REPLICATED DATABASE SYSTEM - DETAILED FUNCTIONALITY DOCUMENT

PAGE 1: CORE SYSTEM AND PRIMARY OPERATIONS

1. SYSTEM ARCHITECTURE
The system consists of 10 database sites managing 20 shared variables with the following key properties:
- Sites operate independently, can fail/recover separately
- Variables x1 through x20 are maintained across sites
- Even-numbered variables are replicated on all sites
- Odd-numbered variables exist on one site (site = i mod 10)
- Each variable xi has initial value 10i

2. CORE COMPONENTS AND PRIMARY FUNCTIONALITY

A. Transaction Manager (Main Controller)
Core Functions:
- Transaction Lifecycle:
  * Begins transactions with unique IDs and timestamps
  * Executes reads/writes while maintaining consistency
  * Manages commit/abort decisions
  * Maintains transaction state and history

- Concurrency Control:
  * Implements First-Committer-Wins protocol
  * Detects and prevents cycles in dependency graph
  * Manages read/write sets for conflict detection
  * Coordinates between multiple sites

- Operation Handling:
  * Routes reads to available sites
  * Manages write propagation
  * Handles site failures during operations
  * Queues operations when sites are unavailable

B. Site Manager (Distribution Controller)
Primary Functions:
- Site Status:
  * Tracks up/down status of each site
  * Manages site failures and recoveries
  * Updates accessibility maps
  * Controls data availability

- Data Distribution:

  * Maps variables to appropriate sites
  * Handles replication rules
  * Manages data consistency across sites
  * Controls access during recovery

C. Data Manager (Storage Controller)
Key Functions:
- Data Storage:
  * Maintains versioned data copies
  * Manages commit logs
  * Provides consistent snapshots
  * Tracks read/write operations

- Version Control:
  * Maintains timestamp-based versions
  * Ensures consistent reads
  * Manages commit history
  * Handles recovery data

3. OPERATION PROCESSING

A. Read Operations (R(Ti,xj)):
1. Validation Steps:
   - Check transaction status (not aborted)
   - Verify site availability
   - Ensure data accessibility
   - Validate read permission

2. Execution Process:
   - Select appropriate site
   - Retrieve consistent version
   - Update read sets
   - Record operation history
   - Return value to transaction

B. Write Operations (W(Ti,xj,v)):
1. Pre-write Checks:
   - Validate transaction state
   - Check site availability
   - Verify write permissions
   - Prepare replication targets

2. Write Process:
   - Buffer write in transaction

- Update write sets
- Track site dependencies
- Prepare for commit
- Handle replication rules

C. Commit Processing (end(Ti)):
1. Validation Phase:
- Check First-Committer-Wins rule
- Verify site availability
- Detect dependency cycles
- Validate read/write sets

2. Execution Phase:
- Apply buffered writes
- Update commit logs
- Modify site data
- Release resources
- Update transaction status

4. FAILURE AND RECOVERY MANAGEMENT

A. Site Failure Handling (fail(site)):
1. Immediate Actions:
- Mark site as down
- Block access to variables
- Queue pending operations
- Update accessibility maps

2. Transaction Impact:
- Abort affected transactions
- Reassign operations if possible
- Update dependency tracking
- Maintain consistency guarantees

B. Site Recovery Process (recover(site)):
1. Recovery Steps:
- Mark site as available
- Process queued operations
- Update site accessibility
- Restore data access

2. Data Handling:

- Non-replicated: Immediate access
- Replicated: Block reads until write
- Process pending operations
- Restore consistency

5. CONCURRENCY CONTROL IMPLEMENTATION

A. First-Committer-Wins Protocol:
Implementation Details:
- Tracks transaction timestamps
- Maintains write sets
- Compares commit times
- Handles conflicts
- Ensures serializability
- Manages dependencies

B. Dependency Graph Management:
Operation:
- Tracks read/write dependencies
- Maintains transaction ordering
- Detects potential cycles
- Prevents deadlocks
- Manages concurrent access
- Ensures consistency

6. SPECIFIC ALGORITHMS

A. Cycle Detection Algorithm:
Implementation:
- Uses depth-first search
- Maintains visited nodes
- Tracks recursion stack
- Identifies cycles
- Triggers appropriate aborts
- Ensures serializability

B. Site Selection Process:
Steps:
- Evaluates variable location
- Checks site availability
- Considers replication
- Handles failures
- Manages recovery
- Ensures accessibility

