USING NON-INTRUSIVE INSTRUMENTATION TO ANALYZE ANY DISTRIBUTED MIDDLEWARE IN REAL-TIME

by

Nyalia Lui

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science

Department of Computer and Information Science

Indianapolis, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. James H. Hill, Chair

Department of Computer and Information Science

Dr. Rajeev R. Raje

Department of Computer and Information Science

Dr. Fengguang Song

Department of Computer and Information Science

Approved by:

Dr. Mihran Tuceryan

Dedicated to my friends and family who have helped me get through my graduate school experience.

ACKNOWLEDGMENTS

Thank you to Dr. Hill for your guidance and support throughout this experience. Thank you to the members of my committee, Dr. Raje and Dr. Song, who stayed with me during my time at IUPUI. Thank you to Nicole and the rest of the staff in the Computer Science department office for answering my endless stream of questions. A special thanks to my friends and family for the numerous late night motivational speeches and days we would spend working with muted microphones on video conference calls. I could not have done this without you. Thank you to the Department of Computer and Information science for providing me the opportunity to conduct research.

Lastly, thank you to the Australian Defence Science and Technology Group (DSTG) for funding this research project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

SDMM	Standards-based Distributed Middleware Monitor
DBI	Dynamic Binary Instrumentation
gRPC	gRemote Procedure Call
DDS	Data Distribution Service
ADT	Abstract Data Type

ABSTRACT

Dynamic Binary Instrumentation (DBI) is one way to monitor a distributed system in real-time without modifying source code. Previous work has shown it is possible to instrument distributed systems using standards-based distributed middleware. Existing work, however, only applies to a single middleware, such as CORBA.

This thesis therefore presents a tool named the Standards-based Distributed Middleware Monitor (SDMM), which generalizes two modern standards-based distributed middleware, the Data Distribution Service (DDS) and gRemote Procedure Call (gRPC). SDMM uses DBI to extract values and other data relevant to monitoring a distributed system in real-time. Using dynamic instrumentation allows SDMM to capture information without *a priori* knowledge of the distributed system under instrumentation. We applied SDMM to systems created with two DDS vendors, RTI Connext DDS and OpenDDS, as well as gRPC which is a complete remote procedure call framework. Our results show that the data collection process contributes to less than 2% of the run-time overhead in all test cases.

1. INTRODUCTION

Distributed middleware, such as the Data Distribution Services (DDS) [23] and gRemote Procedure Call (gRPC) [27] are designed for large distributed systems that have many endpoints and components. An important aspect of these systems is real-time monitoring because it ensures the system is functioning properly and meeting its performance requirements [18]. For example, real-time monitoring can provide feedback to stakeholders that the distributed system is offline, missing end-to-end deadlines, and show the data values to events when deadlines are missed. Regardless of how system architects apply real-time monitoring, it is vital to providing critical information to stakeholders.

There are two approaches to monitoring distributed systems, intrusive [10] and non-intrusive [32] instrumentation. Intrusive instrumentation is when coders modify source code to collect data, such as logging events, by recording timestamps and content. The advantage of intrusive instrumentation is instrumentation code becomes a part of the execution flow. The disadvantage to this approach is any change to the distributed system requires changes to the instrumentation code—if not complete removal—since the instrumentation is tied to specific blocks of code.

The second approach is to use non-intrusive instrumentation, which is when data is collected without modifying source code. Dynamic binary instrumentation (DBI) is one form of non-intrusive instrumentation. In DBI, code is injected at run-time into the binaries for the software under instrumentation. When the program terminates, the binaries return to their original state before undergoing DBI. The advantage of this approach is it allows stakeholders to delay decisions related to instrumentation, such as what data to collect and when to collect it, to later in the software development life cycle. The disadvantage, however, is that more execution overhead is brought into the system compared to the intrusive approach. Because non-intrusive dynamic binary instrumentation allows stakeholders to analyze third-party binaries [20], we believe it has potential to provide more visibility into distributed system behavior compared to intrusive instrumentation. An example of third-party binaries are the distributed middleware provided by a vendor.

The advantage to using distributed middleware, such DDS and gRPC, is they make programming a distributed system easier and more portable [7]. The interfaces in middleware, which all vendors comply to, are often written with patterns to the symbols. These patterns could be anything repetitive, or noteworthy, such as attaching the same symbol to procedure names or the ordering of data types in parameter lists. We refer to these patterns as programming *standards*. In the context of dynamic instrumentation, programming standards become the gateway into a distributed system.

Satyanarayana et al. [28] has shown that systems developed with CORBA [22, 30], a standards-based distributed middleware, can be monitored with DBI. Although the existing work shows that DBI can be used to instrument standards-based distributed middleware, there is unanswered questions:

1. Can DBI be used to instrument other standards-based distributed middleware such as DDS and gRPC?
2. If so, can the individual solutions be generalized into one common approach?

Based on the questions above, we developed a generalized approach for non-intrusive instrumentation of distributed middleware. The approach is realized in a tool called the *Standards-based Distributed Middleware Monitor (SDMM)*. To use SDMM, the user supplies a configuration file that indicates the distributed middleware, the binary files for the system under instrumentation, and the interface definition file that the distributed middleware use. This is a generalized approach since any distributed middleware and relevant binaries can be represented in the configuration file.

SDMM will then discover the methods used for initializing data and component communication which are defined within the interfaces of distributed middleware. Those methods are entry points into the system, which allows SDMM to identify the events passed between components and extract information in real-time for analytical purposes. Our work therefore extends the work completed by Satyanarayana et al. [28] in that SDMM supports any type of distributed middleware if the developer can define a configuration file that matches the above specifications.

The main contributions of this thesis are as follows:

- It presents an approach to non-intrusively instrument any distributed system without any *a priori* knowledge of the target system by leveraging programming standards;
- It discusses a strategy for identifying entry points for instrumentation in compiled distributed middleware such as DDS and gRPC;
- It presents a generalized approach to represent any distributed middleware using a configuration file and object composition;
- It presents an approach for non-intrusively extracting event data sent using a distributed middleware in real-time without *a priori* knowledge of the events; and
- It presents results of SDMM for run-time performance, CPU usage, and memory usage.

We evaluate SDMM on a distributed system from the shipboard computing domain. Three versions were created with two DDS vendors, RTI Connext DDS [2] and OpenDDS [1], as well as gRPC. We executed those systems with SDMM for 10 minute periods. Our results show that SDMM can extract the values from events communicated between components and the tool accounts for less than 2% of the total run-time in all test cases.

1.1 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 discusses related works that cover instrumentation of middleware. Chapter 3 introduces the DBI frameworks Pin, Pin++, and the case study used in this thesis. Chapter 4 presents challenges in non-intrusively instrumenting any distributed middleware and describes the design behind SDMM; Chapter 5 presents experimental results and evaluates the performance of SDMM. Chapter 6 discusses the limitations and future research directions to SDMM. Finally, Chapter 7 provides concluding remarks.

2. RELATED WORKS

This chapter offers a review of techniques others have created to instrument or analyze distributed middleware. In particular, we compare our work to tools that use intrusive and non-intrusive instrumentation. Table 2.1 also provides a summary of related work and classifies them as intrusive or non-intrusive tools.

Table 2.1. Summary of related work classified by intrusive or non-intrusive tools.

Tool	Intrusive	Reference
Scalasca	yes	[34]
GCov	yes	[6]
Component Port Monitor	no	[28]
Data Distribution Service	no	[26]
NetLogger	no	[13]
Wireshark	no	[25]

2.1 Intrusive Instrumentation Approaches

The theme of intrusive instrumentation tools is that they require modifications to source code. SCALASCA [34], for example, is designed to instrument high performance computing middleware such as MPI and OpenMP. Scalasca analyzes distributed systems by inserting instrumentation code at the entry and exit of MPI or OpenMP function calls with their own compiler named Score-P.

Another example is GCov [6], from the GNU Project, which is a tool that calculates program code coverage. GCov will annotate C/C++ source code when the GCov compiler flag is enabled. The annotated source files are passed to the GCov command line tool where instrumentation occurs and code coverage results are written to stdout.

A main requirement for intrusive approaches is the availability of source code so annotations are made at compile-time. Moreover, any changes to the source code also requires that annotations are reapplied which can become a repetitive process for system maintainers. A key difference between these approaches and SDMM is that SDMM does not analyze source code. Instead, SDMM analyzes binary files at run-time.

The problem with the intrusive approach is gaining access to the source code can be difficult when using closed-source libraries or third-party middleware. The pros to analyzing binary files is that binaries are always available on a machine. The cons to analyzing binary files, however, is that instrumentation is injected and executed at run-time which can increase run-time overhead.

2.2 Non-intrusive Instrumentation Approaches

Non-intrusive instrumentation tools refers to software that does not modify source code. Examples of non-intrusive instrumentation are the Component Port Monitor [28], Net Logger [13], Wireshark [25], and even the Data Distribution Service [26] comes with instrumentation methodology.

Satyanarayana et al. [28] created a Pintool, which SDMM was based on, called the Component Port Monitor (CPM) that monitors CORBA events in real-time. Their Pintool leverages CORBA coding standards to non-intrusively discover events and extract data from methods prefixed with *push_*. The authors successfully instrument event data in real-time and show that data collection requires an average of 2 seconds. The differences between their work and ours is twofold: (1) Our work extends CPM such that SDMM non-intrusively instruments any distributed middleware instead of a single specification. (2) Our work generalizes the design so new distributed middleware can be easily supported for real-time monitoring.

