CONTROL AND VALIDATION MECHANISMS FOR INFORMATION, RESOURCES, AND DEPLOYMENTS IN DISTRIBUTED REAL-TIME AND EMBEDDED SYSTEMS

By

James R. Edmondson

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

May, 2012

Nashville, Tennessee

Approved:

Professor Aniruddha Gokhale

Professor Douglas C. Schmidt

Professor Janos Sztipanovits

Professor Gabor Karsai

Professor Sandeep Neema

To my wife Jenny for her unconditional love, encouragement, and support. To my family for forcing me to stop playing in traffic.

ACKNOWLEDGMENTS

I would like to thank my advisers Douglas Schmidt and Aniruddha Gokhale for their many meetings, guidance, and phone calls throughout my graduate career at Vanderbilt University. The tireless effort and patience they exhibited are very much appreciated and were invaluable in shaping my studies, work, and the fruits of this document.

My advisers were only the front lines of the many incredible staff, faculty, and researchers that engaged with me in conversation and research. Larry Dowdy, who recently retired, was a constant source of inspiration, discussion, and guidance in pedagogy, research, and surviving the graduate process. Julie Adams actively engaged me in artificial intelligence concepts and was never too busy to meet with me and speak frankly and directly about technologies and concepts in multi-agent systems.

Similarly, the members of my dissertation committee have been involved with my development as a researcher long before last year when we formed the dissertation committee. Janos Sztipanovits and Gabor Karsai both made themselves available, despite their stringent schedules, to meet with me about formal methods for book chapters and conference papers. Sandeep Neema inspired me with work on the AMMO project and his questions and comments helped me in both the writing of this document and the creation of effective presentations for job talks and the dissertation committee itself. And again, this dissertation would be nowhere near as interesting, informed, or legible without Aniruddha Gokhale and Douglas Schmidt.

My research would also not have been possible without the principal investigators and project collaborators like Joe Loyall and his team at BBN on the QED project. Much of this dissertation work was funded by the QED and AMMO projects, and without companies like BBN, Boeing, Lockheed Martin, and IHMC, and government agencies like the Air Force Research Labs, DARPA, and the National Science Foundation, I would not be the researcher I am today.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Of the types of networked systems, none are more difficult to manage, control, deploy, test, and maintain than distributed, real-time and embedded (DRE) systems. DRE systems are most often characterized as mission-critical applications executed in networked processes across heterogeneous architectures under stringent timing requirements and scarce resources. Though once typically only associated with avionics, manufacturing, and military applications, DRE systems have grown to include any distributed application where information must be delivered according to stringent quality-of-service needs despite scarce resources [46, 48, 79].

In fact, the challenges of DRE systems manifest themselves in most networked applications ranging from simple client-server model applications to mobile ad hoc systems-of-systems and large scale enterprise applications in a local area network or cloud. In a client-server model application, the clients may only be interested in a server if it provides a certain latency and liveliness requirement. A mobile ad-hoc network between unmanned drones in Afghanistan may require network topologies that cater to high bandwidth video feeds and low latency responses from operators near ground units or base stations. Similarly, a cloud provider may agree to a service level agreement (SLA) with an application developer that attempts to guarantee available resources operating within certain latency requirements.

Network engineers can provide static deployments at design time tailored to measured or expected latency, CPU, and memory availability on a target network, but the reality of a network is that resource availability and quality-of-service changes as clients, services, and users begin interacting [66]. The statically-assigned deployment during a design phase may be woefully out of date at deployment or runtime. The software developers and network

engineers are thus faced with a moving problem in DRE system deployment that may be unsuited to static deployments due to changing availability of resources, quality-of-service or even evolving SLA during peak and spike periods (e.g. the day of the Super Bowl for a sports website).

Ideally, the distributed application should be self-aware of its own high level requirements for how the application should interact with its own entities, and the distributed application should be able to dynamically respond to changes to the context of the system in order to adhere to the objective of the DRE system. For instance, consider the situation of soldiers carrying smartphones into a battlefield and capturing video of enemy combatants setting up IEDs. The smart phones should self-organize into efficient routing networks based on latency and quality-of-service workflows required of a routing network or at least inform the soldiers of the best route of information from a soldier back to a commanding officer. In order to self-organize, the DRE system needs to be aware of the environment and then have the capacity to make dynamic decisions based on quality-of-service and application-specific concerns.

Additionally, software developers are faced with the monumental task of validating their DRE systems before they are deployed in the real world. In critical systems like the software drivers of the first failed Mars Rover missions, a lack of adequate software tests featuring fine-grained sequencing of tasks and applications according to real-world scenarios can result in signicant loss of property, money, or even human lives. Consequently, along with latency- and resource-aware deployments, distributed regression testing of those deployments is an extremely important problem in DRE systems, and this regression testing requires the same need for a capacity to make dynamic decisions based on quality-of-service and application-specific concerns in a way that is portable to most architectures.

To compound both the deployment and testing challenges for DRE systems, latency and resource availability may change within a deployment throughout a test or real-world application lifecycle. Consequently, even if a perfect deployment is found for a DRE system

at design or system-startup, the quality-of-service could deteriorate over time to the point that the deployment would be ineffective or faulty. For websites, this may result in breaking an SLA for a quick response time. For a flight controller program, such an ineffective application could cause substantial loss of human lives, money, and property.

I.1 Overview of Research Challenges

In this section we list the research challenges that must be addressed in the context of deployment and testing of DRE systems.

I.1.1 Monitoring and Responding to Environments in Continuous Real-time Systems

For DRE systems to dynamically respond to their environment according to quality-of-service and application-specific concerns, they require powerful tools to quickly and efficiently monitor and mutate environment information that can be readily viewable by other entities in the network. Any solution in this area should be portable to most architectures, feature fast latency for transmission of data, and also the ability to quickly translate and act on new information. In computer systems, we usually call such a system a knowledge and reasoning engine.

On top of having evaluation latencies in the milliseconds and even seconds, most existing knowledge and reasoning engines require expensive ontologies and do not support location transparency between participating entries, which hinders cloud adoption. For example, DRAGO [77] specifically uses TCP and HTTP for connection-oriented communications between each reasoning entity. The SOMEWHERE [2] project is built on P2PIS [1], which creates a network of peer and variable mappings. DDL [6, 76] is similar to the other technologies but also provides data aggregation and the specification of incoming and outgoing peer/variable mappings.

Effective and efficient knowledge dissemination and reasoning in distributed, real-time,

and embedded (DRE) systems remains a hard problem due to the need for tight time constraints on evaluation of rules and scalability in dissemination of knowledge events. Limitations in satisfying the tight timing properties stem from the fact that most knowledge reasoning engines continue to be developed in managed languages like Java and Lisp (e.g., KaOS), which incur performance overhead in their interpreters due to wasted precious clock cycles on managed features like garbage collection and indirection. Limitations in scalable dissemination stem from the presence of ontologies and blocking network communications involving connected reasoning agents.

Chapter II addresses the existing problems with timeliness and scalability in knowledge reasoning and dissemination by presenting a C++-based knowledge reasoning solution that operates over a distributed and anonymous publish/subscribe transport mechanism provided by the OMG's Data Distribution Service (DDS). Experimental results evaluating the performance of the C++-based reasoning solution illustrate microsecond-level evaluation latencies, while the use of the DDS publish/subscribe transport illustrates significant scalability in dissemination of knowledge events while also tolerating joining and leaving of system entities.

I.1.2 Quality-of-Service-Enabled Distributed Mutual Exclusion

DRE systems need differentiated access to networked resources based on the criticality and utility of individual participants in the deployed system. Centralized mutual exclusion options like the master controller in the Google File System [27, 28, 63] offer fine-grained control and differentiated service for resource acquisition, but these controllers are single points-of-failure, do not scale well with large numbers of participants or requests, and suffer throughput issues due to the controller being a bottleneck. Distributed mutual exclusion algorithms like Maekawa [61] offer better scalability but less control and are generally difficult to implement.

Chapter III presents a distributed mutual exclusion algorithm called Prioritizable Adaptive Distributed Mutual Exclusion (PADME) that we designed to meet the need for differentiated services between applications for file systems and other shared resources in a public cloud. We analyze the fault tolerance and performance of PADME and show how it helps cloud infrastructure providers expose differentiated, reliable services that scale. Results of experiments with a prototype of PADME indicate that it supports service differentiation by providing priority preference to cloud applications, while also ensuring high throughput.

I.1.3 Real-time Application Deployment and Execution

For deployment and testing of DRE systems, the system must be portable to diverse heterogeneous architectures and able to work under a stringent deadline. Such a system should be loaded with useful testing and deployment features for not only sequencing distributed application launches but also termination of errant processes and sequencing of tests.

Automated deployment and testing of distributed, service-oriented applications, particularly mobile applications, is a hard problem due to challenges testers often must deal with, such as (1) heterogeneous platforms, (2) difficulty in introducing additional resources or backups of resources that fail during testing, and (3) lack of fine-grained control over test sequencing. Depending on the testing infrastructure model, the testers may also be required to fully define all the hosts involved in testing, be forced to loosely define test execution, or may not be able to dynamically respond to application failures or changes in hosts available for testing.

Most existing testing infrastructures feature a centralized approach [14, 17, 43, 82] that provides sequenced, fine-grained deployment and testing execution, but these solutions do not scale as well as a decentralized solution, are prone to single points-of-failure and bottlenecks, and may not be able to mimic decentralized testing scenarios (*e.g.*, when network latency from the centralized controller increases due to congestion at that node and the

controller is unable to service commands in time.). Examples of decentralized deployment and testing infrastructure can be found [90], but these are fickle, lack full features, and are limited to either only Linux machines or only Windows machines.

To address these challenges, Chapter IV describes an approach that combines portable operating system libraries with knowledge and reasoning, which together leverage the best features of centralized and decentralized testing infrastructures to support both heterogeneous systems and distributed control. A domain-specific modeling language is provided that simplifies, visualizes, and aggregates test settings to aid developers in constructing relevant, feature-rich tests. The models of the tests are subsequently mapped onto a portable testing framework, which uses a distributed knowledge and reasoning engine to process and disseminate testing events, successes, and failures. We validate the solution with automated testing scenarios involving service-oriented smartphone-based applications.

I.1.4 Emergent Redeployment of Real-time Systems

Once a DRE system is deployed, the latency between important nodes in the DRE system may have grown to the point where the system has become useless to users. To meet quality-of-service needs, a redeployment may be required, and this action will require meeting the new deployment constraints.

Other researchers [45, 80] handle detailed deployment resource constraints, such as CPU, bandwidth availability, etc. This approach features a powerful, useful set of algorithms for detailed planning, but the algorithms require users to estimate all usage (e.g., down to individual application timing using) with only monotonic functions, bandwidth specified usage per link (which can be highly variable as the program runs), and CPU and memory usage. Kichkaylo et. al. [45] described STRIPS-like planning routines to solve the planning problem. Kee et. al. [80] introduce a novel selection and binding technique based on hierarchical data using SQL queries. This prior research, however, does not allow

specification of data flow and instead tries to optimize according to estimated bandwidth usage.

In Chapter V, we present heuristics and genetic algorithms that we developed to approximate the subgraph isomorphic problem according to actual latencies for a distributed application informed by a high-level workflow provided by a user. We also present techniques for collecting and aggregegating latencies as well as a distributed voting system for determining the redeployment.

I.2 Dissertation Organization

The remainder of this proposal outlines each of the above research challenges in more detail and describes any advances made in addressing each research challenge and proposes additional research to be completed for the dissertation. Chapter II outlines the creation of a knowledge and reasoning engine for DRE systems. Chapter III introduces a distributed mutual exclusion algorithm for DRE systems that features differentiated levels of service and fault tolerance. Next, Chapter IV describes the efforts to create a portable deployment and testing infrastructure for DRE systems. Finally, Chapter V outlines our efforts to approximate the subgraph isomorphic problem according to a high level workflow and the challenges that still remain in applying this to DRE systems and incorporation into the MADARA middleware, which already includes the knowledge and reasoning engine and the portable deployment and testing infrastructure for DRE systems.

CHAPTER II

MONITORING AND RESPONDING TO ENVIRONMENTS IN CONTINUOUS REAL-TIME SYSTEMS

Effective and efficient knowledge dissemination and reasoning in distributed, real-time, and embedded (DRE) systems remains a hard problem due to the need for tight time constraints on evaluation of rules and scalability in dissemination of knowledge events. Limitations in satisfying the tight timing properties stem from the fact that most knowledge reasoning engines continue to be developed in managed languages like Java and Lisp, which incur performance overhead in their interpreters due to wasted precious clock cycles on managed features like garbage collection and indirection. Limitations in scalable dissemination stem from the presence of ontologies and blocking network communications involving connected reasoning agents. This paper addresses the existing problems with timeliness and scalability in knowledge reasoning and dissemination by presenting a C++-based knowledge reasoning solution that operates over a distributed and anonymous publish/subscribe transport mechanism provided by the OMG's Data Distribution Service (DDS). Experimental results evaluating the performance of the C++-based reasoning solution illustrate microsecond-level evaluation latencies, while the use of the DDS publish/subscribe transport illustrates significant scalability in dissemination of knowledge events while also tolerating joining and leaving of system entities.

II.1 Motivation

II.1.1 Overview of KaRL

Distributed, real-time and embedded systems (DRE) are characterized by scarce resources and high demands on what is available. More often than not, DRE systems are mission-critical systems where decisions on what to disseminate to other entities in the

network must be done within hard deadlines (often in microseconds). Additionally, a deployed DRE system may feature components or agents that are more important than other entities participating in the network or at least have tighter deadlines. Systems that feature such configurable priorities, deadlines, and resource requirements per entity are often called quality-of-service (QoS)-enabled systems.

There have been recent efforts to incorporate knowledge and reasoning into QoS-enabled systems [56, 59], but these efforts are not aimed at mission-critical scenarios, they use proprietary and minimal reasoning, and are implemented in managed languages like Java. These systems are limited in their support to scale to thousands of systems, cannot meet microsecond latency requirements, and are often developed simply as proof-of-concept prototypes or theoretical ideas. Even highly optimized solutions [42] built on C++ engines like RACERPRO operate in dozens of millisecond ranges.

Based on our extensive experience testing such DRE systems and the performance bottlenecks observed in meeting the knowledge reasoning demands for mission-critical QoS-enabled DRE systems, we surmise these limitations to stem from the following reasons: (1) contemporary knowledge reasoning systems are often developed using managed languages in which garbage collection, just-in-time compilation, and other language features tend to cause significantly high latencies during local reasoning operations – generally in the dozens to hundreds of milliseconds when inserting new knowledge into the system (or evaluating new rules); and (2) the dissemination of knowledge between knowledge reasoning peers in a network continues to rely on transports that do not offer quality-of-service differentiation (*e.g.*, UDP and TCP), require blocking communication, and do not scale well to thousands of reasoning peers. Additionally, the current knowledge reasoning engines that are available do not appear to have options for fault tolerance due both to node failures and incorrect knowledge.

To address these issues, we have developed the Knowledge and Reasoning Language Engine (KaRL Engine), which is an open-source knowledge reasoning and dissemination

framework for DRE systems. The design and implementation of KaRL focuses on delivering (1) microsecond latency for knowledge rule evaluations, (2) low latency in knowledge dissemination across a network, and (3) QoS mechanisms for fault tolerance, high priority reasoning peers, and knowledge conflicts.

II.1.2 Challenges for KaRL

We use an example case study to elicit requirements for scalable and real-time knowledge dissemination for DRE systems. Our case study shown in Figure II.1 comprises at least ten components that have been deployed into a network (dozens could be deployed as failover components, in case the primary component fails) and each of these components has dependencies between each other that must be reasoned out before the component is able to fully load. This is not an uncommon problem in DRE systems. One component may require services already be started (like GPS systems, schedulers, etc.) before it may execute its own logic which depend on those services being available in the network.

Each component in our case study has an init method that checks its deployment dependencies and a run method that is executed after all dependencies have been met. The run method will contain an arbitrary but important user program, and the init method may contain user logic in between checking for our component's dependencies being met and signalling that our component is ready to service others.

II.1.3 Challenge 1: Microsecond latencies

Mission-critical services depend on this ten-component deployment, and timing is crucial. Microsecond deadlines exist per component, and the entire deployment should take milliseconds, including local reasoning and dissemination across the network.

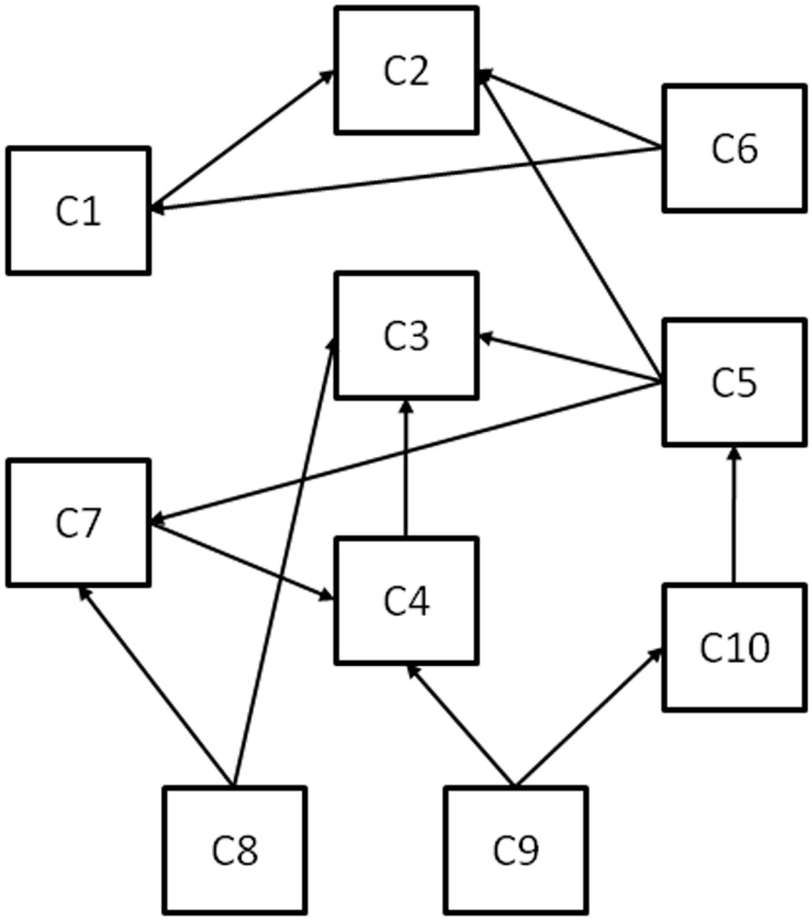

Figure II.1: Deployment plan of ten components with dependencies on each other

II.1.4 Challenge 2: Flexible user-provided logics

The user needs to be able to insert logic in between the reasoning service validating that the component's requirements have been met, and informing other interested components that this component's services are now available (after the user startup logic has been executed).

II.1.5 Challenge 3: Scalability and Failover

The solution should scale and be configurable to add new components, especially failover components that can replace faulty mission-critical components in the network. Some components are more mission-critical than others, and quality-of-service mechanisms should be available for fine-grained reasoning and dissemination of knowledge support.

II.2 Solution

This section describes KaRL, which is our solution to provide a real-time and scalable, distributed knowledge reasoning and dissemination capability. Before we describe the details of our solution we bring about the key requirements for such a capability in Section II.1.

Our solution to meet these requirements can be broken into three main vectors of attack. In Section II.2.1, we describe a highly optimized reasoning language and associated reasoning engine mechanisms that we developed to provide an intuitive abstraction to capture and mutate knowledge for distributed real-time and embedded systems (Challenge 1 and 2). In Section II.2.2, we describe our techniques for further optimization and scalability with a high performance, QoS-enabled anonymous publish/subscribe transport (Challenge 1 and 3). Finally, in Section II.2.3, we describe the quality-of-service mechanisms like knowledge quality and domains that address some key reasoning issues introduced in Challenge 3.

II.2.1 Language and Reasoning Engine Mechanisms

Despite the stringent timing requirements of Challenge 1, we were also interested in usability of the reasoning middleware. The Knowledge and Reasoning Language (KaRL) looks and feels like a modern programming language and allows complex conditionals and mutations to be constructed with multi-modal knowledge variables and constants. Support is provided for programming in both a rule-based (e.g., *condition => resulting knowledge*) and declarative rule programming (e.g., *condition && resulting knowledge*), and KaRL allows for both local variables (knowledge that will not be disseminated but can be referenced and mutated by the reasoning agent) and global variables (which will be dessiminated to interested reasoning agents).

To evaluate KaRL, we provide several engine mechanisms exposed to C++ programmers via an object-oriented library of function calls. Several mechanisms are provided that allow for accessing or mutating individual knowledge variables, interpretating KaRL logics, and providing fine-grained debugging capabilities of global knowledge state from a C++ executable or library. Each mechanism is atomically executed, *i.e.*, no outside entities may change the local knowledge state maintained by the KaRL reasoning engine while the user program is calling these mechanisms, and the library is completely thread-safe. We focus our discussions on the two chief mechanisms that are required to meet Challenge 2: *evaluate* and *wait*.

The evaluate and wait mechanisms both compile the provided KaRL user logic into an optimized format and then evaluate the resulting expression tree. The key difference between the evaluate and wait mechanisms in the KaRL engine is that an evaluate call evaluates the expression tree only once. In contrast to evaluate, the wait mechanism will evaluate the expression tree until the result of the expression tree is non-zero (true). To save precious microseconds on future evaluations, the expression tree is cached and future evaluations or waits on the KaRL user logic will not be compiled again. At the end of an

evaluate or wait call, the knowledge state mutations are aggregated into a knowledge event that is ready to then be disseminated to all interested peer entities.

In a system with periodic nature that needs to run a KaRL logic repeatedly for an extended period of time, the wait mechanism reduces computation overhead by limiting the re-evaluation to only when changes have been made to the global knowledge state. If the same logic is executed with an evaluate mechanism nested within a for loop, the logic will be evaluated constantly, even if no change has been made. Consequently, wait should always be preferred in a periodic system that synchronously waits until certain predicates are met. Evaluate should be used when the results of the logic are not critical to the software entity or there is other work that can be done while waiting for conditions or global state to change.

We mention these distinctions because our motivating scenario requires both mechanisms. The wait mechanism is inserted into the top of of the component's init function, e.g., component C5 requires components C2, C3, and C7 to be ready so it calls *engine.wait ("C2.ready && C3.ready && C7.ready")*. The user is then allowed to insert their own programming logic for setting up the component services and subsequently calls the evaluate mechanism with a global variable mutation to indicate that the service is ready, e.g., for component C5, *engine.evaluate ("C5.ready = 1")*.

After an evaluate or wait call has been made, a tuple of form $S(K,V)$ is created to represent the changes to the local state. K is a vector of length n of all changed knowledge variables and V is a vector of length n of the corresponding values, where $K_i = V_i$. With the local state changes aggregated into a tuple form, we have the first major piece of the knowledge event.

II.2.2 Dissemination of Knowledge in the KaRL engine

Once a knowledge tuple S has been formed via the KaRL language and reasoning engine mechanisms outlined in Section II.2.1, the event is almost ready to be passed to

Table II.1: Equation for updating an entity's Lamport clock c

$$c = \begin{cases} c+1 & \text{if E(S,t,q) is a local event} \\ max(c,t) & \text{if E(S,t,q) is a remote event} \end{cases}$$

interested reasoning agents. In this section, we discuss the mechanisms and additional information required to globally order the knowledge events as well as demonstrate how the anonymous publish/subscribe paradigm is used to disseminate knowledge to entities within a knowledge domain (a partition of the networking entities that might be interested in knowledge from this entity).

The tuple S which represents changes to an entity's local state is now augmented with a Lamport clock time t [51] and a quality q, which is essentially the priority of the entity to write knowledge to this variable and is explained in more detail in Section II.2.3. The Lamport time mechanism requires each entity to keep a local counter for each variable (referred to as V_t) in the knowledge base which represents the time stamp at which the last update occurred to the knowledge variable. Additionally, an entity Lamport clock c is maintained which is incremented only if the state is changed by a KaRL user logic via a KaRL engine mechanism call on the local entity (in which case $c = c+1$) or a knowledge event arrives which has a more recent Lamport clock time t, in which case $c = t$. To clarify this process, we outline the equation for updating the entity clock (c) in Table II.1.

