**Team 6: Distributed Multi-Agent System**

**Technical Specification Document**

**Table of Contents**

1. [Project Overview](https://claude.ai/chat/6de0fb32-38a2-40c0-b003-470028f959a0#1-project-overview)
2. [System Architecture](https://claude.ai/chat/6de0fb32-38a2-40c0-b003-470028f959a0#2-system-architecture)
3. [Core Data Structures](https://claude.ai/chat/6de0fb32-38a2-40c0-b003-470028f959a0#3-core-data-structures)
4. [Component Specifications](https://claude.ai/chat/6de0fb32-38a2-40c0-b003-470028f959a0#4-component-specifications)
5. [Integration Interfaces](https://claude.ai/chat/6de0fb32-38a2-40c0-b003-470028f959a0#5-integration-interfaces)
6. [Development Timeline](https://claude.ai/chat/6de0fb32-38a2-40c0-b003-470028f959a0#6-development-timeline)
7. [Testing Strategy](https://claude.ai/chat/6de0fb32-38a2-40c0-b003-470028f959a0#7-testing-strategy)
8. [Code Examples](https://claude.ai/chat/6de0fb32-38a2-40c0-b003-470028f959a0#8-code-examples)

**1. Project Overview**

**1.1 Mission Statement**

We are building a **Distributed Multi-Agent System** where autonomous drones/robots communicate peer-to-peer to accomplish complex missions through emergent collective intelligence, inspired by bee colonies.

**1.2 Core Objectives**

* **Collective Decision Making:** Agents vote and reach consensus without central control
* **Distributed Communication:** Peer-to-peer messaging with realistic range limitations
* **Emergent Behavior:** Complex coordination from simple local rules
* **Mission Execution:** User assigns high-level goals, swarm coordinates execution

**1.3 Key Deliverables**

* **5-10 agent swarm** with coordinated movement
* **Real-time visualization** with user control interface
* **Voting system** for collective decision making
* **Mission planning** interface with waypoint assignment
* **Fault tolerance** - system continues with agent failures

**2. System Architecture**

**2.1 Component Overview**

┌─────────────────────────────────────────────────────────────────┐

│ DISTRIBUTED MULTI-AGENT SYSTEM │

├─────────────────────────────────────────────────────────────────┤

│ │

│ ┌─────────────────┐ ┌─────────────────┐ │

│ │ Anthony │◄──►│ Lauren │ │

│ │ User Interface │ │ Swarm Intelligence│ │

│ │ & Integration │ │ & Algorithms │ │

│ │ │ │ │ │

│ │ • Ground Control│ │ • Flocking Logic│ │

│ │ • Visualization │ │ • Decision Making│ │

│ │ • System Testing│ │ • Task Assignment│ │

│ └─────────────────┘ └─────────────────┘ │

│ ▲ ▲ │

│ │ │ │

│ │ │ │

│ ▼ ▼ │

│ ┌─────────────────┐ ┌─────────────────┐ │

│ │ Sanidhaya │◄──►│ John │ │

│ │ Core Agent │ │ Communication │ │

│ │ System │ │ System │ │

│ │ │ │ │ │

│ │ • Agent Physics │ │ • Message Protocol│ │

│ │ • Performance │ │ • Network Sim │ │

│ │ • Integration │ │ • Range Limits │ │

│ └─────────────────┘ └─────────────────┘ │

│ │

└─────────────────────────────────────────────────────────────────┘

**2.2 Data Flow**

The system operates in a continuous cycle:

1. **Sanidhaya** updates agent positions and status
2. **John** determines who can communicate with whom
3. **Lauren** makes decisions based on available information
4. **Anthony** displays everything and handles user input

**2.3 Real-Time Operation Loop**

Agent State Updates → Communication Events → Decision Making → Movement Commands → [repeat]

↑ ↓

User Interface ← Visualization Data ← System Status ← Performance Metrics

**3. Core Data Structures**

**3.1 Fundamental Types**

// Basic geometric types used throughout system

public class Point2D {

public double x, y;

public Point2D(double x, double y) { this.x = x; this.y = y; }

}

public class Vector2D {

public double x, y;

public Vector2D(double x, double y) { this.x = x; this.y = y; }

public double magnitude() { return Math.sqrt(x\*x + y\*y); }

public Vector2D normalize() {

double mag = magnitude();

return new Vector2D(x/mag, y/mag);

