Agent Communication Protocol - Deep Dive

Overview

The Agent Communication Protocol is the **most critical component** of the Powerhouse B2B Platform. It enables the modular business model by allowing any combination of agents to communicate seamlessly, regardless of which subset is deployed.

Why This Matters for B2B

In a traditional multi-agent system, agents are tightly coupled and must know about each other at design time. This makes it impossible to offer an à la carte model where clients select which agents they want.

Our Solution: A robust communication protocol that provides:
- **Service Discovery**: Agents can find each other at runtime
- **Loose Coupling**: Agents don't need to know about each other's implementation
- **Dynamic Configuration**: Any combination of agents works together
- **Audit Trail**: All communication is logged for compliance

Architecture

```
┌─────────────────────────────────────────────────────────────┐
│                  Communication Protocol                      │
│                                                              │
│  ┌─────────────────┐  ┌─────────────────┐  ┌──────────────┐ │
│  │  Message Bus    │  │ Agent Registry  │  │    Shared    │ │
│  │                 │  │                 │  │   Context    │ │
│  │  • Routing      │  │  • Discovery    │  │              │ │
│  │  • Queuing      │  │  • Health       │  │  • State     │ │
│  │  • History      │  │  • Capability   │  │  • History   │ │
│  │  • Broadcast    │  │  • Load Bal.    │  │  • Watching  │ │
│  └─────────────────┘  └─────────────────┘  └──────────────┘ │
│                                                              │
└─────────────────────────────────────────────────────────────┘
```

Component 1: Message Bus

Purpose

Route messages between agents with guaranteed delivery and ordering.

Message Format

```python
@dataclass
class Message:
    id: str                     # Unique message ID
    sender: str                 # Sending agent name
    receiver: str               # Receiving agent name (or "broadcast")
    message_type: MessageType   # Type of message
    content: Any                # Message payload (JSON-serializable)
    timestamp: datetime         # When created
    run_id: Optional[str]       # Associated run
    correlation_id: Optional[str]  # For threading messages
    priority: int               # 0-10 (higher = more important)
    metadata: Dict[str, Any]    # Additional metadata
```

Message Types

```python
class MessageType(Enum):
    # Direct Communication
    REQUEST = "request"           # Request for action/information
    RESPONSE = "response"         # Response to a request
    NOTIFICATION = "notification" # One-way notification

    # Coordination
    TASK_ASSIGNMENT = "task_assignment"
    TASK_COMPLETE = "task_complete"
    TASK_FAILED = "task_failed"

    # Collaboration
    QUERY = "query"              # Question to other agents
    ANSWER = "answer"            # Answer to a query
    PROPOSAL = "proposal"        # Suggest an approach
    VOTE = "vote"                # Vote on a proposal

    # System
    HEARTBEAT = "heartbeat"      # Agent health check
    SHUTDOWN = "shutdown"        # Agent shutting down
    ERROR = "error"              # Error notification

    # Broadcast
    BROADCAST = "broadcast"      # Message to all agents
```

Key Features

1. Direct Messaging

```python
# Agent A sends to Agent B
message = protocol.send_message(
    sender="agent_a",
    receiver="agent_b",
    message_type=MessageType.REQUEST,
    content={"query": "What's the status?"}
)

# Agent B receives
messages = protocol.get_messages("agent_b")
for msg in messages:
    if msg.message_type == MessageType.REQUEST:
        # Process and respond
        protocol.respond("agent_b", msg, {"status": "complete"})
```

2. Broadcast Messaging

```python
# Notify all agents
protocol.broadcast(
    sender="orchestrator",
    message_type=MessageType.NOTIFICATION,
    content={"event": "task_started", "task_id": "123"}
)

# All agents receive this message
```

3. Request-Response Pattern

```python
# Agent A requests from Agent B
response = protocol.request(
    sender="agent_a",
    receiver="agent_b",
    content={"question": "What's the answer?"},
    timeout=30.0  # Wait up to 30 seconds
)

if response:
    answer = response.content
```

4. Subscription-Based Routing

```python
# Agent subscribes to specific message types
protocol.subscribe("agent_c", MessageType.TASK_COMPLETE)

# Now agent_c receives all TASK_COMPLETE messages
# even if not directly addressed to it
```

