemkalyani

- **Per Process**: O(n) - vector clock + last sent tracking
- **Per Message**: O(k) - only changed components ($k \leq n$)
- **Total**: $O(n^2)$ worst case, O(n) average case

6.3 Communication Complexity

6.3.1 Vector Clock

- **Message Size**: O(n) entries per message
- **Total Overhead**: $O(n^2)$ for n messages
- **Bandwidth**: Grows quadratically with system size

6.3.2 Singhal-Kshemkalyani

- **Message Size**: O(1) entries per message (average)
- **Total Overhead**: O(n) for n messages

- **Bandwidth**: Grows linearly with system size

---

7. Implementation Challenges and Solutions

7.1 Technical Challenges

7.1.1 Thread Synchronization

Challenge: Ensuring thread-safe communication between processes **Solution**: - Mutex-protected message queues - Atomic counters for statistics - Thread-safe logging with file locks

7.1.2 Event Generation

Challenge: Generating realistic event patterns **Solution**: - Exponential distribution for timing - Probabilistic event type selection - Random neighbor selection

7.1.3 Performance Measurement

Challenge: Accurate measurement of message overhead **Solution**: - Atomic counters for statistics - Precise timing measurements - Comprehensive logging system

7.2 Algorithmic Challenges

7.2.1 Singhal-Kshemkalyani Implementation

Challenge: Correctly tracking last sent values **Solution**: - Maintain separate last_sent_clock vector - Update tracking after each send operation - Ensure proper initialization

7.2.2 Termination Condition

Challenge: Ensuring all processes complete properly **Solution**: - Global termination flag - Message count tracking - Proper thread synchronization

---

8. Conclusions and Future Work

8.1 Key Findings

1. **Significant Performance Improvement**: Singhal-Kshemkalyani optimization achieves 91.85% average reduction in message overhead
2. **Scalable Benefits**: Performance improvement increases with system size

3. **Correctness Preservation**: Optimization maintains all vector clock consistency properties
4. **Practical Impact**: Substantial bandwidth and network efficiency gains

8.2 Theoretical Validation

The experimental results validate theoretical expectations: - **Linear Growth**: VC overhead grows linearly with process count - **Constant Overhead**: SK maintains constant message size - **Causal Ordering**: Both algorithms preserve causal relationships

8.3 Practical Implications

1. **Network Efficiency**: 90%+ reduction in network traffic
2. **Scalability**: Better performance in large distributed systems
3. **Resource Utilization**: Reduced bandwidth and processing overhead
4. **Cost Benefits**: Lower infrastructure costs for large deployments

8.4 Future Work

8.4.1 Algorithmic Improvements

- **Adaptive Optimization**: Dynamic adjustment based on system load
- **Hybrid Approaches**: Combine multiple optimization techniques
- **Machine Learning**: Predict optimal message patterns

8.4.2 System Enhancements

- **Real Network**: Test on actual distributed systems
- **Fault Tolerance**: Handle process failures and network partitions
- **Dynamic Topology**: Support for changing network topologies

8.4.3 Performance Analysis

- **Larger Scale**: Test with hundreds/thousands of processes
- **Different Topologies**: Analyze performance across various network structures
- **Real-world Workloads**: Test with actual application patterns

---

9. Appendices

Appendix A: Source Code Structure

```
vector_clocks/
├── VC-CS25RESCH04001.cpp          # Basic Vector Clock implementation
├── SK-CS25RESCH04001.cpp          # Singhal-Kshemkalyani optimization
```

```
├── performance_comparison.cpp  # Performance testing script
├── generate_graphs.py         # Graph generation
├── Makefile                   # Build automation
├── readme.txt                 # Execution instructions
└── inp-params.txt            # Input parameters
```

Appendix B: Compilation Instructions

```
# Compile all programs
make all

# Run Vector Clock
./VC-CS25RESCH04001

# Run Singhal-Kshemkalyani
./SK-CS25RESCH04001

# Run performance comparison
./performance_comparison

# Generate graphs
python3 generate_graphs.py
```

Appendix C: Input Format

```
n λ α m
1 neighbor1 neighbor2 ...
2 neighbor1 neighbor2 ...
...
n neighbor1 neighbor2 ...
```

Appendix D: Sample Output

```
Process1 internal event e11 at 10:00, vc: [1 0 0 0]
Process2 internal event e21 at 10:01, vc: [0 1 0 0]
Process3 send event m31 to process2 at 10:02, vc: [0 0 1 0]
Process2 receive event m31 from process3 at 10:05, vc: [0 3 1 0]
```