}

}

**3.2 Agent State (Central Data Structure)**

public class AgentState {

// Identity

public int agentId;

public String agentName;

// Physical state

public Point2D position;

public Vector2D velocity;

public double heading; // radians

// Capabilities

public double maxSpeed;

public double maxTurnRate;

public double communicationRange;

// Status

public AgentStatus status;

public double batteryLevel; // 0.0 to 1.0

public long lastUpdateTime;

// Current assignment

public Task currentTask;

public int teamId; // for formations

}

public enum AgentStatus {

ACTIVE, INACTIVE, FAILED, BATTERY\_LOW, MAINTENANCE

}

**3.3 Message System**

public class Message {

public String messageId;

public MessageType type;

public Object payload;

public long timestamp;

public Map<String, Object> metadata;

}

public enum MessageType {

POSITION\_UPDATE, VOTE\_PROPOSAL, VOTE\_RESPONSE,

TASK\_ASSIGNMENT, FORMATION\_COMMAND, EMERGENCY\_ALERT

}

**4. Component Specifications**

**4.1 Sanidhaya - Core Agent System**

**Primary Responsibilities:**

* **Agent Lifecycle:** Create, update, and destroy agents
* **Physics Simulation:** Calculate movement, collisions, boundaries
* **Performance Management:** Ensure system runs smoothly with 10+ agents
* **System Integration:** Coordinate between all components

**Key Classes to Implement:**

public class Agent {

private AgentState state;

private PhysicsEngine physics;

public void update(double deltaTime) { /\* movement calculations \*/ }

public AgentState getState() { return state; }

public void applyMovementCommand(MovementCommand cmd) { /\* execute command \*/ }

}

public class AgentManager {

private Map<Integer, Agent> agents;

public Agent createAgent(Point2D position) { /\* implementation \*/ }

public void removeAgent(int agentId) { /\* implementation \*/ }

public List<AgentState> getAllAgentStates() { /\* implementation \*/ }

}

public class PhysicsEngine {

public void updatePosition(Agent agent, double deltaTime) { /\* physics \*/ }

public boolean checkBoundaries(Point2D position) { /\* boundary checking \*/ }

public List<Agent> detectCollisions(Agent agent) { /\* collision detection \*/ }

}

**What You Receive:**

* SystemCommand from Anthony (spawn agents, configure system)
* MovementCommand from Lauren (flocking behaviors, formations)
* CommunicationEvent from John (message deliveries)

**What You Provide:**

* AgentStateUpdate to John (position changes for communication range)
* AgentCapabilities to Lauren (what each agent can do)
* VisualizationUpdate to Anthony (all data for display)

**4.2 John - Communication System**

**Primary Responsibilities:**

* **Message Protocol:** Design how agents send/receive messages
* **Range Simulation:** Only nearby agents can communicate
* **Network Topology:** Track who can talk to whom
* **Message Routing:** Deliver messages between agents

**Key Classes to Implement:**

public class CommunicationManager {

private Map<Integer, List<Integer>> networkTopology;

private Queue<OutgoingMessage> messageQueue;

public void sendMessage(OutgoingMessage message) { /\* implementation \*/ }

public List<Integer> getNeighbors(int agentId) { /\* implementation \*/ }

public void updateTopology(List<AgentState> agents) { /\* implementation \*/ }

}

public class NetworkSimulator {

public boolean canCommunicate(Point2D pos1, Point2D pos2, double range) {

double distance = calculateDistance(pos1, pos2);

return distance <= range;

}

public double calculateSignalStrength(double distance, double maxRange) {

return Math.max(0, 1.0 - (distance / maxRange));

}

}

public class MessageRouter {

public boolean routeMessage(int senderId, int receiverId, Message message) {

/\* find path and deliver message \*/

}

}

**What You Receive:**

* AgentStateUpdate from Sanidhaya (positions for range calculations)
* OutgoingMessage from Lauren (votes, decisions to broadcast)
* NetworkConfiguration from Anthony (range settings, failure rates)

