Telemetry System Documentation

Overview

The UtilityFog telemetry system provides comprehensive metrics collection and export capabilities for monitoring system performance, behavior, and health across the fractal tree network.

Architecture

Core Components

1. **TelemetryCollector**: Central hub for metrics collection and event recording
2. **MetricsExporter**: Abstract interface for exporting metrics in various formats
3. **PrometheusAdapter**: Prometheus-compatible metrics export
4. **Metric Types**: Counter, Gauge, and Histogram implementations

Key Features

- **Real-time Metrics**: Live collection of system performance data
- **Event Recording**: Structured event logging with metadata
- **Multiple Export Formats**: Prometheus, JSON, and extensible format support
- **System Integration**: Built-in hooks for coordination, messaging, and health systems
- **Thread-Safe**: Concurrent access support with proper locking
- **Configurable**: Flexible collection intervals and export options

Usage

Basic Setup

```
from utilityfog_frontend.telemetry import TelemetryCollector, PrometheusAdapter

# Create collector
collector = TelemetryCollector(collection_interval=30.0)

# Register metrics
counter = collector.register_counter("requests_total", "Total HTTP requests")
gauge = collector.register_gauge("active_connections", "Active connections")
histogram = collector.register_histogram("request_duration_seconds", "Request dura-
tion")

# Start collection
await collector.start_collection()
```

Recording Metrics

```
# Increment counter
counter.increment()
counter.increment(5.0)

# Set gauge value
gauge.set(42)
gauge.increment(1)
gauge.decrement(2)

# Record histogram observations
histogram.observe(0.123)
histogram.observe(0.456)
```

Event Recording

```
# Record structured events
collector.record_event("user_login", {
    "user_id": "12345",
    "timestamp": time.time()
}, labels={"source": "web"})

collector.record_event("error_occurred", {
    "error_type": "connection_timeout",
    "details": "Failed to connect to database"
})
```

Exporting Metrics

```
# Prometheus export
prometheus_exporter = PrometheusAdapter("/tmp/metrics.prom")
await prometheus_exporter.export_metrics(collector)

# JSON export
from utilityfog_frontend.telemetry.exporter import JSONExporter
json_exporter = JSONExporter("/tmp/metrics.json")
await json_exporter.export_metrics(collector)
```

System Integration

Coordination System Hooks

The telemetry system automatically integrates with the coordination system:

```
from utilityfog_frontend.telemetry.collector import setup_coordination_hooks

setup_coordination_hooks(collector)
```

Metrics Registered:

- `coordination_messages_total`: Total coordination messages
- `coordination_active_nodes`: Number of active coordination nodes
- `coordination_message_latency_seconds`: Coordination message latency

Messaging System Hooks

Integration with the messaging system:

```
from utilityfog_frontend.telemetry.collector import setup_messaging_hooks

setup_messaging_hooks(collector)
```

Metrics Registered:
- `messages_sent_total`: Total messages sent
- `messages_received_total`: Total messages received
- `message_queue_size`: Current message queue size
- `message_processing_duration_seconds`: Message processing duration

Health System Hooks

Integration with health monitoring:

```
from utilityfog_frontend.telemetry.collector import setup_health_hooks

setup_health_hooks(collector)
```

Metrics Registered:
- `health_status`: Current health status (0=unknown, 1=healthy, 2=degraded, 3=unhealthy)
- `health_checks_total`: Total health checks performed
- `health_check_duration_seconds`: Health check duration

Metric Types

Counter

Monotonically increasing values (e.g., request counts, error counts):

```
counter = collector.register_counter("http_requests_total", "Total HTTP requests")
counter.increment()        # Increment by 1
counter.increment(5.0)     # Increment by 5
```

Gauge

Values that can increase or decrease (e.g., memory usage, active connections):

```
gauge = collector.register_gauge("memory_usage_bytes", "Memory usage in bytes")
gauge.set(1024000)         # Set absolute value
gauge.increment(100)       # Increase by 100
gauge.decrement(50)        # Decrease by 50
```

Histogram

Distribution of values (e.g., request durations, response sizes):

```
histogram = collector.register_histogram(
    "request_duration_seconds",
    "HTTP request duration",
    buckets=[0.1, 0.5, 1.0, 2.5, 5.0, 10.0]
)
histogram.observe(0.234)   # Record observation
```

