Minimally Intrusive Real-time Software Instrumentation

Gerardo Pardo-Castellote, Ph.D.
Chief Technology Officer

Andrea Sorbini
Software Engineer

www.rti.com
Outline

• Background
• Monitoring of Distributed Applications
• Application Instrumentation Approaches
• DDS-based Application Instrumentation
  – API
  – An example
  – Performance
  – A simple instrumented application, visualized.
• Conclusions
Background

• This presentation describes an approach to observing & monitoring *internal application state (IAS)* in the context of distributed and real-time systems

• *IAS* refers to the applications state known to the application but very hard or impossible to observe from its external interfaces

• The characteristics and format of the IAS are application specific.
  – Examples are state variables internal to the application code. Lengths of internal queues, values of internal loop counters, etc.
Why is observing IAS important?

• During development & testing
  *If you can’t see it you can’t understand it*
  *… If you can’t understand it you cannot fix it*

• During integration and operation
  – Detect or anticipate operational problems
  – React to changes in behavior, adjust operational parameters or take corrective or remedial action
  – Ensure quality of service is maintained

• Post-operations
  – Provide data for validation and certification
  – Diagnose cause of application fault or QoS violations
MOTIVATION AND NEED

Common Practice:

- State reporting interface
- Readiness reporting interface
- Heartbeat interface
- Fault reporting interface
- Performance reporting interface
- Status reporting interface

- Run-time Instrumentation Data propagated:
  - via problem-specific interfaces
  - to specific clients
  - using a variety of middleware technologies
  - via different service APIs
  - using service-specific data formats

- Approach requires both application awareness of:
  - what instrumentation data is needed and
  - the system-level context of what the data will be used for

- As instrumentation requirements evolve, additional problem-specific interfaces and data formats are created

- Components using instrumented data often need to gather the data from multiple direct and indirect sources

Better approach:

- Instrumentation reporting interface
  - Reporting of:
    - State events
    - Readiness events
    - Heartbeat events
    - Fault events
    - Performance events
    - Status events
    - ...  

- Common application-side event reporting API
- Clients register for sets of events
- Decoupling of instrumented apps and client apps
- Metadata approach for interpreting / translating event data

Source: Paul V. Werme, NSWCDD. OMG C4I Briefing, December 2006. c4i/c4i/06-12-09
Distributed Twist

• There are many applications in the system:
  – Need to collect information from multiple applications simultaneously
  – Need to support multiple Platforms/Languages/OSs
• Overall system state is composition of multiple application state:
  – Need to have a single monitoring app receive data collected from multiple sources
• There are many orthogonal ‘live’ uses for the collected instrumentation data:
  – Need to have more than one application observe and/or process the collected information
Real-Time Twist

• Addition of instrumentation, data collection, and transmission must not interfere with and degrade application determinism
• Impact on CPU and memory should be minimized
• Impact on network bandwidth should be minimized
State of the Art: Data Collection

- **Code Instrumentation Interfaces.**
  - App developers instrument code with calls to the instrumentation API passing relevant data.
  - Examples: RTI’s StethoScope (now Wind River’s Workbench Run-Time Analysis Data Viewer), Lawrence Livermore Berkeley Laboratory’s NetLogger, Apache Logging Project’s log4j/log4cxx, OpenGroup ARM

- **Compile-time tools for Code-injection/Modification.**
  - Instrumentation inserted during the compilation or linking phase. The instrumentation can be single-focus for the tool (e.g. profiling and memory checking tools), or generic/programmable via a instrumentation spec file/prgram
  - Examples (single-focus): IBM/Rational Purify and Quantify and Linux Valgrind

- **Dynamic Patching/Code Injection Tools.**
  - Patches the code when it is loaded into memory for execution.
  - Examples: RTI’s MemScope and TraceScope (now Wind River’s Workbench Run-Time Analysis Memory and Function Trace), OC Systems’ Aprobe, and Intel’s PIN tool.