**What You Provide:**

* CommunicationEvent to Sanidhaya (message delivery notifications)
* IncomingMessage to Lauren (received votes and neighbor info)
* NetworkStatus to Anthony (connection info for visualization)

**4.3 Lauren - Swarm Intelligence & Algorithms**

**Primary Responsibilities:**

* **Flocking Behavior:** Separation, alignment, cohesion algorithms
* **Collective Decision Making:** Voting and consensus systems
* **Task Assignment:** Distribute work among agents
* **Formation Control:** Coordinate movement patterns

**Key Classes to Implement:**

public class FlockingController {

public MovementCommand calculateFlocking(int agentId, List<NeighborAgent> neighbors) {

Vector2D separation = calculateSeparation(neighbors);

Vector2D alignment = calculateAlignment(neighbors);

Vector2D cohesion = calculateCohesion(neighbors);

// Combine forces and create movement command

Vector2D totalForce = combine(separation, alignment, cohesion);

return new MovementCommand(agentId, totalForce);

}

}

public class VotingSystem {

private Map<String, VoteProposal> activeVotes;

public void initiateVote(VoteProposal proposal) { /\* implementation \*/ }

public void processVote(String proposalId, int agentId, String choice) { /\* implementation \*/ }

public VoteResult checkConsensus(String proposalId) { /\* implementation \*/ }

}

public class TaskAllocator {

public TaskAssignment assignTask(Task task, List<AgentState> availableAgents) {

// Find best agent for task based on position, capabilities, current load

int bestAgentId = findBestAgent(task, availableAgents);

return new TaskAssignment(task, bestAgentId);

}

}

**What You Receive:**

* IncomingMessage from John (votes, neighbor information)
* AgentCapabilities from Sanidhaya (what agents can do)
* BehaviorConfiguration from Anthony (algorithm parameters)

**What You Provide:**

* MovementCommand to Sanidhaya (flocking forces, formation positions)
* OutgoingMessage to John (voting proposals, task assignments)
* DecisionStatus to Anthony (current decisions, mission progress)

**4.4 Anthony - User Interface & Integration**

**Primary Responsibilities:**

* **Ground Control Interface:** Mission planning and monitoring
* **Real-Time Visualization:** Display agents, communications, decisions
* **System Integration:** Ensure all components work together
* **User Experience:** Intuitive controls and clear information display

**Key Classes to Implement:**

public class MainInterface extends Application {

private Visualizer visualizer;

private ControlPanel controlPanel;

private SystemController systemController;

@Override

public void start(Stage primaryStage) {

// Setup UI components and event handlers

}

}

public class Visualizer {

public void updateAgentDisplay(List<AgentState> agents) {

// Draw agents as circles/shapes on 2D canvas

}

public void drawCommunicationLinks(List<ConnectionInfo> connections) {

// Show lines between communicating agents

}

public void showDecisionProcess(DecisionStatus decision) {

// Visualize voting process and results

}

}

public class ControlPanel {

public void handleWaypointPlacement(Point2D position) {

// User clicked to place waypoint

SystemCommand cmd = new SystemCommand(CommandType.PLACE\_WAYPOINT, position);

systemController.executeCommand(cmd);

}

public void handleParameterChange(String parameter, Object value) {

// User adjusted algorithm parameters

}

}

**What You Receive:**

* VisualizationUpdate from Sanidhaya (agent positions and status)
* NetworkStatus from John (communication links and messages)
* DecisionStatus from Lauren (voting progress, mission status)

**What You Provide:**

* SystemCommand to Sanidhaya (user commands and configuration)
* NetworkConfiguration to John (communication settings)
* BehaviorConfiguration to Lauren (algorithm parameters)

**5. Integration Interfaces**

**5.1 Message Formats**

**SystemCommand (Anthony → Sanidhaya)**

public class SystemCommand {

public CommandType type;

public Map<String, Object> parameters;

public long timestamp;

}

public enum CommandType {

SPAWN\_AGENT, REMOVE\_AGENT, SET\_BOUNDARIES,

START\_SIMULATION, STOP\_SIMULATION, PLACE\_WAYPOINT

}

// Example usage:

SystemCommand spawnCmd = new SystemCommand();

spawnCmd.type = CommandType.SPAWN\_AGENT;

spawnCmd.parameters.put("position", new Point2D(100, 100));

spawnCmd.parameters.put("maxSpeed", 50.0);

**MovementCommand (Lauren → Sanidhaya)**

public class MovementCommand {

public int agentId;

public MovementType type;

public Map<String, Object> parameters;

public CommandPriority priority;

}

public enum MovementType {

FLOCKING\_BEHAVIOR, FORMATION\_POSITION,

MOVE\_TO\_TARGET, AVOID\_OBSTACLE

}

// Example usage:

MovementCommand flockCmd = new MovementCommand();

flockCmd.agentId = 5;

flockCmd.type = MovementType.FLOCKING\_BEHAVIOR;

flockCmd.parameters.put("separationWeight", 1.5);

flockCmd.parameters.put("alignmentWeight", 1.0);

flockCmd.parameters.put("cohesionWeight", 1.0);

**OutgoingMessage (Lauren → John)**

public class OutgoingMessage {

public int senderId;

public int receiverId; // -1 for broadcast

public Message messageContent;

public MessagePriority priority;

}

// Example voting message:

OutgoingMessage voteMsg = new OutgoingMessage();

voteMsg.senderId = 3;

voteMsg.receiverId = -1; // broadcast to all

voteMsg.messageContent = new Message();

voteMsg.messageContent.type = MessageType.VOTE\_PROPOSAL;

voteMsg.messageContent.payload = new VoteProposal("Go left or right?",

Arrays.asList("LEFT", "RIGHT"));

**5.2 Event-Driven Architecture**

**Event System**

public class SystemEvent {

public EventType type;

public String sourceComponent;

public Object eventData;

public long timestamp;

}

public interface EventListener {

void onEvent(SystemEvent event);

}

public class EventBus {

private List<EventListener> listeners = new ArrayList<>();

public void registerListener(EventListener listener) {

listeners.add(listener);

}

public void publishEvent(SystemEvent event) {

for (EventListener listener : listeners) {

listener.onEvent(event);

}

}

}

**6. Development Timeline**

**6.1 Phase 1: Foundation (Weeks 1-4)**

**Week 1: Project Setup**

* **All:** Set up development environment, Git repository
* **Sanidhaya:** Basic Agent class with position and movement
* **John:** Message and communication framework
* **Lauren:** Basic flocking algorithm research and design
* **Anthony:** Basic JavaFX window with agent visualization

**Week 2: Basic Integration**

* **Integration Focus:** Sanidhaya + John
* **Goal:** Agents can share position information
* **Test:** 3 agents sending location updates through communication system

**Week 3: Movement Coordination**

* **Integration Focus:** + Lauren
* **Goal:** Communication enables basic flocking behavior
* **Test:** Agents adjust movement based on neighbor positions

**Week 4: First Full System**

* **Integration Focus:** + Anthony
* **Goal:** Complete system with user interface
* **Test:** User can spawn agents and see coordinated movement

**6.2 Phase 2: Intelligence (Weeks 5-8)**

**Week 5: Decision Making**

* **Focus:** Voting and consensus systems
* **Integration:** John + Lauren (voting messages)
* **Test:** Agents vote on simple decisions

**Week 6: Mission Interface**

* **Focus:** User mission planning
* **Integration:** Anthony + Lauren (mission assignment)
* **Test:** User places waypoints, agents coordinate to reach them

**Week 7: Formation Flying**

* **Focus:** Coordinated movement patterns
* **Integration:** Lauren + Sanidhaya (formation commands)
* **Test:** Agents form lines, circles, and maintain formations

**Week 8: Major Integration**

* **Focus:** Complete mission execution capability
* **Integration:** All components
* **Test:** Complex missions with voting and formation changes

**6.3 Phase 3: Enhancement (Weeks 9-12)**

**Week 9: Fault Tolerance**

* **Focus:** Handling agent failures and network partitions
* **Test:** System continues operating when agents fail

**Week 10: Performance Optimization**

* **Focus:** System handles 10+ agents smoothly
* **Test:** Stress testing with maximum agent count