Before a knowledge event of form $E(S,t,q)$ is constructed, the entity clock is updated according to the equation in Table II.1 and then we set $t = c$. After this step is completed, $E(S,t,q)$ is formed and the knowledge event is finally ready to be disseminated across a knowledge domain d. The domain can be set via the KaRL engine mechanisms to identify partitions of the global knowledge space that the software entity is interested in.

To facilitate the scalable delivery of knowledge events to entities in the domain, we use the OMG Data Distribution Service (DDS) [69]. Specifically, we provide a configurable

transport for both OpenSplice Community Edition, an open source implementation of the DDS standard, and RTI's NDDS, a closed source option.

The anonymous publish/subscribe paradigm used by DDS fits our solution concept in the following ways. First, DDS allows for us to be host and entity agnostic because it relies on an anynomous paradigm, which enables a programming model that intrisically supports fault tolerance (Challenge 3) because we are no longer tied to host/port information and are instead interested in components publishing knowledge relevant to our interests. If a sensor or actuator in a DRE system is tracking a globally recognized knowledge variable, the value of the variable is essentially more important than where it came from.

Second, with the dynamic nature of DRE systems (*e.g.*, failing and joining entities), any solution that requires predetermined URI information for connecting ontologies and variables is detrimental to a real-time mission critical system. Third, many of the features of DDS can be directly mapped to our application domain of distributed knowledge and reasoning, *e.g.*, DDS domains, for the most part, directly map to our concept of knowledge domains. Last, these transports have a history of scaling to thousands of components and data dissemination latency in microseconds [69, 90]. Consequently, in order to meet the scalability requirements in Challenge 3, we need to add very little overhead via local reasoning services (preferably low microseconds to nanoseconds) to the low latency of the DDS transport mechanisms (which are also in the microseconds on local area networks).

II.2.3 Quality-of-Service Mechanisms in KaRL

The knowledge event $E(S,t,q)$ has a parameter q, the knowledge event quality, which indicates the utility of the updates from this software entity, according to the KaRL logic developer. When forming the q of the knowledge event E, the KaRL engine aggregates the qualities of all variables in the knowledge event according to the following equation: $q = max(q, S.V_i)$. By allowing developers to set quality per knowledge variable or entity on each entity, a KaRL logic developer can exert fine-grained control over the entity's knowledge

quality on a per-variable basis. This same mechanism also allows a developer to safely deploy redundant components from our motivating scenario, the system will still function properly, and preference may be given to a component running on the best hardware or one which is running on top of the sensor that knowledge in the network is based on (e.g. the component is running on a GPS sensor).

Quality always overrides the Lamport clock value to dictate global ordering (i.e. t is a tiebreaker for q - not the other way around). This caveat means that regardless of how fast a low quality sensor, actuator, component, etc. is publishing its knowledge events, a high quality sensor's data will always be preferred. We do provide an optional timeout quality-of-service to allow for low quality knowledge events to eventually overwrite high quality knowledge, if new updates by high quality knowledge publishers haven't been received after some user-provided timeout interval. Wherever possible, we map our quality-of-service parameters to the DDS transport under the hood without requiring the user to understand DDS quality-of-service concepts.

If a knowledge variable has been updated via a remote knowledge event (*i.e.*, either q was greater than the quality attribute of the knowledge variable, the higher quality stored knowledge has timed out, or t is greater than the time attribute of the stored knowledge variable of the same quality q), then the time and quality attributes of the variable in the local KaRL context are updated to reflect the q and t of the received $E(S,t,q)$. Unless overridden by the developer, all knowledge variables on a software entity start with a quality of 0 which is considered neutral. Negative quality values are considered defective or lies, if the developer is considering Mechanism Design or untruthful agents as a reasoning possibility, and positive quality values are considered trusted entity events in the KaRL reasoning engine.

For readers familiar with the Data Distribution Service (DDS) [69] quality-of-service features, it may appear at first glance that quality is covered by ownership strength of DDS, but ownership strength is attached to the data writer entity and not a particular instance.

Similarly, deadline or liveliness may appear to be equivalent to the quality timeout of our solution approach, but neither is appropriate because they cannot be tied to our quality attribute for a data instance. Consequently, quality and quality timeout are application specific quality-of-service attributes tied intimately into reasoning engines and our specific solution approach, and we feel they are powerful quality-of-service additions to a distributed reasoning engine in a real-time or mission critical system.

We should note that many of the quality-of-service feature concepts of DDS do not directly map to our reasoning solution space. Consequently, many of our dissemination needs required custom solutions, *e.g.*, global ordering per instance with Lamport clocks, knowledge quality, and timeouts on quality. Because our solution is independent of transport used, DDS is not the only supported transport, and our future work will look into alternative mechanisms. Transports can be added by extending the Transport interface and implementing three of the methods: setup, send_data and send_multiassignment.

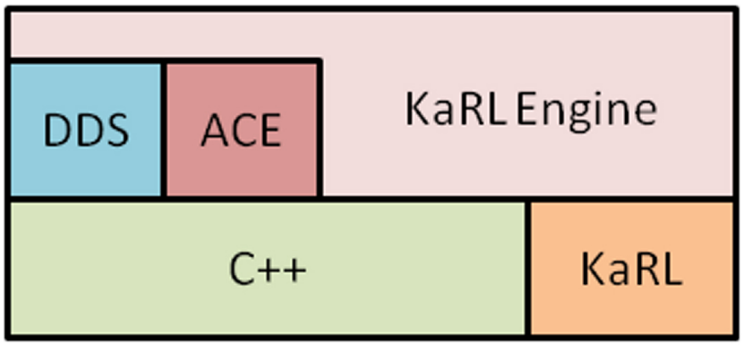

Figure II.2: KaRL Architecture

The resulting solution architecture is shown in Figure II.2. The Adaptive Communication Environment (ACE, project site at `http://www.dre.vanderbilt.edu/ACE`) shown in the diagram is used by the KaRL engine whenever portable operating-system

specific mechanisms are required (e.g. transient threads for receiving knowledge updates while user application logic is being ran in the main thread of execution).

II.3 Results

This section empirically evaluates the KaRL reasoning engine to determine if it is able to support the knowledge reasoning and dissemination requirements of DRE systems, which include microsecond-level latencies for evaluating knowledge events, and scalability to thousands of system entities. To that end, Section II.3.1 investigates the KaRL engine's throughput in interpreting logic. Section II.3.2 investigates the dissemination latency when using Open Splice DDS as a transport.

Unless specified otherwise, all experiments were conducted on five IBM blades with dual core Intel Xeon processors at 2.8 GHZ each and 1 GB of RAM running Fedora Core 10 Linux. The code was compiled with g++ with level 3 optimization, and each test featured a real-time class to elevate OS scheduling priority to minimize jitter during the test runs. Each test scenario was executed 20 times to form an average, and then the tests were repeated 10 times. The code for the test is available via the KaRL project site.

II.3.1 Knowledge Reasoning Latency

In this series of tests we create KaRL logics that test the KaRL engine's ability to process local knowledge without dissemination to test whether or not the engine can support the timing properties (microsecond latencies) of DRE systems. These tests isolate the latency of the KaRL engine from the network dissemination latency of the KaRL engine and DDS transport, which are tested in the next section. The results help to identify the strengths of KaRL to meet the performance requirements.

Although the performance of the KaRL engine in evaluating one knowledge rule is a useful metric, we were also interested in timing information for how efficient KaRL is in handling larger user logics. To test each of these scenarios, we constructed four total

logics. The first is a simple reinforcement (++.*var1*) that is evaluated inside of a C++ *for loop* 1,000,000 times. The second is a predicate guarded reinforcement (*predicate* => ++.*var1* that is evaluated inside of a C++ *for loop* 1,000,000 times. These logics were then expanded into KaRL logics that performed 10,000 simple reinforcements and predicate guarded reinforcements that were evaluated in a C++ *for loop* 100 times. The results of these experiments are shown in Table II.2.

Table II.2: Reasoning Engine Latency

Reinforcement	266 ns
Chained Reinforcement	169 ns
Guarded Reinforcement	269 ns
Chained Guard Reinforcement	196 ns

The results indicate that not only can our solution perform reasoning services within microsecond deadlines, but the KaRL engine can perform reasoning services on larger user logics in nanoseconds of time. The specific optimizations performed here that result in larger user logics performing faster is removing the function call and compiled expression tree lookup that is performed with every call to *engine.wait* or *engine.evaluate* as described in Section II.2.1.

II.3.2 Dissemination Latencies in KaRL

In this series of tests we create a number of KaRL logics that generate knowledge events and disseminate them across a local area network to interested reasoning peers using OpenSplice DDS as the transport. We break down the latencies experienced as the logic is compiled, evaluated, and disseminated. Our aim is to investigate how close KaRL is to the native dissemination protocol, which is an indicator of how much overhead, if any, it imposes on the dissemination and other latencies. To gauge these latencies, we construct tag-along logics shown in Table II.3.

Table II.3: Latency test setups

Process 1	Process 2
P0 == P1 => ++P0	P0 != P1 => P1 = P0

The two tag-along processes modify global knowledge variables and produce knowledge events for dissemination. The first process changes $P0$ whenever $P0 == P1$. The second process changes $P1$ whenever $P1 != P0$. Combined together with the *engine.wait* mechanism, these KaRL logics create a continuous system that we can stop after a certain number of logic evaluations. The logics are evaluated 5k, 25k, 50k, and 100k times and we include the network latencies obtained via a ping utility to guage the latency added by the KaRL engine and DDS dissemination transport.

Table II.4: Average Latency Results

	5k	25k	50k	100k	500k
Ping	114 us	114 us	114 us	114 us	114 us
Dissem	650 us	440 us	437 us	315 us	317 us
Compile	55 us	55 us	55 us	55 us	55 us
Eval	3 us	2 us	3 us	3 us	3 us

The compilation time penalty is only paid once: the first time `wait` or `evaluate` mechanism is called on the KaRL user logic. After the KaRL user logics are compiled with the wait or evaluation mechanisms, the expression is cached, and compilation time is reduced to the time it takes to lookup the KaRL logic in an STL map (part of the C++ standard library), which is a matter of nanoseconds on modern processors. Thus, after compilation time is reduced to lookup time, total evaluation latency is linear to the number of operations performed in the compiled expression tree.

Each dissemination latency average includes KaRL logic evaluation latency because in the dissemination latency we are interested in the time it takes for a user-provided KaRL

logic to be evaluated, and the changes propogated across the network and received by interested entities. Though we compute dissemination latency with roundtrip times, we present the one shot time in Table II.4 by dividing the average roundtrip time by 2, since the roundtrip is the result of a knowledge event going to the subscriber and then returning for 2 total events for a single latency calculation.

These experiments were repeated twenty times and the average latencies could deviate by as many as 70 us. Our lowest average latency for 100k and 500k messages hovered at 240 us and our highest was 380 us. We also saw jitter of 30+ us in the ping tests. We believe the majority of this jitter is being caused by operating system context switching due to I/O operations required for dissemination. The last important note to make about DDS and the dissemination latency is that this latency scales well to thousands of reasoning peers because of its reliance on broadcast and multicast paradigms where available and flexible asynchronous patterns underneath that even handle single-point communications efficiently on multi-core systems [69, 90].

II.4 Related Work

Most existing knowledge and reasoning engines require ontologies and do not support location transparency between participating entries. For example, DRAGO [77] specifically uses TCP and HTTP for connection-oriented communications between each reasoning entity. The SOMEWHERE [2] project is built on P2PIS [1], which creates a network of peer and variable mappings. DDL [6, 76] is similar to the other technologies but also provides data aggregation and the specification of incoming and outgoing peer/variable mappings. All of these technologies require users to build mappings of variables from local variables to other peers by their URIs, do not temporally order events, and do not distinguish between important and unimportant events. In contrast, our solution does not require such peer and variable mappings, it temporally orders events, is built for real-time systems, and allows for priorities.

Partition-based Reasonings, such as the High Performance Knowledge Base (HPKB) [3], allow for variable mappings between logical partitions. The concept of logical partitions is similar to our solution concept of knowledge domains, which separate the dissemination space and reduce message complexity. The consequence finding algorithm employed in the HPKB is, however, very specific to modal (*i.e.*, boolean) logic and will result in conflicts when multiple sensors or actuators are modifying the same variable with different values – an issue that occurs with multiple sensors informing listeners of target tracking or temperature. Our solution to these issues involves quality and multi-modal logic, which is described in Section II.2.

Policy management services, such as KAoS [81, 83], are built to work with semantic web languages based on the Web Ontology Language (OWL) [37]. KAoS essentially provides a policy reasoning engine which can be configured to work with the semantic variable mappings provided by the OWL standard. Recently, this type of policy management service was integrated into the Quality-of-service Enabled Dissemination (QED) project [56, 59], which also uses a publish/subscribe paradigm. However, the QED infrastructure was designed to disseminate user-provided data events and not knowledge and reasoning information and does not globally order events.

Several content-based tools for publish/subscribe ontologies have been created including OPS [87] and other Semantic Message Oriented Middleware [55, 70]. All of these tools feature a content-based querying system that forces subscribers to subscribe to all topics, which results in more messages than should be necessary. Additionally, these systems are typically built to sustain latencies of 1-2 seconds for performing complex queries, parsing complex types like graphs, or other types of semantic matching operations on content. These technologies also do not try to enforce temporal consistency (ordering) of events or event priorities between sensors, actuators, or software entities.

In contrast to these content-based pub/sub ontologies, KaRL allows for querying content through a *wait* mechanism within a specified domain to reduce message complexity

and work within time constraints (microseconds) for real-time systems. Consequently, we use matching on content after subjects (topics) have been matched. Additionally, we temporally order events and allow for knowledge quality, to specify importance and priority, which these technologies do not support.

CHAPTER III

QUALITY-OF-SERVICE-ENABLED DISTRIBUTED MUTUAL EXCLUSION

Popular public cloud infrastructures tend to feature centralized mutual exclusion models for distributed resources, such as file systems. The result of using such centralized solutions in the Google File System (GFS), for instance, reduces scalability, increases latency, creates a single point of failure, and tightly couples applications with the underlying services. In addition to these quality-of-service (QoS) and design problems, the GFS methodology does not support generic priority preference or pay-differentiated services for cloud applications, which public cloud providers may require under heavy loads.

This paper presents a distributed mutual exclusion algorithm called Prioritizable Adaptive Distributed Mutual Exclusion (PADME) that we designed to meet the need for differentiated services between applications for file systems and other shared resources in a public cloud. We analyze the fault tolerance and performance of PADME and show how it helps cloud infrastructure providers expose differentiated, reliable services that scale. Results of experiments with a prototype of PADME indicate that it supports service differentiation by providing priority preference to cloud applications, while also ensuring high throughput.

III.1 Motivation

III.1.1 Overview of PADME

Even enterprise-level applications have quality-of-service and differentiation needs. To highlight the need for quality-of-service enabled distributed mutual exclusion—whether in enterprise or DRE systems, we highlight known problems in a popular cloud file system.

The Google File System (GFS) was designed to support the sustained file throughput capacities of the Google search engine [27, 28, 63]. GFS provides high throughput in

a single cluster of thousands of computers, each servicing the Google search engine. Although the GFS scaled well to hundreds of terabytes and a few million files in append-mode (GFS does not support overwriting a file), other quality-of-service (QoS) properties (*e.g.*, latency, throughput of small files—which is common in many applications, and differentation amongst applications) were not the focus of its initial design.

Scalability problems with GFS began appearing when the centralized master server was forced to process tens of petabytes worth of data requests and appends [27]. As a short-term solution, Google engineers used a centralized master server to manage a *cell* of the overall cluster. Although this approach provided some fault tolerance against the single master failing, some failures still occurred, and throughput and scalability suffered [63].

As Google grew, so did its list of services and applications. Since GFS focused on throughput rather than latency and scalability, performance issues appeared with certain applications, such as GMail, Youtube, and Hadoop [63]. Google's temporary solution to overcome this problem was the BigTable application, which was layered atop GFS and packed small files into the large 64 MB file metadata that had been in place since their Internet crawler was first deployed [27, 63].

For cloud applications (such as Youtube) that can be buffered, the latency of the GFS system has mostly been acceptable. For applications with file accesses and writes on the cloud, however, Google is looking into replacements for GFS that provide better QoS [63]. Deferring these changes can be costly for applications that have a GFS-like interface or these applications face mounting integration issues.

Figure III.1 shows the scalability problems of a centralized server for reading and writing to a segregated file system. In this figure, the light-shaded lines represent the computational and bandwidth resources that are utilized and dark represents the wasted resources. According to Google network engineers, the major bottlenecks of the GFS system include the inability to provide native overwrite operations (GFS supports only append mode and applications have to work around this), the 64 MB metadata for even small files, and the

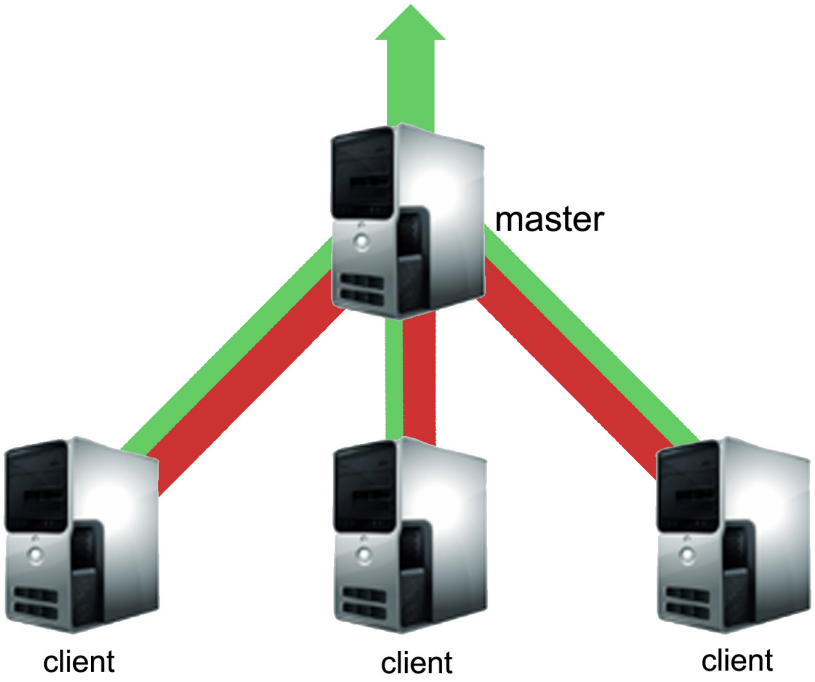

Figure III.1: GFS Centralized Master Controller

centralized master controller that every request and write goes through in each cluster cell. The centralized master controller provides mutual exclusion for read-write operations, garbage collection, and replication and persistence. Even with extra computing resources, this controller reduces scalability and throughput, and significantly increases latency due to queuing [63].

In addition to the latency bottleneck, reduced overall throughput, and lack of fault tolerance, GFS's centralized architecture also treats all application requests equally. Applying this system to cloud providers with scarce resources and steady client application deployments means there is no built-in differentiation between client priorities according to their payscale or other factors.

Ideally, we'd like to utilize a decentralized architecture for file system access and mutation similar to the one shown in III.2. As with Figure III.1, the decentralized algorithm still has a certain amount of overhead (shown in dark) due to the replicas and clients working

together and coming to a consensus. In contrast, however, the overall capacity of the decentralized approach scales with the number of participating nodes, rather than being limited by a single master controller.

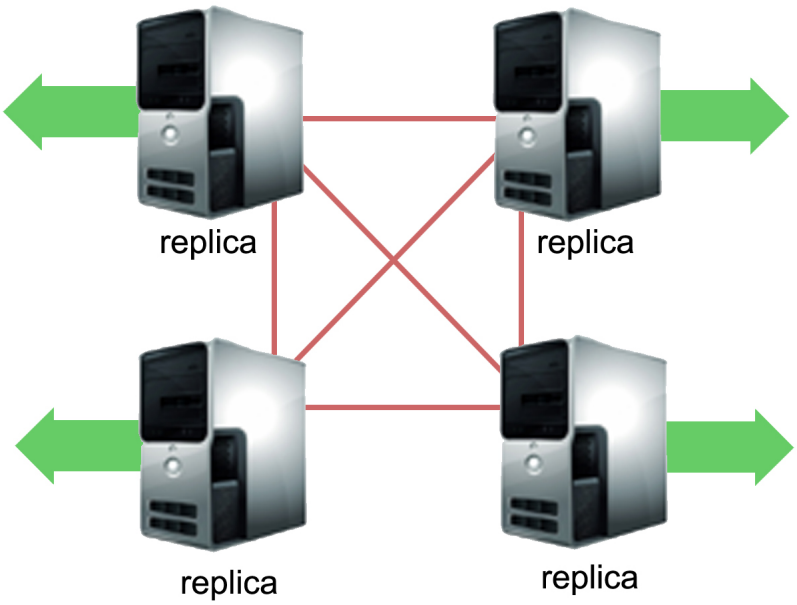

Figure III.2: Scalability of a Decentralized Methodology

III.1.2 Challenges for PADME

III.1.3 Challenge 1: Avoid centralized bottlenecks

Any solution should avoid a centralized bottleneck, such as a master controller which restricts file throughput, and increases latency due to queuing and limited bandwidth through a single host.

III.1.4 Challenge 2: Handle failures of nodes and applications

GFS requires that master controllers be manually reset by an engineer when they fail [63]. This is not an acceptable solution for most DRE system or enterprise application developers.

III.1.5 Challenge 3: Support aggregated priorities

A distributed mutual exclusion algorithm should be flexible to any user-specified priority, including those based on expected cost, payment, etc. of cloud applications.

III.1.6 Challenge 4: Provide differentiated levels of service

Not all applications and users are the same. High priority applications and users should have lower latency and higher throughput when resources are scarce. No other distributed mutual exclusion algorithms appear to do this explicitly.

III.2 Distributed Mutual Exclusion in Public Clouds

This section presents an algorithm called *Prioritizable Adaptive Distributed Mutual Exclusion* (PADME) that we developed to meet the cloud file system challenges described in Section III.1. The PADME algorithm performs two main operations:

1. It maintains a spanning tree of the participants (*e.g.*, applications, customers, or anything that wants access to the file system) in the network. The spanning tree is based upon the customer or application priorities and the root of the spanning tree will be the highest priority entity in the system. Depending on the network topology and customer demands this root can and will change during runtime operations.

2. It enforces mutual exclusion according to user-specified models and preferences for messaging behavior. The models supported by the PADME algorithm include priority differentiation and special privileges for intermediate nodes in the spanning

tree (intermediate nodes are nodes between requesting nodes). Each model may be changed during runtime if required by the cloud provider, applications, or users.

Below we describe the PADME algorithm and show how it can be implemented efficiently in cloud middleware platforms or cloud applications, wherever the distributed mutual exclusion is appropriate.

III.2.1 Building the Logical Spanning Tree

PADME builds a logical spanning tree by informing a cloud participant (*e.g.*, an application that is using the cloud infrastructure to manage the file system) of its parent. Under ideal circumstances, all the spanning tree construction should be performed in the cloud infrastructure without affecting user applications.

A participant need not be informed of its children as they will eventually try to contact their parent, establishing connections on-demand. PADME uses this same mechanism to reorder the tree when optimizing certain high priority participants. Each participant is responsible for reconnecting to its parent through the cloud API, middleware, or however the cloud offers file system services.

To establish which parent to connect to, PADME's joining process multicasts its priority and waits for responses during a (configurable) timeout period. The closest priority above the joining process becomes the parent. The process can arbitrarily connect to a different parent later, in which case the parent will need to remove pending requests from the child. This step can be postponed until the file access permission returns to this node for the specific requester to eliminate the need for a parent to care about children moving to other parents. If the connection no longer exists the token is sent back up to higher priority processes in the logical spanning tree of cloud participants.

Cloud middleware developers could decide to let each cluster form its own priority-based spanning tree and connect roots of each tree to form the cloud file system mutual

exclusion tree. The PADME algorithm presented in Section III.2.2 will work with any spanning tree as long as connected nodes can communicate. Section III.2.3.2 covers PADME's fault tolerance support. Figure III.3 shows the construction of such a spanning tree out of priority information.