- **Operating-Systems/Platform-Supported Tools.**
  - Attach to instrumentation hooks/APIs provided by the executing environment (processor, operating system, or virtual machine)
  - Examples: Solaris “dtrace,” Java Virtual machine Tool Interface (JVMTI).
<table>
<thead>
<tr>
<th>Code Instrumentation Interfaces</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Any kind of application data.</td>
<td>Requires source modification.</td>
</tr>
<tr>
<td></td>
<td>Minimal overhead.</td>
<td>Cannot instrument legacy applications which cannot be recompiled.</td>
</tr>
<tr>
<td></td>
<td>Instrumented code can be validated (matches executable).</td>
<td>Cannot instrument third-party libraries or kernel modules.</td>
</tr>
<tr>
<td></td>
<td>High control and fidelity on the timing</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compile-Time Tools</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No source modification.</td>
<td>Cannot collect some application data.</td>
</tr>
<tr>
<td></td>
<td>Minimal overhead.</td>
<td>Cannot instrument legacy applications which cannot be recompiled.</td>
</tr>
<tr>
<td></td>
<td>Instrumented code can be validated (matches executable)</td>
<td>Cannot instrument third-party libraries or the kernel.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamic Patching Tools</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No source or executable modification</td>
<td>More limited on the kinds of data that can be collected</td>
</tr>
<tr>
<td></td>
<td>Can be used for legacy applications, third-party libraries or kernel code.</td>
<td>Hard/impossible to validate: Modifies executing code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lower performance.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operating System or Platform-supported Tools</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No source or executable modification</td>
<td>Most limited on the kinds of data that can be collected.</td>
</tr>
<tr>
<td></td>
<td>Can be used for legacy applications, third-party libraries or kernel code.</td>
<td>Timing of collection cannot be precisely controlled.</td>
</tr>
<tr>
<td></td>
<td>Instrumented code can be validated (matches executable).</td>
<td>Can miss important events.</td>
</tr>
<tr>
<td></td>
<td>Minimal impact on execution.</td>
<td>Extremely platform specific. Limited and inconsistent availability.</td>
</tr>
</tbody>
</table>
State of the Art: Collected Data Distribution

- Home-grown protocols under an instrumentation API
- File-based replication mechanisms to share data collected by instrumentation
- Tool-specific network protocols.
  - NetLogger (point to point to a central server)
  - Ganglia [22] (unreliable multicast with no filtering)
- Use of standard management protocols (SNMP, CIM)
  - SNMP (e.g. used by OpenNMS, Nagios, Zabbix)
- Standard middleware.
  - JMS. E.g. using the JMS appender in Log4j
  - DDS. E.g. OMG AMSM specification. OMG Application Instrumentation RFP
<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home-grown protocols</td>
<td>Can be customized and optimized for a specific problem/domain</td>
<td>Costly to develop and maintain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 3rd party tools available</td>
</tr>
<tr>
<td>File-based replication</td>
<td>Simple</td>
<td>Not good for live or real-time data</td>
</tr>
<tr>
<td></td>
<td>Can use OS-bundled technology</td>
<td>Heavy use of network bandwidth.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be smart about what to send to whom and when.</td>
</tr>
<tr>
<td>Tool-specific network</td>
<td>Easy to use when supported as comes supported by the tool.</td>
<td>Cannot be used to combine information from multiple</td>
</tr>
<tr>
<td>protocols</td>
<td></td>
<td>sources and tools</td>
</tr>
<tr>
<td>Standard</td>
<td>Well established protocols.</td>
<td>Protocols not optimized for real-time data or for</td>
</tr>
<tr>
<td>management</td>
<td>Ecosystem of tools available to gather and display the information</td>
<td>scalability to many consumers of the information. E.g.</td>
</tr>
<tr>
<td>protocols</td>
<td>Integrates well with monitoring of network appliances</td>
<td>SNMP offers no reliability for “events” and no QoS of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>filtering on the writer side</td>
</tr>
<tr>
<td>Standard</td>
<td>Highest performance (depending on middleware used)</td>
<td>Requires adaptation of each tool to the chosen</td>
</tr>
<tr>
<td>middleware</td>
<td>Can integrate data from multiple sources/tools</td>
<td>middleware.</td>
</tr>
<tr>
<td></td>
<td>Can leverage ecosystem of tools</td>
<td></td>
</tr>
</tbody>
</table>
DDS-Based App- Instr components

