OASIS: A Framework for Real-time Instrumentation of Distributed Real-time and Embedded Systems

Dennis Feiock, Tanumoy Pati, & James H. Hill
dfeiock@iupui.edu, tpati@cs.iupui.edu, billj@cs.iupui.edu
Department of Computer and Information Sciences
Indiana University-Purdue University Indianapolis
Indianapolis, IN USA
Motivation: Real-time Instrumentation of DRE Systems

- Software instrumentation is the primary technique for analyzing distributed real-time and embedded (DRE) system quality-of-service (QoS) properties

Intrusive Software instrumentation
- Tightly coupled to system design & requires decisions about what metrics to collect to be made early in the software lifecycle

Non-Intrusive Software instrumentation
- Does not account for the distributed nature of DRE systems & hard to make real-time decisions
Software instrumentation is the primary technique for analyzing distributed real-time and embedded (DRE) system quality-of-service (QoS) properties.

**Intrusive Software instrumentation**
- Tightly coupled to system design & requires decisions about what metrics to collect to be made early in software lifecycle.

**Non-Intrusive Software instrumentation**
- Does not account for the distributed nature of DRE systems & hard to make real-time decisions.

**Challenge:** Decoupling instrumentation design concerns from overall system design while supporting real-time instrumentation concerns.
Dynamic software instrumentation enables developers to determine at runtime what metrics to collect

- i.e., does not always require decisions at design time

Believed that combining dynamic instrumentation with service-oriented methodologies can make this concern a service

- e.g., data is collected, sanitized, & ready for analysis tools

**Technology Enabler:** *Open-Source Architecture for Software Instrumentation of Systems (OASIS)*
Overview of the OASIS Architecture

Software Probe

- Collects application & system metrics autonomously, or manually
- Forwards collected metrics to Embedded Instrumentation (EI) Node
- Software probes are bound to a specific metric type
**Embedded Instrumentation (EI) Node**

- Forwards collected metrics to the Data Acquisition Controller (DAC)
- Regulates network transmission of collected metrics
- One EI Node is bound to an application context
  - e.g., function, class, component, thread, process, & etc.
Overview of the OASIS Architecture

Data Acquisition Controller (DAC)
- Stores metrics sent from an EI Node
- Supports multiple data handlers that operate on collected metrics
  - e.g., archive metrics, publish metrics on different middleware technologies
Test and Experimentation (T&E) Manager

- Main entry point into OASIS architecture for performance analysis tools
- Naming service for the DACs, the DAC services, and collected metrics
Overview of the OASIS Architecture

Performance Analysis Tools

- User-defined tools that analyze collected metrics
- Pull metrics from the DAC, or register with a DAC service to receive metrics in real-time
- Modify the behavior of software probes at runtime
Storage & analysis of collected metrics occur outside of application domain to minimize impact on application behavior & performance.
OASIS Data Collection Architecture (1/3)
OASIS Data Collection Architecture (2/3)

Entities are bound to the target programming language

Semantics dictate locality-constrained communication
OASIS Data Collection Architecture (3/3)

Entities are bound to an architecture (e.g., CORBA, raw sockets, DDS)

DAC can expose many different endpoints
DAC supports different data handlers that operate on metrics as they are received.

Examples: TENA data handler, DDS data handler, SQLite data handler, Websocket data handler, & etc.
Because OASIS is designed to operate with many technologies & languages, we provide a specification for encoding metrics sent from a software probe to an EINode.

The header below is *always* encoded as a sequence (or array) of octets regardless of the target technology & language.

<table>
<thead>
<tr>
<th>Name (size)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>probeUUID (16)</td>
<td>User-defined UUID</td>
</tr>
<tr>
<td>tsSec (4)</td>
<td>Seconds value of timestamp</td>
</tr>
<tr>
<td>tsUsec (4)</td>
<td>Micro-seconds value timestamp</td>
</tr>
<tr>
<td>sequenceNum (4)</td>
<td>Metric sequence number</td>
</tr>
<tr>
<td>probeState (4)</td>
<td>Probe-defined value (if applicable)</td>
</tr>
<tr>
<td>dataSize (4)</td>
<td>Size of metric data</td>
</tr>
<tr>
<td>data (dataSize)</td>
<td>The actual data content in binary format</td>
</tr>
</tbody>
</table>
The EINode sends collected metrics to the DAC using the specified header below. The encoding of the header is completely dependent on the target architecture used to send data to the DAC. e.g., CORBA, raw sockets, DDS.

