Approximation Techniques for Maintaining Real-time Deployments Informed by User-provided Dataflows Within a Cloud

James Edmondson, Aniruddha Gokhale, Douglas Schmidt
Dept of EECS, Vanderbilt University
Nashville, TN 37212, USA
{james.r.edmondson,a.gokhale,d.schmidt}@vanderbilt.edu

***Abstract*—Distributed applications are increasingly developed by composing many participants, such as services, components, and objects. When deploying distributed applications into a mobile ad hoc cloud, the locality of application participants that communicate with each other can affect latency, power/battery usage, throughput, and whether or not a cloud provider can meet service-level agreements (SLA). Optimization of important communication links within a distributed application is particularly important when dealing with mission-critical applications deployed in a distributed real-time and embedded (DRE) scenario, where violation of SLAs may result in loss of property, cyber infrastructure, or lives.**

To complicate the optimization process, the underlying cloud environment can change during operation and an optimal deployment of the distributed application may degrade over time due to hardware failures, overloaded hosts, and other issues that are beyond the control of distributed application developers. To optimize performance of distributed applications in dynamic environments, therefore, the deployment of participants may need adapting and revising according to the requirements of application developers (*e.g.*, how they inform the cloud of important connections between participants within their distributed application) and the resources available in the underlying cloud environment.

This paper present two contributions to the study of dynamic optimizations of user-provided deployments within a cloud. First, we present a dataflow description language that allows developers to designate key communication paths between participants within their distributed applications. Second, we describe and empirically evaluate heuristics that use this dataflow representation to identify optimal configurations for initial deployments and/or subsequent redeployments within a cloud. We motivate our contributions with a distributed real-time and embedded cloud of airborne drones to highlight the applicability of our solutions and validate our techniques with experiments on simulated network infrastructures of a wireless ad hoc cloud.

***Index Terms*—heuristics; genetic algorithms; clouds optimization; real time; constraint problems**

I. Introduction

Enterprise distributed real-time and embedded (DRE) systems are mission-critical applications that run in networked processes across heterogeneous architectures under stringent timing requirements and scarce resources [2]. Though enterprise DRE systems were originally associated with avionics, manufacturing, and defense applications, they increasingly focus on a broader class of distributed applications where the right answer delivered too late becomes the wrong answer. Information in DRE systems must therefore be delivered according to stringent quality-of-service (QoS) needs, despite failures and resource limitations [17], [20], [16].

Unlike some enterprise cloud-based applications—that deal with service-level agreement (SLA) violations with small surcharges to the cloud infrastructure provider—mission-critical enterprise DRE systems cannot tolerate unresponsiveness. Recurring SLA violations may thus result in financial loss and even deaths. DRE systems often require continuous human vigilance to maintain appropriate end-to-end QoS. Moreover, cloud environments do not optimize distributed deployments according to user-defined application dataflows between important participants. To enable next-generation cloud environments to support DRE applications, therefore, they need the following capabilities:

1) A means to specify key communication paths within a distributed application to inform the underlying cloud of what participant interactions should be optimized.
2) Heuristics for optimizing distributed application participant pathways that are identified as important by the user or a monitoring system that informs the cloud infrastructure of heavily utilized pathways.

This paper describes extensions to the *Multi-Agent Distributed Adaptive Resource Allocation* (MADARA) [6], [7] open-source multi-agent middleware, which provides adaptive deployment tools to support next-generation cloud computing capabilities for DRE systems. We have enhanced MADARA to provide a dataflow description language that allows developers to designate key communication paths between participants within their distributed applications. MADARA now also provides genetic algorithms and heuristics that use this dataflow representation to identify optimal configurations for initial deployments and/or subsequent redeployments within a cloud using real-time latency information. In addition, MADARA now provides developers with methods for aggregating latency information via summations of latencies along important paths in the dataflow, which is useful for other approximation techniques that require similar aggregations of latency.

The remainder of this paper is organized as follows: Section II presents a search-and-rescue scenrio that motivates the need for the specification and heuristics added to MADARA;

Section III describes the dataflow description specification and heuristics MADARA uses to minimize end-to-end latency in a DRE application dataflow within a cloud; Section IV analyzes the results of experiments that evaluate how well the MADARA guided genetic algorithms and heuristics approximate a user workflow; Section V compares MADARA with related work on approximation techniques; and Section VI presents concluding remarks.

