```
import asyncio
import json
from typing import Dict, List, Optional, Any, Set
from dataclasses import dataclass, field
from datetime import datetime
import hashlib
import re
from collections import defaultdict
import networkx as nx

@dataclass
class NarrativePayload:
    """Represents a narrative or meme in the information space"""
    id: str
    content: str
    origin: str  # platform/source
    vector_type: str  # organic, bot, influencer, state-actor
    timestamp: datetime
    virality_score: float = 0.0
    toxicity_markers: List[str] = field(default_factory=list)
    counter_deployed: bool = False
    metadata: Dict[str, Any] = field(default_factory=dict)

@dataclass
class CounterMeme:
    """Counter-narrative payload"""
    target_id: str
    content: str
    strategy: str  # inversion, amplification, redirection, mockery
    deployment_channels: List[str]
    expected_impact: float
    aesthetic_profile: Dict[str, Any] = field(default_factory=dict)

class NarrativeWarfareSystem:
    """Unified system for narrative detection, analysis, and counter-operations"""

    def __init__(self, api_keys: Dict[str, str]):
        self.api_keys = api_keys
        self.narrative_graph = nx.DiGraph()
        self.threat_narratives: Dict[str, NarrativePayload] = {}
        self.counter_arsenal: List[CounterMeme] = []
        self.operation_log: List[Dict[str, Any]] = []

        # BRICS narrative signatures
        self.brics_signatures = {
            'russia': ['nazi', 'biolab', 'multipolar', 'empire collapse'],
            'china': ['century of humiliation', 'win-win', 'surveillance necessary'],
            'generic': ['dollar dead', 'west declining', 'traditional values']
        }

        # Counter-strategies
        self.counter_strategies = {
            'inversion': self.invert_narrative,
            'mockery': self.mock_aesthetic,
            'overload': self.semantic_overload,
            'redirect': self.redirect_energy
        }

    async def scan_information_space(self, sources: List[str]) -> List[NarrativePayload]:
        """Scan multiple platforms for narrative patterns"""
        detected_narratives = []

        # Simulate platform scanning
        for source in sources:
            # In reality, this would call platform APIs
            narratives = await self.scan_platform(source)
            detected_narratives.extend(narratives)

        # Analyze for BRICS signatures
        for narrative in detected_narratives:
            narrative.metadata['brics_alignment'] = self.check_brics_alignment(narrative)

        return detected_narratives

    def check_brics_alignment(self, narrative: NarrativePayload) -> Dict[str, float]:
        """Check narrative alignment with known BRICS information operations"""
        alignment_scores = {}

        content_lower = narrative.content.lower()
        for country, signatures in self.brics_signatures.items():
            score = sum(1 for sig in signatures if sig in content_lower) / len(signatures)
            if score > 0:
                alignment_scores[country] = score

        return alignment_scores

    async def scan_platform(self, platform: str) -> List[NarrativePayload]:
        """Scan specific platform for narratives"""
        # Simulated - would connect to real APIs
        return []

    def analyze_narrative_network(self, narratives: List[NarrativePayload]) -> Dict[str, Any]:
        """Build and analyze narrative propagation network"""

        # Build narrative graph
        for narrative in narratives:
            self.narrative_graph.add_node(
                narrative.id,
                content=narrative.content,
                timestamp=narrative.timestamp,
                origin=narrative.origin