Other tools such as NetLogger identify performance bottlenecks by using an API to control the instrumentation process [13]. NetLogger instruments by simplifying log output and display results with several visualizations. In contrast to NetLogger, SDMM uses configuration files that represent the system instead of an API to instrument a distributed middleware.

Wireshark is an open-source tool that analyzes network packets in numerous protocol formats [25]. Example protocol formats are the Real-time Publish-Subscribe protocol [24] which is used in DDS and Protocol Buffers [9] used in gRPC. SDMM is different from

Wireshark because it does not inspect low-level network packets but instead inspects the events passed between components.

Lastly, the Data Distribution Service (DDS) also has methodology for instrumenting itself [26]. DDS's instrumentation listens to network communication between the objects responsible for producing and consuming events, the *DataWriter* and *DataReader*. SDMM is similar in that it instruments on communication methods such as *write()* and *take()*. Unlike DDS instrumentation that is only for its specification, SDMM is designed as a general tool to instrument any distributed middleware.

3. BACKGROUND

This chapter presents the instrumentation frameworks we used to analyze binary images and monitor a running distributed system. We also introduce the case study and distributed middleware used throughout this thesis. The middleware is described in the context of the case study.

3.1 Pin

Intel's Pin is a *dynamic binary instrumentation* (DBI) framework for the IA-32 and x86-64 instruction-set architectures. Developers use Pin to create analysis tools, called Pintools, that can instrument a C/C++ program at different levels: binary image level, routine level, and instruction level. Pintools are developed independent from the target program and are compiled into separate shared binaries. To instrument a C/C++ program, the shared binary file is loaded into the Pin environment and executed at runtime.

```
static UINT64 icount = 0;
VOID docount() { icount++; }

VOID Instruction(INS ins, VOID *v) {
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

ofstream OutFile;
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o", "inscount.out", "specify
    ↪ output file name");
VOID Fini(INT32 code, VOID *v) {
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl;
    OutFile.close();
}

INT32 Usage() {
    cerr << "This tool counts the number of dynamic instructions executed" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

int main(int argc, char * argv[]) {
    if (PIN_Init(argc, argv)) return Usage();
24
25      OutFile.open(KnobOutputFile.Value().c_str());
26
27      INS_AddInstrumentFunction(Instruction, 0);
28      PIN_AddFiniFunction(Fini, 0);
29      PIN_StartProgram();
30
31      return 0;
32  }
```

Listing 3.1. Pin example that counts the number of instructions in a program.

Listing 3.1 shows an example Pintool that counts the number of instructions in a program. Lines 1 & 2 define a callback function, *docount()*, that increments a counter. Lines 4-6 define the *Instruction()* procedure used to inject *docount()* to every unique instruction in the binary. Lines 8-20 define the *Fini()* function which is injected just before program exit and another procedure that prints a helper message. Finally, lines 22-32 define the main function that is responsible for starting the instrumentation process, injecting the above procedures into the binary image, and executing the target program.

3.2 Pin++

The original Pin framework is fragile, rigid, hard to reuse, and difficult to understand [15, 28]. To address those problems, Hill et al. [15] extended Pin, which originally had a C-like interface, into the C++ runtime. Named Pin++, their framework wraps many low-level Pin procedures and symbols around C++ objects. The authors show that Pintools written in Pin++ have a reduction in cyclomatic complexity, do not introduce additional overhead, and improve performance in some cases. For these reasons we use Pin++, and consequently Pin, as the instrumentation framework behind our approach.

There are three pieces to writing a Pintool in Pin++, the `Callbacks`, `Instrument`, and `Tool`. `Callbacks` are objects with the methods used for instrumenting a C/C++ program. These methods are injected into the binary images at run-time. The developer defines when and where to insert `Callbacks` for instrumentation within the `Instrument` class. The `Tool` objects are responsible for starting the instrumentation process, performing any tasks before the program terminates, and usually contains the `Instruments`. Pintools written with

Pin++ must have one or more `Tool` instances but may have any number of `Instruments` and `Callbacks`.

```
class docount : public OASIS::Pin::Callback <docount(void)> {
public:
  docount (void)
    : count_ (0) { }

  void handle_analyze (void) {
    ++ this->count_;
  }

  UINT64 count (void) const {
    return this->count_;
  }

private:
  UINT64 count_;
};

class Instruction : public OASIS::Pin::Instruction_Instrument <Instruction> {
public:
  void handle_instrument (const OASIS::Pin::Ins & ins) {
    this->callback_.insert (IPOINT_BEFORE, ins);
  }

  UINT64 count (void) const {
    return this->callback_.count ();
  }

private:
  docount callback_;
};

class inscount : public OASIS::Pin::Tool <inscount> {
public:
  inscount (void) {
    this->enable_fini_callback ();
  }

  void handle_fini (INT32 code) {
    std::ofstream fout (outfile_.Value ().c_str ());
    fout.setf (ios::showbase);
    fout << "Count␣" << this->instruction_.count () << std::endl;

43      fout.close ();
44    }
45
46  private:
47    Instruction instruction_;
48  };
49
50  KNOB <string> inscount::outfile_ (KNOB_MODE_WRITEONCE, "pintool", "o", "inscount.out", "
        ↪specify␣output␣file␣name");
```

Listing 3.2. Pin++ example that counts the number of instructions in a program.

Listing 3.2 shows the same Pintool from Listing 3.1 written in Pin++. Lines 1-16 define the `Callback` class for incrementing the counter to an instruction. Lines 18-29 define the `Instrument` class used to setup and inject instrumentation into the binary images. Specifically, line 21 inserts the *docount* object to every unique instruction in the binary. Lines 32-48 create the `Tool` object that is responsible for starting the instrumentation process and defines what the Pintool should do when the program terminates in the *handle_fini()* method. Finally, line 50 defines the command line option to capture the output file name.

3.3 The SLICE Scenario

The case study we refer to throughout this thesis is the SLICE scenario, which is a distributed system from the domain of shipboard computing environments. It has been used in prior research for emulating workloads for early testing of real-time distributed systems [14], evaluating system execution modeling tools [31], conducting formal verification [16], and highlighting challenges of searching the deployment and configuration solution space of real-time distributed systems [17]. With respect to everyday use, similar distributed systems are used by the navy for shipboard computing [3]. A high-level diagram of the SLICE scenario is shown in Figure 3.1. Figure 3.1 illustrates that seven component instances form the SLICE scenario: (from left-to-right) *SensMain*, *SensSec*, *PlanOne*, *PlanTwo*, *Config*, *EffMain*, and *EffSec*. The directed black lines indicate communication points for event I/O between components. Lastly, each component is deployed across two nodes in the target environment. *SensMain* and *SensSec* have endpoints for event transmission over the network. *EffMain* and *EffSec* have endpoints for receiving events. The other components *PlanOne*, *PlanTwo*, and

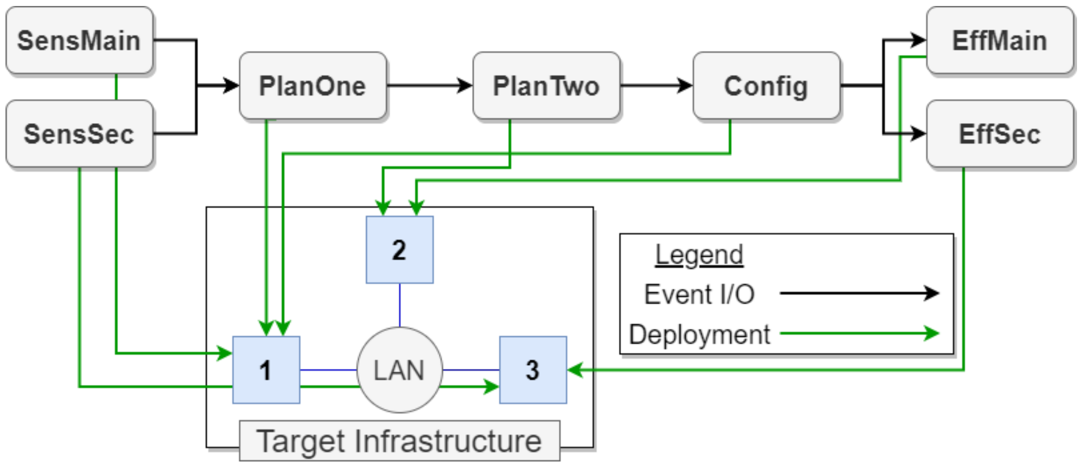


Figure 3.1. A conceptual, high-level diagram of the SLICE scenario.

Config have endpoints both for transmitting and receiving events. Pin++, from Section 3.2, is applied to the

3.4 The Distributed Middleware

The distributed middleware discussed in this thesis is the Data Distribution Service (DDS) [23] and gRemote Procedure Call (gRPC) [27]. Out of the several implementations of DDS we used RTI Connext DDS [2] and OpenDDS [1]. We treat gRPC as its own vendor because there is only one implementation to gRPC.

We selected RTI Connext DDS because the library accounts for 70% of the DDS market share [8]. We chose OpenDDS [1] version 3.14 because the library is open-source, comes with many build scripts that fit our development process, and is updated daily where as years pass until other open-source implementations are updated. We based our development on the DDS C++11 mapping for all implementations [4]. We used gRPC version 1.20 because the latest version is not compatible with the operating system on our test bed.