Export Formats

Prometheus Format

Standard Prometheus text format for scraping:

```
# HELP http_requests_total Total HTTP requests
# TYPE http_requests_total counter
http_requests_total 1027

# HELP memory_usage_bytes Memory usage in bytes
# TYPE memory_usage_bytes gauge
memory_usage_bytes 1048576
```

JSON Format

Structured JSON for programmatic consumption:

```
{
  "timestamp": 1632150000.0,
  "metrics": {
    "http_requests_total": {
      "value": 1027,
      "labels": {},
      "timestamp": 1632150000.0
    }
  },
  "events_count": 150,
  "running": true
}
```

Configuration

Collection Interval

Configure how often metrics are collected:

```
collector = TelemetryCollector(collection_interval=60.0)  # 60 seconds
```

Custom Buckets

Configure histogram buckets for your use case:

```
histogram = collector.register_histogram(
    "api_response_time",
    "API response time distribution",
    buckets=[0.001, 0.01, 0.1, 0.5, 1.0, 5.0]  # Custom buckets
)
```

Periodic Export

Set up automatic periodic export:

```
from utilityfog_frontend.telemetry.exporter import PeriodicExporter

exporter = PrometheusAdapter("/var/metrics/prometheus.txt")
periodic_exporter = PeriodicExporter(exporter, collector, interval=30.0)

await periodic_exporter.start()
```

Best Practices

Metric Naming

- Use descriptive names with units: `request_duration_seconds`, `memory_usage_bytes`
- Follow Prometheus naming conventions: `subsystem_metric_unit`
- Use consistent labeling: `{method="GET", status="200"}`

Performance Considerations

- Use appropriate metric types for your data
- Avoid high-cardinality labels (many unique values)
- Configure reasonable collection intervals
- Monitor memory usage with many metrics

Error Handling

The telemetry system is designed to be resilient:

- Failed metric operations are logged but don't crash the system
- Export failures are retried automatically
- Thread-safe operations prevent data corruption

Monitoring and Alerting

Key Metrics to Monitor

- `telemetry_collection_runs_total`: Collection health
- `telemetry_collection_duration_seconds`: Collection performance
- `telemetry_metrics_count`: Number of registered metrics
- `telemetry_events_total`: Event recording rate

Sample Alerts

```yaml
# Prometheus alerting rules
groups:
- name: telemetry
  rules:
  - alert: TelemetryCollectionFailed
    expr: increase(telemetry_collection_runs_total[5m]) == 0
    for: 2m
    annotations:
      summary: "Telemetry collection has stopped"

  - alert: HighTelemetryCollectionDuration
    expr: telemetry_collection_duration_seconds > 10
    for: 1m
    annotations:
      summary: "Telemetry collection taking too long"
```

API Reference

TelemetryCollector

Main class for metrics collection and management.

Methods

- `register_counter(name, description, labels)` : Register a counter metric
- `register_gauge(name, description, labels)` : Register a gauge metric
- `register_histogram(name, description, buckets, labels)` : Register a histogram metric
- `get_metric(name)` : Retrieve a registered metric
- `record_event(name, value, labels, metadata)` : Record a telemetry event
- `add_hook(hook_type, callback)` : Add event hook
- `start_collection()` : Start periodic collection
- `stop_collection()` : Stop periodic collection
- `get_snapshot()` : Get current telemetry snapshot

MetricsExporter

Abstract base class for metrics exporters.

Implementations

- `PrometheusAdapter` : Prometheus text format export
- `JSONExporter` : JSON format export
- `MultiExporter` : Multiple format export
- `PeriodicExporter` : Automatic periodic export

Troubleshooting

Common Issues

1. **Metrics not updating**: Check if collection is started with `start_collection()`
2. **Export failures**: Verify file permissions and disk space
3. **High memory usage**: Review metric cardinality and event retention

4. **Thread safety issues**: Use the provided thread-safe metric operations

Debug Information

Enable debug logging to troubleshoot issues:

```
import logging
logging.getLogger('utilityfog_frontend.telemetry').setLevel(logging.DEBUG)
```

Performance Tuning

- Adjust collection interval based on your needs
- Use appropriate histogram buckets for your data distribution
- Limit event retention with reasonable buffer sizes
- Monitor system resource usage

Future Enhancements

- **Remote Export**: HTTP endpoints for metrics scraping
- **Alerting Integration**: Built-in alerting rules and notifications
- **Dashboard Integration**: Grafana dashboard templates
- **Advanced Analytics**: Statistical analysis and anomaly detection
- **Distributed Tracing**: OpenTelemetry integration for request tracing