II. Motivation Example

To motivate the need for MADARA, this section presents a scenario that occurs during a search-and-rescue mission where multiple government agencies utilize a cloud of remote-controllable drones within a disaster area. Figure 1 shows this disaster recovery scenario, where remote-controllable drones have been deployed to search for survivors in an earthquake-ravaged metropolitan area. The application dataflow shown

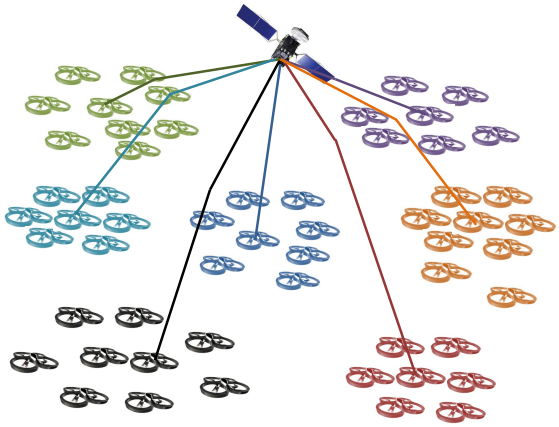

Fig. 1. Motivating Search-and-Rescue Application Scenario

in this figure show segregated groups of remote-controlled drones in the search-and-rescue mission communicating via satellite with human controllers. Due to the destruction, human 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 between humans and the drones searching for signs of life.[1]

Human controllers can thus only maintain communication with a small subset of the drones, called *collector* drones. Each drone has onboard sensors that may allow it to detect radiation, record video, observe and report atmospheric anomalies, detect thermal signatures, and other useful functions, and regardless of which government agency leased time on the drone cloud, each of the automated participants aids in searching for survivors. Each drone is also equipped with a wireless access point that allows it to form/join ad hoc networks and transmit sensor readings, images, or other data.

[1]For simplicity, this figure only shows dozens of drones, but we anticipate thousands of drones being deployed in future scenarios, depending on the size of the search area and the availability of inexpensive, commercially-available remote-controlled drones, such as the Parrot AR.Drone.

With all secondary functions (*e.g.*, radioactivity detection) turned on, the drones will quickly run out of power. Moreover, the faster the drones power down, the less survivors that will be found and useful work accomplished. Data communication is a particularly expensive operation that quickly drains batteries. The closer two drones are to each other, however, the lower the latency and the less data resends required across the communication, which extends battery life.

Each government agency may have its own satellites and secondary functions it is interested in. Application dataflows are therefore defined to reflect (1) the agencies funding (*e.g.*, more funding generally means more drones allocated) and (2) the path that information will make within the allocation, which may reflect a specific flying or roaming formation. This latter refinement means that the application dataflow is arbitrary and may not reflect the Area Coverage Problem [1], [19], [14], [8], which is commonly solved with sensor networks.

An example application dataflow is shown in Figure 2. The data shown in this figure enables two radiation detectors and has two collector drones communicating with human controllers via satellite links. Each edge in an application dataflow shown in this figure is equally important, *i.e.*, no edge is more important than any other edge that is defined in the application dataflow. The absence of an edge means that the datapath is unimportant or at least that no attempt should be made to minimize latency along that datapath. We also assume that each edge is utilized equally (*i.e.*, no edge is monopolizing traffic unevenly), and this caveat helps us to simplify the optimization problem: *the best deployment will be the one that has the lowest total latency summed across all edges*.

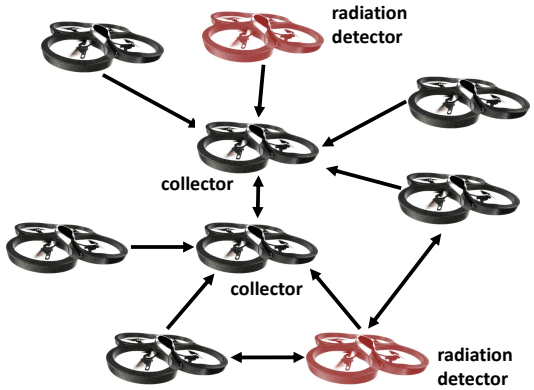

Fig. 2. Application Dataflow Example

As drones move around the area, the optimal deployment of DRE applications that are running on the drones may become outdated. A redeployment of the DRE application may therefore be necessary to ensure these collector drones are in range of their group within the dataflow. 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, as well as draining precious battery life. 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 resources may be lost. For these reasons, the time required for calculating the redeployment should be minimized—preferably a handful of seconds or less.