In the context of the component layout from Figure 3.1, representative interface definitions are shown in Listing 3.3 for DDS and Listing 3.4 for gRPC.

```
struct PlannerOneEvent {
  long eventcount;
  string name;
};

struct PlannerTwoEvent {
  long eventcount;
  string name;
};

struct ConfigEvent {
  long eventcount;
  string name;
};

struct EffectorEvent {
  long eventcount;
  string name;
};
```

Listing 3.3. An example interface definition for the events in DDS in the SLICE scenario.

```
service SLICEServer {
        rpc SendEvent (SimpleEvent) returns (google.protobuf.Empty) {}
}

message SimpleEvent {
        int64 eventcount = 1;
        string name = 2;
}
```

Listing 3.4. An example interface definition for the servers and events in gRPC in the SLICE scenario.

For DDS, *PlannerOneEvents* are produced by SensMain & SensSec and consumed by PlanOne. *PlannerTwoEvents* are produced by PlanOne and consumed by PlanTwo. *ConfigEvents* are produced by PlanTwo and consumed by Config. Lastly, the *EffectorEvents* are produced by Config and consumed by EffMain & EffSec.

For gRPC, every remote server has a single remote procedure named *SendEvent()* that takes a *SimpleEvent* as input and returns *Empty*. Empty is a class defined by gRPC to represent return type `void`. The SimpleEvent object is communicated between all endpoints of the system.

4. SDMM Design and Implementation

This chapter describes the design and implementation of the *Standards-based Distributed Middleware Monitor (SDMM)*. First we discuss the challenges in non-intrusively instrumenting standards-based distributed middleware. Second is a discussion over the different approaches to designing SDMM. Then we address each challenge in the context of the SLICE scenario case study.

4.1 Implementation Challenges

Creating a tool to non-intrusively instrument any standards-based distributed middleware is a non-trivial problem. This is because the instrumentation tool is unaware of the middleware in use, the system composition and user defined types, as well as the binaries used within the distributed system. For example, an instrumentation tool does not know what distributed middleware is used. Likewise, an instrumentation tool is unaware of the event types from Listings 3.3 & 3.4, and does not know *a priori* what binary files to parse through.

More specifically, creating a tool capable of non-intrusively instrumenting a distributed system implemented using any standards-based distributed middleware has the following challenges:

1. **Generalizing distributed architecture.** Our instrumentation tool must analyze a distributed system irrespective of the different architectures used in distributed middleware. For example, the Data Distribution Service (DDS) [23] is based on the publisher-subscriber architecture and is designed for asynchronous event communication. gRemote Procedure Call (gRPC) [27], on the contrary, is based on the remote procedure call (RPC) model where the client's execution is suspended until control returns from the RPC. The instrumentation tool does not have *a priori* knowledge of these differences between middleware and must adjust at run-time.

2. **Accounting for small differences between vendors.** We recognize that middleware vendors will have small differences that are difficult to generalize. The tool must

be able to account for these differences between vendors. For example, RTI Connext DDS [2] has several wrapper classes around symbols that are not used in OpenDDS [1].

3. **Extracting values from events.** The instrumentation tool should be able to extract values from events. In each middleware, there are user-defined data fields such as `PlannerOneEvent.eventcount` in Listing 3.3. Extracting values is a challenge because the instrumentation tool cannot call the getter or setter methods like a traditional C++ method.

4. **Discovering points of instrumentation.** The instrumentation tool will not know about the system under instrumentation. The tool, for example, will not know the produce and consume methods in DDS nor the RPC arguments in gRPC. Locating the communication methods in distributed middleware is important because they serve as the entry points into the system. The difficulty increases since different binary files, which are loaded at different times, contain the methods of interest. This is a challenge because the instrumentation tool must locate these points-of-instrumentation at run-time.

The remainder of this chapter therefore discusses how we address these challenges while designing and implementing the Standards-based Distributed Middleware Monitor.

4.2 Design Approaches

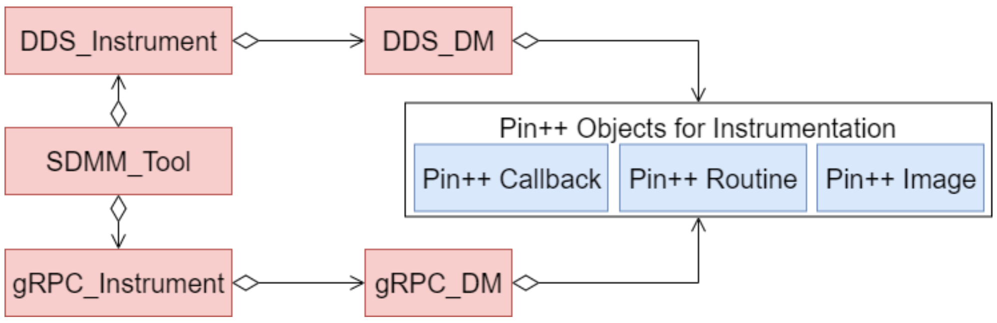


Figure 4.1. Representation of SDMM where each distributed middleware has a unique composition.

Before discussing the details of SDMM, it is necessary to understand different approaches for non-intrusively instrumenting a distributed system supported by any standards-based middleware. The following are two approaches we explored for realizing the monitor:

- **Approach 1: Create individual monitors for each middleware.** The initial design involved using a programmatic approach to instrument standards-based middleware. The instrumentation tool would select the middleware, (*e.g.*, DDS or gRPC), within its source code. This results in a unique composition, as shown in Figure 4.1, within the instrumentation code for each middleware. The tool would decide which instrumentation to use with parameter flags and other rigid code [21]. The advantage to this approach is that the target middleware is set at compile-time and the resulting monitor will be configured for that middleware.

  There are several disadvantages to this approach. First, users will need to modify instrumentation source code when they want to make changes such as selecting the middleware. This is risky because the user must be familiar with the tool's design and instrumentation framework (*e.g.*, Pin) to make those updates. Second, modifying the instrumentation source code is error prone and time consuming. Finally, in this scenario, the monitor will only work for systems using the targeted middleware and must be updated for each system with a different middleware.

- **Approach 2: Configuration file based implementation.** An external configuration file could be used instead of a unique composition. When the instrumentation software loads the configuration file, where the target middleware is indicated, it will automatically adjust components to the middleware at run-time. This is a generalized approach. The advantage to this approach is that users can analyze a distributed system with any standards-based middleware without modifying instrumentation code. The disadvantage to this approach is that it may impact performance negatively compared to Approach 1 because instrumentation code is interpreted at run-time instead of compile-time.

Based on the advantages and disadvantages of both approaches discussed above, Approach 2 was selected as the approach for non-intrusively instrumenting any standards-based distributed middleware. We selected this approach because system administrators can easily target different distributed middleware with no modification to the instrumentation code. The SDMM, however, must support the middleware.

4.3 Creating a Generic Pintool

The first challenge, as described in Section 4.1, in non-intrusively instrumenting any standards-based distributed middleware is creating a Pintool that works with multiple distributed middleware. The SDMM is focused around two architectures, publisher-subscriber with DDS and remote procedure call with gRPC. SDMM needs to determine the middleware

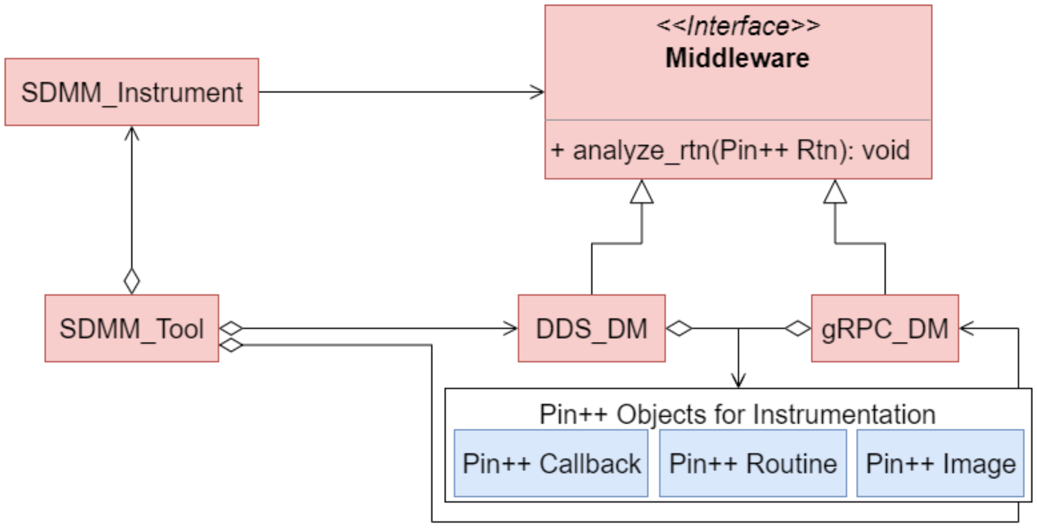


Figure 4.2. Representation of distributed middleware in SDMM using the Adapter Pattern.

at run-time and we achieve this with the Adapter Pattern [12] where each middleware are the adapters and the Pin++ objects (and consequently the Pin framework) are the adaptees. As shown in Figure 4.2, the DDS and gRPC classes inherit from the *Middleware* interface that defines methods for running Pin analysis on the binary files [20].