<table>
<thead>
<tr>
<th>Name (size)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byteOrder (1)</td>
<td>Seconds value of timestamp</td>
</tr>
<tr>
<td>versionNumber (2)</td>
<td>Micro-seconds value timestamp</td>
</tr>
<tr>
<td>einodeUUID (16)</td>
<td>EINode’s unique id</td>
</tr>
<tr>
<td>probeData (?)</td>
<td>Binary data received from software probe</td>
</tr>
</tbody>
</table>
Software probes have metadata document that provides details about itself & collected metrics

XML Schema Definition format

Metadata is requested when software probe registers with EINode

EINode forwards metadata to DAC for storage and acquisition at a later time

```xml
<?xml version='1.0' ?>
<xsd:schema>
  <xsd:element name='probeMetadata' type='component.state' />
  <xsd:complexType name='component.state'>
    <xsd:annotation id='metadata'>
      <xsd:appinfo>
        0A499B6B-7250-4B88-B9DC-360D32639081
      </xsd:appinfo>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name='component' type='xsd:string' />
      <xsd:element name='state' type='xsd:integer' />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```
Software probes have metadata document that provides details about itself & collected metrics

XML Schema Definition format

Metadata is requested when software probe registers with EINode

EINode forwards metadata to DAC for storage and acquisition at a later time

<?xml version='1.0' ?>
<xsd:schema>
    <xsd:element name='probeMetadata'
        type='component.state' />
    <xsd:complexType name='component.state'>
        <xsd:annotation id='metadata'>
            <xsd:appinfo>
                0A499B6B-7250-4B88-B9DC-360D32639081
            </xsd:appinfo>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:element name='component'
                type='xsd:string' />
            <xsd:element name='state'
                type='xsd:integer' />
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>

Allows different performance analysis tools to build parsers on the fly independently of the execution context’s technology & programming language
The Probe Definition Language (PDL) is used to define what metrics are collected by a software probe.

Software probes can be inherited from other software probes to capture hierarchical relationships in metrics.

- e.g., LinuxProcessProbe is a ProcessorProbe

```c
probe ProcessorProbe
{
    uint64 idle_time;
    uint64 system_time;
    uint64 user_time;
};

probe LinuxProcessorProbe
{
    uint64 nice_time;
    uint64 iowait_time;
    uint64 irq_time;
    uint64 soft_irq_time;
};

probe WindowsProcessorProbe
{
    // Windows-specific metrics
};
```
The Probe Definition Language (PDL) is used to define what metrics are collected by a software probe.

Software probes can be inherited from other software probes to capture hierarchical relationships in metrics.

- e.g., LinuxProcessProbe is a ProcessorProbe

PDL is parsed to generate stubs & base implementation for the software probe.

Autonomous probe implementations extend base implementation & implement `handle_collect_data()` method.
class PROCESSORPROBE_IMPL_Export ProcessorProbe_Impl :
#if defined (_WIN32)
  public WindowsProcessorProbe_Impl
#else
  public LinuxProcessorProbe_Impl
#endif
{
  ///...

  /// Collect a new set of data.
  virtual int handle_collect_data (void);

private:
#if !defined (_WIN32)
  /// Parser for extracting information from /proc/stat.
  ACE_Auto_Ptr <Procstat_Parser> parser_;  
#endif
};

Extend base implementation depending on target platform
class PROCESSORPROBE_IMPL_Export ProcessorProbe_Impl :
#if defined (_WIN32)
    public WindowsProcessorProbe_Impl
#else
    public LinuxProcessorProbe_Impl
#endif
{
    ///...

    /// Collect a new set of data.
    virtual int handle_collect_data (void);

private:
#if !defined (_WIN32)
    /// Parser for extracting information from /proc/stat.
    ACE_Auto_Ptr <Procstat_Parser> parser_;
#endif
};

Method for autonomous collection of metrics
The TENA Middleware combines the concepts of CORBA distributed objects and anonymous publish-subscribe data dissemination to provide a programming abstraction known as a stateful (i.e., “full of state or data”) distributed object (SDO).

The DAC contains a TENA bridge, which is a data handler that converts OASIS data to a SDO. The bridge is responsible for publishing data to correct SDO.
WebSocket is a web technology providing for bi-directional, full-duplex communications channels, over a single Transmission Control Protocol (TCP) socket.

The DAC contains a data handler that exposes a WebSocket endpoint for pushing data.

Prototype dashboard leverages HTML5; targeting cloud-computing applications.
Preliminary Performance Results: Data Size vs. Throughput

Performance Tests

- **No DAC** – No data sent to the DAC (i.e., no network overhead)
- **No Handler** – Data sent to DAC, but no data handler loaded
- **TAO Handler** – Data sent to DAC, then forwarded to data handler
Future Research Directions

- Identifying patterns for real-time instrumentation of applications with emphasis on increasing data collection without compromising application performance
- Understanding how multi-core architectures can play a part in supporting real-time instrumentation of software systems
- Explore integrating OASIS with dynamic binary instrumentation tools & frameworks
From the current design & implementation of OASIS, we have learned the following:

- Separating the concerns of data collection, data handling, and data analysis has allowed us to focus on addressing individual challenge problems in real-time instrumentation in isolation instead of in totality.

- We still have more performance benchmarks to execute & design optimizations to implement.
  - e.g., application utilization, network utilization, system impact, & etc.

- We have unofficially discovered a new software design pattern for integrating heterogeneous systems named the Standard Flat-rate Envelope Pattern.

There still remains many open-challenges in realizing a general-purpose middleware solution that can fit the needs of many different application domains!