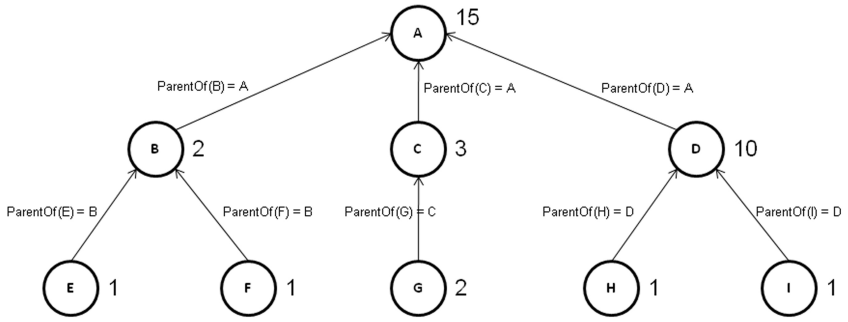

Figure III.3: Building a Balanced Spanning Tree

In this case, the tree is balanced, though it need not be. During runtime, an application or user may add or remove a participant, rename participants, or conduct other such operations to organize the intended tree and dynamically respond to changes in request load or priority changes.

PADME's tree building phase requires updating affected participants with parent information (*i.e.*, informing them of the direction of the logical root of the tree). The messaging overhead of maintaining a logical spanning tree is not included in our algorithm details because cloud providers can simply build a spanning tree once manually if desired. For example, the cloud provider may know that the application deployments are static, or the provider is anticipating a specific scenario like spikes of file access during a major sporting event and wants to give preference to this activity during the championship match.

Even in statically assigned spanning trees, higher priority cloud participants should be pushed towards the root to give them preferential service. The logical token will be passed back to the root of the tree before continuing on to the next participant in our algorithm.

Participants at higher levels of the tree experience lower message complexity, faster synchronization delay, better throughput, and even higher QoS for the target system and a preference for high priority customers, as shown in Section III.2.3.

III.2.2 Models and Algorithm for Distributed Mutual Exclusion

Before accounting for faults, PADME requires just three types of messages: *Request, Reply,* and *Release*. A Request message is made by a participant that wants to acquire a shared resource, such as cloud file system access. A Request message traverses up the spanning tree from the participant node to the root via its parent and ancestor nodes. A Reply message is generated by the root after access to the resource is granted. The Reply message traverses from the root to the requesting participant node. The Release message traverses up the tree from the node that holds the shared resource towards the root once the node is ready to release the resource. The interaction of these messages is shown in Figure III.4.

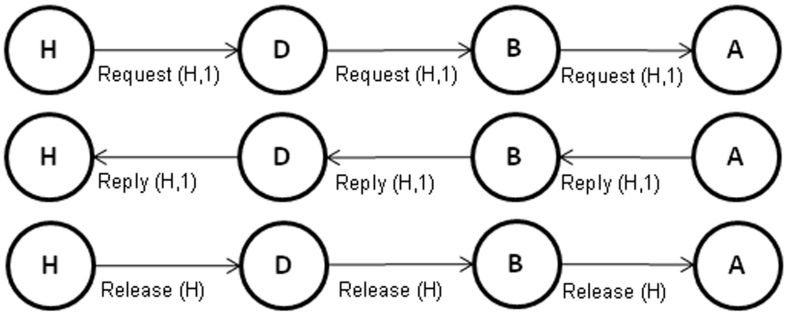

Figure III.4: PADME Messages for Mutual Exclusion

PADME supports four models (*Priority Model, Request Model, Reply Model,* and *Release model*) that describe the semantics of actions performed by any participant in the spanning tree that receives one of the three types of messages, as well as QoS differentiation that must be supported. The latter three models are named according to whether

an intermediate participant can enter its own critical section, *i.e.*, exclusive access to the cloud's distributed file system resource, upon receipt of the message type. These models stem from our approach to distributed mutual exclusion and optimizations that allow shorter synchronization delay between critical section entries and improved QoS via user-specified requirements to middleware.

The configurations of the Request, Reply, and Release models may be changed at run-time to yield different QoS, including

- **Higher critical section throughput** – *i.e.*, the number of file accesses possible to the cloud's distributed file system,
- **Changes in fairness** – *e.g.*, going from preferring higher priority participants to giving everyone a chance at the critical section – a requirement of our motivating scenario in Section III.1,
- **Fewer priority inversions** – *i.e.*, alleviating the situation where a low priority participant gets a critical section entry before a high priority participant, even though a critical section request from a higher priority participant exists, and
- **Lower average message complexity** – *i.e.*, fewer messages being required per critical section entry.

These four models are described below and each are integral components in the algorithm that may be tweaked to affect performance, usually at the cost of possible priority inversions.

III.2.2.1 Request Models

PADME provides two Request Models: *Forward* and *Replace*. The Forward Request Model requires a parent to immediately forward all requests to its own parent. The Replace Request Model requires a parent to maintain a priority queue of child requests, which

should have the same Priority Model as the root participant. Under the Replace Request Model, a node only sends a Request to its parent if there are no Request messages in its priority queue, or if the new Request is of higher priority than the last one that was sent. The Replace Request Model is slightly harder to implement, but it results in messages only being sent when appropriate and may alleviate strain on the root node. It also will result in less message resends if a parent node fails.

III.2.2.2 Reply Models

PADME provides two Reply Models: *Forward* and *Use*. The Forward Reply Model requires a parent to immediately forward a reply to its child without entering its own critical section, regardless of whether or not it has a request pending. The Use Reply Model allows a parent P_c to enter its critical section upon receiving a Reply message from its parent P_p, if P_c currently has a Request message outstanding.

III.2.2.3 Release Models

PADME provides two Release Models: *Forward* and *Use*. The Forward Release Model requires a participant to immediately forward a Release message to its parent without entering its own critical section, regardless of whether it has a request pending. The Use Release Model allows a participant to enter its critical section when it receives a Release message from one of its children, if the participant has an outstanding Request pending.

When applying the Use model, the participant must append its identifier onto the Release message if it entered its critical section (as shown in Figure III.5, where each participant

appends their release information to their parents when the critical sections have already been entered), which may result in a Release message containing multiple instances of the participant identifier. Consequently, appropriate data structures should be used to allow for these duplicates (*e.g.*, multisets). These duplicates enable proper bookkeeping along

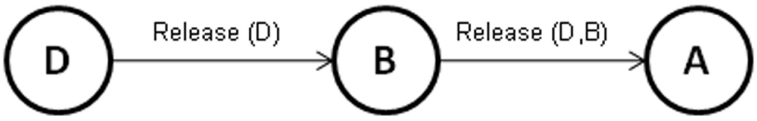

Figure III.5: Appending Identifier to Release Message

the token path since up to two Request messages may require removal from each affected priority queue.

III.2.2.4 Priority Models

PADME provides two Priority Models: *Level* and *Fair*. The Level Priority Model means that one Request of the tuple form Request $< I_m, P_m, C_m >$ should be serviced before Request $< I_n, P_n, C_n >$ if $P_m < P_n$. P_x stands for the priority of the participant identified by I_x, and C_x refers to the request id or clock. If a tie occurs, the clocks C_x are compared first and then the identifiers. This ordering does not guarantee the absence of priority inversions, and priority inversions may happen when the token is in play (walking up or down the tree).

The Fair Priority Model means that one Request of the form Request $< I_m, P_m, C_m >$ should be serviced before Request $< I_n, P_n, C_n >$ if $C_m < C_n$. Upon a tie, the priority levels are compared, followed by the identifiers. The Fair Priority Model will result in all participants eventually being allowed into a critical section (assuming bounded critical section time and finite time message delivery), whereas the Level Priority Model makes no such guarantees.

III.2.2.5 Overview of PADME's Mutual Exclusion Algorithm

When a participant needs to enter its critical section (*e.g.* an agent is requesting exclusive access for writing to a cloud file system), it sends a Request message to its parent, who then forwards this Request up to its parent, until eventually reaching the root node. The

Request message is a tuple of the form Request $< I, P, C, D >$, where I is the identifier of the requesting participant, P is the priority level (level), C is a timer or request id, and D is a user data structure that indicates the shared resource id (*e.g.*, the name of the file that will be accessed in the cloud file system) and any other data relevant to business logic. There is no reason that any of these variables be limited only to integers. For more information on the election of cloud entity and volume identifiers that may be useful for a cloud file system implementation, please see the Renaming Problem [49].

The choice of a timer mechanism (also known as a request id) may result in varying ramifications on the Fair Priority Model, discussed in Section III.2.3. A timer should be updated (1) only when sending a Request or (2) any time a Request, Reply, or Release message with the highest time that of the agent who is receiving message or the time indicated in the message sent. The latter method will result in time synchronization across agents which can be helpful in synchronizing fairness in late joining agents or when switching from Level Priority Model to Fair Priority Model. Resending a Request does not increase the local request count. A Request may be resent if the parent participant faults or dies to ensure that a Request is serviced eventually by the root.

The root participant decides which Request to service according to a priority mechanism. After determining who gets to enter their critical section next, a Reply message is sent of the form Reply $< I, C >$ or $< I, C, D >$ where I is once again the identifier of the requesting participant, C is the count of the Request, and D is an optional parameter that may indicate business logic information, *e.g.*, the name of the file to be overwritten. Once a Reply message reaches the intended requesting participant, the requesting participant enters its critical section.

Upon exiting the critical section, the requesting participant must send a Release message to its parent participant, who forwards this Release message to its parent until the root receives the message. Release messages have the form Release $< I_0, I_1, \ldots I_n >$ or $< I_0, D_0, I_1, D_1, \ldots I_n, D_n >$ where $I_0, I_1, \ldots I_n$ is a list of participant identifiers that used

their critical section along this token path, and $D0, D1, \ldots Dn$ is a parameter that may indicate business logic information *e.g.*, the frequency that is being released. The root participant and any participant along the token path should remove the first entry of each identifier in$< I_0, I_1, \ldots I_n >$ before forwarding the Release to its parent for proper bookkeeping.

III.2.3 QoS Properties of the PADME Algorithm

PADME's Request, Reply, Release, and Priority Models described in Section III.2.2 are orthogonal and may be interchanged by the user to accomplish different QoS, higher fault tolerance, reduced message complexity at key contention points, or critical section throughput during runtime. Each combination has certain QoS properties that may fit an application's needs better than the others, *e.g.*, each has certain synchronization delay characteristics, throughput, and even message complexity differences during fault tolerance. To simplify understanding of the different combinations of these models, we created a system of model combinations that we call Request-Grant-Release settings that codify these combinations.

III.2.3.1 Non-fault Tolerance Case

PADME's most robust Request-Reply-Release setting is the Replace-Use-Use model, which corresponds to the Replace Request Model, Use Reply Model, and Use Release Model. The Replace-Use-Use setting requires each participant to keep a priority queue for child Requests (described further in Section III.2.2), but its primary purpose is to limit the number of message resends during participant failures or general faults to only the most important Requests in the queue. Replace-Use-Use is consequently very useful when reducing the number of messages in the network is a high priority.

PADME's Use Reply Model of the Replace-Use-Use combination allows a participant to enter its critical section before forwarding on a Reply message to an appropriate child. The Use Release Model allows a similar mechanism in the opposite direction, on the way

back to root. Both of these use models work well in conjunction with the Fair Priority Model to not only decrease synchronization delay (and thus increase critical section throughput) but also favor higher priority participants, as those higher priority participants will be closer to root and may have up to two chances of entering a critical section along a token path from root to a requestor and back to root.

Even when the Forward-Forward-Forward combination is used, the higher priority participants closer to root will still have lower message complexity and lower average synchronization delay than lower priority participants (*e.g.*, leaf nodes). This results from the token path being longer from the leaf nodes to root. Consequently, placing frequently requesting participants closer to the root node in the logical routing network can result in increased performance (Section III.2.3 analyzes message complexity and synchronization delay).

All of PADME's Fair Priority Model-based settings inherently may lead to priority inversions. PADME's Level Priority Model by itself, however, does not eliminate priority inversions. To eliminate priority inversions from occuring in PADME, the Level Priority Model must be used in combination with *-Forward-Forward.

If the settings contain Use Models for either the Reply or Release Models when the virtual token progresses towards an entity, it is possible that a higher priority request may be delayed at the root node that arrived after permission was granted to a lower priority entity. In practice, completely eliminating priority inversions is typically not as important as throughput. Settings that enable the Use Model for both Reply and Release models therefore have much higher throughput proportional to the depth of the spanning tree.

III.2.3.2 Fault Tolerance Case

There are several options for fault tolerance that can be supported by PADME, but we focus on the most straightforward to implement and still be robust to failures. We use a classic approach called a Byzantine view change [9] whenever a root node faults (*i.e.*, becomes unavailable). A Byzantine view change is a type of consensus that requires a

majority agreement for electing a primary node. A Byzantine view change would always occur after the initial deployment to elect the defacto root node—the one specified in the deployment. Since only one node should have no parents upon deployment (the initial root node), this election is straight forward and there will only be one candidate. Doing this at the beginning simplifies the fault tolerant scenarios that must be handled.

The only other time a Byzantine view change would be initiated would be when a non-root participant detects that its parent is unresponsive, attempts to connect to a new parent using the protocol discussed in Section III.2.1, and receives no response from a higher priority entity. This can be induced by an administrator if faulty performance is detected or priority on a particular node has been elevated higher than the previous root, but generally, this situation occurs when a child participant believes it may be root. If there really is a need for a new root, an election takes place based on the importance of the available candidates, and the one with the highest priority gets pushed up to the root of the logical token network.

Upon electing a new root node, the children of former root push all pending requests up to the new root, and the system resumes operation. When a token is believed to be lost by the root node, *e.g.*, after a configurable timeout based on a query of the participants for any outstanding tokens, a new token is generated.

If the root did not die—but believes a token has been lost—it can either multicast a query for the outstanding tokens and regenerate, or it can use targeted messages along the path the token took. In the latter case, the token will either pass normally with a release message or the root will have to regenerate a token. This process is shown in Figure III.6.

The root participant knows it sent a permission token to *D* and that it went through child *B* to get there. There is no reason to send a recovery message to anyone other than B and let it percolate down to *D* unless multicast or broadcast is being used and the operation is inexpensive for the network.

Any time a parent connection is lost, the orphaned child establishes a new parent with

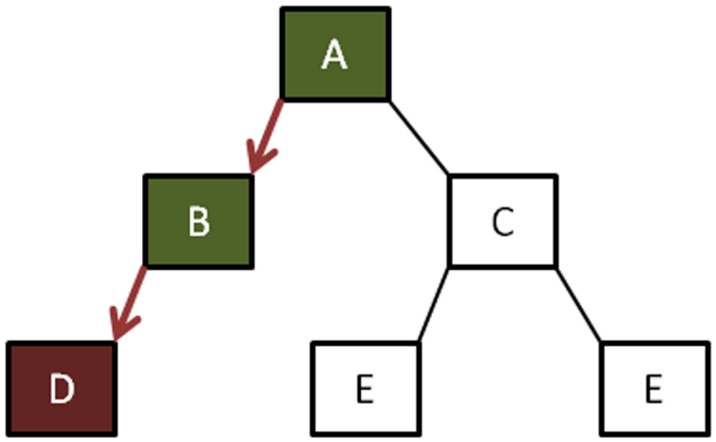

Figure III.6: Targeted Recovery of an Outstanding Token

the protocol outlined in Section III.2.1 and resends all pending requests from itself and its children.

III.3 Results

This section evaluates results from experiments conducted on a simulated message-oriented prototype of the PADME algorithm over shared memory. We simulate critical section time (the time a participant uses its critical section), message transmission time between participants (the time it takes to send a Request, Reply, or Release message between neighboring cloud participants in the spanning tree), and critical section request frequency (how often a participant will request a critical section if it is not already in a critical section or blocking on a request for a critical section). Our experiments focus on the following goals:

1. **Quantifying the degree of QoS differentiation**. The goal of these experiments is to gauge whether or not the PADME algorithm provides QoS differentiation for

participants in a public cloud (see Section III.1) and whether or not the Request-Reply-Release models described in Section III.2.2 have any tangible effects on QoS differentiation and throughput. Our hypothesis is that the PADME algorithm will provide significant differentiation based on proximity to the root participant.

2. **Measuring critical section throughput**. The goal of these experiments is to measure the critical section throughput of the PADME algorithm. Our hypothesis is that the PADME algorithm will provide nearly optimal critical section throughput for a distributed system, which is the situation where synchronization delay is t_m—the time it takes to deliver one message to another participant.

We created a simulator that allowed us to configure the Priority, Reply, and Release Models for several runs of 360 seconds. The experiments ran on a 2.16 GHZ Intel Core Duo 32 bit processor system with 4 GB RAM. The experiments were conducted on a complete binary tree with seven participants and a depth of 3 on a simulated network of seven participants: one high importance, two medium importance, and four low importance.

III.3.1 Quantifying the Degree of QoS Differentiation

The QoS Differentiation experiments quantified the ability of the PADME algorithm to differentiate between cloud customers and applications based on their priority—a derived metric that may directly correlate to money paid by the users of the system or a service's importance in serving cloud customers. We present the results as they relate to algorithm configurations and the tradeoffs between reducing priority inversions and increasing file access throughput.

III.3.1.1 Setup

Two experiments are presented here. The first has a message transmit time of 1 sec and a critical section entry time of 1 sec. The second experiment has a transmit time (t_m) of 0.5

sec and a critical section entry time of 1 sec. The latter experiment more accurately emulates network and Internet traffic since transmit time is rarely 1 sec. We use a convention of referencing models as Priority-Request-Reply-Release when describing PADME settings for brevity.

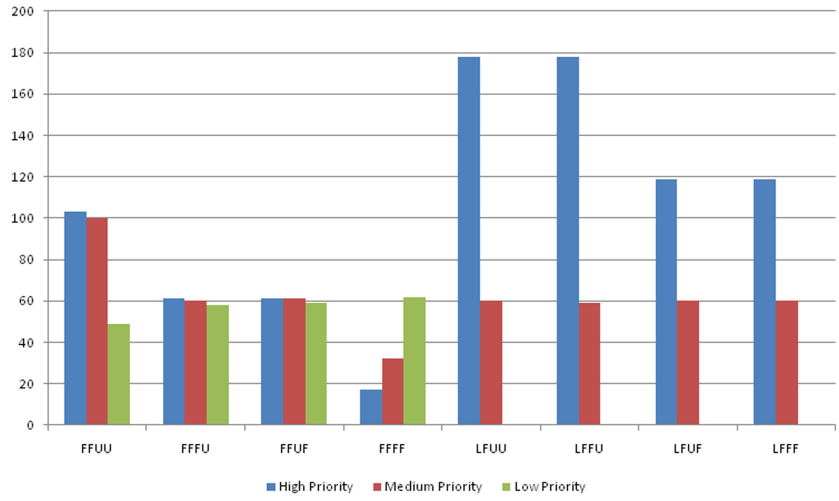

Figure III.7: Priority Differentiation with CS time of 1s and Message latency of 1ms

Our PADME prototype does not yet include the Replace Request Model, so no configurations with the Replace Request Model are included in this section. We expect, however, that throughput and latency for this model should be identical to the Forward Request Model.

III.3.1.2 Analysis of Results

Figure III.8 and Figure III.7 outline the results for this test. The root participant had high priority, the participants on the second level had medium priority, and the leaf nodes on the third level had low priority.

In the analysis below we reference the PADME model configurations as Priority-Request-Reply-Release for brevity. Differentiation increases under certain models as the message time is decreased. This result appears to occur in Fair-Forward-Forward-Use, but is likely

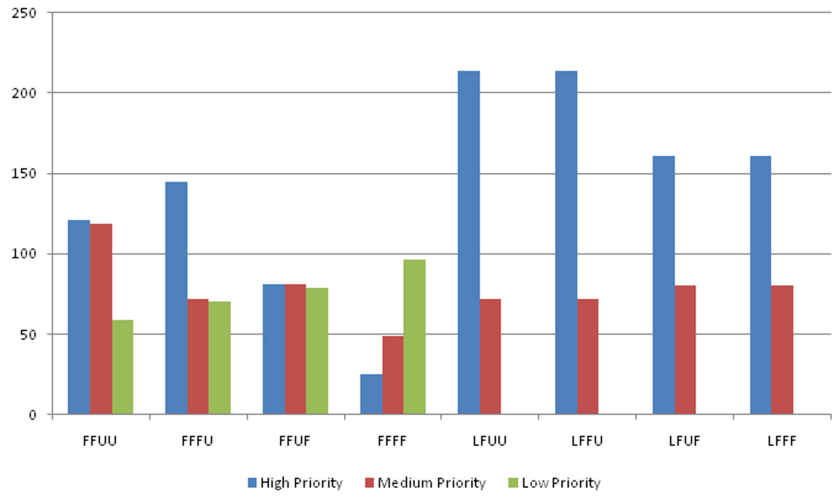

Figure III.8: Priority Differentiation with CS time of 0.5s and Message latency of 0.5ms

true of Forward-Use-Forward. Of the Request-Reply-Release combinations that appear to show the best differentiation amongst priority levels, those with Level Priority Model differentiate the best. Those with any type of Level Priority Model differentiate according to priority levels, which makes sense.

More interesting, however, is how the Fair-Forward-Use-Use, Fair-Forward-Forward-Use, and Fair-Forward-Use-Forward model combinations allow for better QoS in comparison to Fair-Forward-Forward-Forward. Although we are being fair in priority policy, this policy shows favoritism to the lower priority levels, which have more participants, and consequently get more critical section entries under a fair priority policy. Forward-Use-Use, Forward-Forward-Use, and Forward-Use-Forward offset these policy decisions by allowing critical section entries as the Reply and Release messages pass through participants, to allow for higher critical section entries than would have been possible with the more intuitive Forward-Forward-Forward. If we increased the number of high priority and medium priority participants, we would have even better differentiation during Fair Priority Policy because less time is spent in pure overhead states where the token is percolating back up the tree and not being used.

III.3.2 Measuring Critical Section Throughput

Differentiation can be useful, but if the system becomes so bogged down with messaging or algorithm overhead that file system or other shared resource throughput is greatly reduced, then no real benefits are available in the system. The experiments in this section guage the performance of the PADME algorithm in delivering file system access to customers, cloud applications, or persistent services.

III.3.2.1 Setup

Two experiments are presented here. The first experiment sets the message transmission time (t_m) to 1 ms, critical section usage time to 1 s, and we generate a new request once every 1 ms (when not using or blocking on a critical section request). The second experiment has a fast message transmission time of 0.5 ms (*i.e.*, more in line with a local area network transmit for a cluster within a cloud) and generates a new request every 0.5 ms (unless blocking on or using a critical section).

Our PADME prototype does not yet include the Replace Request Model, so no configurations with the Replace Request Model are included in this section. We expect, however, that throughput and latency for this model should be identical to the Forward Request Model.

III.3.2.2 Analysis of Results

Figure III.9 and III.10 show the results for these tests. These results are above our proposed theoretical max where synchronization delay = t_m, because participants are able to enter their critical sections (*e.g.*, access a file) both on a release and reply using the Use models.

Each model equals or outperforms a centralized solution. A centralized solution would have required a critical section entry (1 sec) plus two message transmissions—Release (1 sec) and Reply (1 sec)—per access resulting in only 120 critical section entries in a 360s

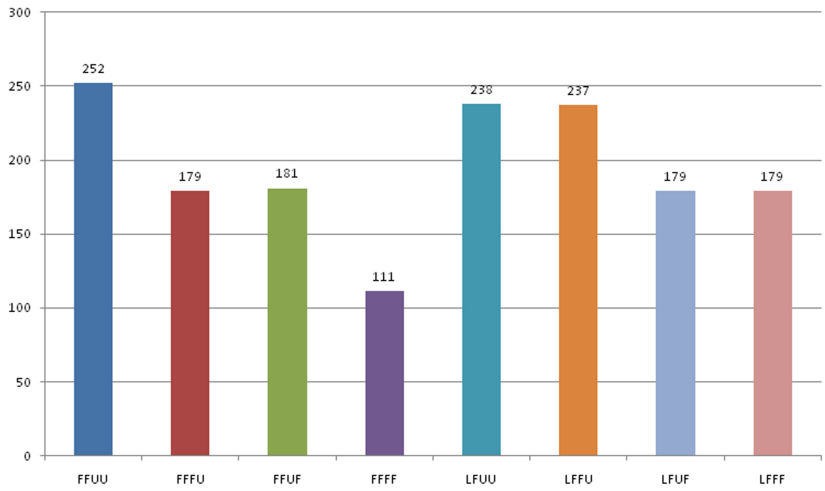

Figure III.9: Throughput with CS time of 1s and Message latency of 1ms

test. The only configuration that performs worse than this one is the Fair-Forward-Forward-Forward combination. A centralized solution would have required a critical section entry (1 sec) plus two message transmissions—Release (0.5 sec) and Reply (0.5 sec)—per access resulting in just 170 critical section entries in a 360 sec test. Every model outperforms or equals a centralized solution in this scenario.