In summary, this motivating DRE application described above has the following requirements:

1) Users must be able to define a flexible deployment dataflow for thousands of drones, which also helps make the techniques described in this paper applicable to other enterprise-scale cloud applications deployed on commercial cloud providers, such as Amazon's EC2 service. Section III-A describes a dataflow description specification used by MADARA to designate key communication pathways.
2) Algorithms for approximating the dataflows against the current network conditions must be able to execute quickly (sub-second runtime is preferred for DRE systems) and failure to do so may result in loss of drones or survivors due to faster battery depletion. Sections III-C, III-D, and III-E describe the heuristics MADARA uses to approximate these dataflows.
3) Any implemented deployment solution suggested to the drones for (re)deployment should result in a noticeable performance increase in the network. Section III-A describes the process of preparing underlying cloud environment latency information for heuristics and the target conditions for redeployment.

III. Approximation Techniques in MADARA

This section describes genetic algorithms and heuristics provided by MADARA to approximate an optimal enterprise DRE application deployment under different constraints. We developed multiple solutions due to memory limitations imposed by different 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. These heuristics can also be run on all hosts in the cloud or on specific hosts, such as collector drones or a master host.[2]

A. Defining the Dataflow and Identifying Degrees

MADARA optimizes DRE application dataflows from a graph perspective. In particular, it encodes a user-defined deployment dataflow 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 incident on 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.

[2]This paper does not specify how cloud hosts agree on a redeployment and assume a distributed voting protocol is used to determine redeployment thresholds (which is how we implement redeployment agreement in MADARA.)

TABLE I
Dataflow Description for Motivating DRE Application

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

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

To show how a degree is imparted from an application dataflow, Table I depicts an actual dataflow description file for our motivating application in Section II, which consists four collector drones, each gathering messages from a quarter of the drone population. The MADARA dataflow description language provides a mapping of directed edges and is ideal for specifying large ranges of values, which maps well to cloud environments. The simplest dataflow description involves a source mapped to a range of destination participants, which are processing elements capable of executing a component or service of a distributed application.

For instance, participant 0 in the first line of Table I has important edges from itself to participant 0 to size / 4, where '[' denotes inclusiveness and ')' denotes non-inclusiveness. Instead of a single participant id, the source participant in the dataflow description language can be a range of IDs, *e.g.*, [0, size/4] → [0, size/4) indicates that important edges exist between each participant in one-fourth of the available participants in the cloud. The number of participants available per host can be potentially infinite, but for DRE systems it should ideally map to the number of processors available or less if threads of execution should be available for certain system threads at all times.

From the dataflow description in Table I, we can make the following observations. 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 3 services 3-6, drone 6 takes care of 6-8, and drone 9 handles information to and from 9-11. This MADARA deployment specification interface addresses requirement 1 of the motivating DRE application in Section I by providing users a flexible mechanism for specifying a deployment dataflow in a DRE system.

Figure 3 visualizes what a degree in a graph is by labeling the high degree nodes in a user-provided DRE application dataflow. The node with a degree of seven has seven 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 I, our solution techniques are relevant to approximating component placement, optimal resource monitoring, routing, and other problems involving connected graphs.

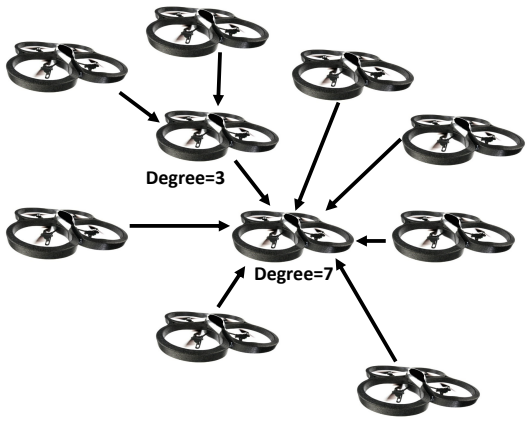

Fig. 3. Degrees in a User-provided DRE Application Dataflow

B. Preparing the Data

Before presenting MADARA's 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.

Algorithm 1 Prepare

Require: DRE application dataflow 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 dataflow **do**
5: utilities[degree][source] ← $\sum_{j=1}^{degree}$ 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 dataflow **do**
11: utilities[degree] ← $\sum_{j=1}^{degree}$ latencies[j]
12: **end for**
13: **end if**
14: sort_ascending(utilities)

Each utility list entry in Algorithm 1 is the sum of the best latencies in the underlying cloud per degree in a user-provided dataflow. This sum is used to fill in participant IDs by an approximation heuristic. After the data is ready, it is disseminated to interested parties in the network, *e.g.*, the drones or a special collector of this type of information—perhaps a command-and-control output.

