ARM 2009
Proceedings of the 8th Workshop on Adaptive and Reflective Middleware

DECEMBER 1st 2009, URBANA CHAMPAIGN, ILLINOIS, USA

CO-LOCATED WITH THE 10th ACM/IFIP/USENIX INTERNATIONAL MIDDLEWARE CONFERENCE

WORKSHOP CHAIRS: PAUL GRACE & FRANK ELIASSEN

ACM International Conference Proceedings Series

ACM Press
FOREWORD

On behalf of the organising committee we would like to welcome you to this 8th edition of the Workshop on Adaptive and Reflective Middleware, which this year is hosted in Urbana Champaign, Illinois, USA.

The goal of the ARM workshop series for this and its seven prior incarnations is to bring together researchers working on techniques and middleware platforms to engineer dynamic adaptations in distributed systems. This year we received ten research paper submissions around this topic; subsequently a rigorous review process was followed with each paper receiving either three or four reviews from international experts. Based upon this process, six papers were finally selected to appear in these proceedings, and be presented at the workshop.

The papers cover a range of topics related to the key issues of dynamic adaptation. These include new techniques to perform and manage adaptation in distributed; and also cover example application domains where the use of adaptive middleware is fundamentally important. We hope these papers will form the basis of lively discussion at the workshop.

There are a number of people we must acknowledge whose help greatly simplified the organisation of this workshop. We would first like to thank the Middleware 2009 conference team, in particular Roy Campbell (General Chair) and Cecilia Mascolo (Workshops Chair) for their help in organising the workshop and creating these proceedings. Secondly, we warmly thank the members of the programme committee whose dedicated work in providing helpful feedback to both ourselves and the submitting authors made the reviewing stages straightforward.

Paul Grace, Lancaster University
Frank Eliassen, University of Oslo

Workshop Chairs (ARM 2009)
PROGRAMME COMMITTEE

Anders Andersen, University of Tromsø, Norway
Gordon Blair, Lancaster University, UK
Licia Capra, University College London, UK
Renato Cerqueira, PUC-Rio, Brazil
Fabio Costa, Federal University of Goias, Brazil
Geoff Coulson, Lancaster University, UK
Edward Curry, Digital Enterprise Research Institute (DERI), Ireland
Frank Eliassen, University of Oslo, Norway
Sebastian Gutierrez-Nolasco, NASA Ames Research Center, USA
Francisco Jose da Silva e Silva, Federal University of Maranhao, Brazil
Fabio Kon, University of Sao Paulo, Brazil
Thomas Ledoux, INRIA Rennes - Bretagne Atlantique, Ecole des Mines de Nantes, France
Philippe Merle, INRIA Lille - Nord Europe, France
Omer Rana, Cardiff University, UK
Romain Rouvoy, University of Lille 1, France
Douglas Schmidt, Vanderbilt University, USA
Richard Staehli, Expeditors International, Seattle, Washington, USA
Alexandre Sztajnberg, State University of Rio de Janeiro, Brazil
Francois Taiani, Lancaster University, UK
Carolyn Talcott, SRI International, USA
Luis Veiga, INESC-ID Lisboa, Portugal
Nalini Venkatasubramanian, University of California, Irvine, USA
Nanbor Wang, Tech-X Corporation, USA
Eiko Yoneki, University of Cambridge, UK

ORGANISING COMMITTEE

Renato Cerqueira, PUC-Rio, Brazil
Francois Taiani, Lancaster University, UK
Fabio M. Costa, Federal U. of Goias, Brazil
Geoff Coulson, Lancaster University, UK
Omer F. Rana, Cardiff University, UK
Nalini Venkatasubramanian, University of California, Irvine, USA
TABLE OF CONTENTS

Mercury: A Reflective Middleware for Automatic Parallelization of Bags-of-Tasks .......................................................... Article No. 1
João Nuno Silva, Luis Veiga, Paulo Ferreira

A Component-based Approach For (Re)-Configurable Routing in VANETs.......... Article No. 2
Vatsala Nundloll, Gordon S. Blair, Paul Grace

A Reflective Middleware Architecture for Simulation Integration ................. Article No. 3
Leila Jalali, Nalini Venkatasubramanian, Sharad Mehrotra

Using Machine Learning to Maintain QoS for Publish/Subscribe Systems in Dynamic Environments ................................................................. Article No. 4
Joe Hoffert, Daniel Mack, Douglas Schmidt

RACED: an Adaptive Middleware for Complex Event Detection .................. Article No. 5
Gianpaolo Cugola, Alessandro Margara

A Semantic Composition Model to Preserve (Re)configuration Consistency in Aspect Oriented Middleware ......................................................... Article No. 6
Bholanath Singh Surajbali, Paul Grace, Geoff Coulson
Mercury: A reflective middleware for automatic parallelization of Bags-of-Tasks

João Nuno Silva  Luís Veiga  Paulo Ferreira
INESC-ID / Technical University of Lisbon
Distributed Systems Group
Lisboa, Portugal
{joao.n.silva,luis.veiga,paulo.ferreira}@inesc-id.pt

ABSTRACT

Today, the development of Bag-of-Tasks, i.e. embarrassingly parallel, applications for execution on multiprocessors or clusters requires the use of APIs not designed for this kind of problem.

For instance, MPI allows the parallel execution of tasks, but was developed for much complex parallel applications, with high data communication between tasks. The use of such APIs requires the programmers to learn them, and add complexity to the final parallel solution.

Mercury provides a platform for the transformation of serial applications into parallel Bag-of-Tasks. Mercury reads a configuration file stating what methods and classes should be parallelized, loads the application, and in run-time transforms it so that the specified methods are executed concurrently. This transformation is performed without user intervention. Its modular design allows the integration of Mercury with different parallel environments.

The initial experiments done show that the overhead is minimal, and that it is possible to take advantage of parallel processing environments (multiprocessors/multicores, clusters, ...) without the use of complex APIs.

Categories and Subject Descriptors
D.1.3 [Concurrent Programming]: Parallel programming;
C.2.4 [Distributed Systems]: Distributed applications

General Terms
Design, Languages, Performance

Keywords
adaptive middleware, Bag-of-Tasks, automatic parallelization

1. INTRODUCTION

There has been an increase on the use of scripting languages (such as Python) for the management of scientific computing jobs. In particular, Python [22] has been used to interact with grid enabled simulations, as with Ganga [12], or to access and manipulate scientific data sets as in PyRAF [16]. Furthermore, the performance obtained when using Python is on par with some other commonly available programming languages or environments [8, 21], making it a suitable language for scientific processing.

The use of Python is not limited to the invocation of sequential simulations. With a suitable middleware it is possible to take advantage of multiprocessors or clusters of computers. For instance, Star-P [17] offers a set of APIs for data and code distribution, while MPI [9] has the usual functions for task creation and synchronization, and data transfer.

For large scale projects, besides the use of parallel kernel (such as BLAS parallel implementations [13]) the use of MPI is the best way for data and work distribution, due to their high availability. Not only the programmer is able to take advantage of the remote resources, but network and computing platforms heterogeneity are hidden from him.

However, for the execution of a pure Bag-of-Tasks the use of the above mentioned approaches may be overkill, as it requires the use of specialized APIs to perform explicit parallelization, synchronization and data transmission. As a matter of fact, the gains achieved may not be enough to justify the extra complexity the programmer has to manage to parallelize his application.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ARM 2009, December 1, 2009, Urbana Champaign, Illinois, USA.
Copyright 2009 ACM 978-1-60558-850-6/09/12 ...$5.00.

Listing 1: Typical serial Bag-of-Tasks version

As an example, consider the effort of parallelizing the application shown in Listing 1 (allowing all processData() methods to execute concurrently) using MPI, or any other parallel execution middleware; the following modifications have to be done: i) identification of master and slave tasks, ii) transmission of data to slaves, and iii) receiving the results.

To accomplish that, the overall organization of the application code would have to be changed. These modifications would require the programmer to know the work distribution API and would introduce error prone-code, thus reducing programmer’s productivity as well as applications robustness.

Thus, our goal is to minimize the modification effort necessary to transform the structure of a sequential application, as the example presented in Listing 1, into a parallel Bag-of-Tasks. For that purpose, we developed Mercury, a middleware that allows the parallelization of independent object methods, allowing their concurrent execution on different local threads or different remote computers.

```python
1 for i in range(1000):
2   inputData = getTaskInput(i)
3   objList[i] = processinObject()
4   objList[i].processData(inputData)
5 for i in range(1000):
6   outputResult = objList[i].getResult()
7   process(outputResult)
```
Our solution transparently transforms and parallelizes sequential applications containing long iterations that process different data sets. This is done with no user intervention. The lengthy tasks executed on each iteration are executed on different threads. These threads can be executed locally, in the case of a multiprocessor (or multicore) computer, or on remote computers.

Using Mercury, the programmer has to adhere to a very simple coding structure: the programmer must only write his data processing application in the form presented in Listing 1, and define in a XML configuration file the class and methods that can be executed concurrently.

The Mercury middleware is responsible for spawning the necessary threads, and synchronize the invocation of the methods. In the previous example, only after the conclusion of processData, the corresponding getResult() method can be executed.

Mercury also allows different execution environments for the parallelized methods. A set of adaptation classes allows the execution of the methods on different threads on the same multiprocessor/multicore computer or the remote execution of faster computers. The indication of what parallel execution environments are available is also made in a descriptive way. The selection of the best location to execute the parallel code is made during run-time and takes into account both the code requirements and the computing resources availability.

The distribution of work among several computers or processors can now be done using libraries such as MPI, or Map-Reduce, but requires the programmer to know their API. With run-time code adaptation this need disappears, as the work distribution code is inserted in the correct places during application execution. Furthermore the application does not become tied to any specific execution environment.

The proposed solution uses metaclasses, allowing the modification of the code to be done in run-time, without any need to transform and recompile the source code. The developed metaclass intercepts all class creations and modifies the implementation of those that are to be parallel, without any user intervention: the user must only state what classes have methods that can be executed concurrently with the rest of the code.

In the next section we present some technologies and systems that address similar issues as our work (parallel execution and reflection). In sections 3 and 4 we describe the architecture and implementation (respectively) of our system. Finally we show performance and functional evaluation as well as the conclusions and future work.

2. RELATED WORK

Middleware overcomes non-functional concerns related to system issues (e.g., distribution, communication, parallelism) and frees developers from repetitive and/or intricate code (unproductive and error-prone). Although full transparency rids programmers from distribution issues, this ignores fundamental differences in failure models (local and distributed) and may be too inflexible. To improve middleware and application behavior, the former provides inspection and (re)configuration abilities (reflective middleware [20]) that may be invoked by applications, or configured by declarative policies. Middleware may even be monitored and triggered automatically, according to surrounding environment and execution context [6], and by autonomic systems (e.g., cluster and cloud computing management). Other non-functional concerns are addressed using aspect-oriented programming for clean software design and reuse, including interaction with middleware API and code [14]. Nonetheless, middleware should not impose heterodox programming models, languages, APIs as it hinders developers’ productivity and precludes portability; thus meta-object protocol approaches should be preferred [7].

The research work more related to ours encompasses middleware projects that attempt to improve application performance by providing support for object distribution, deployment, remote execution, or thread parallelism (and parallelization). This entails that process/thread enrollment, synchronization, scheduling and resource management must be also addressed by the middleware and execution environment, or left for the programmer to resolve.

A natural approach is to extend the built-in distributed invocation mechanism in Java VM (RMI) via middleware with enhanced semantics. In the middleware described in [4], group communication services are provided as a new set of Java classes and methods with multicast and parallel execution with wait-by-necessity semantics (futures). In [10], another type system extension is proposed but employs communication qualifiers as attributes associated to objects to be invoked with different strategies (multicast, reliable multicast). Such approaches either lack transparency as they impose new APIs or are unable to allow automatic parallel method execution. Furthermore, both are unable to dynamically configure the number of participating processes.

The work described in [24] and [18] treats applications as bundles of components that may be dynamically deployed in different processes of a distributed system. In [24], application byte-codes (and comprising class types) are transformed to allow application partition and component distribution following hints by the developer. In [18], the Java Isolate API is employed. Although able to deploy code using native types and allowing lazy creation of distributed objects, they do not provide transparency since a component-based application architecture is implied. Thus, automatic parallelization, as we propose, is not allowed.

A different approach, attempting at greater transparency, is to improve application performance by extending the concept of virtual machine encompassing more machines and resources, thus providing a larger JVM on top of a cluster or distributed infrastructure. This consists in a single system image with shared global object space [2, 3], with global thread scheduling, or at least ensuring global identities for transparent distributed threads for safe synchronization semantics [15], and supporting thread migration [25]. Although providing transparency and code portability, these approaches do not extract parallelism automatically as they only manage threads explicitly created by developer’s code.

The work in [19] aims at supporting parallelism and distribution in Java by offering a new API fusing the semantics of shared memory (OpenMP) and message-passing (MPI). Parallel code is defined inside lambda constructs passed as arguments to API methods. Synchronization is explicitly defined in code and scheduling is based on a task queue. Transparency to the programmer is not intended, and the number of processes involved must be known a priori by participants.

In Map-Reduce [11], programmers develop large parallel applications by providing functions to split input and aggregate results, tailoring it to a specific API and programming model. For efficiency, a distributed shared storage (e.g., distributed file system, table, tuple-space) is assumed, which requires some form of coordination and membership management among participating nodes. Existing applications cannot be parallelized transparently on Map-Reduce. Data splitting and task creation are also defined programmatically, not automatically handled by middleware.
The Pro-Active project and its more recent work [1, 5] offer multicast invocation semantics and futures for Java, task distribution, and deal with distributed synchronization and cooperation semantics [5]. It does not aim at transparency, initially imposing a specific API and type hierarchy, then evolving for a prescribed component model combining a set interfaces specifying interaction/cooperation, active objects, introspection and byte-code enhancement [5]. No adaptability is provided for scheduling and parallelization, since the cardinality of multicast groups and the number of participating processes in distributed deployments is either predefined or stored explicit in middleware data structures. Process location was initially fixed but can be abstracted to allow adaptive deployment.

Globally addressed and analyzed, to the best of our knowledge, none of the proposed approaches in the literature combines the necessary transparency (do not impose new class hierarchy, API or programming model, using reflective approach instead), parallelization for performance improvement (transform local method invocations into distributed ones and execute sequential method invocations in parallel with automatic synchronization enforcement), and flexibility (able to adapt and configure middleware behavior w.r.t. scheduling, and being able to dynamically spawn one or more parallel tasks to processes presently available with no need for prior knowledge on group membership).

3. ARCHITECTURE

The architecture of Mercury is closely mapped to its main functionality: load all class adaptors, load information about parallel classes, transform parallel classes, and adapt transformed classes to the various execution environments.

The organization and linking between the several components is shown in Figure 1.

The Application Transformer module replaces the entry point of the program being transformed, loads the transformation code and initiates the original application transformation. This module starts by loading and creating the Adaptor Loader, Metaclass Loader modules, then initializes the necessary data structures and starts the Application Loader.

The Adaptor Loader reads a configuration file (the Adaptors List in Figure 1) stating the available Adaptors, loads each Adaptor code, and creates and registers the corresponding class. For each available computational resource (threads and other parallel execution environments with suitable middleware) there is one Adaptor. Each of these Adaptors are responsible for the creation and termination of the concurrent tasks on one infrastructure.

The Metaclass Loader reads the ClassTransformer Metaclass code, creates it and registers it for use when Application Loader starts loading and processing the Application Code, taking into account the name of the parallel classes and methods stated on the Application Configuration.

During application execution, the organization of classes and objects is the one shown in Figure 2.

Figure 2: Transformed classes organization

After loading the application, besides the original classes (and their objects), some more auxiliary classes are created and instantiated, as described next.

The Transformed Class is a wrapper for the Original Class, to which it has one reference. When an instance of the Original Class is supposed to be created, it is responsibility of the Transformed Class to decide what kind of Adaptation Object to create. No instances of this Transformed Class exist during execution: only instances of the Original Class (locally or on remote computers) and instances of the Adaptation Classes.

The Adaptor Objects serves the purpose of handling all particularities of the underlying middleware parallel execution mechanisms: threads, or processes on remote computers. These objects act as proxies, being responsible for redirecting all calls to the original objects (instances of the Original Classes), and handling all synchronization issues. During execution unmodified objects interact with Adaptor Objects transparently.

The code loading process and initialization of a transformed application is shown in Figure 3.

Figure 3: Application start fluxogram

The first steps are straightforward. First the supplied meta-
class is loaded and registered for latter use: when the information about the parallel classes is read. The supplied Application Configuration files must contain for each classes to be transformed its name, and the name of its parallel methods. This information is stored and will be used when loading the application class, and when invoking their different methods as will be shown in the next section.

The Adaptor Loader Module is responsible for loading the Adaptor List file and create the Adaptation classes referred on that file. This configuration file contains the list of classes. From those names, the Adaptor Loader gets the file with the class implementation and imports it. From this moment forward, the corresponding class is available for use. The names of all Adaptation Classes are stored in a list.

The last step before the application execution is its loading from disk and transformation. Inside the Application Loader, whenever a class is loaded from disk it is verified if its name was read from the Application Configuration file. If the name was present in the configuration file, a class transformation should occur, on the other hand ordinary classes are created normally.

The Python mechanisms used in the interception of the class loading and the class transformations are presented in the next section.

4. IMPLEMENTATION

In the previous section we made an overview of how the application transformation is made. The implementation details and how the presented steps are carried out is now presented: i) interception of the class load, ii) class transformation and iii) adaptor implementation.