7. ERROR SCENARIOS AND HANDLING

A. Common Error Cases:
1. All Sites Down:
   - Queue operations
   - Maintain consistency
   - Handle recovery
   - Resume processing

2. Partial Failures:
   - Continue operations
   - Manage accessibility
   - Handle replication
   - Maintain consistency

3. Recovery Conflicts:
   - Resolve inconsistencies
   - Process queued operations
   - Restore accessibility
   - Update status

B. Transaction Failures:
Handling:
- Clean state restoration
- Resource release
- Dependency updates
- Queue management
- Recovery processing

8. COMMAND PROCESSING AND OUTPUT

A. Input Commands:
Processing:
- begin(Ti): Start transaction
- R(Ti,xj): Read variable
- W(Ti,xj,val): Write variable
- end(Ti): End transaction
- fail(site): Site failure
- recover(site): Site recovery
- dump(): System status

B. Output Generation:
Types:

- Transaction status
- Read values
- Commit/abort messages
- Site status updates
- Error notifications
- System state dumps

9. PERFORMANCE CONSIDERATIONS

Key Aspects:
- Minimize blocking
- Optimize site selection
- Efficient recovery
- Quick conflict detection
- Smart queuing
- Effective replication

10. CONSISTENCY GUARANTEES

Ensures:
- Serializable execution
- Consistent reads
- Atomic writes
- Failure atomicity
- Recovery consistency
- Data durability

Classes Information :-

A. Transaction Class :-

```
class Transaction {
int transaction_id;          // Unique identifier
int start_timestamp;         // Transaction start time
int commit_timestamp;        // Commit time if successful
vector<operation> operations; // List of operations
bool aborted;                // Transaction status
vector<int> data_vals;       // Written values (size 20, INT_MAX = no write)
vector<int> read_vars;       // Variables read by transaction

// Methods
void addOperation(operation op);
```

```
    bool canCommit();
    void abort();
    void commit(int timestamp);
}
```

B. Operation Class :-

```
class operation {
    int transaction_id;    // Transaction performing operation
    int timestamp;         // Operation timestamp
    char op_type;          // R/W/B/E/F/r (read/write/begin/end/fail/recover)
    int variable_id;       // Target variable (0-19 for x1-x20)
    int site_id;           // Target site (0-9)
    int val;               // Value for writes
    set<int> wait_site_ids; // Sites transaction is waiting for
}
```

C. Transaction Manager :-

```
class TransactionManager {
    // State
    vector<operation> all_operations;
    unordered_map<int, Transaction> transactions;
    vector<set<int>> rw_graph;
    int timestamp;
    set<int> committed_nodes;
    vector<operation> wait_operations;

    // Core Operations
    void beginTransaction(int transaction_id, int timestamp);
    int executeRead(int timestamp_no, int transaction_id, int var_id);
    void executeWrite(int timestamp_no, int transaction_id, int var_id, int val);
    void executeEnd(int timestamp_no, int transaction_id);

    // Validation & Control
    bool can_commit(int transaction_id);
    bool dfs();  // Cycle detection
    bool iscycle(int i, vector<int>&visited, vector<int>&recstack);

    // Dependency Management
    void updateDependencyGraph(Transaction* trans);
    void cleanupTransactionDependencies(int trans_id);
}
```

D. Site Manager :-

```
class SiteManager {
   // State
   vector<bool> site_status;              // Up/down status
   vector<vector<int>> site_data;         // Current values
   vector<vector<int>> site_var_map;      // Variable location
   vector<vector<int>> var_site_map;      // Site mapping
   vector<vector<bool>> can_read;         // Read accessibility
   vector<vector<int>> trans_site_write;  // Transaction writes

   // Core Operations
   void failsite(int siteno);
   void recover_site(int site_id);
   bool checksite(int siteno);

   // Data Access
   int getsite(int variable_id, int transaction_id);
   void update_all_sites(int var_id, int val, int transaction_id);
   void write_all_sites(int var_id, int val);

   // Initialization
   void initialize();
}
```

E. Data Manager :-

```
class DataManager {
   // State
   map<int, vector<vector<int>>> main_data;     // Timestamped data
   vector<operation> commit_logs;               // Operation history
   vector<int> committed_timestamps;            // Commit times
   vector<set<int>> read_trans, write_trans;    // Transaction sets

   // Core Operations
   int readval(int timestamp, int site_id, int var_id);
   void append_commit(vector<operation>& ops, int timestamp,
                      vector<vector<int>>& current_data);

   // Version Management
   void cleanup_old_versions();
   void maintain_versions();
```

```
}
```