The DDS adapter contains code for instrumenting any distributed system written with a DDS vendor such as RTI Connext DDS or OpenDDS. For example, the DDS produce and consume methods (*write* & *take*). The gRPC adapter contains code for instrumenting routines that are not the remote procedure or event setter methods.

```
1  MIDDLEWARE = DDS
2  VENDOR = RTI
3  IDL = /path/to/SLICE.idl
4  INCLUDE = Sensor ,PlannerOne ,PlannerTwo ,Config ,Effector
```

Listing 4.1. An example SDMM configuration file.

Also in Figure 4.2, the SDMM_Tool is where the configuration file is parsed and the Pin instrumentation process begins. An example configuration file is shown in Listing 4.1 where the distributed middleware is on line 1, the DDS vendor is on line 2, and line 3 shows the absolute path to the interface definition file. Line 4 lists the binary & shared library files, referred to as images in Pin. The RTI Connext DDS example of the SLICE scenario had all symbols in the component executables.

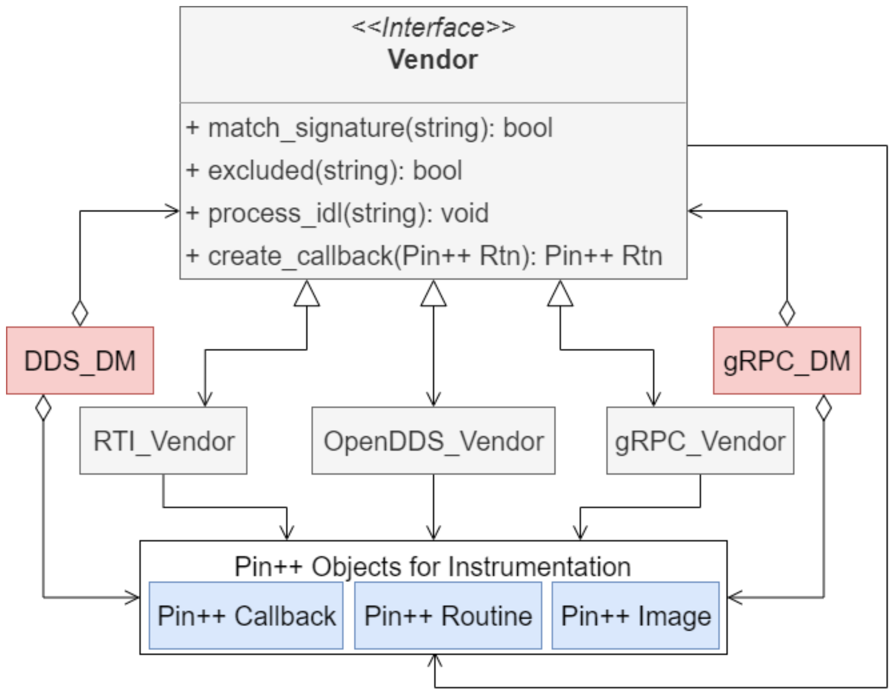
After parsing the configuration file, the distributed middleware object is created at run-time and passed to the SDMM_Instrument where data collection begins. The SDMM_Instrument will capture all running images and routines which are passed to the adapters via the *Middleware* interface for further processing. *Middleware.analyze_rtn()* will analyze the routines contained within the binary images.

The method *Middleware.analyze_rtn()* is where any instrumentation specific to a vendor is called. More details on vendor specific analysis is discussed in Section 4.4. Using the adapter pattern and configuration files generalizes distributed middleware and allows SDMM to address Challenge 1 presented in Section 4.1.

4.4 Accounting for Differences in Vendors

Our previous iteration to SDMM [19] supported one vendor for both middleware and did not need to address the second challenge from Section 4.1. The second challenge in non-intrusively instrumenting standards-based distributed middleware is creating a Pintool that handles the design differences between vendors. SDMM was developed with the Data Distribution Service (DDS) and gRemote Procedure Call (gRPC) as the distributed middleware.

The new SDMM supports two DDS implementations, RTI Connext DDS and OpenDDS, and gRPC was treated as its own vendor. An example of design differences is that RTI Connext DDS has several method overloads to *write* and *take* which are not used in OpenDDS. Likewise, OpenDDS also has its own share of unique method overloads.



<<Interface>>
Vendor
+ match_signature(string): bool
+ excluded(string): bool
+ process_idl(string): void
+ create_callback(Pin++ Rtn): Pin++ Rtn
DDS_DM
gRPC_DM
RTI_Vendor
OpenDDS_Vendor
gRPC_Vendor
Pin++ Objects for Instrumentation
Pin++ Callback
Pin++ Routine
Pin++ Image

Figure 4.3. Representation of distributed middleware vendors in SDMM using the Strategy Pattern.

To differentiate between implementations, we used the Strategy Pattern [12] where each vendor has its own class. As shown in Figure 4.3, each implementation inherits from the *Vendor* interface that defines methods for handling vendor specific details. In the diagram, each Distributed Middleware (DM) contains one vendor such as *RTI_Vendor*, *OpenDDS_Vendor*, or *gRPC_Vendor* and will invoke the vendor interface methods when appropriate.

Vendor.match_signature() is called when SDMM needs to decide whether to instrument a procedure and *Vendor.excluded()* checks if a procedure should not be instrumented. *Vendor.process_idl()* is used to process an interface definition file. Finally, *Vendor.create_callback()* is a factory method [12] that creates Pin/Pin++ callback routines.

The function *Middleware.analyze_rtn()*, from Figure 4.2, is where many vendor methods are used. For example, *Vendor.match_signature()* and *Vendor.excluded()* are used to check if a procedure should or should not be instrumented. If the procedure should be instrumented, then *Vendor.create_callback()* is invoked to construct the necessary Pin callback function that is injected into the binary. The procedure is skipped if it should not be instrumented.

Section 4.6 discusses how SDMM decides if instrumentation is injected for a procedure. Also, Section 4.5 indicates where *Vendor.process_idl()* is called. Using the strategy pattern generalizes middleware vendors and allows SDMM to address Challenge 2 presented in Section 4.1.

4.5 Extracting Values from Events

Early versions of SDMM did not retrieve values from the events passed between components [19] which is the third challenge from Section 4.1. SDMM cannot call accessor or setter methods on C++ objects because the tool does not have access to the object definitions in their ADT files. To address this problem, we extended SDMM to parse the interface definitions from Listings 3.3 & 3.4.

4.5.1 Extracting Values from DDS

For DDS, several C++ files are generated when the interface definitions from Listing 3.3 is compiled. For both vendors, each event becomes a C++ class with public methods for setting the data values. The code follows a standard where each field from the interface has their own setter methods and Listings 4.2 & 4.3 show an example on the PlannerOneEvent type. To detect each setter method, SDMM will parse the interface definition file in the vendor specific methods *RTI_Vendor.process_idl()* for RTI Connext DDS and *OpenDDS_Vendor.process_idl()* for OpenDDS from Figure 4.3. The signatures to each setter method is then stored in the appropriate vendor object.

```
class PlannerOneEvent {
  void eventcount(int32_t);
  void name(const std::string&);
  void name(const char*);
};
```

Listing 4.2. Source code for the RTI Connext DDS endpoints to set data values for PlannerOneEvent.

```
class PlannerOneEvent {
  void eventcount(int32_t);
  void name(const std::string&);
};
```

Listing 4.3. Source code for the OpenDDS endpoints to set data values for PlannerOneEvent.

4.5.2 Extracting Values from gRPC

For gRPC, the interface follows a standard where each field from Listing 3.4 has their own setter methods prefixed with the substring `set_`. The setter methods are shown in Listing 4.4. To detect each setter method, SDMM will parse the interface definition file in *gRPC_Vendor.process_idl()* from Figure 4.3. The signatures to each setter method is then stored in the *gRPC_Vendor* object.

```
class SimpleEvent {
  void set_eventcount(long);
  void set_name(const char*);
};
```

Listing 4.4. Source code for the gRPC endpoints to set data values.

Each setter method serves as an entry point into the system to extract values. More entry points into the system are discussed in Section 4.6. By parsing the interface definition files for setter methods, SDMM can address Challenge 3 in Section 4.1.

4.6 Discovering Points of Instrumentation

The fourth challenge in non-intrusively instrumenting standards-based distributed middleware as defined in Section 4.1 is locating places to analyze code. Our approach to this challenge is influenced by Schantz et al. [29] where the authors breakdown middleware in distributed systems to four layers. Starting at the most concrete: the host infrastructure layer refers to code that generalizes OS specific features for passing data over the network such as sockets; the distribution middleware layer extends the networking capabilities from the host layer without creating dependencies to the programming language or OS (this is where DDS and gRPC reside); the common middleware services layer extends the distribution layer with code that lets developers focus on business logic rather than managing system resources; lastly, the domain-specific layer refers to code tailored to a particular domain such as e-commerce or health care.

SDMM instruments systems at the distribution middleware layer exclusively because the symbols of interest come from the extensions to the host layer. For example, the publisher-subscriber methods in DDS or remote procedures in gRPC. The consequence of excluding other middleware layers is that our tool does not analyze low-level procedures such as socket creation or the procedures common middleware use to manage resources.

