Multi-Agent Simulacra Design Document: Interactive Fiction with Model Context Protocol

This design document outlines a comprehensive architecture for a Multi-Agent Simulacra simulation featuring LLM-powered NPCs in a Japanese RPG village setting, with specific focus on a debate house containing deliberators and electors. The system integrates agentic interactive-fiction architecture, Model Context Protocol (MCP), and advanced memory systems.

System Architecture Overview

The system employs a **three-tier architecture** combining agentic interactive fiction principles with MCP integration and sophisticated memory management. Each NPC operates as an autonomous agent capable of dynamic storytelling, debate participation, and contextual memory retention.

Core Components

Agent Layer: Individual NPC agents with specialized roles (deliberators, electors)
MCP Integration Layer: Standardized protocol for tool and resource access
Memory Layer: CLIN-based episodic memory with IPOCL planning and vector embeddings
Frontend Layer: Real-time village and debate house visualization

Agent Architecture

Individual Agent Design

Each NPC agent implements a **hybrid cognitive architecture** combining:

Perception Module

- Spatial awareness of village environment and debate house positioning
- Social context detection (other agents' states, ongoing conversations)
- Event detection and filtering based on agent role and interests
- Real-time monitoring of debate proceedings and voting status

Memory Module (Detailed in Section 4)

- **CLIN (Causal Link Network)** for episodic memory storage
- **IPOCL (Intentional Partial-Order Causal Link)** planning for goal-driven behavior
- **Vector embeddings** for semantic similarity and retrieval
- Persistent character knowledge and relationship tracking

Reasoning Module

- **Belief-Desire-Intention (BDI)** framework for autonomous decision-making
- Role-specific reasoning patterns (deliberative vs. evaluative)
- Philosophical argument construction and evaluation
- Dynamic goal formation based on debate context

Action Module

- Speech generation for debate participation
- Non-verbal behavior (gestures, expressions, positioning)
- Voting decisions based on accumulated evidence
- Social interaction protocols

Agent Specialization

Deliberator Agents (2)

- **Primary Goals**: Present compelling philosophical arguments, counter opponent positions
- **Specialized Behaviors**: Argument construction, rhetorical strategy, audience awareness
- **Memory Focus**: Philosophical knowledge, debate history, opponent analysis
- **Planning Horizon**: Multi-turn debate strategy with adaptive responses

Elector Agents (5)

- **Primary Goals**: Evaluate arguments fairly, make informed voting decisions
- **Specialized Behaviors**: Critical analysis, evidence weighing, bias detection
- **Memory Focus**: Argument tracking, speaker credibility, philosophical consistency
- **Planning Horizon**: Evaluation criteria development and decision justification

Model Context Protocol Integration

MCP Server Architecture

The system implements **multiple specialized MCP servers** to handle different aspects of the simulation:

Philosophy Knowledge Server

- **Tools**: `query_philosophical_concepts`, `retrieve_arguments`, `analyze_logical_fallacies`
- **Resources**: Philosophical text corpus, argument databases, logical reasoning frameworks
- **Prompts**: Debate templates, argument construction guides, evaluation criteria

Character Memory Server

- **Tools**: `store_memory`, `retrieve_memories`, `update_relationships`, `query_beliefs`

- **Resources**: Character knowledge bases, relationship graphs, belief systems
- **Prompts**: Memory consolidation templates, relationship update patterns

Debate Management Server

- **Tools**: `track_speaking_turns`, `evaluate_arguments`, `manage_voting`, `generate_topics`
- **Resources**: Debate rules, scoring systems, topic databases
- **Prompts**: Moderation templates, evaluation rubrics, procedural guidelines

MCP Client Implementation

Each agent operates as an **MCP client** with the following capabilities:

Dynamic Tool Discovery

```
async def discover_available_tools():
    philosophy_tools = await mcp_client.list_tools("philosophy_server")
    memory_tools = await mcp_client.list_tools("memory_server")
    debate_tools = await mcp_client.list_tools("debate_server")
    return merge_tool_catalogs(philosophy_tools, memory_tools, debate_tools)
```

Context-Aware Tool Selection

- Agents dynamically select appropriate tools based on current situation
- Role-based tool prioritization (deliberators favor argument tools, electors favor evaluation tools)
- Fallback mechanisms for tool failures or unavailability

Resource Access Management

- Secure access to shared knowledge bases through MCP resource protocols
- Agent-specific resource filtering based on character knowledge and clearance
- Real-time resource updates for dynamic debate topics

Memory Module Design

CLIN (Causal Link Network) Implementation

The memory system employs **CLIN for episodic memory storage** with the following structure:

Event Representation

```
@dataclass
class MemoryEvent:
    event_id: str
    timestamp: datetime
    event_type: EventType  # SPEECH, ACTION, OBSERVATION, DECISION
    participants: List[str]
    content: str
    causal_links: List[CausalLink]
    emotional_valence: float
    importance_score: float
    embedding: np.ndarray
```

Causal Link Structure

- **Enablement Links**: Event A enables Event B (prerequisite relationships)
- **Motivational Links**: Event A motivates intention for Event B
- **Consequential Links**: Event A directly causes Event B
- **Temporal Links**: Event A occurs before Event B (strict ordering)

Memory Network Construction

The system builds causal networks by:

1. Identifying temporal relationships between events
2. Detecting enablement patterns through precondition analysis
3. Tracking motivational chains from observations to actions
4. Maintaining character-specific causal interpretations

IPOCL (Intentional Partial-Order Causal Link) Planning

Planning Architecture

The system uses **IPOCL for goal-driven behavior planning** with character-specific adaptations:

Goal Hierarchy Management

- **Primary Goals**: Role-specific objectives (win debate, evaluate fairly)
- **Secondary Goals**: Social objectives (maintain relationships, demonstrate expertise)
- **Reactive Goals**: Situational responses (counter arguments, clarify positions)

Plan Structure

```
@dataclass
class IPOCLPlan:
    goal: Goal
    actions: List[PlannedAction]
    causal_links: List[CausalLink]
    ordering_constraints: List[TemporalConstraint]
    character_intentions: Dict[str, Intention]
    contingencies: List[ConditionalPlan]
```

Character Intention Modeling

- Each agent maintains models of other characters' likely intentions
- Intention attribution based on observed actions and stated positions
- Dynamic intention updating as debates progress

Vector Embedding Integration

Embedding Architecture

The system uses the provided `embeddings.py` module for semantic memory retrieval:

Multi-Modal Embeddings

- **Semantic Embeddings**: Philosophical concepts, arguments, positions
- **Emotional Embeddings**: Sentiment and emotional context of interactions
- **Social Embeddings**: Relationship dynamics and social positioning
- **Temporal Embeddings**: Time-sensitive context and debate progression

Retrieval Mechanisms

```
async def retrieve_relevant_memories(query: str, agent_id: str, k: int = 5):
    query_embedding = get_embeddings([query], "text-embedding-3-small")[^0]

    # Multi-faceted similarity search
    semantic_matches = search_semantic_memories(query_embedding, agent_id, k)
    emotional_matches = search_emotional_memories(query_embedding, agent_id, k)
    social_matches = search_social_memories(query_embedding, agent_id, k)

    # Weighted combination based on context
    return combine_and_rank_memories(semantic_matches, emotional_matches, social_matches)
```

Memory Consolidation

- Periodic embedding updates for evolving concepts
- Cross-agent memory sharing for shared experiences
- Hierarchical clustering for memory organization

Interactive Fiction Integration

Narrative Generation Engine

Story Structure Management

Following the narrative psychology principles from the provided research, the system maintains:

Fabula Layer: Complete event sequence in the story world

- All agent actions, decisions, and interactions
- Environmental changes and debate progression
- Hidden motivations and internal character states

Sjuzhet Layer: Presented narrative elements

- Observable actions and dialogue
- Selected internal thoughts for dramatic effect

- Paced revelation of character motivations

Text Layer: Natural language output

- Real-time dialogue generation
- Descriptive text for actions and environment
- Narrative commentary for dramatic moments

Dynamic Plot Generation

Conflict Generation

The system creates compelling narratives through:

- **Philosophical Disagreements**: Fundamental worldview conflicts between characters
- **Personal Stakes**: Individual character motivations affecting debate outcomes
- **Social Dynamics**: Relationship tensions influencing argument reception
- **Procedural Conflicts**: Disagreements over debate rules and fairness

Character Arc Development

- **Deliberator Arcs**: Evolution of argumentative strategies and philosophical positions
- **Elector Arcs**: Journey from initial biases to informed decision-making
- **Relationship Arcs**: Changing dynamics between characters based on debate performance

Frontend Design

Village Environment

3D Village Representation

- **Architectural Style**: Japanese-inspired buildings with distinct debate house
- **Character Visualization**: Real-time avatar representation with emotional expressions
- **Environmental Storytelling**: Visual cues reflecting character states and relationships

Debate House Interface

- **Seating Arrangement**: Clear distinction between deliberator and elector positions
- **Speaking Indicators**: Visual cues for active speakers and turn management
- **Argument Tracking**: Real-time display of key points and counterarguments
- **Voting Interface**: Transparent voting process with decision justifications