Some settings of the Priority-Request-Reply-Release models allow for the root participant (the highest priority participant) and medium priority participants to enter a critical section twice upon Reply or Release messages. This feature causes an additional critical section entry being possible during Use-Release with a synchronization delay = 0. This result occurs when a new request occurs in cloud applications on the root during or just after the root participant is servicing a separate request.

III.4 Related Work

This section compares our work on PADME with key types of mutual exclusion solutions in networks, grids, and clouds. We begin with discussions on central token authorities and end with descriptions of distributed techniques in clouds.

A basic form of mutual exclusion is a central authority that delegates resources based

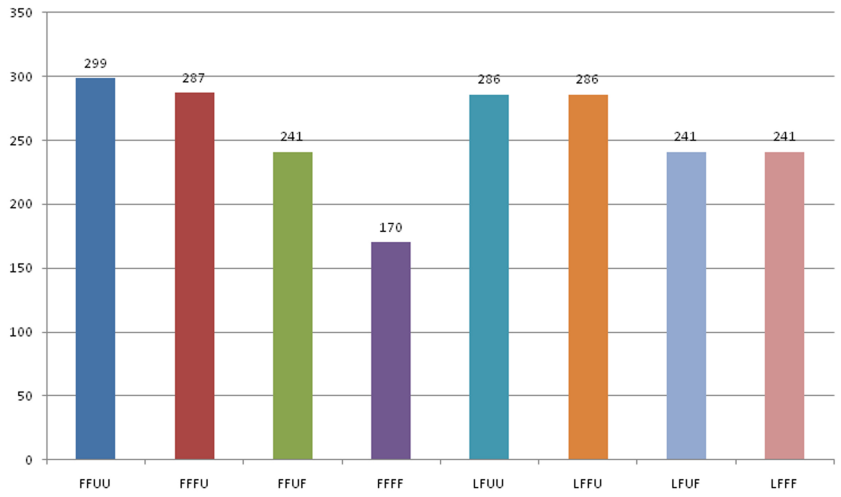

Figure III.10: Throughput with CS time of 0.5s and Message latency of 0.5ms

on priority or clock-based mechanisms. When a participant needs a shared resource, it sends a request with a priority or local timestamp to this central authority, and the central authority will queue up requests and service them according to some fairness or priority-based scheme. The Google File System (GFS) [27] uses such central authorities (called masters) and has tried to address these issues by creating a master per cell (*i.e.*, cluster). The GFS approach only masks the problems with the centralized model, however, and has a history of scaling problems [63].

The Lithium file system [32] uses a fork-consistency model with partial ordering and access tokens to synchronize writes to file meta data. These access tokens require a primary replica (centralized token generator) to control a branching system for volume ownership. Recovery of access tokens when primary replicas die requires a Byzantine view change of O(n) messages before another access token can be generated, and this can be initiated by almost anyone. In PADME, the view change should only be requested by someone along the token path. When the root node dies with a token still in it, the immediate children of the root would be in the token path and could request a view change. If there are no pending writes or reads, no change may even be necessary, and the root could resolve its fault and continue operations.

Distributed mutual exclusion algorithms have been presented throughout the past five decades and have included token and message passing paridigms. Among the more widely studied early distributed algorithms are Lamport [52] and Ricart-Agrawala [72], which both require $O(n^2)$ messages, and Singhal [78], which uses hotspots and inquire lists to localize mutual exclusion access to processes that frequently enter critical sections. These algorithms are not applicable to cloud computing, where faults are expected. Singhal's algorithm also does not differentiate between priorities since it prefers frequent accessors (which might be free or reduced-payment deployments).

Message complexity has been further reduced via quorum-based approaches. In quorum-based approaches, no central authority exists and the application programmer is responsible for creating sets of participants that must be requested and approved for the critical section to be granted. For the Maekawa quorum scheme to function [61], each set must overlap each other set or it will be possible for multiple participants to be granted a critical section at the same time. If the sets are constructed correctly, each participant has a different quorum set to get permission from, and mutual exclusion is guaranteed. The problem is automatable and involves finding the finite projection plane of N points, but suffers performance and starvation problems (potentially of high priority participant) with faults.

More recently, a distributed mutual exclusion algorithm was presented by Cao et. al. [8]. This algorithm requires consensus voting and has a message complexity of O(n) for normal operation (*i.e.*, no faults). In contrast, our PADME algorithm requires O(d) where d is tree depth—$O(log_b n)$ where b is the branching factor of the spanning tree discussed in Section III.3. The Cao et. al. algorithm also appears to require a fully connected graph to achieve consensus, and does not support configurable settings for emulating many of PADME's QoS modes (such as low response time for high priority participants).

Housni and Trehel [38] presented a grid-specialized token-based distributed mutual exclusion technique that forms logical roots in local cluster trees, which connect to other clusters via routers. Each router maintains a global request queue to try to solve priority

conflicts. Bertier et. al. [5] improved upon Housni and Trehel's work by moving the root within local clusters according to the last critical section entry. This improvement, however, could result in additional overhead from competing hot spots, where two or more processes constantly compete for critical sections. Both algorithms treat all nodes and accesses as equals and are susceptible to the problems in Singhal [78] and consequently are non-trivial to implement for a public cloud where paying customers should have differentiation. Moreover, tokens can become trapped in a cluster indefinitely.

CHAPTER IV

REAL-TIME APPLICATION DEPLOYMENT AND EXECUTION

Automated testing of distributed, service-oriented applications, particularly mobile applications, is a hard problem due to challenges testers often must deal with, such as (1) heterogeneous platforms, (2) difficulty in introducing additional resources or backups of resources that fail during testing, and (3) lack of fine-grained control over test sequencing. Depending on the testing infrastructure model, the testers may also be required to fully define all the hosts involved in testing, be forced to loosely define test execution, or may not be able to dynamically respond to application failures or changes in hosts available for testing. To address these challenges, this paper describes an approach that combines portable operating system libraries with knowledge and reasoning, which together leverage the best features of centralized and decentralized testing infrastructures to support both heterogeneous systems and distributed control.

A domain-specific modeling language is provided that simplifies, visualizes, and aggregates test settings to aid developers in constructing relevant, feature-rich tests. The models of the tests are subsequently mapped onto a portable testing framework, which uses a distributed knowledge and reasoning engine to process and disseminate testing events, successes, and failures. We validate the solution with automated testing scenarios involving service-oriented smartphone-based applications.

IV.1 Motivation

IV.1.1 Overview of KATS

Automated testing of distributed, service-oriented applications generally falls into two execution models: centralized and decentralized. In a centralized model, a testing control service sends testing commands to services installed on each host taking part in testing,

generally in a push model. In a decentralized model, there is no centralized control service, and testing occurs in a purely distributed manner. If synchronization is required, decentralized infrastructures may use network lock files, or they may require separate daemons with intimate knowledge of test sequencing or custom messaging protocols.

To understand the two models, let us consider an example where a developer would like to test a service-oriented smartphone-based mobile application which sends a message to an application server. Ideally, the smartphone should not send a message to the server until the server is ready, and suppose this should happen after 15 seconds.

In the centralized model shown in Figure IV.1, a control server sends a command to launch the application server, waits for 15 seconds, and then sends a command to the smartphone or host connected to the smartphone, which runs a script or launches a unit test already on the smartphone and sends a message to the application server. If no such script or unit test exists, a tester might emulate a message that looks like it came from a smartphone.

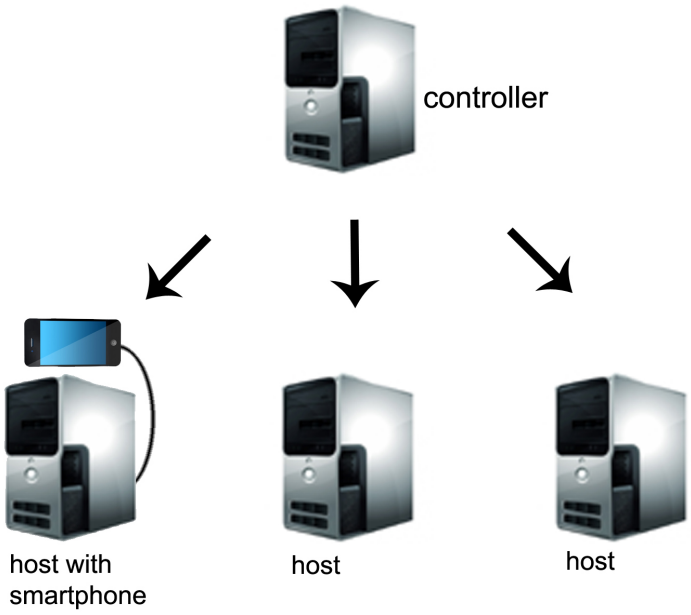

Figure IV.1: Centralized software testing

For simple tests like this example, the centralized model works, particularly when services or applications need to only do one task and then exit. Additionally, the centralized model can allow for fine-grained launch sequencing based on time and provides a single point of configuration when changing tests.

In a decentralized model (Figure IV.2, portable scripts (*e.g.*, in Perl, Python, or Ruby) are written that perform a series of steps appropriate for the host running each script without using a centralized controller. If coordination is needed, the scripts may use network lock files.

In the context of our example, the first script might launch an application server, wait for 15 seconds, and then create a lock file called "server.ready" on a network file system. A second host runs a different script that keeps checking the network file system for a file called "server.ready". When the file is created, the second script may launch another script or service to automate a smartphone sending a message.

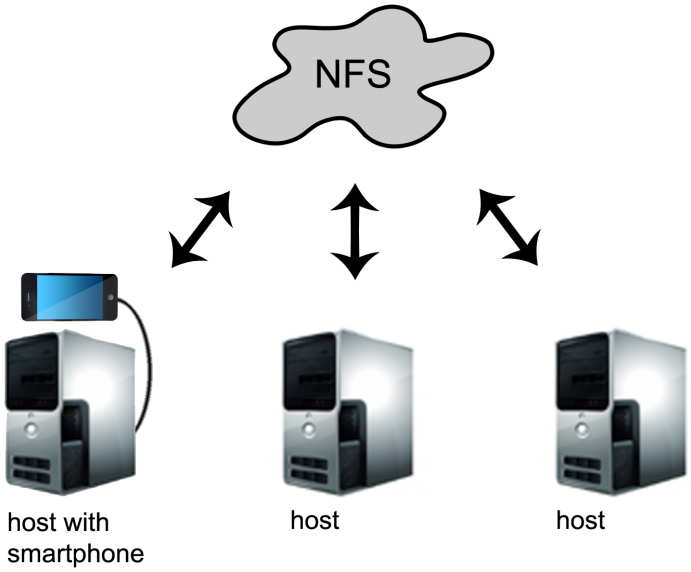

Figure IV.2: Decentralized software testing

The decentralized model is portable, elegant and intuitive and avoids any type of centralized messaging bottleneck or single point-of-failure. It also allows for moving scripts between hosts and not binding the testing infrastructure to a single host or port configuration, which may change as hosts go in and out of service. For clouds, LANs, and grids, decentralized testing infrastructures are highly appropriate, intuitive, and may be the only real available solution when host/port information are unknown before an experiment is swapped in (*e.g.*, in a cloud), without the cloud API exposing the information directly.

One of the major issues with the decentralized model is the usage of network file systems as lock files for testing coordination. Network file systems, however, were not created with lock files in mind. Our extensive experience conducting decentralized testing over local area networks reveals that network file systems, such as the widely-used Network File System (NFS), make no guarantees that local caches would ever be updated with the lock files. Moreover, in practice, timing delays could be thrown off by several minutes, even if the lock file appears at all. A custom communication model built on top of TCP or UDP could potentially solve this issue, but we do not observe this approach being used in traditional testing solutions.

While centralized models can avoid network file systems, they are tightly coupled to hosts, present a single point-of-failure (the testing controller), and the configuration of the controller can become cumbersome as tests become more complex, especially when services need to be launched in parallel with different timing delays. Additionally, most centralized models are not portable to all operating systems and instead cater to a specific architecture (*e.g.*, Windows).

In addition to these sequencing and portability concerns, the testers need to be able to codify and debug their test concerns with an intuitive interface. A cumbersome, unusable infrastructure is unlikely to benefit developers. From all these concerns, we form the following requirements of a distributed, automated testing infrastructure.

IV.1.2 Challenges for KATS

IV.1.3 Challenge 1: Support for Heterogeneous OS and networking platforms and topologies

The solution must be portable to varying operating systems and platform architectures. First-class support should exist for Windows, Linux, and MacOS, and the testing system should be flexible enough to work with most programming languages and executables.

IV.1.4 Challenge 2: Dynamic Sequencing of Tests

The solution should allow for dynamic sequencing of tests based on testing events, successes, and failures. The solution should be flexible enough to meet deverse user scenario needs, including portable wait timers between testing events.

IV.1.5 Challenge 3: Host-agnosticism

With the emergence of cloud providers, there is a real need for a testing solution that will work with dynamically allocated machines. But even outside of clouds, A good testing solution should allow for tests to be run anywhere at any time. Engineers should be able to move the test setup to different machines at a client location or into a real deployment and run the same tests that they ran elsewhere. Consequently, the solution should be host-agnostic and allow for moving test setups easily to other nodes or networks.

IV.1.6 Challenge 4: Backup nodes and Seamless Recovery

The testing system should encourage backup test participants in case of hardware or networking failure within the LAN. It should allow for seamless recovery from node failure, and facilitate the creation and integration of backup testing entities.

IV.1.7 Challenge 5: Visual and Model-based configuration

The testing solution should be easy-to-configure and present multiple ways of instrumenting the system. XML configuration files are useful for batch processing but a visualization and modeling system for creating a new test or maintaining an existing one will cut costs of training new testing engineers and help with diagnosis when a test is setup incorrectly.

IV.2 Solution

This section describes our solution approach to realizing automated testing of distributed, service-oriented applications focusing on mobile services and applications. The contributions of this solution are as follows:

1. A portable process and process group specification that addresses Requirement 1 by utilizing the Adaptive Communication Environment (ACE) [74], which has been ported to most platforms including Linux, Windows, Mac, Android, and iOS (see Section IV.2.1).

2. A portable knowledge and reasoning engine called KaRL [20] that processes testing events, successes and failures (Requirements 2, 4) and distributes knowledge across the testing network via an OpenSplice DDS transport [67]. DDS is an anonymous publish/subscribe protocol that is network architecture portable (Requirement 1) and helps us remain host-agnostic (Requirement 3) (see Section IV.2.3).

3. A middleware solution called the KaRL Automated Testing Suite (KATS) which integrates and configures KaRL and standardizes the reasoning operations into a process lifecycle that facilitates fine-grained sequencing (Requirement 2), and conditions for starting or using backup services (Requirement 4) (see Section IV.2.2).

4. Our solution provides a domain-specific modeling language (DSML) [65] for visualizing test sequences and parameters in an intuitive way (Requirement 5). From this

DSML, we generate test configurations for the decentralized agents to follow (see Section IV.2.4).

The resulting architecture is shown in Figure IV.3.

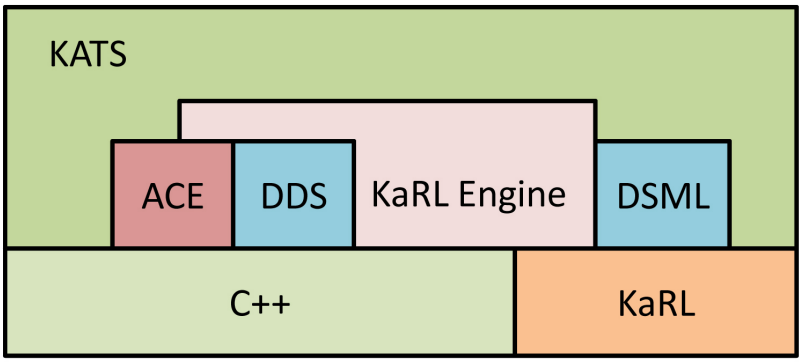

Figure IV.3: Overview of Solution Architecture

IV.2.1 Processes and Process Groups

Processes in the KATS DSML are organized into process groups which dictate how individual processes are configured and launched. Each process has over a dozen configuration settings to affect changes in OS scheduling, file I/O, test sequencing, and logging, and we accomplish OS portability by using the Adaptive Communication Environment (ACE) middleware, which supports a wide range of operating systems–including mobile phone platforms like iOS and Android. Among the more useful settings for services supported in the current version of the KATS DSML are the following:

- Real-time scheduling at highest OS priority, which ensures minimum jitter, fewer context switches, and higher priority for tested applications.
- Executable name, working directory, and command-line arguments for fine-grained control of applications that are launched and what configurations and parameters they will be started with.

- Redirecting stdin, stderr and stdout to files for emulating user input and flexible file logging. Any redirection from or to a file is buffered by the OS, which minimizes jitter and reduces the time spent in OS calls - which are expensive.

- Kill times and signals (POSIX) for errant applications. Even on non-Unix operating systems (*e.g.*, Windows), we allow for termination of applications after a specified time delay if the application does not return first. This feature is configurable by the tester.

- Test sequencing information (*e.g.*, preconditions and postconditions).

These settings allow for flexible management of each application or service launch within a process group. Process groups have additional settings indicating whether to launch the processes in the group in parallel or in sequence. Each process group also includes the settings listed above except for executable name and command-line arguments. This expressiveness for process and process group settings is part of our solution to address Requirement 1 (test settings portable across heterogenous OS platforms).

IV.2.2 Configuring Test Sequencing

Processes in the KATS infrastructure can be sequenced in two complementary ways: temporal delays and conditions. Temporal delays occur after KATS conditions have been checked. These conditions come in three supported types: barriers, preconditions, and postconditions.

The sequence of these temporal delays and conditions is shown in Figure IV.4. Barriers are groupings of processes or process groups that must rendezvous to the same point before moving onward with process launching. Barriers require setting a process id and the number of processes expected to participate in the barrier event.

After the barrier, preconditions are checked against the local context within that represent conditions that must be true before the process may be launched. A temporal delay in

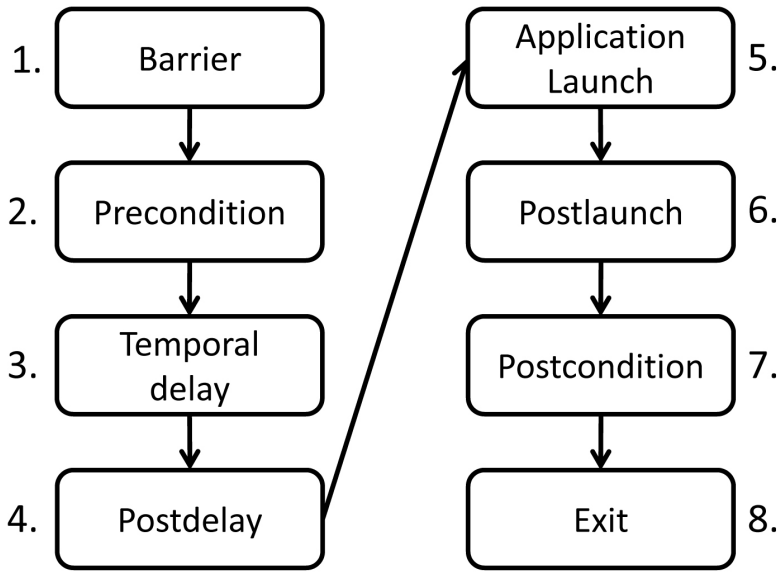

Figure IV.4: KATS Process Lifecycle

seconds is then executed if specified, followed by a postdelay condition which may be set to indicate that the delay is over and the application is ready for launch.

The application is then launched and its return code is saved into the KaRL variable *.kats.return* during the post launch phase, which allows for postconditions to feature logics that check whether or not the application succeeded or failed based on the return code (the exit value returned by the process). During the postlaunch phase, the KATS system may also be configured to kill the application after a specified time with a user-specified kill signal (on Windows, the application will simply be aborted at the time). Postconditions may also make global state modifications, which may satisfy other application or service preconditions.

Temporal delays, barriers, preconditions, postdelays, and postconditions can be specified on either a process or process group level. These lifecycle elements are the foundation of our solution to meet Requirement 2 (dynamic sequencing of tests based on test progress or failure).

IV.2.3 Augmenting Decentralized Testing with Knowledge and Reasoning

The process lifecycle outlined in Section IV.2.2 provides the infrastructure for meeting Requirement 2, but we still need a communication and reasoning mechanism for distributing knowledge and learning from testing events, successes, and failures. In this section, we look at the distributed knowledge and reasoning engine we developed called the Knowledge and Reasoning Language (KaRL) engine [20].

KaRL allows developers to manipulate knowledge via expressions in a programming logic which are evaluated against a local context (see Figure IV.5). The KaRL reasoning engine mechanisms used for evaluating logics allow for accessing or mutating knowledge variables and provide fine-grained debugging capabilities for global knowledge state in an atomic, thread-safe manner.

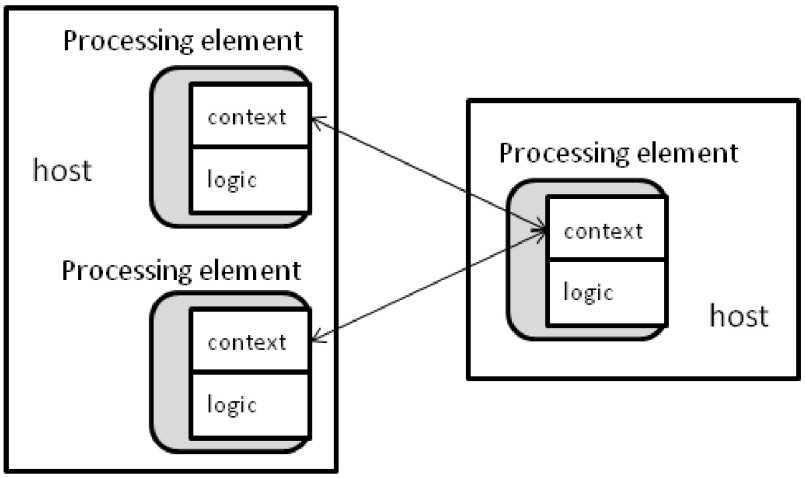

Figure IV.5: KaRL evaluates user logics against a local context, which is then synchronized with other contexts

The underlying reasoning engine mechanisms create knowledge events that are disseminated based on whether they are considered local (*i.e.*, the variable begins with a period) or global (*i.e.*, anything else) and whether or not there are interested software entities in the

network that would like information on the published knowledge domain. Once the knowledge events in KaRL are ready to be disseminated, they can be sent to the appropriate entities using an underlying transport mechanism.

The coupling of KaRL with an anonymous publish/subscribe paradigm fits our knowledge and reasoning needs for testing in the following ways. First, the communication is host-agnostic–which means we do not have to do any configuration when changing hosts or enlarging a service cloud (Requirement 3). Second, the KaRL reasoning service is robust to application or node failures and allows us to seamlessly integrate backup services to take over in testing operations, if necessary (Requirement 4). Third, we wanted an engine that could change preconditions, postconditions, and barriers on the fly, at runtime, and most other knowledge and reasoning engines do not allow this without resetting all knowledge periodically.

The KaRL engine and its underlying transports are seamlessly incorporated into the KATS testing framework with very little interaction or configuration required by the tester. The only time a tester manually works with KaRL is when they are creating preconditions and postconditions. Though barriers use KaRL, the user only has to specify a barrier name, an id for this process, and the number of processes participating in the barrier. KATS builds the KaRL logic from this information provided by the user in the visual DSML of their testing procedures.

We now briefly outline the KaRL solution to encoding testing information into conditions for the expressed purpose of distributed reasoning.

KaRL is a knowledge representation and predicate calculus engine built for real-time, continuous systems. KaRL features first class support for multi-modal knowledge values, and each knowledge variable may be one of 2^{64} possible values. KATS conditions (*i.e.*, barriers, preconditions, and postconditions) are KaRL expressions that take the following form $Predicate => Result$, which can be read as "if the Predicate is true then evaluate Result." By default, $Predicate$ is true in the KaRL language, and KaRL expressions may

be chained together with the *Both* operator ;. This operator ; simply means that both the left and right expression should be evaluated.

For an example of a KaRL expression that can be used in a KATS condition, consider the following possible postcondition: $server.finished => (no.backup => all.tests.stop = 1)$. This KaRL expression will check the variable $server.finished$ and if it is not zero (*i.e.*, true), then we will check if there is a backup server available. If $no.backup$ has been set to anything but zero, we then indicate that all tests need to stop by setting $all.tests.stop$ to 1.