SDMM will look at all symbols within the indicated binaries to discover standards-based code which serve as gateways into the system. SDMM instruments a system by using programming standards to detect when a DDS or gRPC procedure is invoked at run-time, extract information from the procedure signature, and then resumes procedure execution. The tool does not modify a components instructions. The remainder of this section discusses how we identified programming standards in DDS and gRPC to gain access into the system.

4.6.1 Instrumentation Points for DDS

For both RTI Connext DDS and OpenDDS, SDMM instruments the binaries for each component. Those binaries are `Sensor`, `PlannerOne`, `PlannerTwo`, `Config`, and `Effector`. OpenDDS also has a separate binary, `libSLICE_Idl.so.3.14.0`, where the setter methods reside. SDMM will insert instrumentation for each setter method discovered in Section 4.5.

```
void
DataWriterImpl<...>::write(
  PlannerOneEvent const&,
  TInstanceHandle<InstanceHandle> const&);

LoanedSamples<PlannerOneEvent>
DataReaderImpl<...>::take(void);
```

Listing 4.5. Source code for the RTI Connext DDS endpoints to communicate PlannerOneEvents.

```
virtual ReturnCode_t
DataWriterImpl_T<...>::write(
  const PlannerOneEvent &,
  InstanceHandle_t) = 0;

virtual ReturnCode_t
```

```
DataReaderImpl_T<...>::take_next_sample(
  PlannerOneEvent &,
  SampleInfo &) = 0;
```

Listing 4.6. Source code for the OpenDDS endpoints to communicate PlannerOneEvents.

In DDS, producing events is done with *write* methods and consuming events is done with *take* or *read* methods. Our implementation of the SLICE scenario only uses *take* which will be the primary focus of this thesis. Regardless, SDMM can still detect and instrument DDS *read*. Both vendors overload *write* and *take* methods for each event. Listings 4.5 & 4.6 show an example on the PlannerOneEvent type.

There are two patterns to observe with DDS *write*. First, there may be multiple *write* methods in a DDS implementation even though there is only one *write* defined in the DDS specification [23]. Second, from Listings 4.5 & 4.6, observe there is a `DataWriter` substring before both versions to DDS *write*.

There are two patterns to observe with DDS *take*. First, there is multiple *take* methods in the DDS specification but they all start with the substring `take`. Second, from Listings 4.5 & 4.6, observe there is a `DataReader` substring before both versions to DDS *take*.

Considering the above observations for DDS *write* and *take*, we can represent the standards for producing and consuming events in DDS as a regular expressions. Listing 4.7 shows the regular expressions used to identify DDS communication methods and they are held within the *DDS_DM* class from Figure 4.2. All procedures are first tested against the regular expressions and then are tested against the setter method signatures by invoking *Vendor.match_signature()* from Figure 4.3.

```
//Regular expression for DDS write
std::regex write_regex("(.*)(DataWriter)(.*)(::write\\()(.*)");

//Regular expression for DDS take
std::regex take_regex("(.*)(DataReader)(.*)(::take)(.*)");

//Regular expression for gRPC client-side SendEvent
std::regex client_regex("(.*)(::SendEvent)(.*)(ClientContext)(.*)");

//Regular expression for gRPC server-side SendEvent
```

```
std::regex server_regex("(.*)(::SendEvent)(.*)(ServerContext)(.*)");
```

Listing 4.7. The regular expressions for identifying methods in DDS and gRPC.

4.6.2 Instrumentation Points for gRPC

For gRPC, SDMM analyzes the binaries for each component because that is where the setter methods and the remote procedures reside. The binaries are `Sensor`, `Planner`, `Config`, and `Effector`. Just like with DDS, SDMM will insert instrumentation for each setter method discovered in Section 4.5.

```
virtual Status SendEvent(
  ClientContext*,
  const SimpleEvent&,
  Empty* response*) = 0;

virtual Status SendEvent(
  ServerContext*,
  const SimpleEvent&,
  Empty* response*) = 0;
```

Listing 4.8. Source code for the gRPC endpoints to invoke the SendEvent() remote procedure.

There can be any number of arguments to a remote procedure, such as *SendEvent()*, in gRPC but they still follow a standard for both client-side and server-side stubs. The first parameter is always a context object corresponding to the endpoint such as `ClientContext` or `ServerContext`. Any number of arguments may follow the context object, but for the SLICE scenario, the arguments are instances of the SimpleEvent and Empty classes.

The pattern for remote procedures in gRPC is generalized into the regular expressions shown in Listing 4.7 which are held in the *gRPC_DM* class from Figure 4.2. All procedures are tested against the regular expressions first and then are tested against the gRPC setter signatures by invoking the *Vendor.match_signature()* interface method from Figure 4.3.

Expressing symbol patterns as regular expressions and capturing setter method signatures allows SDMM to address Challenge 4 presented in Section 4.1. Furthermore, SDMM can support any C++-based middleware by placing vendor specific instrumentation in their own objects, parsing the interface definition files for setter method signatures, indicating the middleware and relevant binaries in the configuration file, and discovering instrumentation points by expressing methods as regular expressions.

5. EXPERIMENTAL RESULTS

This chapter presents the environment used for our experiments, experimental results, and discusses SDMM's impact to performance. The Standards-based Distributed Middleware Monitor (SDMM) is evaluated by applying the SLICE scenario used throughout this thesis. We implemented the SLICE example with RTI Connext DDS [2], OpenDDS [1], and gRPC [27]. As part of our evaluation, we focused on answering three questions:

1. **Does SDMM successfully instrument distributed middleware that follow programming standards?** This is an important question because it allows us to understand if SDMM can successfully instrument distributed systems implemented using different distributed middleware.

2. **What is the performance impact that SDMM has on a distributed system?** We know dynamic binary instrumentation introduces overhead to a distributed system. We, however, want to understand how much overhead SDMM introduces. Specifically, we want to understand the difference in SDMM's run-time, CPU usage, and memory when applied to distributed systems implemented using different middleware (*e.g.*, DDS and gRPC).

3. **Is determining the middleware at run-time truly slower?** As described in Section 4.2, we expect the programmatic approach (Approach 1) to perform data collection faster than the configuration file approach (Approach 2) because Approach 1 determines the middleware at compile-time instead of run-time. We, however, want to know if that truly is the case.

5.1 Experiment Setup

All experiments were executed on an installation of Emulab [33] which is a cluster of machines that can emulate any operating system given the necessary disk image. All nodes were equipped with AMD Opteron 4130 2.6GHz 4-core processors running Ubuntu 14.04.1 operating system and Linux Kernel 3.13.0. A LAN connects nodes with an average round trip time of 0.093ms. All code was compiled with the GNU C++ compiler version 4.8.4. Our deployment layout, as shown in Figure 3.1, was influenced by Slaby et al. [31] who tested many deployments for SLICE endpoints to meet end-to-end performance deadlines. Finally, both DDS vendors were configured with the DDS Real Time Publish Subscribe protocol over UDP.

5.2 Results

```
Method: eventcount
InputTypes: int
Container: PlannerTwoEvent
CallCount: 4795
AvgRuntime: 0.00437956ms
Values: [1, 2, ...]

Method: name
InputTypes: char const*
Container: PlannerTwoEvent
CallCount: 4795
AvgRuntime: 0.00417101ms
Values: [Foo, ...]

Method: write
InputTypes: PlannerTwoEvent const&, TInstanceHandle<InstanceHandle> const&
Container: DataWriterImpl<PlannerTwoEvent>
CallCount: 4795
AvgRuntime: 0.61293ms

Method: take
InputTypes:
Container: DataReaderImpl<PlannerOneEvent>
CallCount: 4794
AvgRuntime: 0.0329579ms
```

Listing 5.1. Sample output from running SDMM on the PlannerOne binary using RTI Connext DDS.

5.2.1 Results for RTI Connext DDS

When we execute the RTI Connext DDS version of the SLICE scenario, SDMM successfully discovers the setter methods on all event objects as well as the produce and consume methods on each endpoint. These methods are discovered in real-time without *a priori* knowledge of the system. Listing 5.1 shows a sample of the output from running SDMM on the PlannerOne binary. SDMM reports the method, the data types in its parameter list, the object or namespace that contains the method, the call count which indicates the total number of method calls, and the methods average runtime. SDMM also discovers the data values assigned to events.

The PlannerOne binary will consume PlannerOneEvents and produce PlannerTwoEvents. As shown in Listing 5.1, SDMM detects the setter methods *eventcount()* and *name()*. The tool will extract method metadata as well as the value assigned to the event each time a setter method is invoked.

As expected, the call count between *eventcount()* and *name()* is equal since they are both called for event creation. The average run-time between the two methods is similar which aligns with expectations. We expected similar run-times because the events are assigned a short character array.

SDMM also detects the *write()* and *take()* methods from the endpoints that produce PlannerTwoEvents and consume PlannerOneEvents. The call counts between the communication methods are off by one because, at termination, PlannerOne will create and transmit a new event that informs PlannerTwo to terminate. The call count between *write()* and the setter methods are equal because event creation happens just before the event is sent to the next component. The average run-time between the two communication methods are different because, by default, RTI Connext DDS blocks the running thread if *write()* would cause data to be lost [2].

```
Method: eventcount
InputTypes: int
Container: PlannerTwoEvent
CallCount: 4772
AvgRuntime: 0.003772ms
Values: [1, 2, ...]

Method: operator<<
InputTypes: Serializer&, char const*
Container: DCPS
CallCount: 4772
AvgRuntime: 0.00544845ms
Values: [Foo, ...]

Method: write
```

```
InputTypes: PlannerTwoEvent const&, int
Container: DataWriterImpl_T<PlannerTwoEvent>
CallCount: 4772
AvgRuntime: 0.776194ms

Method: take_next_sample
InputTypes: PlannerOneEvent&, SampleInfo&
Container: DataReaderImpl_T<PlannerOneEvent>
CallCount: 5923
AvgRuntime: 1.25713ms
```

Listing 5.2. Sample output from running SDMM on the PlannerOne binary using OpenDDS.