5. Message History

```python
# Get all messages in a conversation
conversation = protocol.message_bus.get_conversation(correlation_id)

# Get messages from/to specific agent
history = protocol.message_bus.get_history(
    agent_name="agent_a",
    since=datetime.now() - timedelta(hours=1),
    limit=100
)
```

Implementation Details

Thread Safety: Uses Python's `threading.Lock` for all operations

Queue Management: Each agent has a dedicated queue (deque)

```python
_queues: Dict[str, deque] = {
    "agent_a": deque([msg1, msg2, ...], maxlen=1000),
    "agent_b": deque([msg3, msg4, ...], maxlen=1000),
}
```

Message Routing Logic:
1. Determine recipients (direct or broadcast)
2. Add subscribers to message type
3. Deliver to each recipient's queue
4. Call registered handlers
5. Add to history

Performance:
- In-memory: ~10,000 messages/sec
- Latency: <1ms for local delivery
- Memory: O(queue_size × num_agents)

Component 2: Agent Registry

Purpose

Enable service discovery and health monitoring so agents can find each other dynamically.

Agent Information

```python
@dataclass
class AgentInfo:
    name: str                      # Unique agent name
    agent_type: str                # Type (e.g., "react", "evaluator")
    capabilities: List[str]        # What this agent can do
    status: str                    # "active", "idle", "busy", "offline"
    metadata: Dict[str, Any]       # Additional info
    registered_at: datetime        # When registered
    last_heartbeat: datetime       # Last health check
    message_count: int             # Messages processed
```

Key Features

1. Registration

```python
# Agent registers itself
protocol.register_agent(
    name="my_agent",
    agent_type="react",
    capabilities=["reasoning", "planning", "tool_use"],
    metadata={"version": "1.0", "model": "gpt-4"}
)
```

2. Discovery by Capability

```python
# Find all agents that can do reasoning
reasoning_agents = protocol.find_by_capability("reasoning")
# Returns: ["agent_a", "agent_b", "agent_c"]

# Get detailed info
for agent_name in reasoning_agents:
    info = protocol.get_agent_info(agent_name)
    print(f"{agent_name}: {info.status}")
```

3. Discovery by Type

```python
# Find all evaluator agents
evaluators = protocol.find_agents(agent_type="evaluator")

# Find all active agents
active = protocol.find_agents(status="active")
```

4. Health Monitoring

```python
# Agent sends heartbeat
protocol.heartbeat("my_agent")

# Check health of all agents
health = protocol.check_health()
# Returns: {"agent_a": "active", "agent_b": "offline", ...}
```

5. Load Balancing

```python
# Get least busy agent with a capability
agent = protocol.agent_registry.get_least_busy_agent(
    capability="reasoning"
)
# Returns agent with lowest message_count
```

Capability Index

The registry maintains an inverted index for fast capability lookup:

```
_capabilities_index = {
    "reasoning": {"agent_a", "agent_b", "agent_c"},
    "planning": {"agent_a", "agent_d"},
    "evaluation": {"agent_e"},
    "tool_use": {"agent_a", "agent_f"}
}
```

This enables O(1) capability lookups instead of O(n) scans.

Component 3: Shared Context

Purpose

Provide shared state management so agents can coordinate without direct messaging.

State Organization

```
Global State (accessible to all agents):
├── task: "Analyze quarterly sales"
├── run_id: "uuid-123"
├── status: "running"
└── results: {...}

Agent Namespaces (private to each agent):
├── agent_a:
│   ├── memory: [...]
│   ├── internal_state: {...}
│   └── cache: {...}
├── agent_b:
│   ├── analysis_results: {...}
│   └── confidence_scores: [...]
```

Key Features

1. Global State

```
# Set global state (any agent can read)
protocol.set_state("task_status", "in_progress")
protocol.set_state("current_step", 3)

# Get global state
status = protocol.get_state("task_status")
step = protocol.get_state("current_step", default=0)

# Update multiple values
protocol.update_state({
    "task_status": "complete",
    "final_score": 0.95
})
```