- Local Instrumentation API
- Mapping of Instrumentation API to DDS Types and Entities
- Distribution of information via standard DDS-RTPS protocol.
DDS-Based Application Instrumentation

Application
InstrumentationLib

Application
InstrumentationLib

Application
InstrumentationLib

Open API

Open Wire Protocol (DDS-RTPS)

Visualize
Record
Process
Analyze
Custom
Instrumented Application
(Mission critical computer)

Real-Time Application Process
- Real-Time Application Code
- Instrumented Code
  - Configure
  - Copy variables
- Instrumentation Library
  - DDS
    - process & consolidate
  - distribute
- Circular Buffers

DDS Middleware
- DDS

Host-Side Tools and Services
- Instrumentation Visualization Tools
- Resource Manager
- Database
- Recorder

Existing LAN Network

Time-critical threads

Low-priority threads
Instrumentation Service API
<table>
<thead>
<tr>
<th>Classifier</th>
<th>Purpose</th>
<th>Related DDS Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Entry point to initialize an configure Application Instrumentation</td>
<td>DomainParticipant</td>
</tr>
<tr>
<td>Schema</td>
<td>Defines the type associated with a single unit of observation. Allows grouping of several data-elements into an aggregate that will be sampled as a unit.</td>
<td>Mutable Type</td>
</tr>
<tr>
<td>Session</td>
<td>Creates the scope for a set of related observations.</td>
<td>Publisher</td>
</tr>
<tr>
<td>ObjectGroup</td>
<td>Creates the scope for a set of related observations sharing a common schema and QoS.</td>
<td>DataWriter</td>
</tr>
<tr>
<td>ObservableObject</td>
<td>Represents a single unit of observation. Allows grouping of related data-elements.</td>
<td>Instance</td>
</tr>
<tr>
<td>Observation</td>
<td>The value of an ObservableObject at a specific point in time</td>
<td>Sample</td>
</tr>
<tr>
<td>DataProcessorPlugin</td>
<td>Function object that can process Observations and decide whether they should be distributed</td>
<td>N/A</td>
</tr>
<tr>
<td>DataProcessor</td>
<td>Concrete instantiation of a DataProcessorPlugin bound to a particular ObservableObjectGroup</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Example: Instrumenting prime-number computation app.

```c
// Compute and save the primes less than “n”
void primes( FILE *fp, long n )
{
    long i, k;
    long m = (long) sqrt( (double) n );
    char *prime = (char *) malloc(n+1);

    for (i=2; i<=m; i++) {
        if (prime[i]) {
            for (k=i*i; k<=n; k+=i) {
                if ( prime[k] ) {
                    prime[k]=FALSE;
                }
            }
        }
    }

    for (i=0; i<n+1; i++) {
        if (prime[i]) {
            fprintf(fp, "%ld : %ld\n", primes_written, (long)i);
        }
    }
}
```
Example: Instrumenting prime-number computation app.

```c
// Compute and save the primes less than “n”
void primes( FILE *fp, long n )
{
    long i, k;
    char *prime = (char *)malloc(n+1);
    long m = (long) sqrt( (double)n );

    for (i=2; i<=m; i++) {
        if (prime[i]) {
            for (k=i*i; k<=n; k+=i) {
                if ( prime[k] ) {
                    prime[k]=FALSE;
                }
            }
        }
    }

    for (i=0; i<n+1; i++) {
        if (prime[i]) {
            fprintf(fp, "%ld : %ld\n", (long)primes_written, (long)i );
        }
    }
}
```
Example: Instrumenting prime-number computation app.