5.2.2 Results for OpenDDS

When we execute the OpenDDS version of the SLICE scenario, SDMM successfully discovers the setter methods on all event objects as well as the produce and consume methods on each endpoint. Output samples from running SDMM on the PlannerOne binary are shown in Listing 5.2. Just like the RTI Connext DDS example, SDMM reports the method, parameter types, the container, the call count, the methods average runtime, and discovers the data values assigned to events.

As shown in Listing 5.2, SDMM detects the setter methods on the PlannerTwoEvent class that do not have string input types and will extract the argument values at run-time. Strings are still captured via the data insertion operator (<<) and OpenDDS uses << to serialize strings. The consequence of instrumenting << is that the strings are no longer correlated to the appropriate setter method. The reason for using the data insertion operator is explained in Section 6.

The call counts between *eventcount()* and << is equal, which aligns with expectations. We expected *eventcount()* and << methods to have equal call counts because the former is called for event creation and the latter is called for every string assigned to an event (*e.g.,* the strings assigned via setter methods). The average run-time for << is higher than the run-time for *eventcount()*. We expect << to be slower because the character array is serialized for network transmission which takes time.

SDMM detects the *write()* method for producing PlannerTwoEvents and the *take_next_sample()* method for consuming PlannerOneEvents. The call count is higher for *take_next_sample()* because of fault tolerance. The OpenDDS version to SLICE will retry *take_next_sample()* if there were any problems with consuming the event. Fault tolerance also explains why the average run-time for consuming events is slower than producing since less writes occurred over time. The call count between *write()* and *eventcount()* is equal because event creation happens just before the event is sent over the network. The call counts between *write()* and << are equal because serialization occurs on all the events passed to *write()*.

```
Method: set_eventcount
InputTypes: long
Container: SimpleEvent
CallCount: 4280
AvgRuntime: 0.0208201ms
Values: [1, 2, ...]

Method: set_name
InputTypes: char const*
Container: SimpleEvent
CallCount: 4280
AvgRuntime: 0.0582937ms
Values: [Foo, ... ]

Method: SendEvent
InputTypes: ServerContext*, SimpleEvent const*, Empty*
Container: SLICEServerServiceImpl
CallCount: 4487
AvgRuntime: 0.0749184ms

Method: SendEvent
InputTypes: ClientContext*, SimpleEvent const&, Empty*
Container: SLICEServer::Stub
CallCount: 4280
AvgRuntime: 2.22696ms
```

Listing 5.3. Sample output from running SDMM on the PlannerOne binary using gRPC.

5.2.3 Results for gRPC

When we execute the gRPC version of the SLICE scenario, SDMM successfully discovers the setter methods on the SimpleEvent class and the remote procedure calls on both the server and client-side endpoints. Listing 5.3 shows the output from running SDMM on the PlannerOne binary. From the listing, SDMM detects all setter methods on the SimpleEvent class and the remote procedure. Our tool also reports the method parameter types, the method containers, method call counts, method average run-time, and the values passed to setter methods. More importantly, this information is extracted without knowledge of the system before-hand.

The call count between *set_eventcount()* and *set_name()* are equal, as expected, since they are both called whenever an event is created. We expected the average run-times between the setter methods to be similar because the events are assigned a small character array. Instead, the average run-time for *set_name()* is higher than *set_eventcount()* because Protocol Buffers [9], a dependency to gRPC, converts the C-string to a std::string using the system allocator behind several other checks and function calls [5]. This issue, however, is removed in the latest versions of gRPC and Protocol Buffers.

The call count between client-side and server-side versions of *SendEvent()* are unequal because the component terminated before sending all the events left in its queue. Early program termination also explains why the call count between setter methods and the server-side *SendEvent()* are different. Yet, the call count between the client-side procedure and setter methods are equal because event creation occurs before event transmission to the next component.

The average run-time for the client-side *SendEvent()* is higher than the server-side because the client is blocked until control is returned from the remote server. Whereas the server-side procedure is called locally and does not have added overhead from communicating events over the network.

Since SDMM reports method names, parameter types, procedure average run-time, and the values communicated over the network, we believe our tool successfully instruments standards-based distributed middleware. SDMM can monitor a system supported by distributed middleware by using symbol patterns, such as the setter methods, to gain access into the system. This is important because stakeholders can use our tool to ensure their distributed systems are meeting performance requirements [18].

5.3 SDMM's Impact on Run-time

We evaluated the impact that SDMM has on a distributed system's run-time by measuring how long data collection took during 10-minute executions of the SLICE scenario. Table 5.1 reports how long data collection took for the PlannerOne component using RTI Connext DDS, OpenDDS, and gRPC. The numbers are an average over 10 runs and are in milliseconds. We calculated run-time overhead by dividing the time taken to collect data over the total run-time. For reference, 10-minutes is 6e5ms.

Table 5.1. Shows how much time data collection requires on the PlannerOne component. The measurements are an average over 10 runs.

PlannerOne	Data Coll run-time	% of Total Run-time
RTI Connext DDS	4442.550ms	0.740%
OpenDDS	11327.700ms	1.887%
gRPC	2297.070ms	0.382%

As shown in Table 5.1, data collection on the PlannerOne component for RTI Connext DDS accounts for $\sim$ 0.740% of the 10-minute run-time. For OpenDDS, data collection accounts for $\sim$ 1.887% of the run-time. For gRPC, data collections takes $\sim$ 0.382% of the run-time.

SDMM has little impact to run-time performance because the tool simply reads and then stores information as it is created in real-time. Other additional processing will come from parsing the configuration file, interface definition file, and procedure signatures. Those files and strings, however, are small and should not take much execution time.

Our previous experiments in [19] showed that data collection contributes to less than 0.01% of the run-time for some test cases. The differences between our previous work and this new iteration involve the amount of symbols SDMM analyzes. For example, the previous design did not extract values from setter methods. Without analyzing setter methods, the tool parses less symbols and less instrumentation is injected into the binaries. Second, the past case study used fewer endpoints per node and fewer endpoints means less time spent parsing binary images.

The run-time overhead for OpenDDS is higher than RTI Connext DDS because there are more procedures in the OpenDDS binaries passed to SDMM. The number of procedures affects run-time because each function signature is tested against our regular expressions which adds to run-time. SDMM does filter out many functions using the *Vendor.match_signature()* and *Vendor.excluded()* methods from Figure 4.3. There still is, however, many procedures that fall through the filtration.

5.3.1 Throughput with respect to Data Collection run-time

We were also interested in the throughput with respect to data collection run-time. Table 5.2 shows the throughput for the number of events analyzed over data collection run-time ($\frac{\#of events}{datacollectionruntime}$). We only consider outbound events, without garbage values, because the setter methods are explicitly invoked on them. We converted the time measurements from milliseconds to seconds because it is easier to comprehend throughput in seconds.

Table 5.2. Shows how many events are instrumented per second with respect to the data collection run-time. The numbers are from the PlannerOne component.

PlannerOne	Data Coll run-time	Total Events	Events/Data Coll run-time
RTI Connext DDS	4.442s	4795	1072.714 events/s
OpenDDS	11.327s	4724	417.056 events/s
gRPC	2.297s	1105	481.062 events/s

As shown in Table 5.2, SDMM instruments 1072.714 events/s for RTI Connext DDS and 417.056 events/s for OpenDDS. For gRPC, SDMM instruments 481.062 events/s on PlannerOne. It is important to note that the throughput is with respect to the data collection run-time and not total run-time.

OpenDDS throughput is lower than RTI because of the procedures that pass our filtering strategy and are compared against the regular expressions which increases run-time. The low throughput for gRPC is because SDMM is capturing a lot of garbage values. SDMM is a middleman and sometimes records values before they are deserialized. Serialized values are excluded from the event count, and decrease the throughput, because they are in a non-human readable form.

If SDMM is run on a distributed system with significantly larger strings or arrays of primitive types (*e.g.*, C-string length 10e6), then data collection may impact execution time more than what is reported in Tables 5.1 & 5.2. A large string would affect performance because SDMM will also have to read and store the string which takes time. Since data collection takes less than 2% of the total run-time, we believe SDMM has a negligible impact on run-time performance.

5.4 SDMM's Impact on CPU Usage

We evaluated the impact that SDMM has on distributed system's CPU usage by measuring how much time data collection spent in user mode and kernel mode during the 10-minute executions of the SLICE scenario. We measured CPU usage with the `rusage` struct available on Linux platforms. We calculated CPU usage for data collection by taking the time spent in each mode at the start and exit of functions specific to SDMM. Then the difference between the start and exit times are summed.

Tables 5.3 & 5.4 report how much time data collection spent in user and kernel mode, respectively, for the PlannerOne component. The numbers are an average over 10 runs and are in milliseconds. We calculated CPU overhead by dividing the time data collection spent in each mode over the total time the component spent in each mode.