The 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 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). Another 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. This 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 dataflow. 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 described above is fully parallelizable since 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 IV.

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 dataflow.

There is a separate preparation step of sorting the user-provided DRE application deployment graph by degree, which helps Algorithms 2, 3, and 5 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 dataflow by degree takes just nanoseconds to a few microseconds for 10,000+ node dataflows. It is therefore a negligible portion of the time needed to perform the approximation (the project site at madara.googlecode.comcontains

complete code examples).

C. Degree-based Heuristics in MADARA

Two heuristics are discussed below, 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.

Algorithm 2 CID Heuristic

Require: Call Prepare (Algorithm 1)

1: **for all** node $\in$ dataflow **do**
2: **if** degree (node) $>$ 0 **then**
3: solution[node] $\leftarrow$ best_candidate (utilities[degree(node)])
4: **end if**
5: **end for**
6: **for all** node $\in$ DRE application dataflow **do**
7: **if** degree (node) $>$ 0 **then**
8: **for** neighbor $\in$ connections(dataflow, node) $\wedge$ neighbor $\notin$ solved(solution) **do**
9: solution[neighbor] $\leftarrow$ best_candidate (latencies[node])
10: **end for**
11: **end if**
12: **end for**
13: **for all** node $\in$ DRE application dataflow $\wedge$ node $\notin$ solved(solution) **do**
14: solution[node] $\leftarrow$ best_candidate (utilities[size])
15: **end for**

Algorithm 2 shows how the CID Heuristic begins by iterating over the deployment and placing candidates based on lowest latency available in the cloud for the degree. See Lines 2-7 of Algorithm 1 that construct the utilities list. This list provides presorted summed latencies per degree.

The latencies list is a sorted list of latencies between all participants. Thus, latencies[node] is the list of latencies involving a certain node. 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 dataflow).

The final phase of the CID heuristic (lines 13-15) deals with nodes that are not connected to the rest of the DRE application dataflow. For example, this phase could be used 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 is shown in Algorithm 3.

Algorithm 3 Blind CID

Require: Call Prepare (Algorithm 1)

1: **for all** node $\in$ dataflow **do**
2: **if** degree (node) $>$ 0 **then**
3: solution[node] $\leftarrow$ best_candidate (utilities[degree])
4: **end if**
5: **end for**
6: **for all** node $\in$ dataflow **do**
7: **if** degree (node) $>$ 0 **then**
8: **for** neighbor $\in$ connections(deployment, node) $\wedge$ neighbor $\notin$ solved(solution) **do**
9: solution[neighbor] $\leftarrow$ best_candidate (utilities[size])
10: **end for**
11: **end if**
12: **end for**
13: **for all** node $\in$ dataflow $\wedge$ node $\notin$ solved(solution) **do**
14: solution[node] $\leftarrow$ best_candidate (utilities[size])
15: **end for**

The Blind CID heuristic 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 and may not find the optimal deployment, which results in less battery life, longer latencies, and more resends of important information.

A 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. In contrast, the Blind CID heuristic only uses aggregation of this knowledge (the utilities list that we obtain from Algorithm 1). The Blind CID heuristic does use deployment information in the dataflow to prioritize which node of the dataflow 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 the Blind CID heuristic is that the drones need not send their individual latency values to other drones that must make 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. Sending fewer messages increases battery life for all participants in the dataflow.

D. Genetic Algorithms in MADARA

Not all DRE application dataflows can be solved optimally by the heuristics described in Section III-C. The CID and BCID heuristics are tailored to solving certain types of dataflows like acyclic collector drones and not more complex dataflows like hierarchical or cyclic dataflows. For more complex dataflows, a randomized search technique may be more appropriate.

To complement the heuristics discussed in Section III-C, we therefore 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—Guided GA shown in Algorithm 4—is guided with degree information.

Algorithm 4 Guided GA

```
Require: Call Prepare (Algorithm 1)
mutations ← min + rand() % (max - min)
orig_utility ← utility(new)
for i → mutations do
    new ← solution
    if rand() % 5 < 4 then
        c1 ← random_degreed_node (dataflow)
        c2 ← location(new[good_candidate(utilities)])
        while c1 ≡ c2 do
            c2 ← location(new[good_candidate(utilities)])
        end while
    else
        c1 ← rand() % size
        c2 ← rand() % size
        while c1 ≡ c2 do
            c2 ← rand() % size
        end while
    end if
    if utility(new) < orig_utility then
        solution ← new
    end if
end for
if utility(solution) < orig_utility then
    return solution
end if
```

The Blind GA Algorithm 5 does not use degree information to mutate solutions and instead uses pure randomness when selecting solution chromosomes to mutate.