KaRL supports most multi-modal logic operations and conditionals including $==$, !=, &&, and $||$ and mutations on knowledge, including $+$, $-$, $*$, $/$, $\%$, where the latter three are multiplication, division, and remainder or modulus. For a more comprehensive listing, please see the project site. See Section IV.3 for specific, simple examples of how KaRL expressions offer powerful sequencing capabilities to a tester.

IV.2.4 Domain-Specific Modeling

KATS provides a modeling front-end (DSML) to overcome mundane and erroneous tasks handled by the testers (Requirement 5). The KATS DSML comprises a UML-based meta-model developed using the open-source and freely available Generic Modeling Environment (GME) [53] to codify the process lifecycle and all reasoning operations in a highly configurable way. This DSML has four key components that users can create and configure: processes, process groups, hosts, and barriers. The DSML in turn generates an XML file that is interpreted by KATS tools. If testers do not wish to use the visual DSML, they can encode simple XML files according to the KATS schema. We show example testing XML files in Section IV.3.

IV.3 Demonstrating KATS on Case Studies

In this section we present experimental scenarios that highlight typical distributed testing configurations. These configurations feature service-oriented applications using smartphones connected to hosts because these scenarios represent the types of experiments we are currently working on for our sponsored R&D projects. To concretize our case studies, we assume that the smartphones used in each of these examples are Android phones that allow for instrumentation of application launches through the Android Debug Bridge [10].

IV.3.1 Sequenced Smartphone Scenario

In our first test case, five smartphones are connected to a server and testers are interested in determining if the server will receive the five smartphone requests in the correct order with no arbitrary temporal delays between them. Specifically, the testers want to start the gateway service, wait for 15 seconds, and then send a message from Phone1, then Phone2, Phone3, etc. We assume that all smartphones are connected by USB to the same host and the server resides on its own host.

In KATS, a model is created that creates a barrier named `SimplePhoneSequence`, which has six processes and IDs from 0..5 inclusive and arbitrarily assigned. The five phone processes can be setup in one of two ways and both require an initial step of launching the server with no preconditions or postconditions. Additionally, we can sequence the phones in two different ways: (1) no pre- or post-conditions and (2) with pre- and post-conditions.

The first option requires that KATS process group launch all the phone instrumentation scripts in parallel (a single element added to the process group XML configuration file). This configuration is by far the easiest to setup, and requires only setting command line arguments for the phone scripts (for the user-defined application logic) and possibly redirecting stderr and stdout to files for logging.

The second configuration possibility requires that Phones 1..5 have preconditions and/or postconditions set according to settings shown in Table IV.1. Phone2 depends on Phone1

```
<group>
    <setup>
        <parallel />
        <domain>case_study_1</domain>
    </setup>
    <process>
        <domain>case_study_1</domain>
        <barrier name="barrier" />
        <id>0</id>
        <processes>6</processes>
        <executable>ServerApp</executable>
        <commandline>--port 55555</commandline>
    </process>
    <process>
        <domain>case_study_1</domain>
        <barrier name="barrier" />
        <id>1</id>
        <processes>6</processes>
        <delay>15</delay>
        <postcondition>phone1.done=1</postcondition>
        <executable>PhoneMessage</executable>
        <commandline>--server 55555 --phoneid 1</commandline>
    </process>
    <process>
        <domain>case_study_1</domain>
        <barrier name="barrier" />
        <id>2</id>
        <processes>6</processes>
        <precondition>phone1.done</precondition>
        <postcondition>phone2.done=1</postcondition>
        <executable>PhoneMessage</executable>
        <commandline>--server 55555 --phoneid 2</commandline>
    </process>
... phones 3-4 are omitted for brevity ...
    <process>
        <domain>case_study_1</domain>
        <barrier name="barrier" />
        <id>5</id>
        <processes>6</processes>
        <precondition>phone4.done</precondition>
        <executable>PhoneMessage</executable>
        <commandline>--server 55555 --phoneid 5</commandline>
    </process>
</group>
```

Figure IV.6: Generated XML from DSML for the Sequenced Smartphone Scenario

completing, Phone3 depends on Phone2, and so on. With this intuitive mapping of preconditions and postconditions, we have accomplished distributed testing execution with no operating system sleep calls and no centralized controller. The XML generated from the visual KATS DSML is shown in Figure IV.6.

Table IV.1: Preconditions, Postconditions, and Temporal Delay in the Sequenced Smartphone Scenario

Name	Pre	Post	Delay
Phone1		Phone1.done=1	15
Phone2	Phone1.done	Phone2.done=1	
Phone3	Phone2.done	Phone3.done=1	
Phone4	Phone3.done	Phone4.done=1	
Phone5	Phone4.done		

IV.3.2 Backup Service for Smartphones Scenario

The testers have acquired a backup server host, and they want to create tests that have phones switch over to the backup server if the main server starts dropping connections. To complicate the scenario, testers bring in three more machines and attach the five phones to random machines. The three phones are blasting clients that send 100 messages a second to the gateway service. If any of these scripts fail to send their message, they exit with a return code of 1. Otherwise, they return a 0 for success.

The two other phones start only if the blasting clients fail with a return code of 1. No phone should try to contact a server until 15 seconds after the gateway service is ready, and all phones should be killed after 3 minutes regardless of whether they succeed in contacting the main server or not. We assume the backup gateway service is launched along with or sometime after the main gateway service.

With KATS, the testers can produce this more complicated sequencing scenario with minimal configuration of preconditions and postconditions shown in Table IV.2.

Table IV.2: Settings for Phones in the Backup Scenario

Name	Pre	Post	Kill
Phone1		.kats.return => backup=1	180
Phone2		.kats.return => backup=1	180
Phone3		.kats.return => backup=1	180
Phone4	backup		180
Phone5	backup		180

".kats.return" is the return value of the launched process (in this case one of the blasting clients). The postconditions here indicate that "if my return value is non-zero, set the global variable *backup* to 1." The other two phones have preconditions that say "if the global variable *backup* is non-zero, launch my process." In this way, we facilitate the testing scenario created by the testers, and we can move these process groups wherever we need to in the network or cloud. The XML generated from the KATS visual model is shown in Figure IV.7.

To implement this same solution with a centralized model would require a custom controller with push and pull capability and may require multiple versions of testing services customized to a particular operating system. In KATS, we get this cross-platform functionality included.

IV.4 Results

In this section we briefly describe performance information regarding our testing infrastructure KATS. Though very few testing infrastructures are concerned with latency and performance, decentralized service-oriented software can often have a high overhead especially in group communications like barriers. Because our solution is novel in how it disseminates data to a testing infrastructure, it is important to investigate whether KATS features come at a high performance cost.

We divide these performance metrics into two sections: condition latency (the time it takes for a precondition or postcondition to reach other interested hosts on a local area

network) and barrier latency (the time it takes for a group of distributed applications to barrier together across a network).

IV.4.1 Experimental Testbed Setup

Unless specified otherwise, all experiments were conducted on five IBM blades with dual core Intel Xeon processors at 2.8 GHZ each and 1 GB of RAM running Fedora Core 10 Linux. Each blade is connected across gigabit ethernet and networked together with a switch. The code was compiled with g++ with level 3 optimization, and each test featured a real-time class to elevate OS scheduling priority to minimize jitter during the test runs. This testbed is similar to the one we are using for sponsored research.

IV.4.2 Measuring Condition Latency

Condition latency is the time it takes for a precondition or postcondition of one application to reach another application across the network. Recall that in testing, a precondition is a condition that must be true before an application can launch. Consequently, this latency between the setting of a variable in one part of the network and the evaluation of this data in interested testing entities is an important metric to have.

Setup. Two applications (App 1 and App 2) were launched on two separate hosts. Each application was configured with conditions indicated in Table IV.3, and then the applications were launched repeatedly. These conditions create a roundtrip latency time on each host. We take this roundtrip latency time and divide by two to get our averaged latency numbers.

Table IV.3: Conditions used for latency tests

App 1	App 2
P0 == P1 => ++P0	P0 != P1 => P1 = P0

Results. The results of our condition latency tests are shown in Table IV.4. These results represent average latency for a condition to be sent and processed from one host to the next. Each column represents the number of round trips completed. The *Ping* row is the latency reported between the nodes via the Linux ping utility. *Dissem* is the average dissemination latency of KATS conditions.

Table IV.4: Condition Results

	5k	25k	50k	100k	500k
Ping	114 us	114 us	114 us	114 us	114 us
Dissem	650 us	440 us	437 us	315 us	317 us

Analysis. The KaRL interpreter and OpenSplice DDS transport provide very low latency for the delivery of KATS conditions. On our network, this allows for sub-millisecond accuracy when a tested application's dependencies are met. This will vary depending on network latency in the developer testbed. The lower the latency, the more fine-grained the test sequencing can be, and the more expressive testers can be with their distributed testing scenarios.

IV.4.3 Measuring Barrier Latency

Barrier latency is the time it takes for a group of KATS processes to execute a barrier operation before launching the user-defined applications. Barrier operations are far more expensive than setting a condition, but barriers are useful when testers need all testing participants to be up and ready. From our experience, all distributed testing scenarios require a barrier, unless the individual tests do not interact with other testing participants. Consequently, the barrier time is an important metric because it is the overhead that must be performed before each distributed test can start.

Setup. Five hosts with two CPUs apiece launch applications (simple sleep statements of five seconds) after reaching barriers under different networked process stress loads. High

resolution timers are used to measure the time it takes to complete a barrier. The tests are repeated 10 times and an average time is reported.

Each host in these tests launched its own barrier set of 2 to 5 applications. We only have two CPUs per host, so as the number of applications increases, no further speed increase is possible (it is pure overhead from context switching past 2 application launches). We will discuss why we did not connect the hosts in a LAN-wide barrier in the Analysis section. Each barrier grouping was executed 10 times per host and averaged. We complement the average latency numbers with minimum and maximum observed barrier time.

Results. The results of our barrier latency tests are shown in Table IV.5. These results represent latencies experienced on a single host launching applications >= the number of CPUs on the host. The minimum latency was the smallest time spent in a barrier in the 50 tests. The maximum was the largest, and the average was also for the 50 tests.

Table IV.5: Barrier Latency Results

Apps per host	Min	Avg	Max
2	0.8 ms	2.0 ms	2.7 ms
3	1.7 ms	9.4 ms	21.5 ms
4	2.4 ms	17.2 ms	17.2 ms
5	3.0 ms	17.3 ms	35.2 ms

Analysis. We use intrahost communication here to highlight the speed and precision that the KATS system is able to achieve with distributed processes. Within a single host, as is the case here, the cause of deviation between the min and max is caused by context switching. Since our experimental machine had two cores, the deviation is very low on the two apps test. Running this same series on a real-time kernel can reduce the deviation, but in a networked test, the benefits will be negligible.

We have found that barrier latencies for networked processes across many hosts are

hugely dependent on when the distributed tests are started on each host, far more than anything dictated by the latency of the underlying KaRL system in disseminating knowledge shown in Section IV.4.2.

As an example, consider the case where KATS tests will be started distributedly via cron jobs on each host at 5:00 p.m., according to the system clock which is synchronized with the network time protocol (NTP). Even under the best of circumstances, this protocol guarantees accuracy between the networked clocks in the hundreds of milliseconds – orders of magnitude larger than how long barriers, preconditions, or postconditions execute in KATS or its KaRL infrastructure outside of situations with heavy context switching between the KATS processes and the OpenSplice DDS daemon.

This may seem like a problem, but this is exactly the reason barriers are recommended among test participants. Barriers ensure that regardless of time synchronization or order of test launches in the network, the exact sequence of testing events will be launched precisely as specified by the testers.

IV.5 Related Work

A vast majority of networked testing infrastructures [14, 17, 43, 82] utilize a centralized controller. This centralized paradigm extends to most deployment [54, 62] and testing instrumentation technologies [30, 54, 85, 86] as well.

Though these related tool suites do not support truly distributed test scheduling and automation, they often do include useful instrumentation tools [85] that allow for real time performance analysis – which we do not include in our tool suite at the moment. Other instrumentation systems place themselves in between applications and the underlying middleware or networking layer [84]. Our solution relies on distributed blackbox testing without any requirement for hooks into the operating system or services to be tested.

Model-driven testing systems exist for centralized or non-distributed systems [7, 25,

29, 44, 75]. Other systems use a generic programming approach [47] for functional testing. Both these types of systems demand intimate knowledge of internal behaviors of the application or service being tested – often requiring code insertion into the application or modeling/reading of source code.

Recent work using domain-specific modeling in testing with mobile phones appears in the literature [73], but there are many key differences between the two approaches. We do not require a hardware modification to directly send key events to the phone and instead rely on the Android Debug Bridge which is part of the Android SDK. Additionally, we offer automated and barriered test cases in contrast to interactive sessions between users and the phone in [73].

The KATS system also acts as a deployment infrastructure for launching applications. Related work in this area would include DAnCE [16, 68], ADAGE [50], and RESERVOIR [11]. None of these deployment systems provide the fine-grained sequencing support that KATS does. DAnCE does provide a step-locked phase mechanism to provide consistent deployments, but it is also imprecise and component deployment latencies can be in the hundreds of seconds. Quema [71] offered a deployment solution that provides activations for component deployments, but it requires a hierarchical deployment structure and does not provide a fine-grained mechanism that can key off application exits or targeted, dynamic logics.

We are unaware of prior work using distributed knowledge and reasoning in automating networked testing, nor have we seen prior art using an anonymous publish/subscribe (pub/sub) paradigm to coordinate and sequence test execution among generic networked processes or programs. Scripted distributed testing using pub/sub exists [90] but it requires a networked file system and customized scripting to sequence tests and suffers from all of the NFS problems mentioned in decentralized testing in Section IV.1.

```
<process>
    <domain>case_study_2</domain>
    <barrier name="barrier" />
    <id>0</id>
    <processes>7</processes>
    <executable>ServerApp</executable>
    <commandline>--port 55555</commandline>
    <kill>
        <time>180</time>
    </kill>
</process>
<process>
    <domain>case_study_2</domain>
    <barrier name="barrier" />
    <id>6</id>
    <processes>7</processes>
    <executable>BackupServerApp</executable>
    <commandline>--port 55556</commandline>
    <kill>
        <time>180</time>
    </kill>
</process>
<process>
    <domain>case_study_1</domain>
    <barrier name="barrier" />
    <id>1</id>
    <processes>7</processes>
    <delay>15</delay>
    <postcondition>.kats.return => backup=1</postcondition>
    <executable>PhoneMessage</executable>
    <commandline>--server 55555 --messages 100</commandline>
    <kill>
        <time>180</time>
    </kill>
</process>
... logic for phones 2 and 3 is similar to phone 1 ...
<process>
    <domain>case_study_2</domain>
    <barrier name="barrier" />
    <id>4</id>
    <processes>7</processes>
    <precondition>backup</precondition>
    <executable>PhoneMessage</executable>
    <commandline>--server 55556 --phoneid 4</commandline>
    <kill>
        <time>180</time>
    </kill>
</process>
... logic for phone 5 is similar to phone 4 ...
```

Figure IV.7: Generated XML from DSML for the Complex Smartphone Scenario

CHAPTER V

EMERGENT REDEPLOYMENT OF REAL-TIME SYSTEMS

Optimizing the deployment of participants in enterprise distributed real-time and embedded (DRE) systems is essential to enforce end-to-end quality-of-service (QoS) properties that affect power usage, execution time, and overall system cost within a LAN or cloud. In previous work, we developed a heuristic for approximating the component placement problem, where a user must optimally place components within a network of processes and do so optimally. Among the remaining challenges remaining in our deployment optimization research were 1) creating new techniques for approximating the optimal redeployment, 2) aggregating and disseminating latency information quickly (our first implementation could take minutes for this operation with only a few dozen participants), and 3) investigating the enforcement of side effects.

This chapter provides three contributions to the state-of-the-art in optimization of enterprise DRE application deployments within a cloud. First, we discuss two new techniques and improvements to our first algorithm for approximating the subgraph isomorphic problem, a NP complete problem that tries to find a user graph G in a networked environment E (and in our case, tries to approximate the optimal one). Second, we outline practices we used to reduce message complexity of latency aggregation, summation, and dissemination tailored to our developed algorithms that reduce computation and messaging time to milliseconds for thousands of participants. Third, we highlight the issues users will face with using such a system to redeploy participants with side effects (*e.g.*, databases).

V.1 Motivation

Enterprise distributed real-time and embedded (DRE) systems are mission-critical applications executed in networked processes across heterogeneous architectures under stringent timing requirements and scarce resources [4]. Though enterprise DRE systems were originally associated with avionics, manufacturing, and defense applications, they increasingly focus on any distributed application where information must be delivered according to stringent quality-of-service needs despite scarce resources [46, 48, 79]. This paper examines continuous optimization of DRE application deployments according to a user-provided workflow, which can be mapped to the subgraph isomorphism problem (*i.e.*, find an optimal graph G if it exists in the graph E) [26].

Motivating DRE application. The application from the domain of enterprise DRE systems that motivates the work described in this paper is shown in Figure V.1. This figure

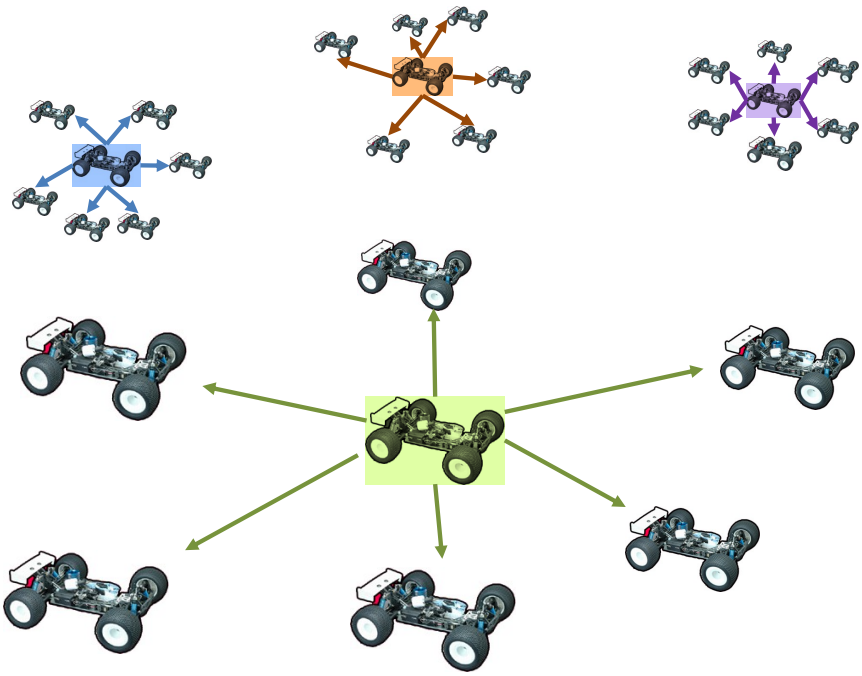

Figure V.1: Example Deployment Workflow of Four Drones Broadcasting In and Collecting from Dynamic Groupings

depicts a disaster recovery scenario where thousands of ground-based drones have been deployed to search for survivors and gas leaks in an earthquake-ravaged metropolitan area. Due to the destruction, controllers of the drones are restricted to satellite connections and the bandwidth available over this limited network resource is sufficient for only a handful of dedicated sessions. Controllers can therefore only maintain communication with a small subset of the drones, called *collector* drones. Other drones may do other tasks, such as weather collection, video recording, or participating in rescue operations.

An application workflow (called a *DRE application* in this paper) is specified to indicate which special drones should serve as collection points for the important sensor readings, images, and audio from the disaster zone, as well as the communication point to human-based controllers. As drones move around the area, the optimal deployment of these collector drones may become outdated. A redeployment of the specialized collection and controller communication logic may therefore be necessary to ensure the special drones are in range of their group within the workflow.

This redeployment time is pure overhead and the associated computation time competes with the CPU and memory resources that the human controller needs to view important data. If the drones remain computation-bound for too long and lock out their controllers from viewing information or issuing commands, survivors may be missed, drones may crash into buildings or other obstacles, and lives and money may be lost. Since the drones are battery-powered any unnecessary computation or communication must be minimized to conserve power. For these reasons, the time required for calculating the redeployment should be kept to a minimum—preferably a handful of seconds or less.

In summary this motivating DRE application has the following requirements:

V.1.1 Challenges for Emergent Redeployment

V.1.2 Challenge 1: Flexible dataflow specification within deployments

Most deployment architectures require the user to specify which participants are deployed to specific hosts, but they do not include information on how participants interact with each other. These latter details are important because they can inform deployment middleware and tools about which participants are likely to be bottlenecks in the deployment should they be placed on faulty or weak hardware.

V.1.3 Challenge 2: Algorithms and heuristics for approximating optimal deployments

Deployments may become worse over time due to faulty computing or networking hardware, or high utilization of an important switch, router, or host. Using latency information between the hosts, we should be able to analyze the current state of the distributed application and inform the middleware when a redeployment is necessary. Because we expect these applications to be deployed on sensors operating in mhz or less ranges in CPU, the techniques will need to be fast and efficient in order to reduce battery consumption and time required to detect a redeployment condition.

V.1.4 Challenge 3: Efficient latency and consensus collection

Communication is one of the most expensive operations (in terms of power consumption and time) that can be performed by a computer. Consequently, informing each participant of changes in latency and reaching a consensus about redeployment conditions in the set of networked participants should be done as quickly and efficiently as possible.

V.2 Approximation Techniques in MADARA

This section presents the dataflow definition format (Challenge 1) and the heuristics and genetic algorithms (Challenge 2) that we developed for MADARA to approximate

Table V.1: Workflow description for four special drones, each collecting from a quarter of the drone population

0	→	[size/4)
size/4	→	[size/4, size/2)
size/2	→	[size/2, 3*size/4)
3*size/4	→	[3*size/4, size)

an optimal enterprise DRE application deployment under differing constraints. Multiple solutions were developed due to memory constraints imposed by differing contexts where the solutions are deployed. These solutions are complementary and can be chained together to produce seeds and candidates for other genetic algorithms or heuristics.

The following sections describe these algorithms and heuristics, focusing on which scenarios they work well in and which they do not.

V.2.1 Degrees in Graphs

This paper treats constraint satisfaction problems (CSPs) and the subgraph isomorphic problem from a graph perspective. We encode a deployment into a graph and use degree information to inform our approximation process. The degree of a node in a graph is the number of connections coming out of the node, *i.e.*, it is essentially a connectivity metric. This concept of degree is derived from graph algorithms, as well as distributed and parallel computing.

The degree of a graph is relevant to our work on MADARA because we seek solutions that minimize the latency or improve the overall utility of the connections between nodes in a DRE application workflow. The node with the highest degree has the most impact on this overall metric. It is therefore often a major bottleneck in a DRE application.

To concretize how a degree is imparted from a deployment description (Challenge 1), consider Table V.1 which shows an actual deployment file that may be used with our solution.

From this deployment description, we can visualize the application workflow that the

user has requested in Figure V.1. There are four special drones, and each are servicing a large portion of the underlying drone network. If the size is set to 12, the logical drone 0 is servicing drones 0-2. Drone 4 services 3-6, drone 7 takes care of 7-9, and drone 10 handles information to and from 10-12. Through this flexible deployment specification interface, we address the challenge of encoding dataflow information for thousands of participants into a deployment specification.