**Week 11: Advanced Features**

* **Focus:** Polish and additional capabilities
* **Test:** Demo scenarios and edge cases

**Week 12: Final Integration & Demo**

* **Focus:** Demo preparation and final testing
* **Test:** Professional demonstration scenarios

**7. Testing Strategy**

**7.1 Unit Testing**

Each component should have comprehensive unit tests:

// Example unit test for Sanidhaya's component

@Test

public void testAgentMovement() {

Agent agent = new Agent(new Point2D(0, 0));

MovementCommand cmd = new MovementCommand();

cmd.type = MovementType.MOVE\_TO\_TARGET;

cmd.parameters.put("target", new Point2D(100, 100));

agent.applyMovementCommand(cmd);

agent.update(1.0); // 1 second update

// Verify agent moved toward target

assertTrue(agent.getState().position.x > 0);

assertTrue(agent.getState().position.y > 0);

}

**7.2 Integration Testing**

Test component interactions:

@Test

public void testCommunicationIntegration() {

// Setup: Create agents with Sanidhaya's system

AgentManager agentManager = new AgentManager();

Agent agent1 = agentManager.createAgent(new Point2D(0, 0));

Agent agent2 = agentManager.createAgent(new Point2D(50, 0));

// Setup: Create communication system (John's component)

CommunicationManager commManager = new CommunicationManager();

commManager.updateTopology(agentManager.getAllAgentStates());

// Test: Send message

OutgoingMessage msg = new OutgoingMessage();

msg.senderId = agent1.getState().agentId;

msg.receiverId = agent2.getState().agentId;

msg.messageContent = new Message();

boolean delivered = commManager.sendMessage(msg);

assertTrue(delivered);

}

**7.3 System Testing**

Test complete scenarios:

@Test

public void testCompleteVotingScenario() {

// 1. Spawn multiple agents

// 2. Create obstacle in path

// 3. Initiate voting on navigation

// 4. Verify consensus reached

// 5. Verify coordinated movement

// 6. Check UI displays process

}

**8. Code Examples**

**8.1 Basic Flocking Implementation**

public class FlockingController {

private static final double SEPARATION\_RADIUS = 30.0;

private static final double ALIGNMENT\_RADIUS = 50.0;

private static final double COHESION\_RADIUS = 80.0;

public MovementCommand calculateFlocking(int agentId, List<NeighborAgent> neighbors) {

Vector2D separation = calculateSeparation(neighbors);

Vector2D alignment = calculateAlignment(neighbors);

Vector2D cohesion = calculateCohesion(neighbors);

// Weight the behaviors

separation = multiply(separation, 1.5);

alignment = multiply(alignment, 1.0);

cohesion = multiply(cohesion, 1.0);

// Combine into final force

Vector2D totalForce = add(add(separation, alignment), cohesion);

MovementCommand cmd = new MovementCommand();

cmd.agentId = agentId;

cmd.type = MovementType.FLOCKING\_BEHAVIOR;

cmd.parameters.put("force", totalForce);

return cmd;

}

private Vector2D calculateSeparation(List<NeighborAgent> neighbors) {

Vector2D steer = new Vector2D(0, 0);

int count = 0;

for (NeighborAgent neighbor : neighbors) {

if (neighbor.distance < SEPARATION\_RADIUS) {

// Calculate vector pointing away from neighbor

Vector2D diff = subtract(getCurrentPosition(), neighbor.position);

diff = normalize(diff);

diff = divide(diff, neighbor.distance); // Weight by distance

steer = add(steer, diff);

count++;

}

}

if (count > 0) {

steer = divide(steer, count); // Average

steer = normalize(steer);

}

return steer;

}

}

**8.2 Basic Voting System**

public class VotingSystem {

private Map<String, VoteProposal> activeVotes = new HashMap<>();

private Map<String, Map<Integer, String>> voteResponses = new HashMap<>();

public void initiateVote(VoteProposal proposal) {

activeVotes.put(proposal.proposalId, proposal);

voteResponses.put(proposal.proposalId, new HashMap<>());