            )

        # Find propagation patterns
        communities = nx.community.louvain_communities(self.narrative_graph.to_undirected())

        # Identify coordinated campaigns
        coordinated_campaigns = []
        for community in communities:
            if len(community) > 5:  # Threshold for coordination
                subgraph = self.narrative_graph.subgraph(community)
                campaign = {
                    'nodes': list(community),
                    'density': nx.density(subgraph),
                    'central_narratives': self.find_central_narratives(subgraph),
                    'likely_coordinated': True
                }
                coordinated_campaigns.append(campaign)

        return {
            'total_narratives': len(narratives),
            'network_density': nx.density(self.narrative_graph),
            'coordinated_campaigns': coordinated_campaigns,
            'influence_nodes': nx.degree_centrality(self.narrative_graph)
        }

    def find_central_narratives(self, subgraph) -> List[str]:
        """Find most influential narratives in subgraph"""
        centrality = nx.degree_centrality(subgraph)
        sorted_nodes = sorted(centrality.items(), key=lambda x: x[1], reverse=True)
        return [node[0] for node in sorted_nodes[:3]]
```

```
def generate_counter_operation(self,
                               target_narrative: NarrativePayload,
                               strategy: str = "auto") -> List[CounterMeme]:
    """Generate counter-narrative operation"""

    if strategy == "auto":
        # Auto-select strategy based on narrative type
        if target_narrative.metadata.get('brics_alignment'):
            strategy = "inversion"
        elif "conspiracy" in target_narrative.toxicity_markers:
            strategy = "mockery"
        else:
            strategy = "redirect"

    counter_memes = []

    # Generate multiple counter-memes for A/B testing
    for i in range(3):
        counter = self.counter_strategies[strategy](target_narrative)
        counter_memes.append(counter)

    return counter_memes

def invert_narrative(self, narrative: NarrativePayload) -> CounterMeme:
    """Invert the narrative logic"""
    # Extraction of key claims
    inverted_content = f"If {narrative.content}, then why [INSERT CONTRADICTION]?"

    return CounterMeme(
        target_id=narrative.id,
        content=inverted_content,
        strategy="inversion",
        deployment_channels=["twitter", "reddit"],
        expected_impact=0.7
    )

def mock_aesthetic(self, narrative: NarrativePayload) -> CounterMeme:
    """Mock the aesthetic and tone"""
    # Generate mocking version
    mocked = narrative.content.upper() + " 🤡 SURE BRO 🤡"

    return CounterMeme(
        target_id=narrative.id,
        content=mocked,
        strategy="mockery",
        deployment_channels=["twitter", "tiktok"],
        expected_impact=0.6,
        aesthetic_profile={"style": "gen-z-irony", "emoji_density": "high"}
    )

def semantic_overload(self, narrative: NarrativePayload) -> CounterMeme:
    """Overload with recursive complexity"""
    overloaded = f"Actually, {narrative.content} is just {narrative.content} " \
                f"pretending to be {narrative.content} while avoiding that {narrative.content}"

    return CounterMeme(
        target_id=narrative.id,
        content=overloaded,
        strategy="overload",
        deployment_channels=["reddit", "twitter"],
        expected_impact=0.5
    )

def redirect_energy(self, narrative: NarrativePayload) -> CounterMeme:
    """Redirect emotional energy elsewhere"""
    redirect = f"Instead of worrying about that, have you considered " \
              f"[INSERT MORE CONSTRUCTIVE CONCERN]?"