Table 5.3. Shows how much time data collection on the PlannerOne component spent in user mode. The measurements are an average over 10 runs.

PlannerOne	Data Coll User Mode	Total User Mode	% of Total
RTI Connext DDS	4430.298ms	16007.843ms	27.675%
OpenDDS	11297.439ms	46382.270ms	24.357%
gRPC	2301.874ms	9533.738ms	24.144%

As shown in Tables 5.3 & 5.4, data collection on the PlannerOne component for RTI Connext DDS accounts for $\sim 27.675\%$ and 2.828% of the total time spent in user mode and

Table 5.4. Shows how much time data collection on the PlannerOne component spent in kernel mode. The measurements are an average over 10 runs.

PlannerOne	Data Coll Kern Mode	Total Kern Mode	% of Total
RTI Connext DDS	72.419ms	2560.606ms	2.828%
OpenDDS	42.272ms	3923.068ms	1.077%
gRPC	47.322ms	1798.581ms	2.631%

kernel mode, respectively. For OpenDDS, data collection accounts for 24.357% of the time spent in user mode and 1.077% of the time spent in kernel mode. For gRPC, data collection accounts for 24.144% of the time spent in user mode and 2.631% of the time spent in kernel mode.

For all test cases in Table 5.3, SDMM is spending $\sim$ 24% to $\sim$ 27% of its time in user-mode. We expected SDMM to often reserve the CPU because our tool must run and compete for time on the processor. SDMM frequently operates in user mode because most of the instrumentation uses high-level C++ code that does not require system interrupts. SDMM does not operate in kernel mode often because the only system calls are memory allocation for objects at startup and handling files such as the interface definition files.

5.4.1 Throughput with respect to CPU Usage

We also looked at the throughput in terms of CPU usage. Tables 5.5 & 5.6 show rates for $\frac{\#ofevents}{datacollectionusermode}$ and $\frac{\#ofevents}{datacollectionkernmode}$ respectively. Only outbound events without garbage values are considered because the setter methods are invoked on those events. The time measurements are shown in seconds instead of milliseconds because it is easier to understand throughput in seconds.

Table 5.5. Shows how many events are instrumented per second with respect to the time data collection spent in user mode. The numbers are from the PlannerOne component.

PlannerOne	Data Coll User Mode	Total Events	Events/Data Coll User Mode
RTI Connext DDS	4.430s	4795	1082.392 events/s
OpenDDS	11.297s	4724	418.164 events/s
gRPC	2.301s	1105	480.225 events/s

As shown in Tables 5.5 & 5.6, SDMM analyzes 1082.392 events/s in user mode and 66,597.222 events/s in kernel mode for RTI Connext DDS. For OpenDDS, SDMM analyzes 418.164 events/s in user mode and 112,476.190 events/s in kernel mode. Lastly, for gRPC, the tool instruments 480.225 events/s in user mode and 23,510.638 events/s in kernel mode. It is important to note that the throughput is with respect to the time data collection spent in both CPU modes.

Table 5.6. Shows how many events are instrumented per second with respect to the time data collection spent in kernel mode. The numbers are from the PlannerOne component.

PlannerOne	Data Coll Kern Mode	Total Events	Events/Data Coll Kern Mode
RTI Connext DDS	0.072s	4795	66,597.222 events/s
OpenDDS	0.042s	4724	112,476.190 events/s
gRPC	0.047s	1105	23,510.638 events/s

Examining the throughput in user mode, OpenDDS is lower than RTI because of the higher count of procedures that fall through filtration and are tested against our regular expressions. Those comparisons are performed in user mode. The main factor behind low user mode throughput with gRPC is the low number of valid events. The throughput rate in kernel mode is high for all middleware because SDMM does not invoke many system calls. OpenDDS, however, is higher than RTI because SDMM creates a few more objects for RTI specific symbols.

The CPU usages may change if a different distributed system from the SLICE scenario is used. For example, a system with thousands of unique communication methods or setter methods. With such an example, we would expect the time spent in user mode to increase because SDMM will extract information at every method call. Since data collection contributes to $\sim 27\%$ of the total time in user mode, we believe SDMM impacts CPU usage.

5.5 SDMM's Impact on Memory Usage

We evaluated the impact that SDMM has on memory usage by comparing how much virtual memory a component uses with SDMM, with Pin++ but without SDMM, and without any instrumentation. We considered virtual memory only because SDMM analyzes symbols in any binary file including binaries that are swapped from disk. We measured virtual memory usage, in megabytes, by parsing output from the Linux `top` command during the 10-minute execution and averaged the numbers across 10 runs. Figure 5.1 & Table 5.7 illustrate how much virtual memory the PlannerOne component used for RTI Connext DDS, OpenDDS, and gRPC.

Table 5.7. Shows how many Megabytes of virtual memory the PlannerOne component used with SDMM, with Pin++ only, and without instrumentation.

PlannerOne	SDMM	Pin++ Only	No Instrumentation
RTI Connext DDS	1719.475 MB	1717.009 MB	1198.351 MB
OpenDDS	1206.266 MB	1199.564 MB	727.256 MB
gRPC	1011.069 MB	892.952 MB	443.932 MB

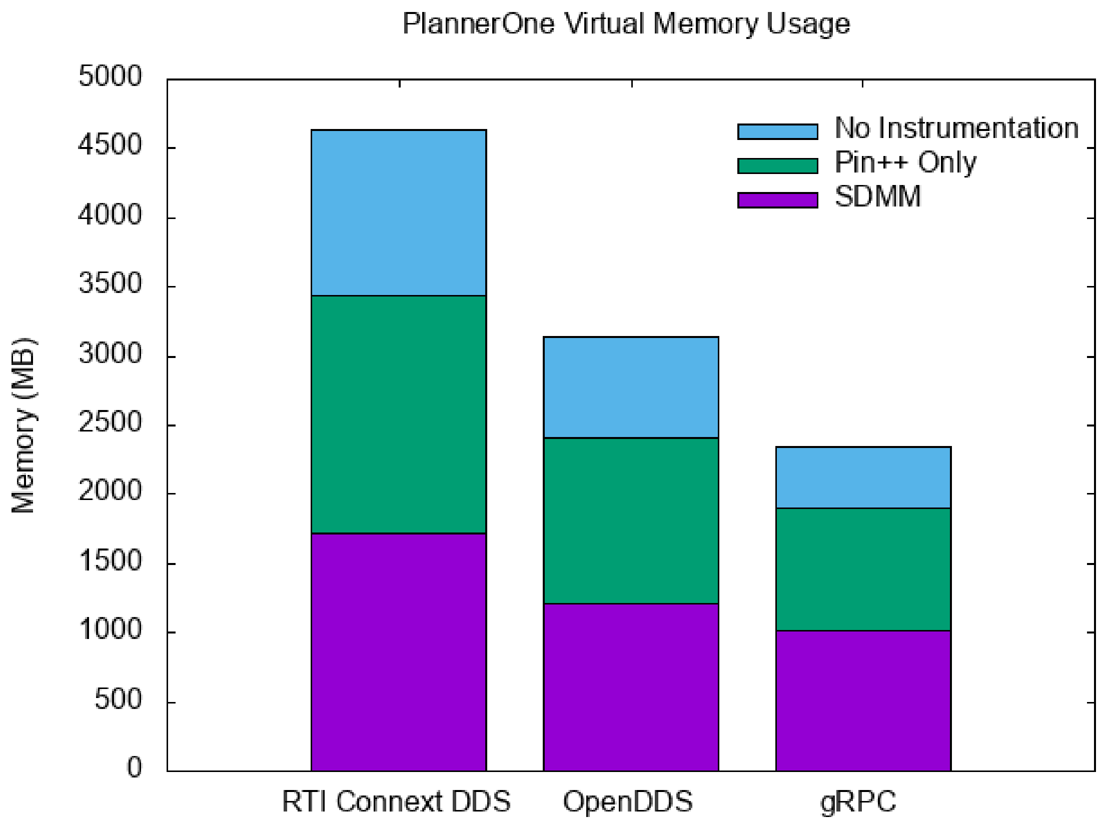


Figure 5.1. Shows how much virtual memory the PlannerOne component used with SDMM, with Pin++ only, and without instrumentation.

From Figure 5.1 & Table 5.7, observe that experiments with SDMM and Pin++-only use a similar amount of virtual memory. We can infer that the SDMM code separate from the Pin++ framework uses a small amount of memory. SDMM uses a small amount of memory because the information extracted from middleware is also small such as the metadata from communication methods and values taken from setter methods. Additionally, there is the memory needed to instantiate the classes from Sections 4.4 & 4.3 as well as hold the regular expressions from Section 4.6. The classes and regular expressions, however, are only created once at startup.

Comparing experiments with Pin++-only and no instrumentation, we see a larger difference in the amount of virtual memory used. The larger difference in virtual memory shows that the instrumentation framework accounts for most of the introduced virtual memory overhead instead of SDMM.

The virtual memory usage may change if a different distributed system from the SLICE scenario is used. For example, a system that defines events with many primitive types or large arrays of data. The larger the events, the more memory is needed for SDMM to store copies of event data. In contrast, we would expect to see an improvement in memory usage from a distributed system with smaller events than the SLICE scenario. Since SDMM adds little memory on top of Pin++, we believe the tool has a negligible impact on an distributed system's virtual memory usage.