// Compute and save the primes less than “n”
void primes( FILE *fp, long n )
{
    long i, k;
    long m = (long) sqrt( (double)n );
    char *prime = (char *) malloc(n+1);

    for (i=2; i<=m; i++) {
        if (prime[i]) {
            for (k=i*i; k<n; k+=i) {
                if ( prime[k] ) {
                    prime[k]=FALSE;
                }
            }
        }
    }

    for (i=0; i<n+1; i++) {
        if (prime[i]) {
            fprintf(fp, "%ld : %ld\n", (long)primes_written, (long)i );
        }
    }
}
Example: Instrumenting prime-number computation app.

// Compute and save the primes less than “n”
void primes( FILE *fp, long n )
{
    long i, k;
    long m = (long) sqrt( (double)n );
    char *prime = (char *)malloc(n+1);

    for (i=2; i<=m; i++) {
        if (prime[i]) {
            for (k=i*i; k<=n; k+=i) {
                if ( prime[k] ) {
                    prime[k]=FALSE;
                }
            }
        }
    }

    for (i=0; i<n+1; i++) {
        if (prime[i]) {
            fprintf(fp, "%ld : %ld\n", (long)primes_written, (long)i);
        }
    }
}
Example Schema Definition

```xml
<schema_library name="SchemaLib">
  <schema name="FindPrimesStatus">
    <member name="stage" type="string8"/>
    <member name="higuest_number_searched" type="uint32"/>
    <member name="outer_loop_step_count" type="uint32"/>
    <member name="outer_loop_step" type="uint32"/>
    <member name="inner_loop_step" type="uint32"/>
    <member name="eliminated_count" type="uint32"/>
    <member name="print_loop_step" type="uint32"/>
    <member name="written_prime_count" type="uint32"/>
  </schema>
</schema_library>
```
Schemas can also be defined in the code

```c
schema = AppInstService_createSchema(service, "FindPrimesSchema", NULL);

AppInstSchema_addPrimitiveField(schema, &id_stage, "stage", APPINST_TYPE_UINT32);

AppInstSchema_addPrimitiveField(schema, &id_highest_number_searched, "highest_number_searched", APPINST_TYPE_UINT32);

AppInstSchema_addPrimitiveField(schema, &id_outer_loop_step_count, "outer_loop_step_count", APPINST_TYPE_UINT32);

...```

4/23/2012 © 2012 RTI • COMPANY CONFIDENTIAL 23
Example Instrumentation: Initialize

```c
service = AppInstService_new("MyService", NULL);

session = AppInstService_createSession(service, "MySession", NULL);

group   = AppInstSession_createObjectGroup(session, "MyGroup",
                                         schema, NULL, NULL);

obj     = AppInstObjectGroup_createObject(group, "MyObject");
```
Example Instrumentation: Save Observations

AppInstObservableObject_setPrimitiveFieldSTRING8(
    obj, id_stage, stage);

AppInstObservableObject_setPrimitiveFieldUINT32(
    obj, id_higuest_number_searched, higuest_number_searched);

AppInstObservableObject_setPrimitiveFieldFLOAT32(
    obj, id_outer_loop_step_count, outer_loop_step_count);

AppInstObservableObject_setPrimitiveFieldUINT32(
    obj, id_outer_loop_step, outer_loop_step)

...
Instrumentation Benchmarks

Observation rate that can be achieved versus number of fields in the ObservableClass

Observations per second

Number of fields in the ObservableClass

Intel Eight-Core i7 @ 2.93Ghz
8Gb RAM
Ubuntu Linux 10.10,
kernel 2.6.35-25-generic
RTI DDS 4.5d
Instrumented application demo

• See internal state from Excel
• Record and export internal state to HTML
Conclusion

• Software application instrumentation is important for embedded systems
• There are challenges in real-time and distributed systems
• Building it as a mapping into a standard like DDS shows promise.