4.1 Class loading interception

Every class transformation is performed by a custom metaclass. While Python allows the use of a global metaclass some libraries do not allow its classes to be created by those metaclasses. So, it was necessary to check if it was required to invoke a metaclass to recreate the class after being loaded. Before starting reading the application code, our initialization code installs a custom import function. Whenever a Python file is included, it is our import function that is executed, whose pseudo-code is shown in Listing 2.

```python
1 def my__import_(fileName):
2     mod = __import__(fileName)
3     for name, object in mod:
4         if isClass(object)
5             mod[name] = classTransformer.newClass(
6                 className)
7     return mod
```

Listing 2: Custom file import

On line 2, the original import function loads the code file, which is stored in the mod variable. Then, on line 4, for every loaded object (constants, function, and classes) we check if it is a class. If the class was referred in the Application Configuration file its name is stored in parallelClasses we replace it for a new class created by our classTransformer metaclass (line 6). After every loaded object being verified, and transformed, the list with the original and replaced objects is returned (line 7).

4.2 Class transformation

As stated earlier, the transformation of the Original Classes is performed by a metaclass. In Python, both classes and metaclasses are first-class objects, and as such have the ordinary class methods: _new_, where the instances are actually created, and _init_ used to initialize the state of its instances.

As we want to intervene on the actual creation of the classes, the classTransformer metaclass must inherit from type and have the _new_ method defined. The actual implementation of the classTransformer metaclass is shown in Listing 3.

```python
1 class classTransformer(type):
2     def __new__(cls, name, bases, dct):
3         oldclass = newClass(name + " old ", cls)
4         proxyclass = newClass(name, transformedClass)
5         proxyclass.originalClass = oldclass
6         return proxyclass
```

Listing 3: classTransformer metaclass pseudo-code

In line 3 we build a copy of the original class but with a different name. From this point forward, the original class can be accessed globally by its new name, or locally through the oldclass variable.

In the following lines, a copy of the transformedClass pre-existent class is made (line 4) and we store it in the Original Class (line 5).

The _new_ method concludes returning a reference to a copy of the transformedClass class. From this point forward, in the original code, whenever the programmer creates an instance of the original class, the object creation will be handled by a copy of the transformedClass.

The transformedClass is an object factory, as actually no instances of this class will ever exist. When trying to create an instance of this class, the returned objects will belong to one of the Adaptation Classes. In this case, its is also the method _new_ (shown in Listing 4) that is executed.

```python
1 class transformedClass(object):
2     def __new__ (cls, *args):
3         adaptorClass = selectAdaptorClass()
4         proxyObj = adaptorClass(cls, originalClassName, parallelClasses[originalClassName], *args)
5         return proxyObj
```

Listing 4: Transformed Class

On line 3 the most suitable Adaptor is selected. This has to be done to optimize the allocation of available resources (processors, memory, ...) taking into account the object’s computational requirements. The way the best Adaptor is chosen is out of scope of this paper, as depends on the underlying parallel execution environment.

On the next line, an instance of the selected Adaptation Class is created. The constructor receives as parameters the Original Class, the list of Parallel Methods (those from the Original Class that can be executed concurrently) and the original arguments that are to be passed to the Original Class constructor.

This method returns a proxy (instance of Adaptation Class), that forwards all method calls to a Original Object.

During normal program operation, different instances of the same class can live concurrently and execute on different computers.

4.3 Adaptor implementation

The Adaptor classes are responsible for a series of management activities: i) evaluation of the adequacy of the execution environment, ii) creation of Original Objects on the target platform, iii) proxying of the method invocation, and iv) synchronization of invoked methods.

The fundamental Adaptation Class is the one that takes advantage of local multiprocessors/multicores to allow the efficient concurrent execution of several Original Classes. Besides allowing the concurrent execution of some methods, this
Adaptation Class must also block other method invocations until parallel methods terminate. Other Adaptation Classes, are built on with this classes to accomplish the same objectives, but must create the Original Objects on different computing infrastructures.

1. def __init__(self, originalClass, parallelMethods, *args):
2. self._proxiedObject=originalClass(*args)
3. self._lock = threading.Lock()
4. self._parallelMethods = parallelMethods

Listing 5: Thread Adaptor - initialization

In listings 5 and 6 we present the Thread Adaptor pseudo-code. The initialization of the Thread Adaptor is shown on Listing 5. The Adaptor will have a reference to an instance on the Original Class. This object is created in line 2 and will be responsible for actually performing all computational tasks, instead of the Adaptor.

The lock created on line 3 will be used to guarantee that no methods executed by the same object will execute concurrently. The parallelMethods attribute is necessary to store which methods should be executed in a separate thread from the rest of the application.

In order to synchronize every method invocation and start the needed threads, it is necessary to intercept all method calls. The necessary code, present in the Adaptor Object, is shown on Listing 6.

1. def __getattr__(self, attr):
2. if type(attr) is MethodType:
3. self._name.append(attr)
4. if attr in self._parallelMethods:
5. return self._invokeParallel...
6. else:
7. return self._invokeSerial...
8. else:
9. self._lock.acquire():
10. ret = getattr(self._proxiedObject, attr)
11. self._lock.release():
12. return ret

14. self._lock.acquire():
15. meth = getattr(self._proxiedObject, methodName)
16. ret = meth(*vargs)
17. self._lock.release():
18. return ret

19. def __invokeParallel...(self, *vargs):
20. self._lock.acquire():
21. self._thread = threading.Thread(self._parallelCode..., *vargs)
22. self._thread.start():

Listing 6: Thread Adaptor - synchronization

Before the execution of any method or access to an object attribute, the method __getattr__ is called. This method either returns the value of the attribute or a reference to the method, in Python methods are first class objects. If the access is to an attribute, the access is forwarded to the original object (in lines 9-12).

If it is a method call, two cases are possible: execution of a parallelizable method or not. In both cases, the method returned does not belong to the Original Object but to the Adaptor (lines 5 and 7). Before returning the references to these methods (__invokeSerial... or __invokeParallel...) the name of the called method is pushed to a stack (line 3).

The __invokeSerial__ method gets the name of the method being called (line 16), obtains from the Original Object the actual method (line 17), and invokes it (line 18).

The __invokeParallel__ acts in a similar way, but concurrently with the rest of the program. The code that gets a reference to the called method on the Original Class and executes it (method _parallelCode_ on lines 28-31) runs on a different thread. This thread is started on lines 24-25 inside the _invokeParallel__ method.

The various acquires and releases of the lock guarantee that the execution of a transformed objects is the same as the one of its unmodified serial version: i) when accessing the attributes of the original object (lines 9-11), ii) during the serial execution of the methods (lines 15-19), and iii) when the parallel methods execute by being acquired before it (line 23) and being release at its end (line 31).

4.3.1 Adaptor extensions

We presented a simple Adaptor for multiprocessors, but it lacks mechanisms for efficient use of the various processors. If more threads are created than the number of processors, their concurrent execution can be disadvantaged. When creating the Thread Adaptor it should be possible to state how many processors are available and guard them with a semaphore, initialized with the number of processors or cores. The semaphore would be acquired before starting the execution threads, leading to a lower number of simultaneous threads.

The development of Adaptors to other parallel execution architectures can be made taking as base the Thread Adaptor. The synchronization mechanisms are the same, only varying the Original Object creation mechanism. The creation of the object should be handled by the job distribution mechanism. In [23] we presented a system that allows remote creation and execution of objects without any code changes to the objects. Its integration with Mercury allows the distribution of objects between several remote computers.

4.4 Execution environment

The Mercury code should be injected into the Python virtual machine before the loading of the application. As a last resort the source code of the virtual machine could have to be changed. The code manipulations performed by Mercury do not oblige that.

The transformation code just needs to be loaded before the application code. For instance if, in order to execute one application, the user would write python app.py, to take advantage of Mercury the command would be python mercury.py app.py. Alternatively the Python virtual machine configuration file could be edited, so that with every execution the mercury.py would be loaded.

This mercury.py file contains the code responsible for loading the configuration files, registering the metaclasses, change the include function, and execute the modified code as described in this section.

5. EVALUATION

The evaluation of Mercury is twofold: i) functional, developing sample applications and executing on different environments, and ii) quantitative, where we show the overhead incurred by using Mercury.

Using Mercury we managed to parallelize a Monte-Carlo computation to integrate one function. Instead of treating each random value in a serial way, each task was responsible for obtaining part of the solution. In order to use Mercury, the definition of a class was necessary, while a more simple
solution would only require a loop with the computation code inside.

By using the Adaptor presented in Section 4.3 we managed to execute this computation on a multicore computer with a linear speedup up to the number of processors. It was also possible to integrate Mercury with our previous work [23] to leverage remote computers: the same application was executed across the network on several remote computers.

The overhead incurred by using Mercury is minimal and easily outdone by the parallelization gains, as can be observed in Table 1.

<table>
<thead>
<tr>
<th>Outside Mercury</th>
<th>Inside Mercury</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>modified</td>
</tr>
<tr>
<td>time(s)</td>
<td>1 CPU</td>
</tr>
<tr>
<td>60.01</td>
<td>61.93</td>
</tr>
</tbody>
</table>

Table 1: Mercury evaluation

This evaluation was performed on an Intel(R) Core 2 Quad CPU with 4 cores running at 2.40GHz. The operating system installed is Ubuntu Hardy Server with a 2.6.28-23.46 kernel. The tested application integrates one complex function using the Monte-Carlo method while generating 50 million random points. The modified version has 16 partial integral calculations (each one performed by a different object).

As seen in the previous table, there is an increase of execution time when running the modified version and using Mercury. One of the reasons for the execution time increase is from the rewriting of the application: the inclusion of objects, and the increase of cycle interaction and method calls.

More overhead is added by Mercury. In the version with 1 CPU, different threads for each object were created but serialized with the help of a lock, guaranteeing that they all executed on the same processor. It is observable an increase of about 2 seconds on the execution time leading to an overhead of about 1/8 of a second for each parallel object.

If tasks are longer, these overheads will have a larger impact. Furthermore, with concurrent working processors all overhead is subduced by the gains of concurrent processing.

6. CONCLUSION

With Mercury we manage to automatically and transparently produce parallel Bag-Of-Tasks. The user must only program a serial version of its job (following simple restrictions), define what objects and methods should be executed concurrently and execute that application within Mercury.

The developed application must have objects that are responsible for the execution of lengthy tasks. The serial version must have one initialization part (where objects are created and data partitioned), one or more execution parts (where interactively and independently each object executes its task) and a data recollection phase. This structure matches most of Bag-Of-Tasks problems. Mercury will transparently execute each of the parallel methods on a different processor or computer.

From the experiments made, we can conclude that Mercury can handle most Bag-Of-Tasks and even some applications with a more complex workflows. The overhead incurred by Mercury is low and easily overcome by the gains on programming ease and speedups.

We managed to parallelize applications allowing the execution of concurrent threads on two different environments: i) multicore computers, executing each concurrent thread on a different core, and ii) cluster of computers, using a object distribution systems to leverage different computer scattered on a local network.

7. REFERENCES


A Component-based Approach For (Re)-Configurable Routing in VANETs

Vatsala Nundloll, Gordon S. Blair, Paul Grace
Computing Department, Lancaster University, Lancaster, UK
{nundloll, gordon, gracep}@comp.lancs.ac.uk

ABSTRACT
With the increasing capability of vehicular communications technology, VANETs (Vehicular Ad-Hoc Networks) have witnessed significant development. Many VANET routing protocols have been proposed and operate well in the specific network they have been developed for such as dense or sparse networks but tend not to operate well or interoperate with different networks. This is a challenging requirement and we address this issue by proposing a (re-)configurable implementation that can adjust the behaviour of the protocol to different operating environments and application requirements. One possible way of achieving this is through a component-based architecture consisting of re-usable components that can be plugged in and out of the system. The paper illustrates how this componentization process can be carried out and how the given architecture is then configured to suit varying network states. The paper also identifies commonalities that exist across different routing protocols, leading up to the design of a generic component-based platform which can be used for VANET routing protocol development.

Categories and Subject Descriptors
C.2.4 [Computer-Communication Networks]: Distributed Systems – Distributed applications

General Terms
Design

Keywords
Vehicular ad-hoc networks, components, dynamic re-configuration, configuration, reflection

1. INTRODUCTION
VANETs are an upcoming type of network that typically consist of vehicles interacting with each other, known as Inter Vehicle Communication (IVC), or with the Internet through access points found along the road, known as Road-Vehicle Communication (RVC). The goal of VANETs is to share information such as traffic data and road safety warnings in order to avoid accidents and traffic congestion. Even though VANETs are considered to be a class of MANETs (Mobile Ad Hoc Networks) [1], they exhibit specific challenges due to their distinctive network characteristics. Their salient features [2] can be summarised as: a frequently changing topology, high node mobility and repeated partitioning of the network leading to network disconnections. The operation of VANETs is also largely dictated by the network density and by other factors including drivers’ behaviour. Dense areas experience an overflow of messages sent out to vehicles while sparse areas encounter loss of messages because of lack of vehicles to propagate them. This implies new buffering techniques to capture the messages both during a limited connectivity and a congested bandwidth. On the other hand, drivers may show different reaction to messages received and also their high speed may cause the network topology to change. Moreover, given the high flux of VANETs, considerable weight is imposed on the successful deployment of routing protocols in such networks.

Consequently, it is imperative to devise new routing protocols for VANETs to ensure a reliable message distribution. However it remains a challenging task to find and preserve a route in VANETs, and as such the implementation of routing protocols is a complex and time-consuming task. Given the highly changing nature of VANETs, such an implementation cannot be a static one i.e., one that should be applied for a given set of conditions. Rather, there needs to be a way to make the implementation more dynamic, so that it can be easily configured to varying network conditions, e.g. as a vehicle moves from a dense to a sparse network, or communicating with other vehicles or the Internet.

A component-based approach is one possible and frequently advocated approach to provide a configurable and re-configurable solution. In [3], Szyperski defines components as “units of composition with contractually-specified interfaces and explicit context dependencies only….that can be deployed independently and are subject to composition by third parties”. A component approach allows the software to be designed in a generic manner for ease of re-use. It also allows an application to be configured at deployment time and even re-configured at run time.

The aim of this paper is to illustrate how a component-based architecture can improve the development of configurable and re-configurable VANET routing protocols. For this purpose, we first present a component architecture that we developed for the BBR routing protocol [11] that operates in sparse areas. We then demonstrate how this architecture can be adapted to changing network conditions, i.e. to dense traffic areas. Subsequently, we identify the commonalities that exist among VANET routing protocols and show how our architecture can be re-used in the implementation of a broader spectrum of VANET protocols.

The rest of the paper is organized as follows: Section 2 presents a survey of the different routing protocols used in VANETs. Section 3 introduces the component model. Section 4 explains the componentization of the BBR protocol. Section 5 evaluates this approach and finally, we conclude the paper in Section 6.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM 2009, December 1, 2009, Urbana Champaign, Illinois, USA.
Copyright 2009 ACM 978-1-60558-850-6/09/12...$5.00.
2. SURVEY OF VANET ROUTING
Finding and preserving a route in VANETs is a highly challenging task. A lot of research is being invested to devise effective routing protocols in VANETs to achieve robust message dissemination. This section focuses on the results of two different surveys carried out by the authors in [4, 5] on the various routing categories applied for VANETs. We use these surveys to identify commonalities and differences in routing protocol behaviour. These form the basis of a common component pattern for developing protocols (as described later). Our analysis is illustrated in Figure 1; this shows categories of routing protocols (described next) and discusses features from example protocols.

Ad-Hoc routing protocols can be adaptive or not; adaptive routing is preferred for VANETs given its capability to adjust to a changing topology and reduce overhead. However, because of the highly dynamic nature of VANETs, it has low throughput.

<table>
<thead>
<tr>
<th>Category</th>
<th>Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PRAODV &amp; PRAODVM [4] - use speed and location information of nodes to predict the lifetime of a connection.</td>
</tr>
<tr>
<td></td>
<td>However, they rely heavily on the accuracy of the prediction method.</td>
</tr>
<tr>
<td>Position-based</td>
<td>GSR [9] - shows better delivery rate, better bandwidth consumption and reduced latency over DSR and AODV.</td>
</tr>
<tr>
<td></td>
<td>A-STAR [17] - computes the best possible sequence of junctions for a packet to reach its destination using street maps. Found to perform better than GSR and GPSR because it can choose paths with higher connectivity to deliver a packet.</td>
</tr>
<tr>
<td>Cluster-based</td>
<td>COIN [4] - produces more stable structures in VANETs though a little overhead is incurred.</td>
</tr>
<tr>
<td></td>
<td>LORA_CBF [10] - performs better than AODV and DSR in a large and highly mobile network.</td>
</tr>
<tr>
<td>Broadcast</td>
<td>BROADCOMM [18] - works with simple highway networks. However, it is found to outperform similar flooding based routing protocols.</td>
</tr>
<tr>
<td></td>
<td>V-TRADE and HV-TRADE [19] - organize nodes into groups, where only a small subset of vehicles is selected to rebroadcast the message. These protocols show considerable improvement in performance, but they incur a routing overhead in selecting nodes to do the rebroadcasting.</td>
</tr>
<tr>
<td></td>
<td>IVG [20] - incorporates a waiting time before deciding about re-broadcasting.</td>
</tr>
<tr>
<td>Delay-Tolerant</td>
<td>VADD [12] - uses a carry-and-forward strategy to route packets. A prioritization scheme is applied at an intersection to route the packet in the best possible path. Packet is stored in case forwarding cannot be done. The routing path is continuously recomputed during the process.</td>
</tr>
<tr>
<td></td>
<td>Max_Prop [13] - is used for sparse networks with limited transfer opportunities. It operates in three basic stages – Neighbour Discovery stage, Data Transfer stage and Storage Management.</td>
</tr>
<tr>
<td></td>
<td>MURU [14] - aims at optimising number of hops and providing a robust route connection by using a metric called EDD based on factors such as vehicle position, speed and trajectory.</td>
</tr>
<tr>
<td></td>
<td>PBR [22] - this algorithm focuses on using mobile gateways instead of a roadside fixed infrastructure in order to provide wireless Internet connection to vehicles on a road.</td>
</tr>
</tbody>
</table>

Figure 1 – Results of the survey on routing protocols
Position-based routing is well suited for VANETs since the nodes are known to move along established paths. Since routing tables are not used, no overhead is incurred when tracing a route. However, it can suffer from routing loops and network delays.