2. Namespaced State (Private)

```
# Agent A sets its private state
protocol.set_state(
    "memory",
    {"last_query": "...", "results": [...]},
    namespace="agent_a"
)

# Only agent_a can read this
memory = protocol.get_state("memory", namespace="agent_a")

# Other agents cannot access it
memory = protocol.get_state("memory", namespace="agent_b")  # Returns None
```

3. State History

```
# Get history of changes to a key
history = protocol.shared_context.get_history(
    key="task_status",
    limit=10
)

# Returns:
[
    {
        "timestamp": "2024-10-06T10:00:00",
        "action": "set",
        "key": "task_status",
        "old_value": "pending",
        "new_value": "running",
        "agent_name": "orchestrator"
    },
    ...
]
```

4. State Watching

```
# Agent B watches a key
protocol.shared_context.watch("task_status", "agent_b")

# When task_status changes, agent_b is notified
# (Implementation can trigger callbacks or add to message queue)
```

5. Bulk Operations

```
# Get all global state
all_state = protocol.shared_context.get_all(namespace="global")

# Get all keys
keys = protocol.shared_context.keys(namespace="global")

# Clear namespace
protocol.shared_context.clear(namespace="agent_a")
```

Thread Safety

All operations use locks to ensure thread-safe access:

```
with self._lock:
    self._global_state[key] = value
    self._add_history_entry(...)
    self._notify_watchers(...)
```

Component 4: Unified Protocol Interface

Purpose

Provide a high-level API that ties all components together.

Complete Example

```python
from communication import CommunicationProtocol, MessageType

# Initialize protocol
protocol = CommunicationProtocol()

# === Agent Lifecycle ===

# Register agents
protocol.register_agent(
    name="analyzer",
    agent_type="react",
    capabilities=["analysis", "reasoning"]
)

protocol.register_agent(
    name="evaluator",
    agent_type="evaluator",
    capabilities=["evaluation", "scoring"]
)

# === Discovery ===

# Find agents by capability
analysts = protocol.find_by_capability("analysis")
print(f"Analysts: {analysts}")

# Get agent info
info = protocol.get_agent_info("analyzer")
print(f"Analyzer status: {info.status}")

# === Messaging ===

# Direct message
protocol.send_message(
    sender="analyzer",
    receiver="evaluator",
    message_type=MessageType.REQUEST,
    content={"data": "...", "question": "Is this good?"}
)

# Broadcast
protocol.broadcast(
    sender="analyzer",
    message_type=MessageType.NOTIFICATION,
    content={"event": "analysis_complete"}
)

# Request-response
response = protocol.request(
    sender="analyzer",
    receiver="evaluator",
    content={"question": "What's the score?"},
    timeout=30.0
)

# === State Management ===

# Set global state
protocol.set_state("analysis_result", {"score": 0.95})

# Set private state
protocol.set_state(
```

```
    "internal_memory",
    {"cache": [...]},
    namespace="analyzer"
)

# Get state
result = protocol.get_state("analysis_result")

# === Health Monitoring ===

# Send heartbeat
protocol.heartbeat("analyzer")

# Update status
protocol.update_agent_status("analyzer", "busy")

# Check health
health = protocol.check_health()

# === Statistics ===

stats = protocol.get_stats()
print(f"Total agents: {stats['agent_registry']['total_agents']}")
print(f"Messages in queues: {stats['message_bus']['total_messages_in_queues']}")
```

Real-World Usage Patterns

Pattern 1: Collaborative Analysis

```
# Analyzer agent requests help from specialists
specialists = protocol.find_by_capability("domain_expert")

results = []
for specialist in specialists:
    response = protocol.request(
        sender="analyzer",
        receiver=specialist,
        content={"data": data, "question": "Analyze this"},
        timeout=60.0
    )
    if response:
        results.append(response.content)

# Aggregate results
protocol.set_state("specialist_results", results)
```

Pattern 2: Voting/Consensus

```
# Coordinator broadcasts proposal
protocol.broadcast(
    sender="coordinator",
    message_type=MessageType.PROPOSAL,
    content={"proposal_id": "123", "action": "approve_plan"}
)

# Agents vote
protocol.send_message(
    sender="agent_a",
    receiver="coordinator",
    message_type=MessageType.VOTE,
    content={"proposal_id": "123", "vote": "yes"}
)

# Coordinator collects votes
votes = protocol.get_messages("coordinator")
yes_votes = sum(1 for v in votes if v.content.get("vote") == "yes")
```