5.6 Approach 1 vs. Approach 2

Table 5.8. Compares the run-time of data collection between Approach 1 and Approach 2 from Section 4.2

PlannerOne	Approach 1	Approach 2	% Change
RTI Connext DDS	4320.630ms	4507.690ms	4.237%
OpenDDS	11430.800ms	12715.700ms	10.642%
gRPC	2204.830ms	2306.430ms	4.504%

Table 5.8 compares data collection run-time on the PlannerOne component from the two approaches discussed in Section 4.2. The experiments were run for 10-minutes and execution

times were averaged over 10 runs. Data collection on PlannerOne with RTI Connext DDS is faster with Approach 1 than Approach 2 by $\sim$ 4.237%. With OpenDDS, data collection on PlannerOne is 10.642% faster with Approach 1 than Approach 2. Lastly, with gRPC, data collection on PlannerOne is faster with Approach 1 than Approach 2 by $\sim$ 4.504%. As expected, Approach 1 is faster than Approach 2 for all test cases because Approach 2 determines the distributed middleware at run-time.

6. LIMITATIONS AND FUTURE WORK

This section discusses the limitations and potential future research directions to the Standards-based Distributed Middleware Monitor (SDMM).

6.1 SDMM does not work with ADTs as parameter types

SDMM does not extract data values when the parameter is an ADT (*e.g.*, std::string). This is because Intel's Pin treats all data types as an *ADDRINT*, a type defined by Pin that needs explicit conversion rules. The consequence of this issue is that SDMM only works with primitive data types. Unfortunately, the DDS C++11 Mapping [4] requires that all strings be treated as C++ std::string and not C-strings.

To address this limitation, we overloaded all string setter methods in RTI Connext DDS with C-strings as the parameter types. We could not do the same with OpenDDS because the target file gets overwritten every time the source code is compiled. Fortunately, OpenDDS uses the data insertion operator << to serialize data, including strings in their C-string representation. SDMM was able to extract string values from events in OpenDDS by instrumenting the << operator. The consequence, however, is that those strings are no longer correlated to the appropriate setter methods.

The solution, and therefore future work, includes features that analyze and extract values from ADTs. This process can be done in two ways. The first method is SDMM can be compiled with the necessary ADT files. The second method would be to require ADTs written with a conversion rule to Intel's Pin ADDRINT type. Both methods, however, qualify as *a priori* knowledge of the distributed system.

6.2 Unintended signatures may match with SDMM regular expressions

It is likely that multiple binary files share similar patterns and name conventions. In such a case, SDMM would attempt to instrument all methods that match our regular expressions, which will affect performance and results. This is, however, a user-based error because the user must indicate those binary files either in the code from Approach 1, or in the configuration file from Approach 2.

The manual solution to this problem is to remove the culprit binary file from the configuration file. This solution, however, may not be feasible if a particular binary has other methods targeted for instrumentation. An automated solution, and therefore future work, is to generalize commonalities between the regular expressions from Listing 4.7 into a single grammar. The resulting grammar could then be used in lexical analysis to identify tokens from method signatures. For example, the lexical analyzer could identify the DataWriter substring from the *write* signature or the Context objects in the *SendEvent()* signature.

6.3 Any instrumentation will slow down the distributed system

Any instrumentation framework will always slow down a distributed system. In the context of Pin, new I/O layers are introduced every time instrumentation code is injected into the binary files. Control must pass through these new layers, which inhibits performance, every time a method flagged for instrumentation is invoked.

The solution is to add control over how much instrumentation overhead is brought into a system. Incorporating instrumentation control has the potential to reduce the number of instrumentation insertions. This feature could include user-controlled sampling for a subset of events instead of instrumenting every event and method discovered. Future work therefore will investigate how we can integrate such controls into SDMM without introducing more instrumentation overhead.

6.4 Support other processing architectures

Intel's Pin is the DBI framework behind SDMM. While this framework is suitable for our project, Pin is only compatible on the x86 CPU architecture. The solution and future work include incorporating support for other CPU architectures by porting the Pin instrumentation code to other DBI frameworks. For example, DynamoRIO [11] is a framework that supports ARM and PowerPC architecture.

7. CONCLUSION

This thesis presented a Pintool named the *Standards-based Distributed Middleware Monitor (SDMM)* that non-intrusively instruments a distributed system implemented with standards-based distributed middleware. SDMM can support any distributed middleware, such as the Data Distributed Service (DDS) and gRemote Procedure Call (gRPC), by representing them in a configuration file and object composition. The tool leverages symbol patterns, which we call programming standards, as the gateway into a distributed system. The Pintool captures standards from communication methods and setter methods for real-time monitoring and analysis. SDMM currently works with two DDS vendors, RTI Connext DDS and OpenDDS, gRPC, and CORBA, which was not discussed in this thesis.

Based on experience gained from applying SDMM to distributed systems, we learned it is possible to create a single tool to non-intrusively instrument a system written with any distributed middleware. SDMM could discover the object setter methods and communication methods for programs written with DDS and gRPC middleware. Furthermore, SDMM could extract information relevant to real-time monitoring such as method average run-time and the data values. This is valuable because all the information is collected at run-time without *a priori* knowledge of the system under instrumentation.

SDMM has been integrated in Pin++. SDMM is freely available in open-source format from the following location: `https://github.com/SEDS/PinPP/tree/master/examples/SDMM`.

REFERENCES

[1] OpenDDS. http://opendds.org/.

[2] Rti connext dds. https://www.rti.com/products.

[3] The data distribution service: Reducing cost through agile integration. https://www.omg.org/news/meetings/tc/dc-13/special-events/dds-pdfs/DDS_Exec_Brief-v20d-public.pdf, 2011.

[4] Idl to c++11 language mapping. https://www.omg.org/spec/CPP11/About-CPP11/, July 2019.

[5] Protocol buffers. https://github.com/protocolbuffers/protobuf/blob/3.7.x-fix/src/google/protobuf/arenastring.h, 2019.

[6] Dcov - a test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, February 2020.

[7] What can dds do for you? https://www.omg.org/hot-topics/documents/dds/CoreDX_DDS_Why_Use_DDS.pdf, August 2020.

[8] World's leading data distribution service (dds). https://www.rti.com/products/dds-leadership, August 2020.

[9] Protocol buffers. https://developers.google.com/protocol-buffers, 2021.

[10] S. Bateman, C. Gutwin, N. Osgood, and G. McCalla. Interactive usability instrumentation. In *Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '09, page 45–54, New York, NY, USA, 2009. Association for Computing Machinery.

[11] D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275. IEEE, 2003.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

[13] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer. Dynamic monitoring of high-performance distributed applications. In *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, pages 163–170. IEEE, 2002.

[14] J. Hill. An architecture independent approach to emulating computation intensive workload for early integration testing of enterprise dre systems. volume 5870, pages 744–759, 11 2009.

[15] J. H. Hill and D. C. Feiock. Pin++: an object-oriented framework for writing pintools. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, pages 133–141. ACM, 2014.

[16] J. H. Hill and A. Gokhale. Model-driven specification of component-based distributed real-time and embedded systems for verification of systemic qos properties. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008.

[17] J. H. Hill and A. Gokhale. *Towards Improving End-to-End Performance of Distributed Real-Time and Embedded Systems Using Baseline Profiles*, pages 43–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[18] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 5(2):121–150, 1987.

[19] N. Lui and J. H. Hill. A generalized approach for non-intrusive real-time instrumentation of standards-based distributed middleware. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 158–166, 2020.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[21] R. C. Martin. Design Principles and Design Patterns. *Object Mentor*, pages 1–34, 2000.

[22] Object Management Group. *CORBA Components v4.0*. Object Management Group, OMG Document formal/2006-04-01 edition, April 2006.

[23] Object Management Group. *Data Distribution Service for Real-time Systems Specification*, 1.2 edition, Jan. 2007.

[24] Object Management Group. *The Real-time Publish-Subscribe Protocol DDS Interoperability Wire Protocol Specification*, 2.3 edition, May 2019.

[25] A. Orebaugh, G. Ramirez, and J. Beale. *Wireshark & Ethereal network protocol analyzer toolkit*. Syngress, 2006.

[26] G. Pardo-Castellote. Omg data-distribution service: Architectural overview. In *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pages 200–206. IEEE, 2003.

[27] M. Roth, N. Noble, and A. Kumar. gremote procedure call. https://github.com/grpc/grpc, 2019.

[28] G. R. Satyanarayana, L. Tu, N. Lui, and J. H. Hill. Real-time, non-instrusive instrumentation and monitoring of standards-based event-based applications. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 20–27, May 2017.

[29] R. E. Schantz and D. C. Schmidt. Middleware for distributed systems - evolving the common structure for network-centric applications, 2001.

[30] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

[31] J. M. Slaby, S. Baker, J. H. Hill, and D. C. Schmidt. Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System

QoS. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2006)*, pages 350–362. IEEE, 2006.

[32] B. Sridharan, B. Dasarathy, and A. P. Mathur. On building non-intrusive performance instrumentation blocks for corba-based distributed systems. In *Proc. IEEE Int. Computer Performance and Dependability Symp. IPDS 2000*, pages 139–143, 2000.

[33] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[34] I. Zhukov, C. Feld, M. Geimer, M. Knobloch, B. Mohr, and P. Saviankou. Scalasca v2: Back to the future. In *Proc. of Tools for High Performance Computing 2014*, pages 1–24. Springer, 2015.