Cluster-based routing is performed in clusters. A group of nodes identifies themselves to be part of a cluster and the node designated to be the cluster-head will broadcast the packet to the cluster. Good scalability can thus be provided for large networks but network delays and overhead are incurred when forming clusters in highly mobile VANETs.

Broadcast sends a packet to all nodes in the network, typically using flooding. This ensures the delivery of the packet but bandwidth is wasted and nodes receive duplicates. In VANETs, it performs better for a small number of nodes.

Geocast is considered as a multicast service within a specific geographic region. It normally defines a forwarding zone where it directs the flooding of packets in order to reduce message overhead and network congestion caused by simply flooding packets everywhere. In the destination zone, unicast routing can be used to forward the packet. One pitfall of Geocast is network partitioning and also unfavourable neighbours which may hinder the proper forwarding of messages.

Delay-Tolerant protocols are mostly applied in sparse networks which are found mainly in rural areas but also in dense areas where there can be low concentration of vehicles especially at night time. In such cases, establishing an end-to-end route is not possible. This makes the routing of packets more difficult which is why such routing protocols need to implement a delay-tolerant aspect in order to cater for cases when there are no nodes available to forward a packet. The technique adopted is called carry-and-forward whereby a packet is forwarded only when nodes are available. Otherwise, it is simply carried.

A Quality of Service protocol normally guarantees a good performance through adequate resource reservation and availability of proper infrastructure. In an ad-hoc network such as the VANET, it is very difficult to provide such service unless there is a roadside infrastructure. Nonetheless, the protocols developed for VANETs estimate the stability of a route by analysing factors such as link delay, link reliability, vehicle velocity and trajectory, vehicle position and distance between vehicles; and calculate how long a route can remain connected and reduce the time required to repair a broken connection.

It is clear that there exist a range of routing requirements and network conditions in VANET applications; hence, there is potential for configuration and re-configuration of routing protocols. We revisit this survey in Section 5, identifying the commonalities and differences underlying these protocols; such commonalities form the basis of a common architecture which can be used in the development of VANET routing protocols.

3. COMPONENT MODEL

This section talks about the component model OpenCom used in our implementation. This is a lightweight model to tailor software architectures in a configurable manner to suit the requirements of a system. It also supports run-time re-configuration through reflection, enabling a high level of adaptability of the system. OpenCom employs a small runtime kernel to manage the lifecycle of the components. The authors in [15] provide three case studies showing the capability of OpenCom to adapt a system to different environments. Figure 2 explains the component model; a component advertises its services through an interface and binds to another component through a receptacle. An interface-receptacle binding is shown in Figure 2. The receptacle is used by B to bind to the interface of A in order to access the services provided by A. Likewise, B may also publicize its services through an interface which can be accessed by other components.

4. COMPONENTIZATION OF BBR

As a first step in our investigation, we present a component-based architecture for a protocol called Border Node Based Routing (BBR). This protocol [11] is designed for sparse areas and mainly uses broadcast to reach the maximum number of nodes in the network and unicast when required to send messages from one node to any other node. Its aim is to ensure a reliable message delivery with minimum delay in an ad-hoc network with low node density and high node mobility. It consists of two algorithms: Neighbour Discovery and Border Node Selection algorithms.

Figure 3 shows our component architecture used to implement and execute the BBR protocol; we now explain the operation of BBR in terms of the component implementation.

The Neighbour Discovery algorithm is used to discover the neighbours of the current node and is implemented by the NeighbourDiscovery component. This component seeks the services of the PacketSender and Timer components in order to process the NeighbourDiscovery algorithm. Generally, the neighbouring nodes are found one-hop away within the radio transmission range of the actual node. All nodes advertise their presence by sending out “Hello” beacon messages periodically.

The PacketSender component takes charge of broadcasting these beacon messages to the surrounding nodes. Since this needs to be done periodically, the Timer component is used to schedule the periodic transmission of these “Hello” messages. The information thus obtained is then stored into a Neighbour table by the NeighbourDiscovery component.

The Border Node Selection algorithm is the core process of this protocol used to designate the next node to broadcast a packet received and is implemented by the PacketProcessor component. This node is known as the border node as it lies furthest away within the transmission range of the source node. An example is illustrated in Figure 4. If s is the first node to broadcast a data packet in this scenario, then it is a border node by default. The circle delineates the transmission range of s (labelled as R). C1, C2 and C3 denote the transmission range of s, d and v respectively. Once s has broadcasted a data packet, the next border node needs to be identified among the neighbours of s so as to further forward the packet out of the transmission range.

The DataPacketHandler component is responsible for creating the data packet according to the format stated by the routing protocol. In case the packet is being broadcasted for the first time,
the PacketSender component will simply disseminate the packet to the surrounding network. When a data packet has been received, the DataPacketHandler component decides whether to carry or forward the packet. First, it uses the Forward component to check whether the packet received is a duplicate, in case of which, it is simply discarded. In case it needs to forward the packet, it uses the PacketProcessor component to trigger the Border Node Selection process based on the following two scenarios:

**Case 1:** Only 1 neighbour in neighbour list of received packet.

No border node selection is made. Instead, the PacketProcessor component triggers a periodic broadcast of the message \( p \) times through the services of the PacketSender and Timer components.

**Case 2:** > 1 neighbour in neighbour list of received packet.

Border node selection is triggered and two timer processes are initiated: \( T_{\text{ad}} \) (access delay timer) and \( T_{\text{max}} \) (maximum delay timer), which are coordinated by the Timer component. At the end of \( T_{\text{ad}} \) timer, the node decides whether it has to re-broadcast, depending on the number of neighbours it has at that time. At the end of \( T_{\text{max}} \) timer, the node decides whether it is a border node, in that case, it re-broadcasts or stores the packet, subject to availability of neighbouring nodes.

---

**Figure 4. Border Node Selection**

All nodes lying around the border of the transmission range of \( s \) (\( d, v, t \)) (see Figure 4) represent the next border nodes. These have the least common neighbours (normally <= 1) compared to the neighbours of \( s \). Table 1 shows how this Border Node Selection algorithm is computed for \( s \), based on information obtained from the Neighbour table. If the border nodes do not encounter any neighbours, the DataPacketHandler component will simply decide to store the packet using the Store component until potential new neighbours are met. Note that the Forward component is also responsible for maintaining the buffer table of the data packets. The Timer component acts as a clock and is in charge of scheduling various activities such as the broadcast of the data packets, periodic broadcast of “Hello” messages and coordinating the Border Node Selection process.

<table>
<thead>
<tr>
<th>Node</th>
<th>Neighbours</th>
<th>Common Neighbour #</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>{t, a, c, d, v}</td>
<td>Nil</td>
</tr>
<tr>
<td>d</td>
<td>{s, c}</td>
<td>1</td>
</tr>
<tr>
<td>v</td>
<td>{s, a}</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>{s, v, c}</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>{s, d, a}</td>
<td>2</td>
</tr>
<tr>
<td>t</td>
<td>{s}</td>
<td>0</td>
</tr>
</tbody>
</table>

The BBR protocol adopts a carry-forward strategy in order to minimize any packet loss when there are no nodes. Since packet loss is highly likely to occur in environments where there is low node density, providing measures to counter this problem in a sparse VANET is a fundamental task.
5. EVALUATION

The aim of this section is to evaluate our architecture for BBR at three levels. First, we evaluate how it can be configured to adapt to changing environment conditions. Second, we illustrate the re-configurability based upon the use of fine-grained components. Third, we identify the commonalities that exist among the various routing protocols presented in Section 2, and show how this architecture is re-usable. We show that the implementation of VANET routing can be geared towards a common architecture.

5.1 Configurability and Re-configurability

In the BBR protocol, when no border node is found (see Section 4), the packet received is broadcasted p times. This parameter p is configurable such that it can be increased or decreased to suit the given network density. Similarly, there are other parameters that affect the functioning of the routing protocol. For example, during the Border Node Selection process, at one point in time, nodes have to take decisions about whether they should re-broadcast a packet (T_d) or whether they are the next border node to carry or forward a packet received (T_max). This decision is based on a given number of neighbours. These parameters can be configured in the architecture at design and run-time; e.g. the number of neighbours tested in T_max can be decreased in a low density so that a border node may still decide to re-broadcast.

Moreover, although the BBR component architecture is developed for a sparse network, it can be adapted to be deployed in a dense network. In this case, there is no need to carry packets given the availability of nodes, implying changes to the configuration such as removing the Store component and adjusting the controller parameters as mentioned above.

The Packet Processor component can be divided into sub-components to perform tasks such as:
1) Broadcasting a packet p times when no border node is selected.
2) Initiating the Border Node selection process.

In the implementation, these tasks are used with the same frequency and are encapsulated within the same component. However, in dense areas, it is possible that there will always be neighbouring nodes, implying that the second task will be executed most. This is encapsulated within a sub-component that is plugged into the system. This shows that the component architecture can be configured flexibly to integrate fine-grained components without affecting the operation of the protocol.

5.2 Commonalities in VANET Routing

We have considered 3 scenarios to identify commonalities and variabilities among routing protocols. All VANET routing methods perform sending and receiving of a packet, scheduling the execution of particular tasks or discovering the surrounding neighbours. The re-usable components that perform these functions have been enclosed within a dashed line in Figure 3 to highlight how we exploit these commonalities.

Scenario 1: Routing protocols from the same category. The category is Broadcast; we compare TRADE [16] to BBR.

TRADE: Since this protocol is similar to the BBR protocol, all the components of BBR can be re-used to adapt it to the requirements of TRADE. The latter also mentions a border node allocation process in order to broadcast a packet in the network, which is same as the BBR border node selection process. The other routing protocol in this category, BROADCOMM [18], also exhibits the same features as BBR, hence, implying that the same component architecture can be utilised as well.

Result: For protocols from the same category, a minimal change needs to be brought to the architecture such as the creation of components that process data packets as per the protocol specification. Our architecture can easily be adapted to another protocol, thus facilitating the new protocol’s implementation.

Scenario 2: Protocols from different categories. The category is Delay-Tolerant and the protocols are VADD [12] and Max_Prop [13].

VADD: In VADD, three different modes are applied for packet dissemination: Intersection, StraightWay and Destination. Upon analyzing VADD, it is found that its implementation consists of the following components: NeighbourDiscovery, PacketSender, DataPacketHandler, Store, Forward, StreetMapData (to obtain street map data), IntersectionMode, StraightWayMode and DestinationMode. These last three components show different ways of forwarding a packet at an intersection, on a straight road and at a destination respectively. If no node is found in any of these modes, then the packet is stored. This analysis shows that most of these components are already available from the given architecture of BBR and are re-usable. The two components labelled “BBR” in Figure 3 can be replaced in VADD in the following way: The DataPacketHandler component will need to be customized so that it creates packet formats for VADD. The PacketProcessor component can be replaced by the Intersection, StraightWay and Destination modes, which represent the core function of the protocol. All of these will be supplied with street map information from a StreetMapData component. [12] also mentions variations in the VADD algorithm, namely L_VADD, D_VADD, MD_VADD and H_VADD. Each of these has a different approach to deal with the Intersection mode. This means that the Intersection component can be further refined into sub-components. H_VADD, being a hybrid of the other versions, can be implemented as a controller to coordinate the actions.

MAX_PROP: Max_Prop operates in three basic stages (Neighbour Discovery, Data Transfer and Storage Management). The common components that we have identified are NeighbourDiscovery, PacketSender, Timer, Store, Forward and DataPacketHandler. However, the last two components need to be adapted: the Forward component must accommodate deletion of packets based on whether they have been acknowledged as delivered. The DataPacketHandler must be customized to the packet format of Max_Prop. The new component to be added is a Prioritizing component (used to allocate priority to messages before they are actually distributed). Since the functions coordinated by the latter component are complex, it may need to be refined into sub-components. For instance, one of its functions is to calculate the probability of node meetings, required in message routing. This can be separately developed as a sub-component to be used in other protocol implementations.

We repeated the scenario; this time with a different category i.e. Cluster-Based and the protocol analyzed is LORA_CBF [10].

LORA_CBF: This protocol can be summarized in four stages: Cluster Formation, Location Discovery, Routing of Data Packets.
and Maintenance of Location Information. These can in turn be mapped onto the following components (similar to those in BBR): NeighbourDiscovery, Timer, PacketSender, Store, Forward. The PacketProcessor component needs to be replaced by a new one to perform Cluster Formation instead of Border Node Selection.

**Result:** The scenario shows the potential for re-usable components. The component architecture can be configured to accommodate the variabilities of different categories of routing, combining reusable components with the addition of fine-grained components specific to individual protocol behaviour.

**Scenario 3:** Show different component architecture of a routing protocol from a different category, yet identifying a percentage of commonality. The category is QoS (Quality of Service) and the protocol analyzed is MURU [14].

MURU: The functionalities of MURU are: calculate the EDD metric, capture street map data, calculate shortest trajectory to the destination, find vehicle location, check vehicle speed, generate RREQ message and perform a pruning mechanism. Each of these can be mapped onto individual components, among which those common with BBR are: DataPacketHandler when customized to properly format packets and also create RREQ messages, PacketSender and Timer (must be adapted to perform the pruning to allow a node to delay the forwarding of a RREQ message). No Store and Forward components are required.

**Result:** This shows that when variabilities exceed commonalities for a particular routing strategy, then there is little potential for configuration of the component architecture. However, the common architecture remains useful particularly when adapting behaviour e.g. if it was required to move from the simple BBR behaviour to more complex MURU behaviour.

6. CONCLUSIONS & FUTURE WORK

In this paper, we have highlighted the benefits of a component-based approach in the implementation of VANET routing protocols. We have demonstrated that most protocols within our categories (for example position-based, cluster-based) share common patterns and we also demonstrated how these can be mapped on to common components. The variabilities can also be mapped on to new components specific to the protocol and can be plugged into the system. The common components can be separated from the more protocol-specific ones and hence provide a common platform in the implementation of VANET routing. We have shown that, through such an approach, the implementation is configurable at deployment time. As future work, we intend to show how such component architectures can be dynamically re-configured in order to adapt the system to the ongoing changing network conditions; hence facilitating both the development and deployment of more flexible VANET routing protocols.

7. ACKNOWLEDGEMENTS

This work is funded under the CONNECT Project (http://connect.forever.eu) (FP7 Theme / ICT-2007.8.6, FET Proactive 6: ICT Forever Yours).

8. REFERENCES

A Reflective Middleware Architecture for Simulation Integration

Leila Jalali  
Department of Computer Science  
University of California, Irvine  
CA 92697 USA  
jalalil@uci.edu

Nalini Venkatasubramanian  
Department of Computer Science  
University of California, Irvine  
CA 92697 USA  
nalini@ics.uci.edu

Sharad Mehrotra  
Department of Computer Science  
University of California, Irvine  
CA 92697 USA  
sharad@ics.uci.edu

ABSTRACT
This paper presents a reflective middleware architecture for simulation integration based on structural reflection and metamodel concepts. The proposed architecture extracts the simulator information as metamodels from the base-level simulators, determines the required features and modules using semantic constraints, and reflects the modified features to the base-level. It is shown that the reflective middleware architecture addresses various challenges in simulation integration. It also enables a design that is more adaptable, flexible and easier to extend. We present a detailed case study from the emergency response domain, where simulations are critical, to illustrate the potential benefits of applying the proposed architecture.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures Patterns (Reflection)

General Terms
Design

Keywords
Reflective Middleware, Simulation Integration, Structural Reflection, Metamodel

1. INTRODUCTION
Modeling and simulation (M&S) are accepted problem solving methodologies for the solution of many real-world problems. There is a wide variety of methods for studying models of real-world systems using software designed to imitate the system’s operations or characteristics, often over time. There are several advantages of using simulations instead of experimenting with the real system itself: simulation is cheaper, quicker, and enables what-if analyses for better system design [1, 12]. This is particularly true in domains such as emergency response where the need to simulate disasters, their impact and the efficacy of response methods are often validated via simulations. These simulations (e.g. earthquake simulators, fire simulators) are often developed by domain experts who have a clear understanding of the field and are often difficult to recreate.

Building complex simulations to understand the joint effect of multiple phenomena (spread of hazardous material as a result of an earthquake, impact of earthquake on the cellular network in a region) is useful. Consider an existing simulator in the crisis management domain, HAZUS-MH, which estimates losses from potential hazards such as earthquake, wind, flood or release of hazardous material [18]. HAZUS-MH can be used in a scenario in which a hazardous material has been released as a consequence of an earthquake. In this scenario, authorities may wish to block traffic, the blocked roads may be captured within a transportation simulator, e.g. the INLET (Internet-based Loss Estimation Tool) transportation simulator [19]. Such integration is useful to understand various factors that can adversely delay evacuation times or increase exposure and consequently used to make decisions that can improve safety and emergency response time. In the example above, one can study the actual effect of blocked roads on the number of people affected by the hazmat release. Additionally, traffic loads in the blocked area are reduced enabling emergency response teams to reach the crisis sooner.