Figure V.2 visualizes what a degree in a graph is by labeling the high degree nodes in a user-provided DRE application workflow. The node with a degree of seven has seven

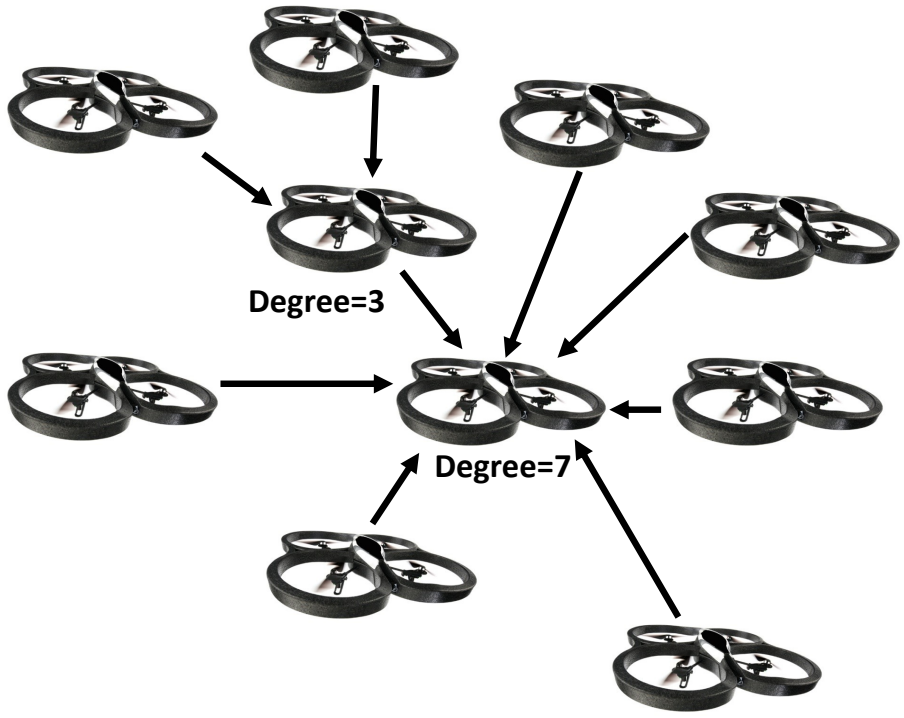

Figure V.2: Degrees in a User-provided DRE Application Workflow

directional edges coming in or out of the node. A degree with three signifies that the node has a connectivity of three. Though this paper focuses on using degree information for the motivating DRE application in Section V.1, our solution techniques are relevant

to approximating component placement, optimal resource monitoring, routing, and other problems involving connected graphs.

V.2.2 Preparing the Data

Before presenting our heuristics, genetic algorithms, and their hybrids, we first summarize the process of data collection and preparation that is needed for these solution approaches. Our heuristics discussed later depend on a notion of *utility*, so the first phase of data preparation collects the latencies (the utilities) and aggregates them according to deployment degrees. After data has been collected, the prepare algorithm (shown in Algorithm 1) is called to aggregate, sort, and process the latency and utility information with respect to the DRE application graph. After the data is ready, it is disseminated to interested

Algorithm 1 Prepare

Require: DRE application workflow nodes are sorted by descending degree
Require: Gather latencies via network and place in an array or a double array

```
1:  if Storing all latencies involving all processes then
2:      for all source ∈ latencies do
3:          sort_ascending(latencies[source]) ;
4:          for all degree ∈ DRE application workflow do
5:              utilities[degree][source] ← ∑_{j=1}^{degree} latencies[i][j]
6:          end for
7:      end for
8:  else if Storing only latencies involving this process then
9:      sort_ascending(latencies)
10:     for all degree ∈ DRE application workflow do
11:         utilities[degree] ← ∑_{j=1}^{degree} latencies[j]
12:     end for
13: end if
14: sort_ascending(utilities)
```

parties in the network, *e.g.*, the drones or a special collector of this type of information—perhaps a command-and-control output.

This preparation routine should be performed before the heuristics are called. Algorithm 1 need not be called continually throughout the life of the deployment infrastructure. It should be called at some point, however, before calling the heuristics to avoid approximating a solution with stale data.

Before executing data collection or Algorithm 1, developers must determine whether or not the network of drones should be fully informed with latency information or if they should only send latency degree-based utility information (the aggregates of their latency tables). A third option is to have all drones send latency information to a single, powerful drone that is fully informed and have it perform all of the data aggregation, approximating, and any other function. Such an option avoids over-broadcasting data, but will also result in a single point-of-failure in the enterprise DRE system, so failover drones or replication mechanisms should be utilized to handle the case where the single, powerful drone becomes unresponsive or destroyed.

With fully-informed collection, each drone must send O(N) latency information and O(M) utility information to each latency/utility collection point, where N is the size of the network and M is the number of degrees in the deployment workflow. During the preparation phase, the drone or computer that is preparing the data performs N sorts of N elements (O(N^2 log N)) and also a summation of latencies (D O (N^2)), where D is the number of degrees available in the deployment.

Each step is fully parallelizable because each operation has no side effects. This work can be performed on each drone to reduce the execution complexity to O(N log N) and O(DN) for sorting and summation, respectively, which is what Algorithm 1 does on lines 8-13. The computation differences between these approaches are highlighted in Section V.4.

This preparation time decision dictates the runtime of the prepare algorithm, but does not necessarily affect whether or not the drones are fully informed. If developers desire a fully informed drone network (*e.g.*, to use Algorithm 2), each candidate can sort its latencies, perform summations on the degrees necessary for the deployment, and transmit the

sorted latency list and aggregation information to all other drones (or the current collection drones that then transmit this information to their local group). In this way, each drone transmits O(N) latencies and O(M) data entries, where N is the number of drones and M is the number of different degrees in the provided deployment workflow.

Algorithm 2 CID Heuristic

Require: Call Algorithm 1

```
1:  for all node ∈ DRE application workflow do
2:      if degree (node) > 0 then
3:          solution[node] ← best_candidate (utilities[degree(node)])
4:      end if
5:  end for
6:  for all node ∈ DRE application workflow do
7:      if degree (node) > 0 then
8:          for neighbor ∈ connections(DRE application workflow, node) ∧ neighbor ∉
            solved(solution) do
9:              solution[neighbor] ← best_candidate (latencies[node])
10:         end for
11:     end if
12: end for
13: for all node ∈ DRE application workflow ∧ node ∉ solved(solution) do
14:     solution[node] ← best_candidate (utilities[size])
15: end for
```

There is a separate preparation step of sorting the user-provided DRE application deployment graph by degree, which helps Algorithms 2, 3, and 4 pick deployment nodes to solve in a more intelligent order. This step can be performed offline, however, during DRE application modeling phases. Even when done online, the preparation step of sorting the DRE application workflow by degree takes just nanoseconds to a few microseconds for 10,000+ node workflows. It is therefore a negligible portion of the time needed to perform the approximation (the project site at `madara.googlecode.com` contains complete code examples).

V.2.3 Degree-based Heuristics in MADARA

Two heuristics are discussed in this section, each targeting a different context of the motivating DRE application. The Comparison- based Iteration by Degree (CID) Heuristic (shown in Algorithm 2) is useful for seeding genetic algorithms when the drone has enough memory to hold latency information of all other drones ($O(N^2)$ space requirement), which can become hundreds of megabytes when thousands of drones or processes are involved.

The CID Heuristic shown in Algorithm 2 begins by iterating over the deployment and placing candidates based on lowest latency for the degree. We place our lowest total latency candidates on the nodes with the highest connectivity (lines 1-5) and then iteratively fill in their closest neighbors when possible on lines 6-12 (*i.e.*, when it does not conflict with other high degreed nodes in the DRE application workflow). The final phase of the CID (lines 13-15) deals with nodes that are not connected to the rest of the DRE application workflow, *e.g.*, for worker drones that do not communicate with the drone collector and serve as sentries, data analyzers, or passive entities whose results can be processed or collected offline (non-mission critical).

A variant of the CID heuristic we developed called the Blind CID heuristic (Algorithm 3) is useful for deployments where drones do not have as much memory ($O(N)$ space instead of $O(N^2)$). The drawback is that the Blind CID heuristic is a less informed approximation of the solution than the CID Heuristic.

The key difference between the CID heuristic and the Blind CID heuristic (Algorithm 3) is that the CID heuristic uses the fine-grained latency information from all drones in the network, whereas the Blind CID heuristic only uses aggregation of this knowledge. The Blind CID heuristic does use deployment information in the workflow to prioritize which node of the workflow to approximate next. It always selects from the best total latency value (essentially the aggregate of a full broadcast from the node), however, rather than the aggregate of best latencies from this node for the degree. The benefit of this heuristic is that the drones need not send their individual latency values to other drones that must make

Algorithm 3 Blind CID

Require: Call Algorithm 1

```
1: for all node ∈ DRE application workflow do
2:     if degree (node) > 0 then
3:         solution[node] ← best_candidate (utilities[degree])
4:     end if
5: end for
6: for all node ∈ DRE application workflow do
7:     if degree (node) > 0 then
8:         for neighbor ∈ connections(deployment, node) ∧ neighbor ∉ solved(solution) do
9:             solution[neighbor] ← best_candidate (utilities[size])
10:        end for
11:    end if
12: end for
13: for all node ∈ DRE application workflow ∧ node ∉ solved(solution) do
14:    solution[node] ← best_candidate (utilities[size])
15: end for
```

redeployment decisions (O(N) total message complexity unlike the other algorithms). Each node using Algorithm 3 alone has a message complexity of O(1), a message containing an aggregrate latency value for a full broadcast from the node.

V.2.4 Genetic Algorithms in MADARA

Not all DRE application workflows can be solved optimally by the heuristics described in Section V.2.3. The CID and BCID heuristics work well for disjointed subgraphs that have few 2+ degreed nodes in the workflow, but are not as ideal for highly-interdependent subgraphs, such as multi-leveled hierarchical trees. To complement the heuristics discussed in Section V.2.3, we developed two genetic algorithms to hone the approximated solution before deciding if a redeployment is necessary for the special drones. Only one of these genetic algorithms (Algorithm 5) is guided with degree information. Algorithm 4 uses no degree information to mutate solutions and instead uses pure randomness when selecting solution chromosomes to mutate.

Algorithm 4 Blind GA

Require: Call Algorithm 1

```
mutations ← min + rand() % (max - min)
new ← solution
orig_utility ← utility(new)
for i → mutations do
    c1 ← rand() % size
    c2 ← rand() % size
    while c1 ≡ c2 do
        c2 ← rand() % size
    end while
    swap(solution[c1], solution[c2])
end for
if utility(new) < orig_utility then
    return new
else
    return solution
end if
```

Algorithm 5 Guided GA

Require: Call Algorithm 1

```
mutations ← min + rand() % (max - min)
new ← solution
orig_utility ← utility(new)
for i → mutations do
    if rand() % 5 < 4 then
        c1 ← random_degreed_node (DRE application workflow)
        c2 ← location(solution[good_candidate(utilities)])
        while c1 ≡ c2 do
            c2 ← location(solution[good_candidate(utilities)])
        end while
    else
        c1 ← rand() % size
        c2 ← rand() % size
        while c1 ≡ c2 do
            c2 ← rand() % size
        end while
    end if
end for
if utility(new) < orig_utility then
    return new
else
    return solution
end if
```

Both algorithms select chromosomes (nodes/drones) of the proposed solution (the approximated deployment) to mutate and then perform a random number of mutations before returning the best solution (either the original or the improved solution). While the Guided GA does converge much more quickly, the randomness inherent in the Blind GA can be better for the hybrid approaches we discuss in Section V.2.5 below.

The Guided GA (GGA) takes longer per iteration due to its added intelligence. After some initial timing, we determined that the maximum mutations available to the GGA implementation in a second might be 500 per solution, while the less-informed BGA solution could manage over 2,000 in the same time period. We analyze the effectiveness of both solutions in Section V.4.

V.2.5 Hybrid Approaches in MADARA

A guided genetic algorithm need not be directly codified with degree information, as we did with Algorithm 5. We can also seed the GAs with heuristic results to help local searches converge much faster than they might have otherwise. This paper therefore combines the two heuristics in Section V.2.3 with each genetic algorithm presented in Section V.2.4 to produce four methodologies: (1) CID with Blind Genetic Algorithm (CID-BGA), (2) CID with Guided Genetic Algorithm (CID-GGA), (3) Blind CID with Blind Genetic Algorithm (BCID-BGA), and (4) Blind CID with Guided Genetic Algorithm (BCID-GGA). Implementations involving seeding genetic algorithms with candidates from both heuristics are not covered in this paper, but are a focus of future work.

V.3 Supporting Latency Collection and Voting in MADARA

This section presents the techniques utilized to efficiently collect latencies, aggregate and disseminate them to voting participants, and conduct votes concerning redeployment (Challenge 3). The collection, aggregation and dissemination methodologies are meant to complement the approximation techniques outlined in Sectionr̃efsec:redeploy:solution:approx

by providing these algorithms and heuristics with latency and degree-based summations that the CID heuristic and others require for operation. The voting mechanisms are used to reach a consensus among voting participants about whether or not to redeploy the distributed application to different hosts or cloud participants.

V.3.1 Collecting Latency

In previous work collecting latencies, we had employed a three-round technique for collecting latencies that was based on an initiator, a reply, and a follow up. With this techique, we were able to collect latency from two participants in three messages using a TCP transport.

Though efficient in message complexity, this approach to gathering latency was extremely slow due to the blocking and unicast nature of TCP. We were also using reactor-based timers that fired every 5 seconds and randomly initiated a new round of latency collection from two participants. Getting full coverage with this type of system took an inordinate amount of time—even for a handful of participants. In early tests, we found that because of the randomization of endpoints, it could take over two minutes just to fully collect latencies from twenty participants. For an enterprise DRE system, we might have 10,000 participants that needed to have latency information between each connected participant.

To facilitate the aim of collecting latencies quickly from thousands of participants, we decided to use the underlying DDS transport of the KaRL engine. DDS uses multicast and broadcast technologies, whenever possible, and can accomplish reliable group communications within microseconds—making it ideal for the task at hand.

As with our TCP-based implementation, we used a three round approach. A participant called an originator starts a timer for each participant, and then sends a message of the form (t, i, c, s), where t = latency, i = id of the initiator, c = lamport clock (incremented from the value before sending the message), and s = 0 (signaling that this is the initiator latency

message from the source). Every participant receives this message, starts their own timers for each participant, and then sends a reply of the form (t, i, o, c, s), where t = latency, i = id of the replier, o = the originator of the initial message, c = lamport clock of the initiating message, and s = 1 (signifying that this is a reply message). Upon receiving this reply, the originator stops the timer for the id of the participant who replied and records the elapsed time. Every participant that receives this reply message (labeled with s = 1), responds to these repliers with a message of the form (t, i, o, c, s), where t = latency, i = id of the participant doing a follow up, o is the id of the replier that sent a message with s = 1, c is the originator's lamport clock, and s = 2. After the replier receives a message with s == 2 and itself indicated as the originator (the o parameter), it stops the timer associated with the follow up participant and records the elapsed time. Because we're using a reliable group protocol for these three messages, this establishes a full latency collection within three messages and is only limited by the speed of the participants in processing these messages. Consequently, we limit the message complexity of this phase to O(N), specifically 3N—where N is the number of participants.

To speed up message processing, two optimizations can be done: 1) adding more threads to read messages, or 2) process larger batches of messages from a queue instead of only reading one message at a time. For listener or callback-based mechanisms, the first option is generally best. For polling and waitset-based middleware, a combination of the first and second options can result in better performance.

V.3.2 Disseminating Aggregate Latencies

After performing the collection of latencies, each participant in the network has a list of latencies between itself and all other participants it can reach. If it wasn't able to reach a participant, it inserts a default latency (our default is 200 seconds, but this can be arbitrarily large). Otherwise, we need to have the participants socialize once more to inform each other of all latencies in the network.

Disseminating aggregate latencies is an O(N), specifically N, operation where each participant sends a message containing a list of its latencies to all other participants. Our implementation encodes the list in a string in the form a=*id1=latency1;id2=latency2;...*, meaning the latency between myself and id1 is latency1 and the latency between myself and id2 is latency2, and then inserts this into a message of the form (t, i, a, c), where t = aggregate, i = id of the participant, a = the aggregate we formed, and c is the clock from the latency round done in Section V.3.1.

The dissemination of aggregate latencies can be skipped if the user only wants to perform the Blind CID heuristic from Section V.2. All other algorithms require fully informed participants in order to guage the utility of a deployment.

V.3.3 Degree-Informed Summations of Latencies

Before the CID-based algorithms can be run, the participants need to process the latency lists into summations by degree of the user-provided dataflow graph. For instance, if a user-provided dataflow had 10,000 participants with degrees (connectivity) of 1-3 edges per participant, then this summation step would require creating a vector of 10,000 x 3 integers. Each row corresponds to a participant id. Each column to a degree. If using a hash map or table, the key could be the degree (1-3), and the value could be a vector of 10,000—where each element of the vector is the summed latency at that id. We use this latter strategy because it allows us to easily sort the vector by latency in the degree, which is necessary for CID, BCID, and the degree-guided genetic algorithm.

Because we've disseminated the aggregate latencies before this step occurs, we have two options available: 1) have each participant process the latencies and incur no messaging overhead or 2) have each participant follow up the dissemination of aggregate latency with another message that performs the summation of latencies by degree that involve this participant. At first glance, option 1 may appear to be the best solution for the enterprise DRE system, but it's important to remember that processing these latencies takes time. In

fact, from our testing in Section V.4, this summation of all latencies by degree for all participants is the most expensive operation in the process of latency collection and voting. We implemented option 2 instead because it reduces the computational workload by factor N, the number of participants in the network, and limits the time required for this step from a few seconds for 10,000 participants to microseconds plus the time required to disseminate the summation message.

The summation message for option 2 takes the form (t, i, s, c), where t = summation, i is the id of the participant, s is the stringified summations calculated by the participant, and c is the clock of the latency rounds that were performed for these summations. Because most dataflows have only a handful of degrees, this message tends to be very small—from tens to hundreds of bytes—which means it can often fit inside of a single message transmission unit in Ethernet traffic and is very inexpensive to resend if using a faulty connection medium like wireless.

V.3.4 Voting for Redeployment

The final step of multi-agent redeployment is to have the participants agree or disagree on a redeployment. We accomplish this with a very simple voting system, but before we describe the interactions of the participants, we first outline some considerations that implementers should take into account before implementing their own redeployment system.

First, redeployment of applications is computationally expensive. In our case, a heavyweight executable has been forked, and killing the running program and launching the application on both Windows and Linux takes at least milliseconds. This is manageable overhead if a redeployment happens once every few minutes, but it can result in operating system-based thrashing if the redeployments happen often. Consequently, care should be taken to only redeploy when necessary and not simply every time a better deployment is found. We recommend managing this overhead with threshold values (demarkations that

signal boundaries for redeployment). Our default threshold in MADARA for a redeployment is when the resulting deployment would be more than 10% faster than current. This parameter may be tweaked within the MADARA Transport Settings class.

Second, the data structures and identifiers used for participant identification and indexing can make a huge difference in both CPU and memory expenditure. Strings as identifiers should be avoided if at all possible for two reasons: 1) they cause mallocs/news with each deep copy, 2) they force hash table lookups to add an additional O(M), where M is the maximum size of the string in the case that users have specified similar string ids—which is very common with participant identifiers (think host1, host2, host3), and 3) for the reasons noted earlier and because compilers and CPUS are much better at optimizing for integers, using strings in latency aggregation and voting is orders of magnitude slower than using integer identifiers. For these reasons, we used a separate data structure to hold our string-based host information (*i.e.*, host:port pairings) and used a pointer-like system with integers for the identification of participants within each step. Doing so resulted in the three latency rounds alone (initiator, reply, and follow up) dropping from seconds to microseconds for dozens of participants. Similar results should come from using integer-based identification throughout the process of redeployment (including the approximation techniques discussed earlier in Section V.2.

With those caveats out of the way, we'll discuss our approach to voting. Because of the nature of this problem, we're only interested in the best deployment candidate—regardless of how many different approximation algorithms we run, if they're worse than the current deployment or the best deployment, we don't care about them. For this reason, voting is a simple group-wide min operation. Each participant submits a vote for the best deployment they have, and the winning vote determines if a redeployment happens (based on the threshold we discussed earlier—the one that was 10% by default in MADARA). Despite the all-to-all communication of latencies and summations, these votes may be different because 1) the degree-guided genetic algorithm depends on randomization, which

will result in different answers per participant, and 2) some participants may not be able to communicate together and may miss out on information from the earlier latency collection, aggregation, or summation.

In our MADARA implementation, we allow for both a group-wide voting system and a user-specified list of voters. The former means that all participants run the algorithms (*e.g.*, CID, BCID, CID with a blind genetic algorithm, BCID with a guided genetic algorithm, etc.) specified by the user. The latter means that only certain participants will run the algorithms and vote on the best deployment available. We obviously assume that all participants in the network are being truthful and that none of them have reason to lie about the best deployment. For some researchers looking into Mechanism Design and untruthful agents, precautions would need to be taken to ensure that no agent is gaming the system to cause frequent redeployments to thrash the distributed application or take advantage of the optimistic nature of the voting algorithm. A plurality vote instead of a min would be an effective way to accomplish this, as long as the plurality is not allowed to be gamed by inserting fake participants into the network.

V.4 Analysis of Experimental Results

This section analyzes the performance of—and utility produced by—the MADARA algorithms and heuristics described in Section V.2. All experiments were conducted on an Intel Core2 Duo clocked at 2.53 GHz and 4 GB of RAM running Windows 7 32-bit operating system. The C++ code was compiled in MS Visual Studio 2008 under the optimized release mode. All experimental code and configuration information can be found online on the MADARA project site at `madara.googlecode.com`.

We offer two types of metrics for our experiments: (1) *runtime*, which is important to Requirement 2 of the motivating DRE application in Section V.1 and (2) *system slowdown*, which reflects the runtime performance of the resulting deployment (Requirement 3 of the

motivating DRE application). All experiments are repeated ten times and the averages are reported.

V.4.1 Experimental Setup

The first experiment created a hand-coded network configuration where four drones in four disjoint groups have 500us latency to their local drones and these special drones have complete coverage of the network topology at the 500us latency. Every other link has 1s latency, which is typical for radio-based communication in a disaster area. The underlying network has exactly one perfect configuration for this deployment of four special drones collecting from equal divisions of the swarm. Consequently, system slowdown will be high if the algorithm does not find the optimal deployment.

For the second experiment, we added noise to the underlying network that allowed for thousands of local minima and maxima to exist. A perfect configuration with 500us latency links is present, but we have added a uniform distribution of latencies from 600us to 3s to the network to confuse the tested algorithms.

We examine this experiment in three configurations—two specialized drones with two optimal local groups, three specialized drones with three optimal local groups, and four specialized drones with four optimal local groups. These tests expect that the guided heuristics and algorithms will far outperform the unguided ones. The motivating DRE application—and consequently these experiments—also demand that results come in within dozens of seconds at least, and milliseconds if possible.

System slowdown is defined by the equation "slowdown = 2 * system_latency / (1,000,000 * size)" in these experiments. With the optimal configuration, slowdown == 1. Anything greater than one is a factor of slowdown. For example, 2.0 is a 100% slowdown in the overall system.

V.4.2 Analysis of Results

Due to space limitations, we show only the four drone results for the first experiment (the results for one to three drones were similar in scale for both runtime clock and system slowdown). The first experiment (one perfect configuration and everything else is bad) is highlighted in Figures V.3 and V.4.

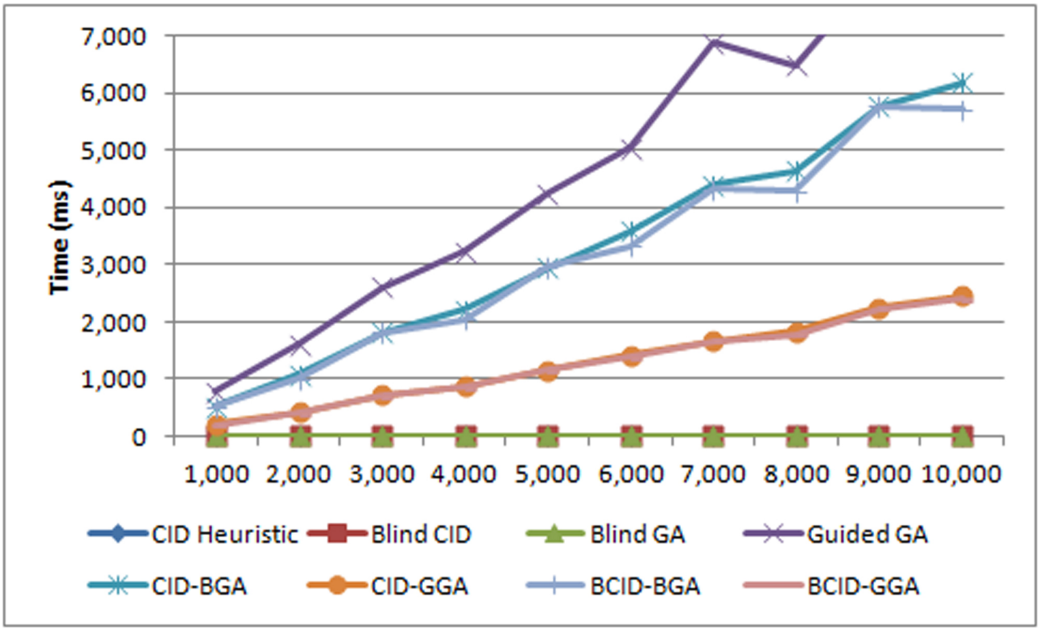

Figure V.3: Runtime Required Under the First Experiment

The second experiment (one perfect configuration and everything else is bad or a local minima) has results reported in Figure V.5 for runtime information and Figures V.6, V.7, and V.8.