Pattern 3: Pipeline Processing

```
# Agent A processes and passes to Agent B
result_a = agent_a.execute(data)
protocol.set_state("stage_1_result", result_a)

# Agent B picks up from shared state
result_a = protocol.get_state("stage_1_result")
result_b = agent_b.execute(result_a)
protocol.set_state("stage_2_result", result_b)
```

Pattern 4: Error Handling & Fallback

```
# Try primary agent
primary = protocol.find_by_capability("primary_analysis")
if primary:
    response = protocol.request(
        sender="orchestrator",
        receiver=primary[0],
        content={"task": "analyze"},
        timeout=30.0
    )

    if not response:
        # Fallback to secondary
        secondary = protocol.find_by_capability("secondary_analysis")
        if secondary:
            response = protocol.request(
                sender="orchestrator",
                receiver=secondary[0],
                content={"task": "analyze"},
                timeout=30.0
            )
```

Database Persistence

All messages are automatically persisted to the database:

```
class AgentMessage(Base):
    __tablename__ = "agent_messages"

    id = Column(String(36), primary_key=True)
    run_id = Column(String(36), ForeignKey("runs.id"))
    sender = Column(String(255), index=True)
    receiver = Column(String(255), index=True)
    message_type = Column(String(50), index=True)
    content = Column(JSON)
    correlation_id = Column(String(36), index=True)
    timestamp = Column(DateTime, index=True)
```

This enables:
- **Audit Trail**: Complete history of all communication
- **Debugging**: Replay conversations
- **Visualization**: Build communication graphs
- **Analytics**: Analyze agent interaction patterns

Performance Optimization

Message Bus

- **Batching**: Get multiple messages at once
- **Priority Queues**: High-priority messages first
- **Cleanup**: Automatic removal of old messages

Agent Registry

- **Capability Index**: O(1) capability lookups
- **Caching**: Cache frequently accessed agent info
- **Lazy Loading**: Load agent details on demand

Shared Context

- **Namespacing**: Reduce global state size
- **History Limits**: Configurable history size
- **Selective Watching**: Only watch needed keys

Testing the Protocol

```python
def test_communication_protocol():
    protocol = CommunicationProtocol()

    # Register agents
    protocol.register_agent("agent_1", "react", ["reasoning"])
    protocol.register_agent("agent_2", "evaluator", ["evaluation"])

    # Test messaging
    protocol.send_message(
        sender="agent_1",
        receiver="agent_2",
        message_type=MessageType.REQUEST,
        content={"test": "data"}
    )

    messages = protocol.get_messages("agent_2")
    assert len(messages) == 1
    assert messages[0].sender == "agent_1"

    # Test discovery
    agents = protocol.find_by_capability("reasoning")
    assert "agent_1" in agents

    # Test state
    protocol.set_state("test_key", "test_value")
    value = protocol.get_state("test_key")
    assert value == "test_value"
```

Best Practices

1. **Always Register**: Agents must register before communicating
2. **Use Capabilities**: Discover agents by capability, not by name
3. **Handle Timeouts**: Request-response may timeout
4. **Send Heartbeats**: Keep registry updated
5. **Clean Up**: Deregister when shutting down
6. **Use Correlation IDs**: Thread related messages
7. **Namespace Private State**: Use agent-specific namespaces
8. **Log Everything**: Communication is automatically logged

Future Enhancements

1. **Distributed Message Bus**: Redis pub/sub for multi-instance
2. **Message Encryption**: End-to-end encryption for sensitive data
3. **Rate Limiting**: Prevent message flooding
4. **Message Priorities**: Advanced priority queuing
5. **Dead Letter Queue**: Handle undeliverable messages
6. **Circuit Breaker**: Prevent cascading failures
7. **Message Replay**: Replay messages for debugging
8. **Real-time Monitoring**: WebSocket for live communication view

This communication protocol is the **secret sauce** that makes the modular B2B model work. Any combination of agents can communicate seamlessly, enabling true à la carte agent selection for clients.