One approach to studying joint impact is to build monolithic simulations; this is a cumbersome process; economic and organizational constraints make it infeasible to build these complex distributed simulations entirely from scratch. The second approach is to leverage existing simulators (developed by experts); since each simulator has its own models and entities, the integration is a big challenge. In this paper we focus on using reflective middleware solutions for integration and enhancement of simulations built by domain experts that are likely available. We design a reflective architecture for simulation integration in which interoperability of different simulators can be ultimately achieved via shared metadata. In Section 2, we discuss the related work in simulation integration and its limitations. In Section 3, we describe the reflective architecture, challenges, and our metamodel. We illustrate the power of the reflective model using real-world case studies in Section 4 and compare the reflective approach with a popular simulation integration framework (the High level architecture). Finally we conclude in section 5 with future research directions.
2. RELATED WORK

To the best of our knowledge, simulation integration has been applied in two domains – (a) military command-and-control and (b) games. The U.S. Department of Defense (DoD) has promoted the development of distributed simulation standards to provide a common framework in which simulators can be integrated. These include standards such as SIMulator NETworking (SIMNET) [4], Distributed Interactive Simulation (DIS) [15], Aggregate Level Simulation Protocol (ALSP) [5], High Level Architecture (HLA) [6, 7]. These standards provide specific services for interoperability in niche applications, for example DIS for human-in-the-loop simulators or ALSP for war games. The recent HLA effort has become the defacto standard technical architecture for military simulations in the United States – it aims to promote interoperability, scalability, and reusability between simulators. One of the central components of HLA is the Runtime Infrastructure (RTI); ORBs and CORBA services are candidate tools for implementing HLA RTIs. While HLA led to some new insights in simulation integration, its broader applicability for general simulation integration is questionable. It is a complex standard designed specifically for the military domain and is not transparent enough – too much low level knowledge is needed from the practitioner. Additionally it requires the participants to agree on a common interpretation of data that is produced and exchanged between them. Recently simulation integration methods have been used in the game community [8, 9, 10, 11] primarily to support interoperability. As in the case with the HLA architecture, solutions here are prescriptive - they force the developers to provide a particular functionality or to conform to specific standards to participate in the integration process.

In contrast to the above approaches where the participating simulators must conform to predefined standards, our goal is to leverage existing simulators, as is, while enabling data interchange between them. Existing middleware frameworks such as CORBA, DCOM or RMI [2, 3], XML tools support data exchange between software applications; supporting semantic interoperability requires capabilities beyond what these frameworks can provide. Time synchronization across multiple simulations is one such challenge. Time synchronization services offered by traditional middleware frameworks typically need the participants to agree on a common interpretation of time and on common time advancement methods. Although existing simulators model time in disparate ways, it is difficult to achieve a joint integration without a clear understanding of how each simulator represents and advances time – this is essential for accurate simulation.

3. A REFLECTIVE ARCHITECTURE FOR SIMULATION INTEGRATION

There are several challenges that must be addressed to fully realize simulation integration. The first challenge in modeling complex scenarios using multiple simulators is the analysis of cause-effect relationships between those simulators. The second challenge arises from the fact that each simulator uses its own models and entities; these must now be integrated in the context of a single simulation. The simulators need to exchange the data and have a correct interpretation of the data they send and receive. It is necessary to analyze the data types used internally by the simulators. Therefore, there are challenges in data transformation and data integration. Time synchronization is yet another challenge. When integrating simulators, there is a need for synchronization of time between the different models. The simulation clock that controls simulation time during execution of a simulation resides within each simulator itself. Time synchronization mechanisms are needed to ensure causal correctness for models that use different time advancement mechanisms.

Given the potential black-box nature of simulators developed by experts in diverse domains (earthquake engineering and transportation systems in the introductory example), we believe that achieving a completely automated plug-and-play integration of simulators is a very difficult, if not infeasible challenge. Our goals are more modest – we intend to develop enabling tools that
will simplify the task of simulation integration with a wide range of simulators that vary in the degree to which they expose their interfaces and implementations. Our solution does not require simulator developers to adhere to a strict programming interface or conform to particular design styles - the ability to flexibly interoperate with multiple simulators is our goal.

We will use reflective middleware solutions to provide a principled, yet flexible approach to support the development of simulation integration platforms. There are two main forms of reflection in middleware platforms: structural and behavioral reflection [16]. In solutions for simulation integration, we are primarily interested in abstracting out the structural aspects of the underlying simulators. Structural reflection is concerned with the reification of the underlying structure of objects or components, e.g., in terms of the needs of the high-level integration task. The expected mode of operation is that the actual interactions between simulations are specified and altered by a specialist who understands the purpose of the different simulators (and the rationale for their integration) but not necessarily their details.

### 3.1 Reflection for Simulation Integration

In this section we present a two-level reflective architecture for simulation integration, as illustrated in Figure 1. In the proposed architecture, integration of different simulators can be ultimately achieved by using the meta-level for specifying/modeling the properties of the different simulators and reasoning about the interactions among the different simulators – i.e. what we intend to design and develop is a meta-simulator. The meta-level is built on base-level simulators; reification of base-level entities yield data structures at the meta-level, modified features of these structures that implement the integration are then reflected to the base-level. A closer look at the base-level simulators themselves reveals that the structural aspects of the simulation application are not merely in the simulator code - backend databases and models stored in domain-specific formats contain aspects of the simulators that may need to be explored as well -- in general, there can be many kinds of meta-level entities to cover various integration aspects. By using the metamodeling capability the model elements that need to be integrated can be extracted.

A major challenge toward the realization of reflective architectures for simulation integration is related to the complexity associated with reification. Many reflective systems [2, 3] provide access to their internal operations in terms of a composition graph, describing the dependencies between their components. Such an approach requires the specification of all interfaces and objects involved. However, in using existing available simulators we may not have access to all details of the simulators including the specification of their interfaces and objects. In our approach, we formulate the metamodel that captures concepts of interest using a publish-Subscribe mechanism for data exchange – here, subscribers (the simulation integration tasks) express interest in aspects that they want to observe (implemented by base-level simulators) – when changes in these monitored aspects occur at the base simulators, the meta-level entities receive information or updates of interest via publishers. A pre-existing set of ontology models assist in the matching process for the pub-sub implementation of the simulation integration task – these include domain ontologies that are representations of knowledge in a well-circumscribed domain. Interoperability of different simulators can be achieved by sharing and understanding the metamodels. Implementing the semantic constraints for simulation integration is a human in the loop process which results in the annotations that are invisible to base-level computation and are provided to the meta-level. In the following section we propose our metamodel.

### 3.2 Meta-model

In this section we attempt to construct a metamodel that helps to reify the main features of the simulation platforms. Metamodels make the underlying simulators more understandable by abstracting out lower-level details of integration and interoperability. The main challenge in deriving a metamodel is what features need to be present in the metamodel. Since our metamodel needs to take several domain expert simulators into account, the metamodel should be comprehensive, yet extensible. The careful examination of the features in various simulators of the different domains has allowed us to identify and categorize common features using key classes. We describe the process and tools used in deriving key structural aspects of simulators to assist the integration process.

Given a set of sample simulators, we used Creole as an Eclipse plug in to examine source code dependencies and to extract the simulator’s features. Since Creole did not help us with complex and large simulators, we implemented a parser using a tool for large scale code repositories search [17] to extract the entities and attributes from a Java simulator using the simulator’s source code, interfaces, and databases. Then we group extracted information into features to capture the structure of the simulator. The features are put into the same class if they are considered equivalent. The key classes in our metamodel will be: model elements, features which could be structural features or behavioral features, and constraints. Model elements are the main elements of a simulation and can be captured from the interfaces, the source code, or databases. Since we are interested in structural reflection, currently we only use structural features which include classes and attributes. We may also take behavioral features into account to represent operations and associations in future. Constraints are the number of limits for the simulation parameters in the simulation model.

Figure 2 shows the key classes that comprise our metamodel. We construct our metamodel using UML (Unified Modeling Language). The reflective UML metamodel has several metamodel propositions including class diagrams for describing the main elements and the static relations among them. We used the Eclipse Modeling Framework (EMF) to define and customize our metamodel which also allows the automatic generation of...
tools (such as a repository). We will explain some examples of the key classes in our case study in next section.

4. CASE STUDY
In this section we develop a case study for simulation integration using two available existing simulators – the primary goal is to validate the proposed reflective architecture and understand issues in its realization. The two simulators are (a) a fire simulator that simulates the effects of fire and smoke inside a building and (b) an activity simulator that model a response activity – evacuation.

4.1 Fire Simulator: CFAST
CFAST, the Consolidated Model of Fire and Smoke Transport, is a simulator that simulates the impact of fires and smoke in a specific building environment and calculates the evolving distribution of smoke, fire gases, and temperature [14]. CFAST have several interfaces to input the parameters that contain information about the building geometry (compartment sizes, materials of construction, and material properties), connections between compartments (horizontal flow openings such as doors, windows), fire properties (fire size and species production rates as a function of time), and specifications for detectors. The simulator produces outputs that contain information about temperatures, ignition times, gas concentrations such as CO and CO2, and etc.

Figure 3-a shows the representation of CFAST using our metamodel. Since in CFAST we only have access to several interfaces and we do not have access to the source code and databases, the model elements are captured from the interfaces. In the figure the model that we try to capture is CFAST-fire. The model elements include time, fire, and geometry. Each element has its own structural features. We try to make our model general and easy to extend to be used for any other fire simulator. A major property associated with any simulation is time. CAFST is a time stepped simulator which has a simulation environment interface in which the simulation time duration and time intervals can be defined. Structural features of time are simulation start time, time interval, and current time. The second model element is the fire which has several structural features, such as location, fire type, time, temperature, etc. Finally, geometry is another model element in CFAST which includes information on the building geometry such as compartments and connections between compartments. CFAST has also some limitation on its parameters that can be specified in the metamodel using constraints. There are two significant properties of the metamodel that we want to briefly mention: flexibility and scalability. First our metamodel is flexible because it is easy to add other key classes to the metamodel using UML specifications. Second, we made our metamodel scalable by developing the meta-adaptors. When using another fire simulator we can adapt its metamodel to the CFAST metamodel using the adaptor. Detailed information about the meta-adaptors is beyond scope of this paper.

4.2 Activity Simulator: Drillsim
Drillsim is a simulation environment that plays out the activities of a crisis response (e.g., evacuation), which is a multi-agent system that simulates human behavior in a crisis [13]. Agents represent an evacuee, a building captain, etc. Every agent has a set of properties associated with it, such as physical and perceptual profile (e.g., range of sight, speed of walking) and the current health status of the agent (e.g., injured, unconscious). At any given time, agents are associated with a given location in the

Figure 3. (a) CFAST Metamodel, (b) Drillsim Metamodel
geographical space. Indoor space consists of floors, rooms, corridors, stairways, etc. Outdoor space is represented by a grid in Drillsim. Figure 3-b shows Drillsim using our metamodel key classes extracted from the source code and databases.

4.3 Drillsim-CFAST integration

In order to integrate Drillsim and CFAST using our reflective architecture we first need to extract semantic constraints by using the metadata to capture where we need to integrate the two simulators. The careful examination of the features in the metadata and the cause-effect analysis allow us to extract the following constraints. The main constraint is that fire from CFAST can affect an agent’s health in Drillsim. Therefore we need to extract the harmful condition caused by fire and smoke from CFAST and update agent health condition in Drillsim. The following are the examples on how the integration enables information interchanged between two simulators:

- Harmful condition from CFAST can affect someone’s health in Drillsim.
- Agents in Drillsim can talk about the fire and its location – this will prevent agents from entering dangerous areas.
- Smoke from CFAST can decrease someone’s visual distance in Drillsim.
- Harmful conditions from CFAST can affect the evacuation process in Drillsim e.g. increase walking speed.

Since the harmful condition in CFAST is associated with the specific time and location, we need a time synchronizer and geometry transformer. Each simulator has its own internal time management mechanism. This implies that we need a time synchronization method to guarantee the logical correctness and causality during simulation. The two simulators also use different geometry representations – translators are required to translate specific locations in CFAST to corresponding locations in Drillsim. In the following sections we will describe each integration module in more detail.

4.3.1 Data Issues

We discussed one example of the semantic constraints between the two simulators. In our example harmful conditions extracted from CFAST can affect the health of agents in Drillsim. In the data management module we need to update an agent’s health level in Drillsim based on the harmful condition caused by fire and smoke in CFAST. In general, the data management module provides data transfer that preserves the meaning and relationships of the data exchanged between two simulators. Since we are working with existing simulators, we cannot use the methods based on the common representation of data. Each simulator may have its own data representation which can not be easily modified. We used data translators that work based on the constraints. Clearly one of the most difficult portions is to extract these constraints. Finally if the data translators are implemented correctly, they can provide immediate conduits to publish or subscribe to information.

4.3.2 Time Synchronizer

Time synchronization can be implemented differently in simulators: clock synchronization and timescale transformation are two common techniques. In clock synchronization the simulators’ clocks have the same time at any given moment which is a costly approach and sometime impossible because the internal time advance manner of a simulator might not be accessible. In timescale transformation we can transform the internal time of one simulation into the internal time of another simulation. When a message or update is sent to another simulator, it has a timestamp which is transformed to the timescale of receiver. For messages sent over multiple simulators the transformation is repeated. Time stamp transformation can be achieved by computing the difference from message creation and its arrival. Since CFAST and Drillsim are both time stepped simulators the timescale transformation can be performed by means of time calibration.

4.3.3 Geometry Transformer

Geometry translators have the responsibility of performing coordinate conversions between geographies that use different coordinate systems. To create this translator module we create a set of guide points in both geographies and determine a coordinate transform matrix. Figure 4 illustrates how to use the proposed architecture in our case study 1) it allows to extract the simulator design information as metamodels from the base-level simulators using their interfaces, source code, and databases. In CFAST we do not access to the simulator’s source code and databases but we can access several interfaces that contain simulation parameters. On the other hand, Drillsim does not provide us with powerful interfaces but we can access the source code and databases; 2) by using the metamodels capability the model elements that need to be integrated can be extracted. This step is a human-in-the-loop process and we use the semantic constraints for integration. In our case study, the fire and smoke from the fire simulator can affect someone’s health in the activity simulator; 3) using the main model elements involve in the integration we need: (i) the data translator to transform the data on fire and smoke and to update agent’s health, (ii) the time synchronizer to synchronize the time of fire and smoke to the time in Drillsim, (iii) the geometry transformer to perform coordinate conversion between the two simulators. Finally; 4) by using reflection we reflect the modified features to the base-level that means we modify health conditions using the source code and databases in Drillsim.

Comparing with HLA: Table 1 presents a brief comparison of the reflective architecture to HLA. Using HLA outside the defense domain such as our case study is very complex, if not impossible. In HLA low level knowledge needed from participants. Each simulator must use the common data format that leads to simulations that are very closely coupled to an underlying database. Since the HLA environment is a fully distributed simulation environment, the simulators must fully conform to the designated features of the HLA standard. Note that transforming existing simulators to conform to the standard may not always be feasible. In our reflective architecture each simulator can have its own data representation, internal time
management, and data management. Therefore, we do not force
the simulators to change their internal properties. Another
advantage of our reflective architecture is separation of concerns,
that is, separate the concerns related to the simulation domain
from those related to the integration mechanisms. Additionally it
provides a design that is more adaptable, flexible and easier to
extend.

Table 1. Comparison between HLA and Reflective
Architecture for simulation integration

<table>
<thead>
<tr>
<th>Criterion</th>
<th>HLA</th>
<th>Reflective Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective</td>
<td>– Interoperability</td>
<td>– Semantic Interoperability</td>
</tr>
<tr>
<td></td>
<td>– Reusability</td>
<td>– Reusability</td>
</tr>
<tr>
<td></td>
<td>– Flexibility</td>
<td>– Flexibility</td>
</tr>
<tr>
<td>Domain</td>
<td>– Defense</td>
<td>– Flexible via use of domain ontologies</td>
</tr>
<tr>
<td>Complexity</td>
<td>– Low level knowledge needed</td>
<td>– No need to conform the internal properties</td>
</tr>
<tr>
<td></td>
<td>– Lack of semantic interoperaity</td>
<td>– Semantic constraints implemented at the meta-level</td>
</tr>
<tr>
<td>Time Management</td>
<td>– Optimistic and conservative</td>
<td>– Allows Timescale transformation</td>
</tr>
<tr>
<td></td>
<td>methods</td>
<td></td>
</tr>
<tr>
<td>Separation of Concerns</td>
<td>– Merges domain-specific and</td>
<td>– Separate concerns related to simulation domain to those</td>
</tr>
<tr>
<td></td>
<td>integrated simulation aspects</td>
<td>related to integration mechanisms</td>
</tr>
</tbody>
</table>

5. CONCLUSIONS AND FUTURE WORK
In this paper we proposed a reflective middleware architecture for
simulation integration that implements structural reflection to
alleviate the flexibility issues in current simulation integration
techniques. In this architecture, the meta-level is structured as a
series of metamodels representing the various simulators. We
have implemented a detailed case study using two available
simulators and illustrated the utility of the reflective architecture.
In the near term, we intend to extend our approach to integrate
more than two simulators. Future research will focus on
addressing challenges in the complexity associated with
reification, generalizing the metamodels for other evacuation and
fire simulators, integrating simulators in other domains including
earthquake and transportation simulators as well as supporting
non-Java simulators.