Real-Time Communication

WebSocket Architecture

```
class DebateHouseClient {
    constructor() {
        this.socket = new WebSocket('ws://localhost:8080/debate');
        this.setupEventHandlers();
    }

    setupEventHandlers() {
        this.socket.onmessage = (event) => {
            const message = JSON.parse(event.data);
            this.handleAgentAction(message);
        };
    }

    handleAgentAction(message) {
        switch(message.type) {
            case 'SPEECH':
                this.displaySpeech(message.agent, message.content);
                break;
            case 'GESTURE':
                this.animateGesture(message.agent, message.gesture);
                break;
            case 'VOTE':
                this.updateVotingDisplay(message.agent, message.vote);
                break;
        }
    }
}
```

System Workflow

Initialization Sequence

1. **MCP Server Startup**: Initialize philosophy, memory, and debate management servers
2. **Agent Creation**: Instantiate 7 agents with role-specific configurations
3. **Memory Loading**: Load character backgrounds and philosophical knowledge
4. **Environment Setup**: Initialize village and debate house environments
5. **Topic Selection**: Generate or select philosophical debate topic
6. **Session Initialization**: Establish agent connections and begin simulation

Debate Execution Loop

Turn Management

```
async def execute_debate_turn(current_speaker: Agent, topic: str):
    # Retrieve relevant memories and knowledge
    context = await current_speaker.memory.retrieve_context(topic)

    # Generate response using MCP tools
    response = await current_speaker.generate_response(context, topic)

    # Update all agents' memories
    for agent in all_agents:
        await agent.memory.store_event(
            MemoryEvent(
                event_type=EventType.SPEECH,
                speaker=current_speaker.id,
                content=response,
                timestamp=datetime.now()
            )
        )

    # Update frontend
    await broadcast_to_frontend({
        'type': 'SPEECH',
        'agent': current_speaker.id,
        'content': response
    })
```

Decision Making Process

Elector Evaluation

1. **Argument Collection**: Gather all arguments from both deliberators
2. **Criteria Application**: Apply philosophical evaluation frameworks
3. **Bias Detection**: Identify and account for personal biases
4. **Decision Formation**: Synthesize evaluation into voting decision
5. **Justification Generation**: Create explanation for vote choice

Security and Privacy Considerations

MCP Security Implementation

Following the security guidelines from the MCP research paper:

Server Authentication

- Cryptographic verification of MCP server identities
- Secure communication channels for all agent-server interactions
- Regular security audits of installed MCP servers

Tool Access Control

- Role-based permissions for different agent types
- Sandboxed execution environments for external tool calls

- Monitoring and logging of all tool invocations

Memory Protection

- Encrypted storage of sensitive character information
- Access controls preventing unauthorized memory access
- Regular backup and integrity verification

Performance Optimization

Scalability Considerations

Concurrent Processing

- Asynchronous agent processing for parallel decision-making
- Efficient memory retrieval through optimized vector search
- Load balancing across multiple MCP servers

Memory Management

- Hierarchical memory storage with hot/cold data separation
- Periodic memory consolidation to prevent storage bloat
- Intelligent caching of frequently accessed embeddings

Network Optimization

- Compressed message formats for frontend communication
- Batch processing of memory updates
- Connection pooling for MCP server interactions

Testing and Validation

Agent Behavior Testing

Unit Tests

- Individual agent response validation
- Memory system consistency checks
- MCP tool integration verification

Integration Tests

- Multi-agent interaction scenarios
- End-to-end debate simulations
- Frontend synchronization validation

Performance Tests

- Concurrent user load testing
- Memory retrieval latency benchmarks
- Real-time response time validation

Narrative Quality Assessment

Automated Metrics

- Dialogue coherence scoring
- Argument logical consistency
- Character behavior consistency

Human Evaluation

- User engagement surveys
- Narrative quality assessments
- Educational effectiveness studies

Future Extensions

Advanced Features

Multi-Topic Debates

- Support for complex philosophical topics requiring multiple sessions
- Cross-topic knowledge transfer and consistency maintenance
- Dynamic topic evolution based on agent interests

Audience Participation

- Human observer integration with question submission
- Real-time polling and feedback mechanisms
- Educational mode with guided learning objectives

Character Development

- Long-term personality evolution based on debate experiences
- Relationship dynamics affecting future interactions
- Skill development and expertise accumulation

Technical Enhancements

Advanced AI Integration

- Multi-modal input processing (voice, gesture, facial expression)
- Emotional intelligence enhancement for more nuanced interactions

- Advanced reasoning capabilities for complex philosophical arguments

Expanded MCP Ecosystem

- Integration with external knowledge bases and research tools
- Real-time fact-checking and source verification
- Collaborative filtering for argument quality assessment

This comprehensive design document provides a robust foundation for implementing a sophisticated multi-agent simulacra system that combines cutting-edge AI technologies with engaging interactive fiction elements, creating an immersive and educational philosophical debate experience.

⁂