A key observation from these tests is that the fully-informed CID heuristic is well-suited for both experiments. The CID heuristic—and consequently the CID-BGA (CID with Blind Genetic Algorithm) and CID-GGA (CID with Guided Genetic Algorithm)—find the optimal configuration in all network sizes (1k to 10k drones) within a second of computation (the CID heuristic takes no more than 7.5ms to return an answer even on the 10,000 drone tests).

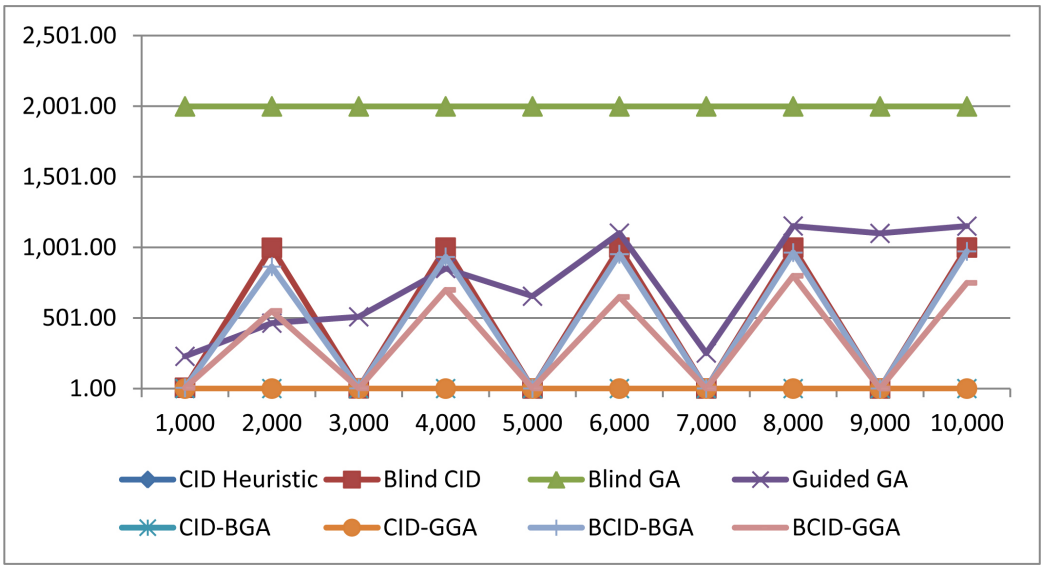

Figure V.4: System Slowdown under the First Experiment

The other algorithms had difficulty with the first experiment. The Blind CID algorithm (the one that works without being fully informed of all network latencies) came in second place on all tests, followed shortly by the Guided Genetic Algorithm (GGA), which was within 10% of system slowdown of Blind CID but took significantly longer to reach the approximation. In particular, Blind CID returns an answer in 5.6ms and 7.5ms on each 10,000 drone test, respectively, and the GGA requires 8s and 36s to come that close on the same drone network sizes.

The Blind Genetic Algorithm (BGA) is negligibly better (1-2% less than the maximum slowdown possible) than random deployments as far as utility and preventing system slowdown are concerned when used by itself in all four of the test configurations shown in Figures V.4, V.6, V.7, and V.8.

The Guided Genetic Algorithm (GGA) shown in Algorithm 4 performs much better than the BGA. By infusing degree information into the choice mechanism for mutation candidate selection, the system slowdown experienced is roughly 50% of the results of the BGA in the first experiment (one perfect solution with no noise in the network). In the two, three, and four drone configurations of the second experiment, the BGA managed a worst

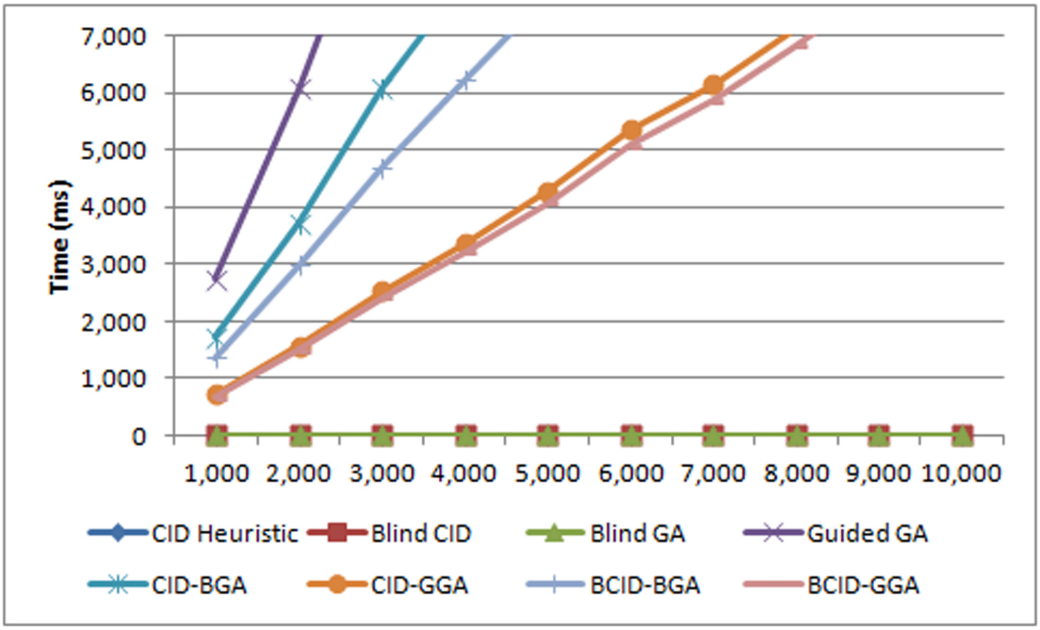

Figure V.5: Runtime Required Under the Second Experiment

case system slowdown of 120%, 140%, and 150%, respectively, compared to the 180% slowdown produced by GGA in each of these tests.

We also included four implementations of guided genetic algorithms that began their mutations with candidates produced by the CID heuristic and the Blind CID heuristic. These implementation are labeled CID-BGA, CID-GGA, BCID-BGA, and BCID-GGA in each of the figures. Each CID heuristic seeded implementation has the best configuration to begin with, so the solutions cannot be improved.

More interesting results come from the BCID-BGA and BCID-GGA implementations. In the first experiment (where only 1 excellent configuration is available), both of these manage to augment the Blind CID heuristic, but BCID-GGA decreases the system slowdown dramatically—from 200% to 447% improvement in the Blind CID heuristic-produced candidate when an improvement is possible.

The second experiment sees a flip-flop in this relationship. The BCID heuristic candidate performs well when many OK solutions are available, as is the case in the noisy experiments. The Guided GA focuses on mutating chromosomes with high degrees in the

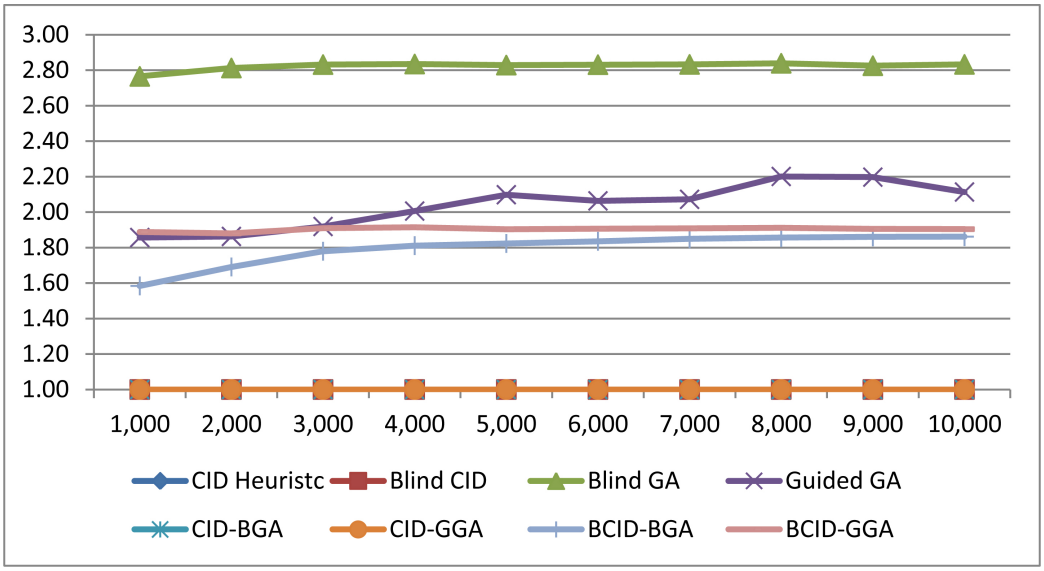

Figure V.6: System Slowdown with 2 Specialized Drones in a Noisy Environment

user-provided workflow (80% of the time), but with BCID seeding the candidates with the same focus, we may waste some time, as shown in the results. This wasted work is quite expensive given the time constraints available to approximate a solution (GGA is only allowed to perform maximum of 500 mutations, while the unguided BGA can perform up to 2,000 in the same time frame).

Both BCID-BGA and BCID-GGA eventually converge to similar system slowdowns, but the unguided option makes more impactful mutations at lower sizes. This result occurs because BGA has many more chances for high impact with 2,000 mutations. As the size of the deployment increases, the BGA has less coverage and the quality of the overall improvement decreases to the same level as GGA's 500 mutations with a preference on high-degreed processes in the workflow.

These results suggest potential future enhancements for the guided genetic algorithms or potential replacements. If BCID and CID heuristics produce excellent candidates for local searches, we can modify the GGA to use incoming degree information rather than just outgoing degree information to make mutation candidate choices. This result may

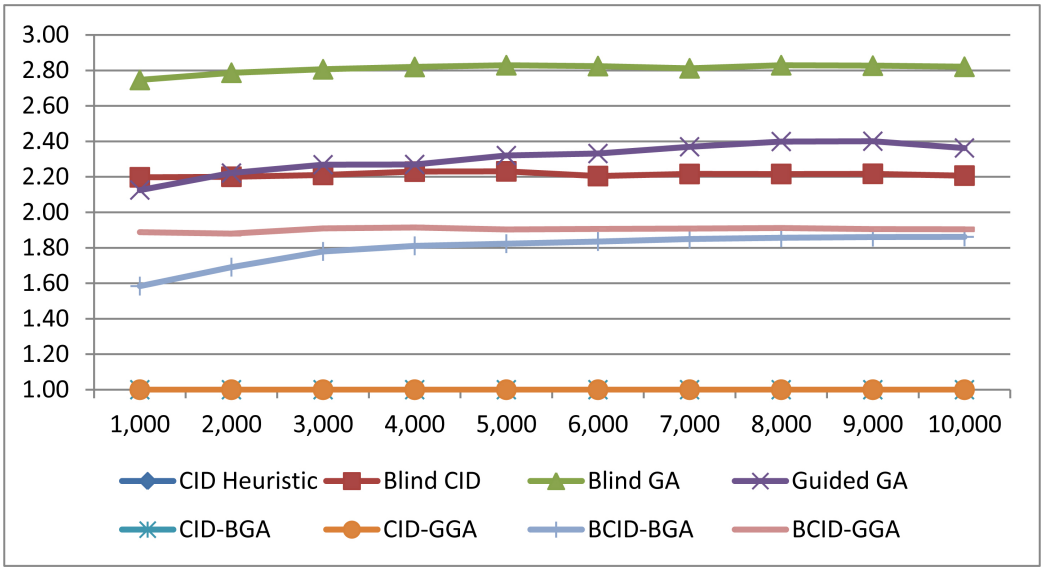

Figure V.7: System Slowdown with 3 Specialized Drones in a Noisy Environment

also motivate a backtracking solution that takes BCID and CID candidates as seeds to start from—especially if we can assume a fully-informed drone network.

The final analysis we make from the data concerns the latency preparation phase defined in Algorithm 1. Figure V.9 shows the results for having each drone run a prepare operation. When a drone must run one of the algorithms discussed in this paper the preparation only takes between 10us on a 1,000 drone solution and 100us for a 10,000 drone solution if it only has to sort and aggregate its own latency information. A drone takes between 10ms for a 1,000 drone solution and 1.5s for a 10,000 drone solution if it must perform preparation for all latencies according to the degrees of the deployment.

V.5 Related Work

This section compares related work on deployment problems based on (1) constraint satisfaction problem solving, (2) genetic algorithms, and (3) heuristics with MADARA.

Constraint satisfaction problem solving. Haldik et. al. [36] presents a constraint programming technique to solve static allocation problems in real-time tasks. Cucu-Grosjean

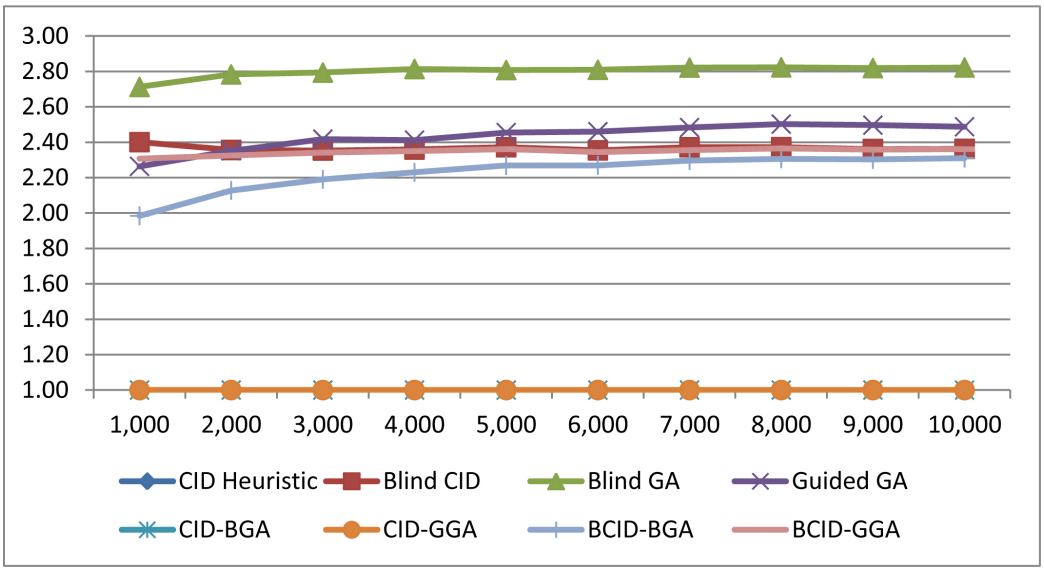

Figure V.8: System Slowdown with 4 Specialized Drones in a Noisy Environment

et. al. [13] proposes two approaches to addressing real-time periodic scheduling on heterogeneous platforms (a CSP problem). The first method requires an encoding of the problem into a basic format that is then passed into state-of-the-art CSP solvers. The other approach encodes problems in an optimized way to obtain solutions faster. Despite being faster than traditional CSP solvers, both techniques take dozens to hundreds of seconds to solve even small number of constraints, which does not meet the requirements of our motivating DRE application in Section V.1 that exhibits thousands of constraints defined on the deployment between the collection drone and its group.

White et. al. [88] scale a CSP solver to work with 5,000 features in a software product line. The time required for doing this activity ranged from 50 seconds in an incomplete bounded worst case to 170 seconds to find an optimal configuration. In contrast, our motivating DRE application in Section V.1 requires runtime solutions within milliseconds or at most seconds. Section V.2 presents techniques provided by MADARA that can meet these time constraints.

Genetic algorithms. Whereas CSP solving typically involves backtracking through potential matches, genetic algorithms are a type of local search that tries to approximate

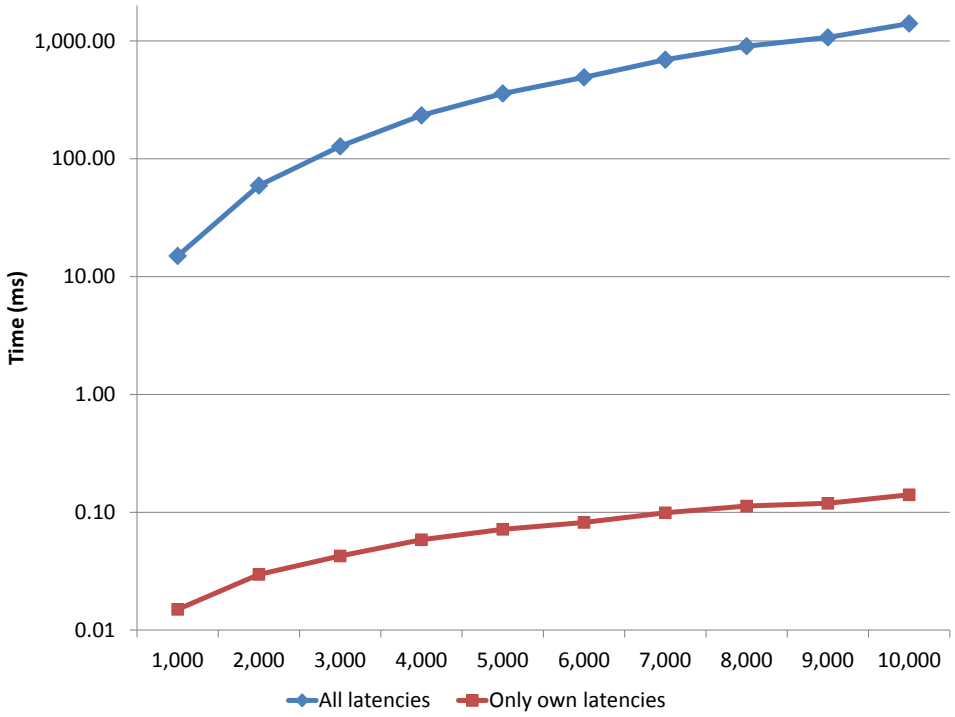

Figure V.9: Preparation Runtime for 4 Specialized Drones in a Noisy Environment

an optimal match through mutations, fitness functions, and crossbreeding best candidates according to the fitness criteria. Heward et. al. [33] recently used genetic algorithms to optimize configurations of monitors in a web services application. This method is unsuitable for our motivating DRE application, however, since it requires roughly an hour to compute an approximated good configuration.

Wieczorek et. al. [89] use a genetic algorithm to schedule scientific workflows in Grid environments, but their mutation-based scheme similarly required at least hundreds of seconds (some of their tests showed requirements of tens of thousands of seconds—several hours). Other implementers have used combinations of genetic algorithms and neural networks [41] and even knowledge and reasoning [39] to converge to optimal solutions. These approaches concentrate on offline or human-interactive solutions, however, and thus are not suitable for DRE application problem solving because they require many minutes or hours to approximate a solution.

Though these results appear to discount genetic algorithms as solution possibilities, recent publications [15, 40, 64] and our on experience with the MADARA guided genetic algorithms (via heuristics) described in Section V.2, indicates that this approach has merit.

Heuristics. Heuristics approximate good solutions and often serve as guides for local search techniques, such as genetic algorithms, simulated annealing, or backtracking and depth-first searches. Some researchers use these heuristics to directly approximate scheduling [33] in grids and workflow solutions [12] for real-time solutions. The latter is of interest to us since the heuristic approximates a constraint problem involving a set of workflows within milliseconds. The solution [12], however, was demonstrated on only five hosts and not thousands, so it is not readily apparent how to migrate our motivating DRE application to the heuristic defined in either of these papers.

The heuristic-based anytime A* search [31] is similar to our MADARA approach to genetic algorithms for the motivating DRE application. In particular, both solutions may be stopped at any time and a solution is presented to the user (though it may not be optimal). The MADARA heuristics offered in this paper can be used with an A* search, which is the focus of future work.

In summary, we are unaware of any deployment frameworks, other than MADARA, that approximate this kind of problem in enterprise DRE systems within the time constraints required from the motivating DRE application.

V.6 Future Work

The redeployment infrastructure and heuristic-based approaches described in earlier sections do not solve all redeployment problems in distributed real-time and embedded systems. In this section, we discuss unresolved challenges in redeployment infrastructure.

V.6.1 Side Effects

The MADARA redeployment infrastructure assumes that all applications have no side effects. Side effects are modifications to host or participant state that are local to the participant. Examples of side effects include databases, files, and shared memory segments.

For continuously-online applications, like those found in most sensor environments, side effect redeployment is often not a problem because new data is being regenerated constantly and the transfer of these artifacts may be counterproductive. However, many other applications do require side effects and consequently, they will require transfer of state between the old and new participant instances within a redeployment.

From our perspective, developers should be aware of two major issues with side effect migration. First, if the state is absolutely vital to the operation of the participant and the network, then the new participant must stay offline until its redeployment is successful, and the old participant should remain online until the new participant is ready (i.e., it has received all side effects and is not blocking on the availability of other required new participants). Second, consistency will always be a problem unless the system is completely halted when a redeployment is detected and the underlying infrastructure is allowed to synchronize between the old participants and their new instances. If a redeployment is agreed upon by the participants in the network, and state is initially transferred to the new instances of participants at time 0, the old participant and the new participant will be consistent in state as long as the old participant refuses new information or state changes. If the old participant continues to perform operations and makes major modifications through continued transactions, the state of the old participant at time 10, for example, will be different from the new participant who is still operating under the state at time 0 and waiting for its own dependencies.

Professional-grade redeployment infrastructure should provide users with management options for side effects per participant, so real-time system developers can customize the redeployment to their database, file, and other state information concerns.

V.6.2 Custom Constraints

The subgraph isomorphic problem is a useful one to target for redeployment, but it does not solve all redeployment criteria. Ideally, a redeployment infrastructure should support custom constraints and advanced constraint satisfaction scenarios. For instance, a deployment may require not only optimal latencies between certain participants, but certain participants may require more memory or CPU power than others, especially ones that must service requests from all participants. For this reason, the heuristic-based approaches that we developed as part of this dissertation should ideally be augmented with a fast, targeted constraint engine for meeting the types of deployment constraints required of most real-world applications.

CHAPTER VI

CONCLUDING REMARKS

In this dissertation we have outlined DRE research challenges in three areas: 1) monitoring and responding to environments in continuous, real-time systems, 2) real-time application deployment and execution, and 3) emergent redeployment of real-time systems.

The challenges in the area of monitoring and responding to environments resulted in the creation of a real-time knowledge and reasoning engine for DRE systems called the MADARA Knowledge and Reasoning Language Engine (KaRL) that disseminates knowledge over a publish-subscribe transport with a proven track record of scaling well in DRE systems. The engine works on Windows and Linux operating systems, evaluates knowledge rules an order of magnitude faster than competing reasoning engines and provides diverse engine mechanisms for blocking and non-blocking evaluations.

The challenges in real-time application deployment and execution resulted in the creation of a deployment and testing engine called the MADARA KaRL-Automated Testing Suite (KATS) which is built on ACE and KaRL. KATS provides a robust, flexible process lifecycle for distribution coordination and sequencing of processes across the DRE system. KATS features fine-grained controls over execution, low latency, and portability across heterogeneous operating system and network architectures.

The challenges in emergent redeployment of enterprise DRE systems resulted in the development of three heuristics-based approaches. The Comparison-based Iteration by Degree (CID) heuristic uses the connectivity information from a user-provided dataflow to approximate an optimal redeployment using $O(N^2)$ memory and is very effective at optimizing dataflows that describe broadcasts, aggregators, or centralized networks. The Blind CID heuristic uses only O(N) memory to approximate an optimal redeployment but is less informed than CID. Still, it is useful when less memory usage is desired, and it may result

in better approximations of hierarchical and highly interconnected networks. The other heuristic-based approach is a degree-informed genetic algorithm that mutates a solution based on degree information in a user-provided dataflow for a specified amount of time.

To facilitate these redeployment approximation approaches, we developed a framework for latency collection, aggregation, and summation and incorporated it into the MADARA network transport. The heuristic-based algorithms were incorporated along with a voting system based on a group-wide global minimum. Together, these mechanisms and techniques take a user-defined dataflow, collect latencies from the network, and redeploy the distributed application once a performance threshold is met.