6. REFERENCES
computer simulation, 2nd ed., Springer, New York, USA.
Modeling: A Comparison of HLA, CORBA and RMI,
as a Real-time Logistic Control System: AGV Control with
Simple++, The New Simulation in Production and Logistics
Prospects, Views and Attitudes ed., Germany, 245-255.
Technical Report 7102, MA: BBN Systems and
Technologies, Cambridge, Massachusetts.
Level Simulation Protocol, Summer Computer Simulation
Conference, Baltimore, Maryland, 953-958.
Architecture(HLA) and Run Time Infrastructure(RTI), SICE
Annual Conference, Japan, 1136-1139.
Computer Simulation Systems: An Introduction to the High
Level Architecture, New Jersey, Prentice Hall.
Model Searching Algorithm Based on Response Order and
Access Order in War-Game Simulation Grid, Edutainment,
Springer-Verlag Berlin Heidelberg, 627-637.
Migration in Grid-Based Virtual Wargame Collaborative
Environment, 606-615.
Processing in Multithreading and Multiprocessor
gaming architecture for incident management training,
Simulation, Proceedings of the Winter Simulation
Conference, 904-913.
[13] Balasubramanian, V., Massaguer, D., Mehrotra, S.,
Venkatasubramanian, N., 2006. DrillSim: A Simulation
Framework for Emergency Response Drills, Intelligent and
CFAST– Consolidated Model of Fire Growth and Smoke
Transport (Version 6) User’s Guide, NIST Special
Publication.
in the Evolution of DoD Warfare Modeling and Simulation,
Proceedings of the IEEE 83(8), 1138-1155.
for Reflective Middleware, Communications of the ACM,
45(6), 33–38.
tool for test-driven source code search. In Companion To the
22nd ACM SIGPLAN Conference on Object Oriented
Programming Systems and Applications Companion.
Montreal, Quebec, Canada, 917-918.
Methodology. HAZUS-MH User Manual, FEMA.
Development of a Web-based Transportation Modeling
Platform for Emergency Response. 8th Conference on
Earthquake Engineering, San Francisco.
Using Machine Learning to Maintain Pub/Sub System QoS in Dynamic Environments

Joe Hoffert, Daniel Mack, and Douglas Schmidt
Department of Electrical Engineering & Computer Science, Vanderbilt University, Nashville, TN
jhoffert@dre.vanderbilt.edu, dmack@isis.vanderbilt.edu, schmidt@dre.vanderbilt.edu

ABSTRACT
Quality-of-service (QoS)-enabled publish/subscribe (pub/sub) middleware provides powerful support for scalable data dissemination. It is hard, however, to maintain specified QoS properties (such as reliability and latency) in dynamic environments (such as disaster relief operations or power grids). For example, managing QoS manually is often not feasible in dynamic systems due to (1) slow human response times, (2) the complexity of managing multiple interrelated QoS settings, and (3) the scale of the systems being managed. For certain applications the systems must be able to reflect on the conditions of their environment and adapt accordingly.

Machine learning techniques provide a promising adaptation approach to maintaining QoS properties of QoS-enabled pub/sub middleware in dynamic environments. These techniques include decision trees, neural networks, and linear logistic regression classifiers that can be trained on existing data to interpolate and extrapolate for new data. By training the machine learning techniques with system performance metrics in a wide variety of configurations, changes to middleware mechanisms (e.g., associations of publishers and subscribers to transport protocols) can be driven by machine learning to maintain specified QoS.

This paper describes how we are applying machine learning techniques to simplify the configuration of QoS-enabled middleware and adaptive transport protocols to maintain specified QoS as systems change dynamically. The results of our work thus far show that decision trees and neural networks can effectively classify the best protocols to use. In particular, decision trees answer questions about which measurements and variables are most important when considering the reliability and latency of pub/sub systems.

Categories and Subject Descriptors
C.2.4.b [Distributed Systems]: Distributed applications

General Terms
Performance

Keywords
QoS, Pub/Sub, Machine learning, Dynamic environments

1. INTRODUCTION

Emerging trends and challenges. The number and type of distributed systems that utilize publish/subscribe (pub/sub) technologies have grown due to the advantages of performance, cost, and scale as compared to single computers [8, 10]. Examples of pub/sub middleware include Web Services Brokered Notification (www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn), the Java Message Service (JMS) (java.sun.com/products/jms), the CORBA Event Service (www.omg.org/technology/documents/formal/event_service.htm), and the Data Distribution Service (DDS) (www.omg.org/spec/DDS). These technologies support data propagation throughout a system using an anonymous subscription model that decouples event suppliers and consumers.

Pub/sub middleware is used in a wide spectrum of application domains, ranging from shipboard computing environments to grid computing. The middleware supports policies that affect the end-to-end QoS of the system. Common policies across different middleware include persistence (i.e., saving data for current subscribers), durability (i.e., saving data for subsequent subscribers), and grouped data transfer (i.e., transmitting a group of data atomically).

While tunable policies provide fine-grained control of system QoS, several challenges emerge when developing pub/sub systems deployed in dynamic environments. Middleware mechanisms used to ensure certain QoS properties for one environment configuration may not be applicable for another configuration. For example, a simple unicast protocol (such as UDP) may provide adequate latency QoS when a publisher sends to a small number of subscribers. UDP could incur too much latency, however, when used for a large number of subscribers due to the publisher needing to send to each individual subscriber.

Challenges also arise when managing multiple QoS policies that interact with each other. For example, a system might specify low latency QoS and reliability QoS, which can affect latency due to data loss discovery and recovery. Certain transport protocols (such as TCP) provide reliability but unbounded latencies due to acknowledgment-based retransmissions. Still other protocols (such as TCP) provide reliability but bounded latencies due to acknowledgment-based retransmissions. Still other protocols, (such as lateral error correction protocols [1]) balance reliability and low latency, but only provide benefits over other protocols for specific environment configurations.

Determining when to modify parameters of a particular transport protocol or switch from one transport protocol to another is hard. Moreover, human intervention is often not responsive enough for the timeliness requirements of the system. The problem of timely
response is exacerbated by increasing the scale of the system, e.g., increasing the number of publishers or subscribers.

Solution approach → ADaptive Middleware And Network Transports (ADAMANT). ADAMANT is QoS-enabled pub/sub middleware that uses machine learning techniques to (1) adaptively configure transport protocols and (2) manage specified QoS by reflecting on a system’s dynamically changing environment and adapting accordingly. By utilizing machine learning techniques trained on data collected from multiple configurations, ADAMANT adjusts middleware mechanisms to provide transport protocol and parameter settings that maintain the specified QoS. The remainder of this document describes how ADAMANT addresses key challenges of maintaining QoS for pub/sub systems in dynamic environments via machine learning. In particular, we evaluate several machine learning techniques for providing adaptation guidance.

2. MOTIVATING EXAMPLE

Search and rescue (SAR) operations help locate and extract survivors in a large metropolitan area after a regional catastrophe, such as a hurricane, earthquake, or tornado. Figure 1 shows an example SAR scenario where infrared scans along with GPS coordinates are provided by unmanned aerial vehicles (UAVs) and video feeds are provided by existing infrastructure cameras to receive, process, and transmit event stream data from sensors and monitors to emergency vehicles that can be dispatched to areas where survivors are identified. These infrared scans and video feeds are then sent to a datacenter, where they are processed by fusion applications to detect survivors and develop three-dimensional views and highly accurate position information for rescue operations.

To motivate the need for integrating machine learning techniques with QoS-enabled pub/sub middleware, this section describes the following key research challenges associated with SAR operations in dynamic environments:

2.1 Challenge 1: Timely Adaptation to Dynamic Environments

Due to the dynamic environment inherent in the aftermath of a disaster, SAR operations must adjust in a timely manner as the environment changes. If SAR operations cannot adjust quickly enough they will fail to perform adequately given a shift in resources. If resources are lost or withdrawn—or demand for information increases—SAR operations must be configured to accommodate these changes with appropriate responsiveness to maintain a minimum level of service. If resources increase or demand decreases, SAR operations should adjust as quickly as possible to provide higher fidelity or more expansive coverage. Manual modification is often too slow and error-prone to maintain QoS.

2.2 Challenge 2: Managing Interacting QoS Requirements

SAR operations must manage multiple QoS requirements that interact with each other, e.g., data reliability so that enough data is received to be useful and low latency for soft real-time data so that infrared scans from UAVs or video from cameras mounted atop traffic lights do not arrive after they are needed. The streamed data must be received soon enough so that successive dependent data can be used as well. For example, MPEG I frame data must be received in a timely manner so that successive dependent B and P frame data can be used before the next I frame makes them obsolete. Otherwise, not only is the data unnecessary, but sending and processing the data has consumed limited resources.

2.3 Challenge 3: Scaling to Large Numbers of Receivers

For a regional or national disaster, a multitude of organizations would register interest not only in the individual video and infrared scans for various applications, but also in the fused data for the SAR operations. For example, fire detection applications and power grid assessment applications can use infrared scans to detect fires and working HVAC systems respectively. Likewise, security monitoring and structural damage applications can use video stream data to detect looting and unsafe buildings respectively. Moreover, federal, state, and local authorities would want to register interest in the fused SAR data to monitor the status of current SAR operations.

2.4 Challenge 4: Specifying Standardized and Robust QoS

SAR applications should be developed with the focus on application logic rather than on complex or custom formats for specifying QoS. Time spent learning a customized or complex format for QoS is time taken from developing the SAR application itself. Moreover, learning a custom format will not be applicable for other applications that use a different QoS format. Application developers also need support for a wide range of QoS to handle dynamic environments.

3. OVERVIEW OF RELATED WORK

This section analyzes related research efforts in light of the challenges presented in Section 2.

Support for adaptive middleware. The Mobility Support Service (MSS) [3] provides a software layer on top of pub/sub middleware to enable endhost mobility. The purpose of MSS is to support the movement of clients between access points of a system using pub/sub middleware. In this sense, MSS adapts the pub/sub middleware used in a mobile environment. Mobile clients notify MSS when mobility starts and ends. MSS buffers messages and manages connections while the client moves to a different access point. MSS is designed to support multiple pub/sub technologies, e.g., implementations of JMS, and adapt to the technology-specific characteristics.

MSS is solely focused on supporting mobility of pub/sub, however, and therefore does not address Challenge 2 in Section 2.2. Moreover, MSS fails to address Challenge 4 in Section 2.4 since it does not present a standardized and robust interface for QoS.

Gridkit [5] is a middleware framework that supports reconfigurability of applications dependent upon the condition of the environment and the functionality of registered components. Gridkit focuses on grid applications which are highly heterogeneous in nature. For example, these applications will run on many types of computing devices and across different types of networks.
To register components, application developers use Gridkit’s API which is based on binding contracts. Gridkit then uses the contract information along with a context engine to determine which components to include in the application. The context engine takes into account the context of the host machines, e.g., battery life, network connectivity.

Gridkit focuses on reconfiguration for installing an application and does not address Challenge 1 in Section 2.1. Within Gridkit no consideration is given to making timely adaptations based on the environment changing for a single application installation. Moreover, Gridkit fails to address Challenge 4 in Section 2.4 as it provides no standardized QoS specification.

David and Ledoux have developed SAFRAN [4] to enable applications to become context-aware themselves so that they can adapt to their contexts. SAFRAN provides reactive adaptation policy infrastructure for components using an aspect-oriented approach. SAFRAN follows the structure of a generic AOP system by supporting (1) a base program which corresponds to a configuration of components, (2) point-cuts which are invoked in response to internal events (e.g., invocations on interfaces) and external events (e.g., change in system resources), (3) advice which defines functionality to be executed for point-cuts, and (4) adaptation which uses adaptation policies to link join points to advice.

The SAFRAN component framework, however, only provides development support of maintaining specified QoS. The adaptive policies and component implementation are the responsibility of the application developer. Moreover, SAFRAN does not specifically address Challenge 3 in Section 2.3 since it does not focus on scalability. SAFRAN also does not address Challenge 4 in Section 2.4 since it provides no standard QoS specification.

Machine learning in support of autonomic adaptation. Vienne and Sourrouille [12] present the Dynamic Control of Behavior based on Learning (DCBL) middleware that incorporates reinforcement machine learning in support of autonomic control for QoS management. Reinforcement machine learning not only allows DCBL to handle unexpected changes but also reduces the overall system knowledge required by the system developers. System developers provide an XML description of the system, which DCBL then uses together with an internal representation of the managed system to select appropriate QoS dynamically.

DCBL’s customized QoS specification, however, does not address Challenge 4 in Section 2.4 and DCBL focuses on single computers rather than addressing scalable distributed systems, as outlined with Challenge 3 in Section 2.3. Moreover, DCBL requires developers to specify in an XML file the selection of operating modes given a QoS level along with execution paths, which leaves handling Challenge 2 in Section 2.2 to developers.

Tock et al. [11] utilize machine learning for data dissemination in their work on Multicast Mapping (MCM). MCM hierarchically clusters data flows so that multiple topics are mapped onto a single reliable session and multiple sessions are mapped onto a single reliable multicast group. MCM’s approach manages the scarce availability of multicast addresses in large-scale systems. MCM leverages machine learning to adapt as user interest and message rate change during the day. MCM is just designed to address the scarce resource of IP multicast addresses in large-scale systems, however, rather than Challenge 2 in Section 2.2 or Challenge 4 in Section 2.4.

Autonomic adaption of service level agreements. Herssens et al. [6] describe work that centers around autonomically adapting service level agreements (SLAs) when the context of the specified service changes. This work acknowledges that both offered and the requested QoS for Web services might vary over the course of the interaction and accordingly modifies the SLA between the client and the server as appropriate. This work does not address Challenge 1 in Section 2.1, but rather negotiates the QoS agreement to fit the dynamic environment.

Autonomic adaption of networks. The Autonomic Real-time Multicast Distribution System (ARMDS) [2] is a framework that focuses on decreasing excessive variance in service quality for multicast data across the Internet. The framework supports the autonomic adaptation of the network nodes forming the multicast graph so that the consistency of service delivery is enhanced. ARMDS does not address Challenge 2 in Section 2.2, however, nor does it address Challenge 4 in Section 2.4.

4. ADAMANT OVERVIEW AND RESULTS

This section describes ADAMANT, our experimental setup, and the results for evaluating machine learning techniques in providing adaptation guidance to select the most appropriate protocol and configuration settings for a particular dynamic environment.

4.1 Overview of ADAMANT

The ADAMANT QoS-enabled pub/sub middleware uses machine learning techniques to adjust the underlying transport protocols and associated parameter settings to maintain specified end-to-end QoS. ADAMANT addresses the challenges presented in Section 2 to resolve gaps in related work described in Section 3 via the following integrated techniques.

• The Adaptive Network Transports (ANT) framework addresses Challenge 1 in Section 2.1 and Challenge 2 in Section 2.2 by providing the flexibility to maintain interrelated QoS even within dynamic environments. For some environment configurations one particular transport protocol provides the required QoS. For other environment configurations a different transport protocol provides the specified QoS. ANT not only supports fine-grained control of a protocol’s parameters, but also switching from one protocol to another to provide the adaptation needed within dynamic environments. Moreover, ANT works to address Challenge 3 in Section 2.3 by supplying appropriate transport protocols and protocol settings as the number of senders and receivers in the system fluctuate.

We chose ANT due to its infrastructure for composing transport protocols. ANT builds upon the properties provided by the scalable reliable multicast-based Ricochet transport protocol [1]. It also provides a modular framework whereby protocol modules can be tuned, enhanced, and replaced to maintain specified QoS.

• Machine learning techniques help address Challenge 1 in Section 2.1 and Challenge 2 in Section 2.2 by selecting in a timely manner an appropriate transport protocol and protocol parameters given specified QoS and a particular environment configuration. Machine learning can interpolate and extrapolate its learning based on the current environment configuration, which might not have been included originally. Thus, machine learning provides increased flexibility over manual or policy-driven approaches.

The first learning technique investigated is a decision tree (DT). This algorithm attempts to create a tree where a set of decisions leads down to a leaf node that can accurately classify a new example. A DT will attempt to produce the shortest and smallest tree possible while maintaining accuracy by looking for features that best split the data as completely as possible and use them closer to the root. DTs are designed for data sets with more than a binary set of classes, i.e., where there are more than two possible classifications of an appropriate transport protocol and parameter settings.

The second technique we investigated is an Artificial Neural Network (ANN). ANNs work well on sets with a small number of features and can produce highly accurate results with medium-sized data sets. ANNs also can generally produce results in less time.
than other machine learning techniques. The learning produced when using ANNs is not as accessible as a DT, however, since the factors that are used and the importance placed on each factor are difficult to present in a human understandable form.

The third technique we investigated is a Linear Logistic Regression Classifier (LLRC), which uses a weighting of the various collected metrics to determine the appropriate protocol and parameters. The results from LLRCs have increased comprehensibility as compared to ANNs since how the environment configuration influences the selection of transport protocol and parameter settings is less opaque. Moreover, LLRCs can be optimized to reduce the time to determine an optimal protocol and settings.

- The OMG Data Distribution Service (DDS) middleware (www.omg.org/spec/DDS) addresses the scalability of Challenge 3 in Section 2.3 by decoupling data senders from data receivers. DDS enables applications to communicate by publishing information they have and subscribing to information they need in a timely manner. DDS is a standards-based anonymous QoS-enabled pub/sub middleware for exchanging data in event-based distributed systems. It provides a global data store in which publishers and subscribers write and read data, respectively.

DDS provides flexibility and modular structure by decoupling: (1) location, via anonymous publish/subscribe, (2) redundancy, by allowing any number of readers and writers, (3) time, by providing asynchronous, time-independent data distribution, and (4) platform, by supporting a platform-independent model that can be mapped to different platform-specific models.

The DDS architecture consists of two layers: (1) the data-centric pub/sub (DCPS) layer that provides APIs to exchange topic data based on chosen QoS policies and (2) the data local reconstruction layer (DLRL) that makes topic data appear local. Our work focuses on DCPS since it is more broadly supported than the DLRL.

The DCPS entities in DDS include Topics, which describe the type of data to be written or read, Data Readers, which subscribe to the values or instances of particular topics; and Data Writers, which publish values or instances for particular topics. Various properties of these entities can be configured using combinations of the 22 QoS policies. Moreover, Publishers manage groups of data writers and Subscribers manage groups of data readers.

Additionally, utilizing DDS addresses the QoS standardization of Challenge 4 in Section 2.4. Table 1 summarizes the DDS QoS policies. DDS provides 22 QoS policies applicable to various entity types. Each QoS policy has \( \sim 2 \) attributes with the majority of the attributes having a large number of possible values, e.g., an attribute of type long or character string.

Figure 2 also shows how we integrated and enhanced the OpenDDS implementation (www.opendds.org) of the OMG Data Distribution Service (DDS) with ANT, which supports various transport protocol properties, such as NAK-based and ACK-based reliability and flow control. ADAMANT leverages ANT to appropriately modify transport protocols and parameters settings as needed to maintain QoS.