// Broadcast vote proposal through communication system

OutgoingMessage voteMsg = new OutgoingMessage();

voteMsg.senderId = -1; // System message

voteMsg.receiverId = -1; // Broadcast

voteMsg.messageContent = new Message();

voteMsg.messageContent.type = MessageType.VOTE\_PROPOSAL;

voteMsg.messageContent.payload = proposal;

communicationManager.sendMessage(voteMsg);

}

public void processVote(String proposalId, int agentId, String choice) {

if (activeVotes.containsKey(proposalId)) {

voteResponses.get(proposalId).put(agentId, choice);

// Check if we have consensus

VoteResult result = checkConsensus(proposalId);

if (result.consensusReached) {

executeVoteResult(result);

}

}

}

public VoteResult checkConsensus(String proposalId) {

VoteProposal proposal = activeVotes.get(proposalId);

Map<Integer, String> votes = voteResponses.get(proposalId);

// Count votes for each option

Map<String, Integer> voteCounts = new HashMap<>();

for (String choice : votes.values()) {

voteCounts.put(choice, voteCounts.getOrDefault(choice, 0) + 1);

}

// Find majority

int totalVotes = votes.size();

int requiredVotes = (int) Math.ceil(totalVotes \* 0.6); // 60% consensus

for (Map.Entry<String, Integer> entry : voteCounts.entrySet()) {

if (entry.getValue() >= requiredVotes) {

return new VoteResult(proposalId, true, entry.getKey());

}

}

return new VoteResult(proposalId, false, null);

}

}

**8.3 Basic Agent Update Loop**

public class Agent {

private AgentState state;

private Queue<MovementCommand> commandQueue = new LinkedList<>();

public void update(double deltaTime) {

// Process pending movement commands

while (!commandQueue.isEmpty()) {

MovementCommand cmd = commandQueue.poll();

applyMovementCommand(cmd);

}

// Update physics

updatePosition(deltaTime);

updateBattery(deltaTime);

// Publish state update

AgentStateUpdate update = new AgentStateUpdate();

update.agentId = state.agentId;

update.newPosition = state.position;

update.newVelocity = state.velocity;

update.timestamp = System.currentTimeMillis();

eventBus.publishEvent(new SystemEvent(EventType.AGENT\_STATE\_UPDATE, update));

}

private void updatePosition(double deltaTime) {

// Basic physics integration

state.position.x += state.velocity.x \* deltaTime;

state.position.y += state.velocity.y \* deltaTime;

// Apply speed limits

double speed = state.velocity.magnitude();

if (speed > state.maxSpeed) {

state.velocity = multiply(normalize(state.velocity), state.maxSpeed);

}

// Check boundaries

if (state.position.x < 0 || state.position.x > WORLD\_WIDTH ||

state.position.y < 0 || state.position.y > WORLD\_HEIGHT) {

// Bounce off boundaries

if (state.position.x < 0 || state.position.x > WORLD\_WIDTH) {

state.velocity.x \*= -1;

}

if (state.position.y < 0 || state.position.y > WORLD\_HEIGHT) {

state.velocity.y \*= -1;

}

}

}

}

**9.1 Week 1 Setup Tasks**

**All Team Members:**

* Set up project structure with Maven/Gradle
* Create basic package structure
* Implement basic data structures (Point2D, Vector2D, AgentState)

**Sanidhaya:**

* Create Agent class with basic constructor
* Implement simple position updates
* Create AgentManager for creating/removing agents
* Set up basic physics calculations

**John:**

* Create Message and CommunicationManager classes
* Implement basic neighbor detection based on distance
* Set up message queue system
* Create simple broadcast functionality

**Lauren:**

* Research flocking algorithms (separation, alignment, cohesion)
* Create FlockingController class structure
* Implement basic vector math utilities
* Design voting system architecture

**Anthony:**

* Set up JavaFX project structure
* Create basic window with canvas for agent display
* Implement mouse click handling
* Create basic agent visualization (circles)

**9.2 Integration Preparation**

* All: Agree on shared data structure formats
* All: Set up EventBus for component communication
* All: Create mock implementations for testing
* All: Schedule Week 2 integration session

*Technical Specification Document for Team 6*  
*Distributed Multi-Agent System for Autonomous Drones/Robots*  
*Software Engineering Graduate Project*