APPENDIX A

LIST OF PUBLICATIONS

Refereed Journal Publications

J3 Loyall J.P., Gillen M., Paulos A., Bunch L., Carvalho M., Edmondson J., Schmidt D.C., Martignoni III A., and Sinclair A. Dynamic Policy-Driven Quality of Service in Service-Oriented Information Management Systems. Component and Service-Oriented Distributed Embedded Real-Time Systems, Special Issue on Software: Practice and Experience. 2011.

J2 Edmondson, J. and Schmidt, D.C. Multi-agent distributed adaptive resource allocation (MADARA). International Journal of Communication Networks and Distributed Systems, Special Issue on: Grid Computing, Vol. 5, No. 3, pp.229–245. November, 2010.

J1 Hill, J. H., Schmidt, D.C., Edmondson, J., and Gokhale, A. Tools for Continuously Evaluating Distributed System Qualities. IEEE Software, Vol. 27, No. 4. August, 2010.

Refereed Conference Publications

C8 Edmondson, J., Gokhale, A., Neema, S. Automated Redeployment of Real-Time Systems Informed by User-Provided Workflows. RTAS WIP 2012.

C7 Edmondson, J., Gokhale, A., Neema, S. Automating Testing of Service-oriented Mobile Applications with Distributed Knowledge and Reasoning. Proceedings of Service-Oriented Computing Architectures (SOCA) 2011. Irvine, CA.

C6 Edmondson, J., Schmidt, D., Gokhale, A. QoS-enabled Distributed Mutual Exclusion in Public Clouds. Proceedings of the 1st International Conference of Secure Virtual Infrastructures in Distributed Object Architectures. Crete, Greece.

C5 Edmondson, J., Gokhale, A. Design of a Scalable Reasoning Engine for Distributed, Real-time and Embedded Systems. Proceedings of the 5th International Conference on Knowledge, Science, Engineering and Management. Irvine, CA.

C4 Loyall, J., Edmondson, J. Bunch, L., Martignoni III, A., Gillen, M., Paulus, A. Varsheneya, P. Schmidt, D., Carvalho, M. Dynamic Policy Driven Quality of Service in Service Oriented Systems. Proceedings of the 13th IEEE International Symposium on Object/component/service-oriented Real-time distributed computing. Carmona, Spain.

C3 Hill, J. H., Turner, H., Edmondson, J. and Schmidt, D.C. Unit Testing Non-Functional Concerns of Component-based Distributed Systems. Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation (ICST). Denver, CO.

C2 Loyall, J., Carvalho, M., Schmidt, D.C., Gillen, M., Martignoni III, A., Bunch, L., Edmondson, J., and Corman, D. QoS Enabled Dissemination of Managed Information Objects in a Publish-Subscribe-Query Information Broker. The SPIE Defense Transformation and Net-Centric Systems conference. Orlando, FL.

C1 Xiong M., Parsons J., Edmondson J., Nguyen H., and Schmidt D.C. (2007, April). Evaluating Technologies for Tactical Information Management in Net-Centric Systems. Proceedings of the Defense Transformation and Net-Centric Systems conference. Orlando, Florida.

Book Chapters

BC1 Edmondson, J., Schmidt, D. (2011). Towards Accurate Simulation of Large-Scale Systems via Time Dilation. Real-time Simulation Technologies: Principles, Methodologies, and Applications. CRC Press.

Submitted for Publication

S4 Edmondson, J., Gokhale, A., Neema, S. Heuristics and Genetic Algorithms for Adaptive Deployments in Large-scale, Real-time Systems. IAAI 2012, Emerging Applications track.

S3 Edmondson, J., Gokhale, A., Schmidt, D. Guided Genetic Algorithms for Real-time, Continuous Optimizations of User-Provided Deployment Workflows. GECCO 2012.

S2 Edmondson, J., Gokhale, A., Schmidt, D., Neema, S. Automated, Adaptive Deployments for Enterprise Distributed, Real-time, and Embedded Applications. ASE 2012.

S1 Edmondson, J., Gokhale, A., Schmidt, D. Facilitating Consistency in Online Reasoning With Knowledge Event Semantics. DEBS 2012.

REFERENCES

[1] Nada Abdallah and François Goasdoué. Non-conservative extension of a peer in a p2p inference system. *AI Communications*, 22:211–233, December 2009. Available from World Wide Web: http://portal.acm.org/citation.cfm?id=1662603.1662604.

[2] P. Adjiman, P. Chatalic, F. GoasdouÃl', M. c. Rousset, and L. Simon. Distributed reasoning in a peer-to-peer setting. In *In de Mantaras*, pages 945–946, 2004.

[3] Eyal Amir and Sheila Mcilraith. Partition-based logical reasoning for first-order and propositional theories. *Artificial Intelligence*, 162:49–88, 2000.

[4] Jaiganesh Balasubramanian, Sumant Tambe, Balakrishnan Dasarathy, Shrirang Gadgil, Frederick Porter, Aniruddha Gokhale, and Douglas C. Schmidt. Netqope: A model-driven network qos provisioning engine for distributed real-time and embedded systems. In *RTAS' 08: Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 113–122, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[5] Marin Bertier, Luciana Arantes, and Pierre Sens. Distributed mutual exclusion algorithms for grid applications: A hierarchical approach. *J. Parallel Distrib. Comput.*, 66:128–144, January 2006. Available from World Wide Web: http://dx.doi.org/10.1016/j.jpdc.2005.06.020.

[6] Alex Borgida and Luciano Serafini. Distributed description logics: Assimilating information from peer sources, 2003.

[7] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. In *IN PROC. INTERNATIONAL SYMPOSIUM ON*

SOFTWARE TESTING AND ANALYSIS (ISSTA, pages 123–133. ACM Press, 2002.

[8] Jiannong Cao, Jingyang Zhou, Daoxu Chen, and Jie Wu. An efficient distributed mutual exclusion algorithm based on relative consensus voting. *Parallel and Distributed Processing Symposium, International*, 1:51b, 2004.

[9] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Third Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, Louisiana, February 1999. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS.

[10] Guiran Chang, Chunguang Tan, Guanhua Li, and Chuan Zhu. Developing mobile applications on the android platform. In Xiaoyi Jiang, Matthew Ma, and Chang Chen, editors, *Mobile Multimedia Processing*, volume 5960 of *Lecture Notes in Computer Science*, pages 264–286. Springer Berlin / Heidelberg, 2010.

[11] Clovis Chapman, Wolfgang Emmerich, Fermín Galán Márquez, Stuart Clayman, and Alex Galis. Software architecture definition for on-demand cloud provisioning. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 61–72, New York, NY, USA, 2010. ACM. Available from World Wide Web: http://doi.acm.org/10.1145/1851476.1851485.

[12] Tommaso Cucinotta and Gaetano Anastasi. A heuristic for optimum allocation of real-time service workflows. In *Service Oriented Computing and Applications, 2011. SOCA '11. International Conference on*, pages 169–172, 2011.

[13] L. Cucu-Grosjean and O. Buffet. Global multiprocessor real-time scheduling as a constraint satisfaction problem. In *Parallel Processing Workshops, 2009. ICPPW '09. International Conference on*, pages 42–49, sept. 2009.

[14] JosÃl' C. Cunha, JoÃčo LourenÃğo, Tiago R. AntÃčo, Jo Ao Lourenco, and Tiago R. Ant Ao. An experiment in tool integration: the ddbg parallel and distributed debugger. In *EUROMICRO Journal of Systems Architecture, nd Special Issue on Tools and Environments for Parallel Processing*, pages 708–717. Springer, 1999.

[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182 –197, apr 2002.

[16] Gan Deng, Jaiganesh Balasubramanian, William Otte, Douglas C. Schmidt, and Aniruddha Gokhale. DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In *Proceedings of the 3rd Working Conference on Component Deployment (CD 2005)*, pages 67–82, Grenoble, France, November 2005.

[17] Catalin Dumitrescu, Ioan Raicu, Matei Ripeanu, and Ian Foster. Diperf: An automated distributed performance testing framework. In *in 5th International Workshop in Grid Computing*, pages 289–296. IEEE Computer Society, 2004.

[18] James Edmondson and Aniruddha Gokhale. Design of a scalable reasoning engine for distributed, real-time and embedded systems. In *Proceedings of the 5th International Conference on Knowledge, Science, Engineering and Management (KSEM)*.

[19] James Edmondson and Aniruddha Gokhale. Qos-enabled distributed mutual exclusion in public clouds. In *Proceedings of the 1st International Conference of Secure Virtual Infrastructures in Distributed Object Architectures*.

[20] James Edmondson and Aniruddha Gokhale. Design of a scalable reasoning engine for distributed, real-time and embedded systems. In *Proceedings of the 5th conference on Knowledge Science, Engineering and Management, KSEM 2011*, Lecture Notes in Artificial Intelligence (LNAI). Springer, 2011.

[21] James Edmondson, Aniruddha Gokhale, and Sandeep Neema. Automating testing of service-oriented mobile applications with distributed knowledge and reasoning. In *Proceedings of the Service-Oriented Computing and Applications (SOCA)*.

[22] James Edmondson, Aniruddha Gokhale, and Sandeep Neema. Automated redeployment of real-time systems informed by user-provided workflows. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium Work-In-Progress (RTAS WIP)*, 2012.

[23] James Edmondson and Douglas Schmidt. Multi-agent distributed adaptive resource allocation (madara). *International Journal of Communication Networks and Distributed Systems, Special Issue on: Grid Computing*, 5(3):229–245, 2010.

[24] James Edmondson and Douglas Schmidt. Towards accurate simulation of large-scale systems via time dilation. In Katalin Popovici and Pieter J. Mosterman, editors, *Real-time Simulation Technologies: Principles, Methodologies, and Applications*. CRC Press, 2011.

[25] Falk Fraikin and Thomas Leonhardt. Seditec " testing based on sequence diagrams. In *Proceedings of the 17th IEEE international conference on Automated software engineering*, ASE '02, pages 261–, Washington, DC, USA, 2002. IEEE Computer Society. Available from World Wide Web: http://portal.acm.org/citation.cfm?id=786769.787022.

[26] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[27] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. Available from World

Wide Web: http://doi.acm.org/10.1145/945445.945450.

[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37:29–43, October 2003. Available from World Wide Web: http://doi.acm.org/10.1145/1165389.945450.

[29] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *PLDI*, 2005.

[30] Weiming Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu. Falcon: on-line monitoring and steering of large-scale parallel programs. In *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers '95., Fifth Symposium on the*, pages 422 –429, feb 1995.

[31] Eric A. Hansen and Rong Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research (JAIR*, 28:267–297, 2007.

[32] Jacob G. Hansen and Eric Jul. Lithium: virtual machine storage for the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 15–26, New York, NY, USA, 2010. ACM.

[33] Garth Heward, Jun Han, Jean-Guy Schneider, and Steve Versteeg. Run-time management and optimization of web service monitoring systems. In *Service Oriented Computing and Applications, 2011. SOCA '11. International Conference on*, pages 294–299, 2011.

[34] James H. Hill, Douglas C. Schmidt, James R. Edmondson, and Aniruddha S. Gokhale. Tools for continuously evaluating distributed system qualities. *IEEE Software*, 27(4):65–71, 2010.

[35] James H. Hill, Hamilton A. Turner, James R. Edmondson, and Douglas C. Schmidt.

Unit testing non-functional concerns of component-based distributed systems. In *ICST*, pages 406–415, 2009.

[36] Pierre-Emmanuel Hladik, Hadrien Cambazard, Anne-Marie Daplanche, and Narendra Jussien. Solving a real-time allocation problem with constraint programming. *Journal of Systems and Software*, 81(1):132–149, 2008. Available from World Wide Web: http://www.sciencedirect.com/science/article/pii/S0164121207000672.

[37] Ian Horrocks, Peter F. Patel-Schneider, and Frank Van Harmelen. From shiq and rdf to owl: The making of a web ontology language. *Journal of Web Semantics*, 1:2003, 2003.

[38] Ahmed Housni and Michel Trehel. Distributed mutual exclusion token-permission based by prioritized groups. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, pages 253–, Washington, DC, USA, 2001. IEEE Computer Society. Available from World Wide Web: http://portal.acm.org/citation.cfm?id=872017.872214.

[39] Yanrong Hu and S.X. Yang. A knowledge based genetic algorithm for path planning of a mobile robot. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 5, pages 4350 – 4355 Vol.5, april-1 may 2004.

[40] Lester Ingber and Bruce Rosen. Genetic algorithms and very fast simulated reannealing: A comparison. *Mathematical and Computer Modelling*, 16(11):87 – 100, 1992. Available from World Wide Web: http://www.sciencedirect.com/science/article/pii/089571779290108W.

[41] A.A. Javadi, R. Farmani, and T.P. Tan. A hybrid intelligent genetic algorithm. *Advanced Engineering Informatics*, 19(4):255 – 262, 2005.

[42] Alissa Kaplunova, Ralf Möller, Sebastian Wandelt, and Michael Wessel. Towards scalable instance retrieval over ontologies. In *Proceedings of the 4th international conference on Knowledge science, engineering and management*, KSEM'10, pages 436–448, Berlin, Heidelberg, 2010. Springer-Verlag. Available from World Wide Web: http://portal.acm.org/citation.cfm?id=1885721.1885761.

[43] M. J. Katchabow. Making distributed applications manageable through instrumentation. *Journal of Systems and Software*, 1999.

[44] Sarfraz Khurshid and Darko Marinov. Testera: A novel framework for testing java programs. In *In IEEE International Conference on Automated Software Engineering (ASE*, pages 22–31, 2003.

[45] Tatiana Kichkaylo and Vijay Karamcheti. Optimal resource-aware deployment planning for component-based distributed applications. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 150–159, Washington, DC, USA, 2004. IEEE Computer Society. Available from World Wide Web: http://dl.acm.org/citation.cfm?id=1032647.1033301.

[46] John S. Kinnebrew, William R. Otte, Nishanth Shankaran, Gautam Biswas, and Douglas C. Schmidt. Intelligent Resource Management and Dynamic Adaptation in a Distributed Real-time and Embedded Sensor Web System. In *Proceedings of the 12th International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC '09)*, Tokyo, Japan, March 2009.

[47] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. In *The 14th International Workshop on the Implementation of Functional Languages, IFLâĂŹ02, Selected Papers, volume 2670 of LNCS*, pages 84–100. Springer, 2002.

[48] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Real-time Systems. Springer, 2011. Available from World Wide Web: http://books.google.com/books?id=oJZsvEawlAMC.

[49] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[50] Sï£¡Îl'bastien Lacour, Christian Pï£¡Îl'rez, and Thierry Priol. Generic application description model: Toward automatic deployment of applications on computational grids. In *In 6th IEEE/ACM International Workshop on Grid Computing (Grid2005*. Springer, 2005.

[51] Leslie Lamport. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978. Available from World Wide Web: http://doi.acm.org/10.1145/359545.359563.

[52] Leslie Lamport. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978. Available from World Wide Web: http://doi.acm.org/10.1145/359545.359563.

[53] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.

[54] Tianchao Li and Toni Bollinger. T.: Distributed and parallel data mining on the grid.

In *Proc. 7th Workshop Parallel Systems and Algorithms*, page 2003.

[55] Yu-Cheng Norm Lien and Wen-Jong Wu. A lexical database filter for efficient semantic publish/subscribe message oriented middleware. In *Proceedings of the 2010 Second International Conference on Computer Engineering and Applications - Volume 02*, ICCEA '10, pages 154–157, Washington, DC, USA, 2010. IEEE Computer Society. Available from World Wide Web: http://dx.doi.org/10.1109/ICCEA.2010.185.

[56] Joseph Loyall, Marco Carvalho, Douglas Schmidt, Matthew Gillen, Andrew Martignoni III, Larry Bunch, James Edmondson, and David Corman. Qos enabled dissemination of managed information objects in a publish-subscribe-query information broker. In *SPIE Defense Transformation and Net-Centric Systems*, 2009.

[57] Joseph Loyall, Marcos Carvalho, Douglas Schmidt, Matt Gillen, Andy Martignoni III, Larry Bunch, James Edmondson, and David Corman. Qos enabled dissemination of managed information objects in a publish-subscribe-query information broker. In *Proceedings of the SPIE Defense Transformation and Net-Centric Systems Conference*, 2009.

[58] Joseph Loyall, Matt Gillen, Aaron Paulos, Larry Bunch, Marcos Carvalho, James Edmondson, Douglas Schmidt, Andy Martignoni III, and Asher Sinclair. Dynamic policy-driven quality of service in service-oriented information management systems. *Component and Service-Oriented Distributed Embedded Real-Time Systems, Special Issue on Software: Practice and Experience*, 41(12):1459–1489, 2011.

[59] Joseph P. Loyall, Matthew Gillen, Aaron Paulos, Larry Bunch, Marco Carvalho, James Edmondson, Pooja Varshneya, Douglas C. Schmidt, and Andrew Martignoni III. Dynamic policy-driven quality of service in service-oriented systems. In *In Proceedings of the 13th International Symposium on Object/Component/Service-oriented*

Real-time Distributed Computing (ISORC âĂŹ10, 2010.

[60] Joseph P. Loyall, Matthew Gillen, Aaron Paulos, James R. Edmondson, Pooja Varshneya, Douglas C. Schmidt, Larry Bunch, Marco M. Carvalho, and Andrew Martignoni. Dynamic policy-driven quality of service in service-oriented systems. In *Proceedings of the International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 1–9, 2010.

[61] Mamoru Maekawa. A n algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3:145–159, May 1985. Available from World Wide Web: http://doi.acm.org/10.1145/214438.214445.

[62] Andrew Stephen McGough, Asif Akram, Li Guo, Marko Krznaric, Luke Dickens, David Colling, Janusz Martyniak, Roger Powell, Paul Kyberd, and Constantinos Kotsokalis. Gridcc: real-time workflow system. In *Proceedings of the 2nd workshop on Workflows in support of large-scale science*, WORKS '07, pages 3–12, New York, NY, USA, 2007. ACM. Available from World Wide Web: http://doi.acm.org/10.1145/1273360.1273362.

[63] Marshall Kirk McKusick and Sean Quinlan. Gfs: Evolution on fast-forward. *Queue*, 7:10:10–10:20, August 2009. Available from World Wide Web: http://doi.acm.org/10.1145/1594204.1594206.

[64] Xiangzhong Meng and Baoye Song. Fast genetic algorithms used for pid parameter optimization. In *Automation and Logistics, 2007 IEEE International Conference on*, pages 2144 –2148, aug. 2007.

[65] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[66] K. Nahrstedt, Dongyan Xu, D. Wichadakul, and Baochun Li. Qos-aware middleware

for ubiquitous and heterogeneous environments. *Communications Magazine, IEEE*, 39(11):140 –148, nov 2001.

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

[68] William R. Otte, Douglas C. Schmidt, and Aniruddha Gokhale. Towards an Adaptive Deployment and Configuration Framework for Component-based Distributed Systems. In *Proceedings of the 9th Workshop on Adaptive and Reflective Middleware (ARM '10)*, Bengaluru, India, November 2010.

[69] G. Pardo-Castellote. OMG data-distribution service: architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003.*, volume 0, pages 200–206, May 2003. Available from World Wide Web: http://dx.doi.org/10.1109/ICDCSW.2003.1203555.

[70] Milenko Petrovic, Ioana Burcea, and Hans-Arno Jacobsen. S-topss: Semantic toronto publish/subscribe system. In *IN: PROC. OF CONF. ON VERY LARGE DATA BASES*, pages 1101–1104, 2003.

[71] Vivien Quéma, Roland Balter, Luc Bellissard, David Féliot, André Freyssinet, and Serge Lacourte. Asynchronous, hierarchical, and scalable deployment of component-based applications. In *Proceedings of Second International Working Conference on Component Deployment*, pages 50–64, Edinburgh, UK, May 2004.

[72] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24:9–17, January 1981. Available from World Wide Web: http://doi.acm.org/10.1145/358527.358537.

[73] Youssef Ridene, Nicolas Belloir, Franck Barbier, and Nadine Couture. A DSML for Mobile Phone Applications Testing. In *proceedings of 10th Workshop on*

Domain-Specific Modeling in SPLASH 10th Workshop on Domain-Specific Modeling in SPLASH, page nc, France, 10 2010. Available from World Wide Web: http://hal.archives-ouvertes.fr/hal-00530082/en/.

[74] Douglas C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *Proceedings of the* 12^{th} *Annual Sun Users Group Conference*, pages 214–225, San Jose, CA, December 1993. SUG.

[75] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *FASE*, 2006.

[76] Luciano Serafini, Alex Borgida, and Andrei Tamilin. Aspects of distributed and modular ontology reasoning. In *International Joint Conferences on Artificial Intelligence*, pages 570–575, 2005.

[77] Luciano Serafini and Andrei Tamilin. Drago: Distributed reasoning architecture for the semantic web. In *Extended Semantic Web Conference*, pages 361–376, 2005.

[78] M. Singhal. A dynamic information-structure mutual exclusion algorithm for distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 3:121–125, January 1992. Available from World Wide Web: http://portal.acm.org/citation.cfm?id=628899.629072.

[79] V. Subramonian, G. Deng, C. Gill, J. Balasubramanian, L.J. Shen, W. Otte, D.C. Schmidt, A. Gokhale, and N. Wang. The design and performance of component middleware for QoS-enabled deployment and configuration of DRE systems. *The Journal of Systems & Software*, 80(5):668–677, 2007.

[80] Yang suk Kee, Ken Yocum, Andrew A. Chien, and Henri Casanova. Improving grid resource allocation via integrated selection and binding. In *In Proceedings of the*

ACM/IEEE Conference on Supercomputing, 2006.

[81] Gianluca Tonti, Jeffrey M. Bradshaw, Renia Jeffers, Rebecca Montanari, Niranjan Suri, and Andrzej Uszok. Semantic web languages for policy representation and reasoning: A comparison of kaos, rei, and ponder. In *International Semantic Web Conference*, pages 419–437, 2003. Available from World Wide Web: http://www.informatik.uni-trier.de/~ley/db/conf/semweb/iswc2003.html#TontiBJMSU03.

[82] John Tufarolo, Jeff Nielsen, Susan Symington, Richard Weatherly, Annette Wilson, and Timothy C. Hyon. Automated distributed system testing: designing an rti verification system. In *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future - Volume 2*, WSC '99, pages 1094–1102, New York, NY, USA, 1999. ACM. Available from World Wide Web: http://doi.acm.org/10.1145/324898.325003.

[83] Andrzej Uszok, Jeffrey M. Bradshaw, Matthew Johnson, Renia Jeffers, Austin Tate, Jeff Dalton, and Stuart Aitken. Kaos policy management for semantic web services. *IEEE Intelligent Systems*, 19:32–41, July 2004. Available from World Wide Web: http://dx.doi.org/10.1109/MIS.2004.31.

[84] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of mpi applications with umpire. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society. Available from World Wide Web: http://portal.acm.org/citation.cfm?id=370049.370462.

[85] Abdul Waheed and Diane T. Rover. A structured approach to instrumentation system development and evaluation. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.

Available from World Wide Web: http://doi.acm.org/10.1145/224170.224271.

[86] Abdul Waheed, Diane T. Rover, and Jeffrey K. Hollingsworth. Modeling, evaluation, and testing of paradyn instrumentation system. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society. Available from World Wide Web: http://dx.doi.org/10.1145/369028.369065.

[87] Jinling Wang, Beihong Jin, and Jing Li. An ontology-based publish/subscribe system. In Hans-Arno Jacobsen, editor, *Middleware 2004*, volume 3231 of *Lecture Notes in Computer Science*, pages 232–253. Springer Berlin / Heidelberg, 2004.

[88] J. White, D.C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated diagnosis of product-line configuration errors in feature models. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 225–234, sept. 2008.

[89] Marek Wieczorek, Radu Prodan, and Thomas Fahringer. Scheduling of scientific workflows in the askalon grid environment. *SIGMOD Rec.*, 34:56–62, September 2005. Available from World Wide Web: http://doi.acm.org/10.1145/1084805.1084816.

[90] Ming Xiong, Jeff Parsons, James Edmondson, Hieu Nguyen, and Douglas Schmidt. Evaluating technologies for tactical information management in net-centric systems. In *Proceedings of the Defense Transformation and Net-Centric Systems conference*, 2007.

[91] Ming Xiong, Jeff Parsons, James Edmondson, Hieu Nguyen, and Douglas Schmidt. Evaluating technologies for tactical information management in net-centric systems. In *Proceedings of the Defense Transformation and Net-Centric Systems Conference*,

2007.