OpenDDS provides a standards-based anonymous QoS-enabled pub/sub middleware for exchanging data in event-based distributed systems. It provides a global data store in which publishers and subscribers write and read data, respectively, so applications can communicate by publishing information they have and subscribing to information they need in a timely manner. OpenDDS supports various transport protocols, including TCP, UDP, IP multicast, and reliable multicast. OpenDDS also provides a pluggable transport framework that allows integration of custom transport protocols within OpenDDS. We chose the OpenDDS implementation due to (1) its source code being freely available, facilitating modification and experimentation and (2) its pluggable transport framework allowing integration of OpenDDS with the ANT framework.

- The Waikato Environment for Knowledge Analysis (Weka) is data mining software (www.cs.waikato.ac.nz/ml/weka) leverages key metrics captured from the ADAMANT prototype as shown in Figure 2. We use Weka to analyze ADAMANT’s behavior for various transport protocols. Specifically, Weka captures (1) data update latency times (i.e., the time from when the data writer writes the data to the time the data reader receives the data), (2) the number of updates received compared to the number of updates sent, and (3) network bandwidth usage statistics (e.g., total bytes on the network and min/max/avg bandwidth usage).

Table 1: DDS QoS Policies

<table>
<thead>
<tr>
<th>DDS QoS Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Data</td>
<td>Attaches application data to DDS entities</td>
</tr>
<tr>
<td>Topic Data</td>
<td>Attaches application data to topics</td>
</tr>
<tr>
<td>Group Data</td>
<td>Attaches application data to publishers, subscribers</td>
</tr>
<tr>
<td>Durability</td>
<td>Determines if data outlives the time when written or read</td>
</tr>
<tr>
<td>Durability Service</td>
<td>Details how durable data is stored</td>
</tr>
<tr>
<td>Presentation</td>
<td>Delivers data as group and/or in order</td>
</tr>
<tr>
<td>Deadline</td>
<td>Determines rate at which periodic data is refreshed</td>
</tr>
<tr>
<td>Latency Budget</td>
<td>Sets guidelines for acceptable end-to-end delays</td>
</tr>
<tr>
<td>Ownership</td>
<td>Controls writer(s) of data</td>
</tr>
<tr>
<td>Ownership Strength</td>
<td>Sets ownership of data</td>
</tr>
<tr>
<td>Liveliness</td>
<td>Sets liveliness properties of topics, data readers, data writers</td>
</tr>
<tr>
<td>Time Based Filter</td>
<td>Mediates exchanges between slow consumers and fast producers</td>
</tr>
<tr>
<td>Partition</td>
<td>Controls logical partition of data dissemination</td>
</tr>
<tr>
<td>Reliability</td>
<td>Controls reliability of data transmission</td>
</tr>
<tr>
<td>Transport Priority</td>
<td>Sets priority of data transport</td>
</tr>
<tr>
<td>Lifespan</td>
<td>Sets time bounds for &quot; stale&quot; data</td>
</tr>
<tr>
<td>Destination Order</td>
<td>Sets whether data sender or receiver determines order</td>
</tr>
<tr>
<td>History</td>
<td>Sets how much data is kept to be read</td>
</tr>
<tr>
<td>Resource Limit</td>
<td>Controls resources used to meet requirements</td>
</tr>
<tr>
<td>Entity Factory</td>
<td>Sets enabling of DDS entities when created</td>
</tr>
<tr>
<td>Writer Data Lifecycle</td>
<td>Controls data and data writer lifecycles</td>
</tr>
<tr>
<td>Reader Data Lifecycle</td>
<td>Controls data and data reader lifecycles</td>
</tr>
</tbody>
</table>

Figure 2: ADAMANT System Architecture

We chose the Weka data mining software due to its intuitive in-
terface, ease of use, robust analysis tools, and support for a wide range of machine learning techniques. These techniques include decision trees, multilayer perceptrons, and support vector machines. We input collected metrics and configuration information for the environment and transport protocol used into Weka. We have classified and analyzed the data using the various machine learning techniques to determine which techniques provide the best guidance in selecting a transport protocol for a given environment.

4.2 Evaluation Setup

To evaluate the behavior of ADAMANT with various transport protocols and protocol configuration settings, we ran experiments and collected metrics using the Emulab network testbed (www.emulab.net). Emulab provides computing platforms and network resources that can be easily configured with the desired computing platform, OS, network topology, and network traffic shaping. It also provides facilities to capture network bandwidth usage.

Table 2 outlines the points of variability for the Emulab experiments. The NAKcast timeout period configures the amount of time that elapses before a receiver notifies the sender of lost packets. The Ricochet R value determines the number of packets received by an individual receiver before error correction data is sent to other receivers. The Ricochet C value determines the number of receivers to which an individual receiver sends error correction data. Table 3 outlines the data that is being collected to classify and evaluate middleware performance.

<table>
<thead>
<tr>
<th>Point of Variability</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td># of data receivers</td>
<td>3, 25</td>
</tr>
<tr>
<td>Frequency of sending data</td>
<td>10Hz, 25Hz, 30Hz, 100Hz</td>
</tr>
<tr>
<td>End-to-end network loss</td>
<td>0% to 5%</td>
</tr>
<tr>
<td>Processor speed</td>
<td>350 MHz, 3 GHz</td>
</tr>
<tr>
<td>Network speed</td>
<td>100 Mbps, 1 Gbps</td>
</tr>
<tr>
<td>Protocols used</td>
<td>NAKcast, Ricochet</td>
</tr>
<tr>
<td>NAKcast timeout</td>
<td>0.5, 1.0, 0.05, 0.025 seconds</td>
</tr>
<tr>
<td>Ricochet R value</td>
<td>4.8</td>
</tr>
<tr>
<td>Ricochet C value</td>
<td>3.6</td>
</tr>
</tbody>
</table>

Table 2: Emulab Experiment Variables

The ReLate2 value [7] is a metric that evaluates both reliability and latency, we focus on the ReLate2 value discussed in Section 4.2 to provide relevant subsampling. For a given environment configuration we selected the transport protocol and parameter settings that performed the best, i.e., had the lowest ReLate2 value. For a set of 5 experiments runs for each environment configuration with varying transport protocols and parameter settings, the protocol and settings that performed the best remained. More specifically, 5 data points are left per experimental setup when including only the data from the protocol and parameters that produced the lowest ReLate2 value for the 5 runs. Across all experiment configurations, the collected metrics were reduced to roughly 150 data points that we used to train the machine learning techniques.

4.3.1 Subsampling the Data

Since ADAMANT is concerned with reliability and latency, we focus on the ReLate2 value discussed in Section 4.2 to provide relevant subsampling. For a given environment configuration we selected the transport protocol and parameter settings that performed the best, i.e., had the lowest ReLate2 value. For a set of 5 experiments runs for each environment configuration with varying transport protocols and parameter settings, the protocol and settings that performed the best remained. More specifically, 5 data points are left per experimental setup when including only the data from the protocol and parameters that produced the lowest ReLate2 value for the 5 runs. Across all experiment configurations, the collected metrics were reduced to roughly 150 data points that we used to train the machine learning techniques.

4.3.2 Analysis of Results

To explore which protocols and protocol parameters perform best under different configuration environments, we used three machine learning techniques described in Section 4.1. The techniques utilized the reduced data set where the best performing protocols and protocol parameters were selected. All three techniques were trained using n-fold cross validation, where each fold partitions the data into a training set and testing set. The cross validation then averages the accuracy of the technique over all n folds, which provides greater coverage and increases the technique’s robustness.

We selected 10 as the number of folds. The learning techniques were then trained and tested on each of the 10 folds. Since the reduced data points are not evenly distributed among the different experiment variables, n-fold cross validation is the best approach to maximizing data coverage while not skewing the learning results toward a particular transport protocol or parameter setting [9].

We used two metrics for evaluating the effectiveness of a machine learning technique. The first metric is the basic accuracy (also known as 1-loss accuracy) which captures how well the technique determines the appropriate protocol and parameters. Basic accuracy has its greatest utility when the number of different environment configurations and protocols used by 5 experiments are evenly distributed across all the types of experiments. The experimental data we collected, however, was not evenly distributed, i.e., there were some protocol parameters that were used in more environment configurations than others. For example, we ran more experiments with NAKcast having a timeout value of 0.025 seconds than with a timeout value of 0.5 seconds.

The second metric for evaluating the effectiveness of machine learning techniques to determine appropriate protocols and parameters is the area under the curve (AUC), which plots sensitivity vs. specificity. For comparison, a learning technique that would select a transport protocol and parameters at random would graph as a straight line with a slope of 1 and the AUC would be 0.5. As a learning technique improves, its AUC increases. AUC attempts to provide some balance between learning techniques for ADAMANT since ADAMANT requires more complexity than a simple boolean yes/no response from the learning technique, i.e., the specific trans-
port protocol and parameters to use. Moreover, a higher AUC value provides an indication of greater robustness.

Applying the three machine learning techniques outlined in Section 4.1 on the reduced set of 150 data points, we see clear differences in the results. The DT produced the best basic accuracy for determining appropriate transport protocols and parameters at 87% and the worst AUC score at .925. The ANN produced an accuracy that was lower at 85.3%, but provided the highest AUC at .966. The LLRC posted the lowest accuracy at 80% but also a higher AUC than DT at .935.

In general, all three techniques provide high accuracy results. While the differences in accuracy between the techniques are low, they are still significant. In machine learning with high levels of accuracy, a non-trivial amount of effort is required to modify a less accurate technique to match a more accurate technique. Our results indicate that ANNs are the most robust learning technique for ADAMANT, but DTs post a higher base accuracy. In particular, DTs exhibit some brittleness in being able to handle new, untrained environment configurations, whereas ANNs are more likely to provide ADAMANT resiliency for environment configurations not previously encountered. We are collecting more data to explore this assessment. LLRC appears to provide the worst results with the lowest base accuracy and only slightly better AUC than DTs.

While it appears ADAMANT might leverage ANNs initially with LLRC being the least useful in determining appropriate protocols and parameters, the results from DTs answer and raise interesting questions. Due to the nature of DTs, one can look at the implications the tree finds about the features. As shown in Figure 3, the tree utilizes relatively few features, e.g., the amount of bandwidth used in bytes, the controlled variable of packet loss, and the controlled variable of number of receivers in the environment. While some of the controlled variables of # of receivers, % packet loss, and sending rate are used, the most important discriminator of the measured environment is the bandwidth usage.

for classifying the best protocols and protocol parameters to use. In particular, decision trees provide human readable details about which variables are most important to consider. Our future work will empirically evaluate the most appropriate techniques for ADAMANT under various dynamic environment conditions.

6. REFERENCES


ABSTRACT
While several event notification systems are built around a publish-subscribe communication infrastructure, the latter only supports detection of simple events. Complex events, involving several, related events, cannot be detected. To overcome this limitation, we designed RACED, an adaptive middleware, which extends the content-based publish-subscribe paradigm to provide a complex event detection service for large scale scenarios. In this paper we describe its main aspects: the event definition language; the protocol enabling efficient and distributed detection of complex events through a network of service brokers; the mechanism that enables RACED to dynamically adapt to network traffic. A preliminary evaluation shows the benefits of RACED w.r.t. more traditional publish-subscribe infrastructures.

Categories and Subject Descriptors
C.2.2 [Computer Communication Networks]: Network Protocols—Routing protocols; C.2.4 [Computer Communication Networks]: Distributed Systems—Distributed applications

General Terms
Design, Languages, Measurement

Keywords
Publish-Subscribe, Complex Event Detection

1. INTRODUCTION
In the last years, the publish-subscribe communication paradigm [12], and in particular its content-based incarnation [7], has shown its effectiveness in a wide range of scenarios, by providing a strong decoupling among communication parties, which simplifies the design of loosely coupled systems. For this reason publish-subscribe has been widely adopted as a natural substrate to build event notification systems; in this field, however, single events are often useless on their own, while they become relevant when their mutual relations are considered as well.

The traditional publish-subscribe paradigm lacks the expressive power to express and detect complex events [10], i.e. events defined as patterns involving relations between other events. To overcome this limitation, different systems have been proposed recently, which extend publish-subscribe with the ability to cope with complex event detection [6, 14, 15]. However, these works are mainly focused on the definition of a rich language for event specification and only few of them address the problem of efficiently distributing events in large scale networks.

In this paper we describe a novel approach for the design of a complex event detection middleware, called RACED (Rate-Adaptive Complex Event Detection). The contribution of our work is threefold: i. we propose a simple language for complex event definition which is suitable for distributed processing; ii. in order to support large scale scenarios involving thousands of nodes, we developed a protocol to let a set of service brokers, connected in an overlay network, cooperate to efficiently provide the complex event detection service to their clients; iii. we augment this protocol with a mechanism that enables RACED to dynamically adapt to network traffic, and in particular to event generation rates.

The rest of the paper is organized as follows: in Section 2 we present the general architecture and the API of our system; in Section 3 we describe our complex event definition language; in Section 4 we present our protocol for distributed event detection, focusing on its capability to dynamically adapt to event generation rates; in Section 5 we evaluate such protocol, using the Omnet++ network simulator [16]. Finally in Section 6 we survey related work, providing some concluding remarks in Section 7.

2. SYSTEM ARCHITECTURE
The architecture of RACED is shown in Figure 1.
Sources notify the dispatching service about events. Examples of sources are temperature and smoke sensors sending notifications about some location, or a traffic monitoring system, notifying about congested routes. We call such event notifications messages. System administrators define complex events: for example they can define that an event of type fire is detected when the system receives both a message about high temperature and a message about smoke coming from the same room. Finally, subscribers ask the system to be notified about the detection of certain (complex) events. Internally, the dispatching service is built around different brokers, connected in an overlay network, which cooperate to detect and route events from sources to subscribers. Table 1 presents the API of RACED.

**Table 1: The RACED API**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advertise(MessageType t)</td>
<td>Publish(Message m)</td>
</tr>
<tr>
<td>Subscribe(EventType t)</td>
<td>DefineEvent(EventType t, Pattern p)</td>
</tr>
</tbody>
</table>

Before sending messages, sources have to invoke the advertise operation, to declare the type of messages they will publish. This build a contract between sources and the dispatching service: only messages whose type has been advertised can be published; such contract is exploited by the dispatching service to enable distributed detection of complex events, as explained in Section 4. The Publish operation is used to inject new messages into the dispatching system, while the Subscribe operation is invoked by subscribers to express the type of events they are interested in. Finally, complex events are defined by system administrators using the DefineEvent operation, which specifies the type t of the new event and a Pattern that expresses when an event of type t occurs. Next section introduces the syntax and semantics of such patterns.

### 3. EVENT DEFINITION LANGUAGE

Several languages have been recently proposed to define complex events [6, 14, 15]. In most cases they privilege expressiveness over simplicity. Being interested in providing a complex event detection system tailored to large scale scenarios, we took the opposite approach and designed a low level language with few operators, optimized to allow detection of complex events in a distributed way.

The RACED language provides just five constructs: i. message filters, ii. composition operators, iii. parameters, iv. windows, and v. event definitions.

Higher level constructs can be easily built on top of it (e.g. sequences and repetitions can be represented by combining message filters and composition operators).

**Message filters.** Filtering is the capability, offered by content-based publish-subscribe systems, to select or discard single messages according to their content. Our language offers the same functionality. Messages have a type and a set of attributes represented as key-value pairs as in 1. Message filters define the type of the matched messages and the possible values for attributes through a predicate (a conjunction of constraints on single attributes). As an example, the filter given in 2 matches the message given in 1.

(1) Temp[Value>20 AND Location="office 1"]

**Composition operators.** Message filters can be combined through the operators AND and NOT to define events whose occurrence requires different messages to be published. As an example, in 3 we combine two messages that have to occur and a third that have not to occur for a certain event to happen.

(2) Temp[Value>10 AND Location="office 1"]

(3) Temp[Value>30] AND Alert[Type="smoke"] AND NOT Weather[State="rain"]

**Parameters.** Consider Example 3: in many scenarios the fact that a message containing a high temperature and a smoke alert have been published in a non raining condition may be meaningless. However, it becomes useful if the three notifications are all related to the same area, as they may indicate a possible fire. To express patterns in which different messages are selected only if they satisfy mutual relations, our language enables the definition of parameters and constraints on them. Example 4 shows a pattern that satisfies the aforementioned requirements using a parameter $X$.

(4) Temp[Value>30 AND Location=$X$] AND Alert[Type="smoke" AND Location=$X$] AND NOT Weather[State="rain" AND Location=$X$]

**Windows.** Sometimes the composition of multiple messages is meaningful only if they are published in a limited amount of time. For example high temperature and smoke notifications from the same location are not relevant if they are generated in different days while they become significant if generated within 5 minutes. Additionally there exist patterns that cannot be processed without time constraints. Consider again Example 3: how long does the system have to wait until it can decide that no rain messages have been received? The NOT operator cannot be evaluated without explicit timing constraints.

For these reasons our language includes windows defined using the WITHIN operator as in Example 5. When the WITHIN clause is not specified we assume that a default value is used.

(5) Temp[Value>30] AND Alert[Type="smoke"] WITHIN 5 min

It is worth mentioning that the exact semantics of the windowing mechanism depends on the time model provided by the underlying system. In particular, in our system published messages are time-stamped by the first broker receiving them, while brokers’ clocks are kept in sync with an error that we assume being not significant for the kind of applications we focus on.

**Event definitions.** System administrators define events invoking the DefineEvent operation (part of the system API). It includes a pattern defined using the operators above together with the definition of the attributes valid for the new event. As an example, in 6 we define the new event Fire with two attributes: Temp and Location.

(6) DefineEvent(Fire[Temp:$X$, Location:$Y$], Temp[Value>$X$>30 AND Location=$Y$] AND Alert[Type="smoke" AND Location=$Y$] WITHIN 5 min)

\(^1\)NOT is known as a blocking operator [8].
4. EVENT DETECTION PROTOCOL

To support large scale scenarios, we designed a protocol that defines how multiple service brokers cooperate in RACED. Such protocol delivers subscriptions exploiting the shortest path tree rooted at the subscriber; during delivery, subscriptions are partitioned at each hop, letting each source receive only those parts that are relevant for the message types it advertised. Messages follow the opposite route ascending the tree up toward the relevant subscribers, being filtered and combined along the route.