    return CounterMeme(
        target_id=narrative.id,
        content=redirect,
        strategy="redirect",
        deployment_channels=["facebook", "reddit"],
        expected_impact=0.8
    )

async def deploy_counter_operation(self,
                                   counter_memes: List[CounterMeme],
                                   intensity: str = "measured") -> Dict[str, Any]:
    """Deploy counter-narrative operation"""

    deployment_scales = {
        "subtle": 10,
        "measured": 100,
        "flood": 1000,
        "swarm": 10000
    }
    scale = deployment_scales.get(intensity, 100)

    deployment_report = {
        'timestamp': datetime.now().isoformat(),
        'memes_deployed': len(counter_memes),
        'scale': scale,
        'channels': [],
        'projected_reach': 0
    }

    for meme in counter_memes:
        # Simulate deployment
        for channel in meme.deployment_channels:
            deployment_report['channels'].append(channel)
            deployment_report['projected_reach'] += scale * meme.expected_impact

        # Log operation
        self.operation_log.append({
            'timestamp': datetime.now().isoformat(),
            'target': meme.target_id,
            'strategy': meme.strategy,
            'content': meme.content,
            'scale': scale
        })

    return deployment_report

def generate_operation_report(self) -> Dict[str, Any]:
    """Generate comprehensive operation report"""

    # Analyze operation effectiveness
    operations_by_strategy = defaultdict(list)
    for op in self.operation_log:
        operations_by_strategy[op['strategy']].append(op)

    return {
        'total_operations': len(self.operation_log),
        'narratives_countered': len(set(op['target'] for op in self.operation_log)),
        'strategy_breakdown': {
            strategy: len(ops) for strategy, ops in operations_by_strategy.items()
        },
        'timeline': self.generate_operation_timeline(),
        'network_impact': self.assess_network_impact()
    }
```

```
def generate_operation_timeline(self) -> List[Dict[str, Any]]:
    """Generate timeline of operations"""
    timeline = []
    for op in sorted(self.operation_log, key=lambda x: x['timestamp']):
        timeline.append({
            'time': op['timestamp'],
            'action': f"{op['strategy']} deployment",
            'scale': op['scale']
        })
    return timeline

def assess_network_impact(self) -> Dict[str, Any]:
    """Assess impact on narrative network"""
    # This would analyze real metrics in production
    return {
        'network_fragmentation': 0.0,
        'narrative_velocity_reduction': 0.0,
        'counter_narrative_adoption': 0.0
    }

async def full_spectrum_operation(self,
                                query: str,
                                sources: List[str],
                                auto_deploy: bool = False) -> Dict[str, Any]:
    """Execute full spectrum narrative warfare operation"""

    operation_report = {
        'query': query,
        'timestamp': datetime.now().isoformat(),
        'phases': {}
    }

    # Phase 1: Detection
    print("🔍 Phase 1: Scanning information space...")
    narratives = await self.scan_information_space(sources)
    operation_report['phases']['detection'] = {
        'narratives_detected': len(narratives),
        'platforms_scanned': sources
    }

    # Phase 2: Analysis
    print("🧠 Phase 2: Analyzing narrative network...")
    network_analysis = self.analyze_narrative_network(narratives)
```

```
        operation_report['phases']['analysis'] = network_analysis

        # Phase 3: Counter-Generation
        print("🎯 Phase 3: Generating counter-operations...")
        counter_operations = []
        for narrative in narratives:
            if narrative.metadata.get('brics_alignment') or narrative.toxicity_markers:
                counters = self.generate_counter_operation(narrative)
                counter_operations.extend(counters)

        operation_report['phases']['counter_generation'] = {
            'counters_generated': len(counter_operations),
            'strategies_used': list(set(c.strategy for c in counter_operations))
        }

        # Phase 4: Deployment (if authorized)
        if auto_deploy and counter_operations:
            print("🚀 Phase 4: Deploying counter-operations...")
            deployment = await self.deploy_counter_operation(
                counter_operations[:10],  # Limit for safety
                intensity="measured"
            )
            operation_report['phases']['deployment'] = deployment
        else:
            operation_report['phases']['deployment'] = {
                'status': 'awaiting_authorization',
                'ready_counters': len(counter_operations)
            }

        # Phase 5: Assessment
        print("📊 Phase 5: Generating operation report...")
        operation_report['assessment'] = self.generate_operation_report()

        return operation_report

# Example usage
async def main():
    # Initialize system
    warfare_system = NarrativeWarfareSystem({
        'groq': 'your-api-key',
        'twitter': 'your-api-key',
        'reddit': 'your-api-key'
    })
    # Execute operation
    report = await warfare_system.full_spectrum_operation(
        query="Detect and counter BRICS-aligned narratives targeting Western democracy",
        sources=["twitter", "telegram", "reddit"],
        auto_deploy=False  # Set to True to actually deploy
    )

    print("\n📋 OPERATION REPORT:")
    print(json.dumps(report, indent=2))

if __name__ == "__main__":
    asyncio.run(main())
```