In particular, each broker runs a link-state protocol to collect information about the topology of the dispatching network. It exploits such information to compute its shortest path tree (SPT) using Dijkstra. The computed SPT is forwarded to all other brokers in the network, so that everyone could store its position (i.e., children and parent) in the defined tree. The SPT is initially used to propagate new event types defined using the DefineEvent primitive so that every broker could store them.

Next sections detail how advertisements, subscriptions and messages are forwarded. For simplicity, we consider a single subscriber and its SPT.

4.1 Forwarding of Advertisements

Advertisements are forwarded from sources up to the subscriber. Each broker saves all the message types contained in the advertisements coming from its descendants in an advertisement table. In Figure 2 we show the advertisement table of broker 2 after it has been filled (we denote the set of message types advertised by broker x as types(x)).

```
<table>
<thead>
<tr>
<th>Broker</th>
<th>Message Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>types(2)</td>
</tr>
<tr>
<td>4</td>
<td>types(4) u types(6)</td>
</tr>
<tr>
<td>5</td>
<td>types(5) u types(7)</td>
</tr>
</tbody>
</table>
```

Figure 2: Forwarding of advertisements

4.2 Forwarding of Subscriptions

When a client connected to a broker B calls the Subscribe operation for an event type t, the broker B looks at the Pattern defining t and creates a packet we call subscription that contains the following fields:

- **Positive Filters (PF):** it is the set of all non-negated message filters that appear in p.
- **Negated Filters (NF):** it is the set of all negated (preceded by the NOT clause) filters that appear in p.
- **Window (W):** is the timing window expressed in the WITHIN clause of p.

For simplicity we forget about subscribers and sources to focus on the overlay network of brokers; for this reason in the following we use the term subscriber (resp. source) to indicate the broker to which a subscriber (resp. source) is connected.

4.3 Forwarding of Notifications

Each broker stores the data contained in all subscriptions it receives and uses them to filter and combine messages. In particular, when a broker B receives a message m from one of its clients, it first checks, for each stored subscription s, whether m matches one of the filters contained in s. If this happens, then m is stored inside a data structure called History, which B uses to detect the matching sets for s, i.e. the sets of messages satisfying all the filters in s.

An example of message processing is shown in Figure 4. Consider the subscription s shown there; it requires the detection of a message matching filter A and one matching filter B in a window of size 3. At time T=1 the broker receives the message A1 that matches filter A; at time T=2 it receives the message B1 that matches filter B. Messages A1 and B1 together form a matching set for s. When, at T=3, the broker receives the message B2, it forms a new matching set with A1 and B2. Finally, at T=4, A1 is deleted from the
A broker 2 receives a subscription containing two positive messages matching negated filters in the window. Subscription, matching sets are detected only if there are no messages matching negated filters in the window.

4.4 Push vs. Pull-Based Forwarding

History, as it is too old for the detection window considered, while A2 arrives, resulting in two new matching sets, one with A2 and B1 and one with A2 and B2.

Notice how in the case of negated filters inside the subscription, matching sets are detected only if there are no messages matching negated filters in the window.

Detected matching sets are delivered to the parent node inside packets called notifications. Each notification may include multiple matching sets, each including multiple messages. At the parent node, messages contained in the received notifications are processed again using the same procedure described above. Whether message sets matching a subscription s are delivered immediately when detected, or stored and delivered later, is determined by the type of s as described below.

4.4 Push vs. Pull-Based Forwarding

Consider now the situation shown in Figure 5, in which broker 2 receives a subscription containing two positive filters, one for messages of type A and one for messages of type B. According to its advertisement table, it delegates the detection of messages of type A to brokers 3, 4, and 5, and the detection of messages of type B to broker 6. In this case broker 2 can satisfy all the constraints of its subscription only if it receives both a message matching the first filter and a message matching the second filter generated within w. This means that, if broker 6 never sends notifications, all messages received from brokers 3, 4, and 5 are useless and only waste network resources.

Starting from these considerations we introduced in our protocol the concept of pull-based forwarding as opposed to the more common push-based approach. In particular, every subscription has an associated type(T) which can be either push or pull. A push subscription requires the broker receiving it to promptly send all matching sets of messages up to its parent; on the contrary a pull subscription requires the broker to store matching sets of messages until the parent explicitly asks for them. So, in the example of Figure 5, broker 2 could decide to send the subscription for messages of type A as a pull subscription and to ask for the delivery of stored messages only after receiving messages of type B from broker 6.

More specifically, the mechanism to decide the type of subscriptions to send to children and to ask for stored messages (in case of pull subscriptions) works as follows. When a broker B receives a subscription s, it processes and partitions it as explained above. For each newly generated subscription s′ it looks at the sets PF and NF. If PF is empty then the subscription is processed in a special way: its type is set to push and the receiving children is asked to promptly send up messages matching the filters in NF. Among all the subscriptions (if any) having a non empty PF only one is selected as the master subscription, while all the others are considered slave subscriptions. The master is sent as a push subscription while the slaves are sent as pull. When broker B detects a matching set of messages for the master subscription, it sends an open packet to all children processing slave subscriptions. The open packet causes children to send all message sets they had detected so far (if any) and to continue to send new sets using a push approach for a period long w, where w is the window of s.

It is worth mentioning that in case of shared parameters between filters of the master subscription and filters of the slave subscriptions, the open packet asks only for messages having the right values for the shared parameters (the values that appear in the messages that matched the master). This reduces the number of message sets that have to be sent in reply to an open packet.

4.5 Adaptive Selection of Masters

The right choice for the master vs. slave subscription may strongly influence the performance of our protocol. In fact, if messages satisfying the master subscription are received sporadically, then fewer requests are sent to children holding slave subscriptions, which may drop several packets locally (i.e., those exiting the window) resulting in less network traffic. On the contrary a master subscription that continuously receives notifications eliminates the benefit of the pull-based approach. To address this issue, our protocol monitors traffic flowing in the network and let each broker adapt its choice of master subscriptions to the traffic monitored in the previous time frame. More specifically each broker B stores, for each subscription s it has received, the number n of matching sets it has detected in a given amount of time t and computes the generation rate of s, gr(s) = n/t. At the same time, periodically B decides the master selection for s by asking to its children the generation rates of all the partitions of s that it sent them. The part having the lowest generation rate is chosen as the new master.

In summary, the mechanism combining push and pull-based forwarding, coupled with this adaptive mechanism in the choice of which part of a subscription to manage as push and which to treat as pull, results in the ability for our protocol to optimize complex event detection to the actual traffic, minimizing the route followed by messages to be matched and combined together.

5. EVALUATION

To evaluate the benefits of the distributed detection protocol of RACED, we compared it with PADRES [9]. As explained in Section 1, only few works have addressed the prob-
lem of distributing detection of complex events; PADRES represents probably the most promising effort in this direction. Similarly to RACED, PADRES defines an advertisement mechanism and exploits it to partition subscriptions in a network of brokers. However, it delivers all notifications using a push-based approach; as a consequence, brokers cannot prevent the delivery of useless notifications, as it happens with our push/pull mechanism. Additionally, PADRES does not provide any mechanism to adapt to network traffic, as RACED does.

To compare the two protocols, we implemented both in Omnet++ [16] and tested them in a scenario of 120 brokers and 600 clients, each one publishing messages of a single type chosen among 100 different types, with different contents. We analyzed the cost of forwarding by counting the total number of notifications generated in the network after all clients have published 1000 messages.

In particular, we analyzed the impact of three parameters: the size of the detection window, the number of filters composing the event to be detected and the distribution of publications rates. In Figure 6 we show the number of notifications generated by the two protocols when the complex event to match does not contain any negated filter. In Figure 7 we present the improvement of RACED over PADRES in the same scenarios showing how it changes when the number of negated filters in the event definition \(nf\) varies from 0 to 2.

Figure 6(a) shows how the number of notifications increases in both protocols when the size of the detection window grows. Looking at Figure 7(a) we notice how small windows favor RACED more, as notifications coming from slave subscriptions are stored for less time, increasing the chance that an open packet is not followed by any reply.

Figure 6(b) shows how the number of notifications increases with the number of positive filters in the event. On one hand adding positive filters decreases the chance of capturing the event, but this trend is dominated by the huge number of matching sets detected when many single messages have to be combined in the matching event. As shown in Figure 7(b), large numbers of filters increase the gain of RACED, as they promote the recursive decomposition of the subscription while it moves from subscriber to sources, thus maximizing the advantages of our push/pull and adaptive mechanisms.

Figure 6(c) shows how the traffic decreases when we increase the variance of publication rates. Increasing the variance, in fact, also increases the possibility of having messages generated so rarely that they are hardly ever captured inside a detection window. Figure 7(c) shows the benefits of our adaptive mechanism with the advantage of RACED over PADRES increasing from 20% up to more than 35%.

If we look at Figure 7 we may also observe how the number of negated filters \(nf\) reduces the advantage of RACED over PADRES. There are two reasons for this: first, negated filters reduce the propagation of useless notification, problem that affects PADRES more than RACED; at the same time, negated filters limit the possibility to define pull-based subscriptions. Finally, it is worth mentioning how our simulations did not take into account parameterization; considering it would bring even better results, as it would enable finer grain selection of notifications from pull-based subscriptions.
6. RELATED WORK

In the last years a large number of content-based publish-subscribe systems have been developed [12, 7, 11, 5]; proposed solutions were, at the beginning, based on a centralized dispatcher, but soon they moved to distributed solutions to improve scalability.

All these systems share the same communication model, in which messages bring data and subscriptions filter single messages according to their content. Recently, a few works have been proposed that extend the expressive power of traditional content-based publish-subscribe to take into account information contained in multiple messages [6, 14, 15]. However, these works are mainly focused on the definition of rich languages and usually don’t address the problem of event matching in large scale systems. Among the few exceptions, the PADRES system [9] adopts a distribution protocol similar to ours; in particular it exploits a tree-based topology to distribute information and advertisements to filter subscriptions. However, all notifications flow using a push-based approach and the dynamics of network traffic is never taken into account.

The problem of combining information coming from multiple sources and to distribute it to users has been addressed also in the field of so called DSMSS (Data Stream Management Systems) [3]. These systems define highly expressive languages, usually derived from SQL [2, 4], which are suitable for generic data manipulation. Even if some of these systems [1] address the problem of increasing scalability by distributing processing, the proposed solutions focus on clustering scenario, in which a set of co-located machines shares the load of processing, while we take the network cost into account and focus on processing messages as close as possible to the sources.

Finally, in [13], authors propose an algorithm to distribute operators in a network of service brokers trying to minimize a cost function, which involves data generation rates. This proposal, however, does not contain any mechanism to inhibit useless data to be propagated in the network, like our master-slave algorithm.

7. CONCLUSIONS

In this paper we presented the design of RACED, an adaptive middleware that extends the content-based publish-subscribe paradigm to provide a complex event detection service for large scale scenarios. In particular, we introduced an event definition language and we described a protocol enabling efficient and distributed detection of complex events inside a network of service brokers. To increase performance, our protocol includes an adaptive mechanism that allows brokers to dynamically adapt their behavior to network traffic. Our tests show that it provides evident benefits over more traditional solutions.

We plan to extend our work in two directions: on one side we are investigating optimization techniques for multiple subscriptions; at the same time we are extending our event definition language to provide not only event detection, but also event processing, for example to compute aggregate values.

Acknowledgment

This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom; and by the Italian Government under the projects FIRB INSYEME and PRIN D-ASAP.

8. REFERENCES

A Semantic Composition Model to Preserve (Re)configuration Consistency in Aspect Oriented Middleware

Bholanathsingh Surajbali, Paul Grace and Geoff Coulson
Computing Department, Lancaster University, Lancaster, UK
{b.surajbali, p.grace, geoff} @comp.lancs.ac.uk

ABSTRACT
Aspect-Oriented Programming enables the isolation and modularisation of crosscutting concerns that are typically implemented in a tangled fashion within the base system. However, the composition of these aspects is not completely orthogonal; with interactions between aspects involving direct and indirect dependencies, and conflicts that can cause runtime inconsistencies when those interactions are not detected. This is particularly true of the dynamic composition and adaptation of aspects within distributed systems; therefore in this paper we propose a semantic composition model to detect and solve these interaction issues at runtime. Our approach can be employed in dynamic AOP middleware, and we measure the overhead incurred by the semantic composition model when performing safe dynamic reconfigurations.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures – Patterns (Reflection).

General Terms
Design, Management

Keywords
Middleware, reflection, aspects, dynamic reconfiguration.

1. INTRODUCTION
Aspect Oriented Programming (AOP) is a method of tackling the problems of tangled code i.e. the basic system implementation becomes tangled with code for features such as security, caching, and monitoring. These concerns are implemented as aspects which are made up of individual code elements that implement the concern (advices). Advices are deployed at multiple positions in a system (join points) which are expressed by pointcuts—a particular form of composition language. Dynamic AOP, for example as provided by the JAC [7] and DyReS [12] middleware then allow aspects to be composed and adapted at runtime. Further, [4] advocates the use of reflection to perform fine-grained adaptation of the aspect elements.

Aspects are designed to have orthogonal properties so they can be deployed obliviously from one another; however, it is clear that semantic interactions can cause compositional inconsistencies to occur i.e. composing aspects that syntactically match with each other, or with the base system may produce behaviour that is in conflict with the original system operation. For example, the composition of both an authorisation and authentication aspect, where the authentication aspect checks the credentials of the user produces a negative interaction if the authorisation aspect gives access to users before the credentials are verified. Another example of semantic inconsistency is the weaving of an encryption aspect at the sending end join point without the corresponding decryption aspect at the receiving end join point. While these examples can be detected with knowledge of the aspect types, more complicated semantics may prove more difficult to detect. For example a running system having a cache and a security aspect could result in an increase in resource usage resulting in an aspect with real-time constraints to miss deadlines.

The majority of dynamic AO middleware (AOM) approaches focus solely on dealing with syntactic inconsistencies; that is detecting if aspects have the same type, version, and interfaces. However, semantic inconsistencies are non-trivial to detect, as the semantics of aspects may interfere with each other without sharing a common element. In this paper, we present a semantic composition model (SCM) for dynamic aspect-oriented, component-based middleware; this provides the capability to describe the various kinds of built-in and external interactions that affect aspect semantic composition at runtime. This is coupled with a method for semantic interaction resolution which detects inconsistencies at run-time and supports methods to resolve semantic interactions.

We evaluate our approach within the AO-OpenCom platform for developing dynamic reconfigurable middleware solutions; this demonstrates the following contributions of our approach:

- Conflict resolution. We show that complex semantic inconsistencies can be resolved for a number of case-studies with minimal performance overhead.
- Transparency. We apply consistent (re)configuration with minimal programmer effort or change to the underlying component model.
- Flexibility. New semantic conflicts can be described dynamically to evolve with the running application or domain context. Moreover, the approach can be applied in different compositions approaches and tools; for example we show how both node-local and distributed (re)configuration semantic consistency can be achieved in this paper.

The remainder of this paper is organised as follows. Section 2 examines the types of aspect interaction that may occur. Then, section 3 describes the design of our semantic composition model, followed by section 4 which validates the proposed semantic composition model. Finally, in section 5 we describe the related research and offer our conclusions in section 6.
2. ASPECT-COMPONENT INTERACTION

In this section, we describe the potential sources of composition inconsistencies. To present the different types of semantic interactions we use a use case scenario to describe the different types of (re)configuration interactions. We then describe the general built-in and external interactions influencing the (re)configuration in the aspect-component model.

2.1 Use Case Scenario

To motivate the requirement for semantic interaction consistency we present an online gaming system case scenario. The system allows users to play games via a central server that co-ordinates multiple online players. When a user logs in at the start of a game a list of available users based on their preferences are provided such that the user can contact peers and request them to play. In such an environment, there are various types of application requirements in terms of multi-player, mobile-player, real-time and non-real-time gaming; further, peers may operate in different network domains e.g. Internet, Wi-Fi, or ad-hoc wireless networks. To cope with the application and environmental demand a number of dynamic (re)configurations may be required:

(i) new mobile users with limited bandwidth may join, requiring a fragmenter aspect to be configured to split data before being sent;
(ii) when monitoring users a persistence aspect may be deployed to keep track of authorised users connected to the server;
(iii) data may be required to be encrypted to protect the users’ privacy;
(iv) the authorisation module may be replaced with an updated one filtering users by privileges (e.g. by administrator, game user, etc).

2.2 Built-in Interaction

In aspect-component middleware, aspects (which are themselves implemented as component modules) are composed with the base components (hereafter termed components) using AO-Connectors [4, 10, 11, 12]. A list of advices is attached to the connector between a receptacle and a provided interface. This capability is illustrated in Figure 1.

![Figure 1: Aspect-Component Interaction](image)

We now identify and classify the types of interaction that can occur in this local aspect-component model; these fall into three categories: dependency, conflict and indirect interactions.

2.2.1 Dependency Interaction

Dependency represents a required interaction between aspects i.e. one aspect instance must be present for the other to operate. In an aspect-component model, such dependencies may occur from the different parties involved; we now show the potential dependencies between different elements in the model:

- **Component to component dependency.** The ‘caller’ component passes invocations through its required interface to the ‘callee’ component’s provided interface. Required interfaces are dependencies that need to be fulfilled in order to guarantee correct semantics of the component model.

- **Component to aspect dependency.** In general one assumption is that components must be present first, and aspects being non-functional services are composed later. However, aspects are often integral, with components depending upon specific aspects. For example, a synchronisation aspect may be required to guarantee the real-time properties of the system, where the communication component is dependent on the synchronisation aspect.

- **Aspect to aspect dependency.** Similar to component to component dependency, aspects may depend upon each other. For example, an encryption aspect depends upon the corresponding decryption aspect to operate effectively.

2.2.2 Conflict Interaction

A conflict interaction represents a negative interaction [5] between two aspects, where the operation of one aspect is detrimental to the other. The causes of these conflicts in the aspect-component model can be classified in terms of:

- **Component to component conflict.** The interaction between two components can conflict following reconfiguration. This can be due to a number of factors: incompatible component types, invalid state of component e.g. replacing a component generating unique users’ identifiers and the latter generates ids’ that were previously created by the old component.

- **Component to aspect conflict.** The composition of an aspect with the base system may also cause semantic conflicts. For example, weaving the encryption aspect at the communication component join point may cause the system’s throughput to fall below the required level.

- **Aspect to aspect conflict.** Composing multiple aspects at a join point is a common source of conflict. This can be caused mainly from:
  - **Ordering.** The order in which aspects are composed influence their execution order. Consider a join point having two security aspects, an authentication aspect and an authorisation aspect. The order in which the aspects are invoked influence the correct system execution. Invoking the authorisation aspect before the authentication aspect may give access to non-authenticated users.
  - **Mutual-exclusion.** This involves two aspects that implement concerns having contradictory semantics such that either one of them can be used but not both. For example, in the gaming scenario, when multiple users are playing a live-game with the server using a real-time aspect to guarantee proper updates to be sent across the users, reconfiguring the system by adding a synchronisation aspect may cause both aspects to block; the real-time aspect waiting for the synchronisation and vice-versa, resulting in deadlock at the server.

2.2.3 Indirect Interaction

A more complicated interaction that can occur is the indirect interaction between aspects involving multiple oblivious parties. For instance, if an aspect “AC-1” depends on another aspect “AC-2”, which itself depends on another aspect “AC-3” but AC-1 has semantic conflicts with “AC-3”. Such indirect interactions are harder to detect, and require reasoning at the AO-Connector level to find indirect interactions.
2.3 External Factors affecting Aspect Interactions

Importantly these interactions are influenced from the following:

- **Application-specific influences** arising from specific constraints and requirements of the application. For example, composing an encryption and logging aspect depends on the application-specific requirements. In an untrusted environment, all data needs to be encrypted before the logging aspect reads the data, to preserve the data safety while in a trusted environment unencrypted data suffice.

- **Domain-specific interactions**. Aspect compositions differ across domains. Each domain imposes specific policies about compositions, for example the combination of two aspects may cause incorrect synchronisation in a real-time domain e.g. applying a caching aspect with a synchronisation aspect is likely to make the software system miss real-time deadlines. In a non-real-time domain the interaction of the two aspects does however cause no interaction concern. Moreover, aspect weaved at the aspect-connector may also contain remote reference such that the order in which they are woven in one domain may differ to that woven in a different domain. Applying the same weaving order in a different domain may result in conflict semantics. In such cases, the local node policy needs to be checked to ensure the correctness of the (re)configuration.

3. SEMANTIC COMPOSITION MODEL

In this section we describe our semantic model for supporting the detection of interaction issues in the aspect-component model and resolution of emerging runtime semantics by supporting the following dimensions: (i) describing semantic aspect interactions; (ii) attaching metadata to entities in the aspect-component model; (iii) using a resolution engine and policies to resolve inconsistencies. Each of the dimensions is now examined in turn.

3.1 Aspect Interaction Semantic Metadata

In order to detect possible semantic conflicts and dependencies each aspect-component is attached with metadata that describes and explains its functionality. This is used to inform the selection and deployment of the aspect—i.e. to help manage compositional and reconfiguration interaction between aspect-components in the aspect-component model as illustrated. These descriptions are written in the format as illustrated in the BNF form of Figure 2.

The aspect-component interaction model consists of the aspect scope, composition rules, and interaction policies. The aspect scope refers to the aspect-component instance of whether it is deployed on the local host, or is remote, or is replicated. More specifically, an aspect-component is assigned to a particular composition policy and has a specific aspect-scope in which it is defined. The composition policies deal with the types of policies options to constrain aspect instances on each particular (or multiple) address space. The interaction policies defines the aspects in which the underlying aspect-component is either dependent on or conflicts with, or the set of conditions that can lead to indirect interactions.

Dependency-specific interactions define the coordination-rules and enforcement rules that the aspect-components need. The coordination and enforcement-rules specify the aspect parties with which the composed aspect must coordinate with. For example for an encryption aspect, the underlying aspect must specify enforcement rules for a decryption aspect to be also added to the system when the encryption composition takes place and coordination rules specifying that both should be added to preserve the system consistency.

The conflict-specific interactions refer to the set of orders and mutual exclusive aspects that an aspect must be composed with respect to other aspects. Finally, the indirect-interactions define the set of conditions on how the aspect can be composed when dependency-interactions and conflict-specific interaction occur in the system.

![Figure 2: BNF SCM for Aspect Interaction](image)

3.2 Attaching Metadata

Aspect-components are considered as black-boxes which provide advises in the form of operations within the provided interface (but hide their implementation). Three implications of this black-box property are:

1. Metadata can be attached to the interfaces and receptacles of components and aspects, as they are the only access points available to other aspect-components at be inspected and inform runtime decisions;

2. After aspect-components have been woven, they are invoked through their operations such that metadata is also required to be annotated at the aspect-component operations to detect runtime interactions; This is because when reconfiguration is performed at runtime, already woven aspect metadata might be required to detect semantic interactions with the reconfiguration aspect(s) and at the join point the aspect is accessed through its operations;

3. The tagged metadata needs to be kept separate from the main source functionality. This is because, an aspect represents crosscutting functionality such that adding descriptions by extending the implementation, e.g. through a new interface, will restrict its applicability to different applications and domains because it couples the consistency checking with the aspect-component functionality. Thus, keeping metadata separate allows both the core functionality and metadata to be reconfigured independently and transparently from each other.

3.3 Semantic Resolution Engine & Policies

A Semantic Resolution Engine (SRE) provides the tool to query and reason about the annotated aspect-components; and resolve possible sources of inconsistency that may result from a dynamic reconfiguration. The latter retrieves the associated aspect-component metadata as illustrated in Figure 3, by getting the annotation file path from the aspect-component and parsing the Aspect Metadata file (retrieved from the Aspect Metadata Repository) to extract respective semantic composition tags for
the aspect-component (structured as described by the BNF semantic composition model from Figure 2). Then, the SRE checks the reconfiguration aspect against a set of composition policy on each reconfigured address space (referred to as a node) to ensure reconfiguration follow the specified domain or running application policies. The composition policy uses a ‘condition-action’ approach to ensure the associated metadata and the join point aspects metadata are valid by not causing any domain or application inconsistency in the node in which the reconfiguration take place. In case the validation is successful the (re)configuration is allowed to proceed. However, in case of any interaction issue is found, based on the policies specification the necessary remedy action is taken. Two alternatives of remedy actions can be taken by the SRE in terms of: either stopping reconfiguration from proceeding by calling the rollback operation to drive the system to the state prior to when the reconfiguration started; or if appropriate resolution policies are specified these can be deployed by the SRE and the reconfiguration can proceed (e.g. resolving the correct order of advices).

Figure 3: Semantic Resolution Framework

4. VALIDATION
In this section we validate our approach using AO-OpenCom [11]. We first provide some background on AO-OpenCom and then validate the extent to which our Semantic Composition Model (SCM) achieves the stated goals of semantic composition resolution, transparency and flexibility. Finally we measured the overhead of deploying the SCM.

4.1 AO-OpenCom
AO-OpenCom is an extension of the OpenCom [1] component model and provides a distributed AO composition service while allowing aspectual compositions to be dynamically reconfigured. The purpose of AO-OpenCom is to build on OpenCom and its associated reflective meta-models and component frameworks architectures [11], to provide a distributed AO composition service, and to allow aspectual compositions to be dynamically reconfigured. The programming model employs components to play the role of aspects—i.e. an aspect is simply an OpenCom component. The AO-OpenCom aspect framework comprises a set of components that are instantiated across each host. The set of components is as follows (see Figure 4):

The Configurator manages the other components in the framework as it is responsible for accepting and handling (re)configuration requests that will apply to a set of hosts. The Configurator also caches join point information it receives from Pointcut-Evaluators in case similar behaviour needs to be applied in the future. The Aspect-Repository holds a set of instantiate aspect-components e.g. the cache aspect, encryption aspect, etc.

The Pointcut-Evaluator evaluates the pointcuts provided by the Configurator and returns a list of the matching join points found within the local address space. Finally, the Aspect-Handler acts on instructions from the Configurator to weave advices at join points as well as supporting the invocation of remote aspects.

The main API provided by an AO-OpenCom-enabled instance for AO (re)configuration is as follows:

```java
Configurator.reconfigure(pc, command, aspect);
```

The `pc` argument specifies a pointcut that picks out the join points in the target nodes at which the desired reconfiguration should occur. The `command` argument offers options for the action to be taken at the indentified join points: the ‘add’ action is used to weave the specified aspect at the join points; ‘remove’ is used to remove it, and ‘replace’ is used to add the specified aspect after removing an existing aspect of the same type that is assumed to be already there. The `aspect` argument can be a direct reference to a local aspect-component, or an indirect reference to an aspect stored in an Aspect-Repository, or a reference to an already-instantiated remotely-accessible singleton aspect. The aspect weaving order and the type of aspect in terms of (before, after, around) are also specified in the aspect argument.

Figure 4: AO-OpenCom platform Architecture

4.2 Applying the SCM to AO-OpenCom
To ensure the semantic consistency, the SRE and the Composition-Policy aspect are both encapsulated as an aspect and weave at the AO-connector component join point connecting the Configurator and the pointcut component as an ‘after’ advice in the AO-OpenCom platform. Moreover, the Aspect Metadata file of the SCM is implemented in an XML file with each aspect annotated with the path to the XML metadata file.

4.3 Qualitative Validation
To illustrate the semantic reconfiguration consistency, we consider the following reconfiguration: the application programmer needs to reconfigure the online gaming system by adding an encoder to all mobile wireless node member; a fragmenter and a logger aspect have previously been woven at that join point. To perform this reconfiguration, the application programmer would provide a reconfiguration request by writing code as shown in Figure 5 (the code is simplified for presentational purposes).

```java
Configurator.reconfigure(pc: new Pointcut("mobile").method("encoder").parameter("mobileEncoder").annotation("学期期末")), command:"add".
```

Figure 5: Aspect Reconfiguration specification example

Since AO-OpenCom also supports remote aspects [11], the respective URL path to the XML file Annotation Metadata Repository is provided for remote aspects.
The Configurator.reconfigure() call takes the given pointcut and aspect specifications and also specifies that the specified aspect should be added. This reconfiguration specification however fails to capture the semantics of the reconfiguration in terms of the: ordering between the three aspects with the fragmenter aspect needing to be woven before the encryption aspect; dependencies involved by weaving an encryption aspect requiring the corresponding decryption aspect to be woven in a coordinated manner to ensure good running of the application; conflicts due to the weaving of the logging and encryption aspects.

4.3.1 Semantic Interaction Resolution

The encoder aspect in the AO-OpenCom Application Repository is tagged with appropriate metadata describing it’s semantic interactions, that is: the encoder aspect interface is tagged with the location path of the xml file containing the metadata having: the type of the aspect, dependency-interaction tags specifying a corresponding decoder aspect is dependent and must be woven when the encoder is applied; conflict-interaction tags specifying two constraints are specified in terms of: encoder aspect conflicts with a logging aspect in an untrusted domain if woven after the logging aspect; the encryption aspect can be allowed in an untrusted domain and can be based on the underlying domain policies. The Composition-Policy Aspect, as illustrated in Figure 6, then contains the ‘condition-action’ rules in terms: (i) a fragmenter must be woven before data is encrypted and similarly containing another policy describing that the decryption aspect must be woven before the reassembler aspect; (ii) the weaving of fragmenter and reassembler aspects must be coordinated across nodes to ensure both needs to be woven as they are dependent; and (iii) the weaving of encryption and decryption aspects must be coordinated across nodes to ensure that encrypted messages can be decrypted.

Figure 6: Composition Policy Example

When Configurator.reconfigure() is called on the Configurator of one of the nodes (referred as the ‘initiator’), the latter calls the Pointcut-Evaluator to locate all the target join points. On returning the located join points, the SRE aspect gets invoked. Using the target join points, metadata, that is from the fragmenter and logging aspects metadata, together with the encryption metadata, the SRE first parses their respective metadata to detect any semantic interactions. With the encryption aspect containing metadata with an order conflict for weaving with aspect type fragmenter, the SRE checks the Composition-Policy Aspect to determine if the constraint is valid for the reconfigured node and checks for any other application-specific or domain-specific restrictions. With the fragmenter metadata constraint matching the Composition-Policy Aspect metadata, and no application-specific or domain-specific constraints for this case, the SRE aspect instructs the AdviceHandler to weave a corresponding reassembler-aspect with the encryption-aspect woven in the second order to ensure the reconfiguration can be successfully done and thus semantically consistent. Moreover, since the original pointcut specification has been updated, the Configurator caches an updated version of the pointcut specification. In case remedy policies were not specified, the reconfiguration would be aborted with the rollback operation deployed for any changes.

4.3.2 Transparency

The approach naturally supports a selectively transparent approach as the SRE aspect and the Composition-Policy aspect can be pre-configured at application start-up time so that the application programmer who wishes to initiate a run-time reconfiguration needs only to make the appropriate call to Configurator.reconfigure(). This achieves complete transparency of consistency-related mechanisms from the code to invoke a reconfiguration. At the other extreme, the programmer can be explicit specifying the SRE and Composition-Policy aspects should be put in place for each reconfiguration. In this case, both aspects are woven on-the-fly (if they are not already present) before proceeding to perform the requested reconfiguration. Note that this extreme is still partially transparent as the programmer is protected from the low level details of actually weaving the SCM.

4.3.3 Flexibility

The use of a separate Aspect Metadata file to attach semantics of the aspect-components allows new metadata updates to be applied without having to recompile existing source-code. Moreover, our approach adds the SCM as an independently-deployable service which can be used for both local and distributed (re)configuration. This means that the SCM imposes no overhead when it not used, and can be dynamically woven/unwoven where and when required. We also believe that the approach, being based upon applying metadata and behaviour at common architectural elements (i.e., interfaces), can be applied generally to other AOM elements not just AO-OpenCom; indeed we see important future work in the deployment of our model in a wider range of systems.

4.4 Quantitative Evaluation

We next evaluate the overhead incurred by the SCM to perform dynamic reconfiguration. The baseline for our experiments is as follows; we reconfigure aspects at one join point using AO-OpenCom without SCM (in this case there are no conflicts to detect). This was performed locally on one node and then repeated with the aspects to be reconfigured spread across 4 client nodes with another node acting as the initiator of reconfiguration. Each node ran on a separate Core Duo 2 processor 1.8 GHz PC with 2GB RAM, using the Java-based version of the AO-OpenCom platform. Each measurement was repeated ten times and the mean value was calculated to discount anomalous results.

We then performed four separate reconfiguration cases and measured the performance overhead compared to the base-line measure above. In each of these cases we increased the scale of the experiment by increasing the number of aspects woven at the join point. Figure 7 shows the results of these four cases: (i) SCM manages reconfiguration where there is no conflict on a local node; (ii) SCM manages reconfiguration where there is no conflict across 5 nodes; (iii) we introduce an aspect conflict and SCM manages reconfiguration on a local node; iv) SCM manages a conflicting reconfiguration across 5 nodes. We identify the following results:
5. RELATED WORK

Few AO middleware platforms have addressed the challenges of performing consistent semantic (re)configuration. CAM/DAOP [10] and DyReS [12] are prominent examples of AOP middleware platforms but do not consider validation for aspect composition. Other prominent platforms such as JAC [7] and DyMac [6] are limited to solving aspect semantic by checking aspect ordering only. custAOMware [5] is a runtime AO component middleware allowing aspect interactions to be evaluated. The interaction model allows the detection of conflicts, dependencies and resolution of aspects by storing and accessing aspects metadata in the runtime kernel repository. However, compared to our approach, the platform offers only aspect configuration; the semantic validation of distributed dynamic reconfigurations is not supported. Spoon [8] is a Java program-transformation framework that uses AOP and compile-time reflection to ensure semantic consistency of the middleware components. However, this approach can only provide compile-time validation and is therefore unsuitable for adaptive software.

Outside the domain of AO middleware there a number of language-based AO approaches. CompAr [9] is a language-based AO approach to detect and solve aspect-composition issues. However, the approach only detects aspect-ordering interactions issues. Douence et al., [2] offers a similar approach for the automatic detection and explicit resolution of aspect interactions. However, the approach only allows for static analysis of aspect interactions. SECRET [3] uses a similar reasoning mechanism as our SCM by analysing all advices at a join point before composing them. However, the approach is language dependent requiring advices to be written in Aspect-J language for the conflicting patterns to be detected. Our approach differs from these related approaches, in that we introduce a general SCM within the field of AOM which can be applied to each of the aforementioned platforms.

6. CONCLUSION AND FUTURE WORK

In this paper we have demonstrated the need to consider aspect semantics to better support and ensure consistent composition and reconfiguration in dynamic AO middleware. We have illustrated our general approach to SCM for validating distributed dynamic reconfiguration, catering for semantic differences in terms of application-specific and domain-specific conditions. Moreover, our solution also prevents the combinatorial explosion that may result as a consequence of coupling metadata with the core aspect functionality allowing aspects semantics to be dynamically evolving without changing the source-code of running aspects.

Turning to future work, we first plan to investigate using our approach in a self-managing autonomic environment in which reconfiguration requests are initiated by the platform itself as opposed to the user. Then, we also plan to extend our semantic philosophy to use an ontology model such that the concepts between applications and domain can better be understood when building large-scale distributed middleware applications.

7. REFERENCES