Before describing the Guided GA Algorithm 4 and Blind GA Algorithm 5 solutions we briefly describe what constitutes a mutable chromosome in the deployment. Each of these algorithms considers a chromosome as a mapped participant of the final deployment solution. For instance, if a user-provided dataflow contained five participants, then five chromosomes would exist in the solution list and the genetic algorithms will attempt to optimize the deployment by mutating chromosomes until a time limit is reached. The best generated solution that contained the lowest summed latency according to the edges in the user-provided dataflow would be returned by these genetic algorithms as the solution list (this list is actually a vector in MADARA for performance reasons).

Both algorithms select chromosomes (*i.e.*, nodes/drones) of the proposed solution (the approximated deployment) to mutate and then perform mutations for a specified time interval or number of allowed mutations before returning the best solution (either the original or the improved solution). The Guided GA in Algorithm 4, however, targets the higher degreed nodes 80% of the time and selects from the best available participants in the underlying cloud, which allows it to make more intelligent mutations by targeting highly degreed chromosomes more often. While the Guided GA does converge much more quickly than the Blind GA, the randomness inherent in the Blind GA can be better for the hybrid approaches we discuss in Section III-E below.

Algorithm 5 Blind GA

```
Require: Call Prepare (Algorithm 1)
mutations ← min + rand() % (max - min)
orig_utility ← utility(new)
for i → mutations do
    new ← solution
    c1 ← rand() % size
    c2 ← rand() % size
    while c1 ≡ c2 do
        c2 ← rand() % size
    end while
    swap(new[c1], new[c2])
    if utility(new) < orig_utility then
        solution ← new
    end if
end for
if utility(solution) < orig_utility then
    return solution
end if
```

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

E. Hybrid Approaches in MADARA

A guided genetic algorithm need not be directly codified with degree information, as we did with Guided GA Algorithm 4. We can also seed the Guided GA algorithm with heuristic results to help local searches converge much faster than they might have otherwise. We therefore combine the two heuristics in Section III-C with each genetic algorithm presented in Section III-D to produce four methods: (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).

IV. Experimental Validation of the Heuristics

This section analyzes the performance of—and utility produced by—the MADARA algorithms and heuristics described in Section III. We used two types of metrics for our experiments: (1) *runtime*, which evaluates the time required to approximate an optimal large-scale deployment dataflow within a simulated mobile DRE adhoc cloud with varying

latencies between hosts and maps directly to requirement 2 of the motivating DRE application in Section II and (2) *system slowdown*, which evaluates the runtime performance of the resulting deployment after a redeployment occurs and maps to requirement 3 of the motivating DRE application.

A. Experiment Setup

The first experiment creates 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 DRE cloud. Consequently, system slowdown will be high (*i.e.*, performance will be poor) if the heuristic does not find the optimal deployment.

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

We examine the second experiment in three configurations—two collector drones that communicate with size/2 local participants, three collector drones that communicate with size/3 local participants, and four collector drones that with size/4 local participants. These tests expect that the guided heuristics and algorithms will far outperform the unguided ones. The motivating DRE application favors techniques that put as little strain on the CPU as possible to conserve battery, so smaller runtimes are preferred to allow for longer drone uptime.

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, which drain a battery more significantly than an optimal deployment.

All experiments were repeated ten times and the averages are reported. Each experiment was conducted on an Intel Core2 Duo clocked at 2.53 GHz and 4 GB of RAM running Windows 7 32-bit operating system. We allow the heuristics to run on this processor configuration with virtual latencies that mimic real-world large-scale cloud infrastructures of 1,000 to 10,000 hosts and then run the heuristics on these virtual cloud environments. The C++ code was compiled in MS Visual Studio 2008 under the optimized release mode. Code for all the experiments and the configuration information is available from the MADARA project site at madara.googlecode.com.

B. Analysis of Results

Below we analyze the experimental outcomes in regards to runtime and system slowdown, which reflect requirement 2 and 3 from the motivating scenario in Section **??**, respectively.

Runtime. Figure 4 depicts the runtime performance of the heuristics in approximating deployments for the first experiment, the scenario where only one good solution exists and every other solution is highly suboptimal. Figure 5 depicts the runtime performance of heuristics in approximating deployments for the second experiment, where thousands of good approximations exist but only one is optimal.

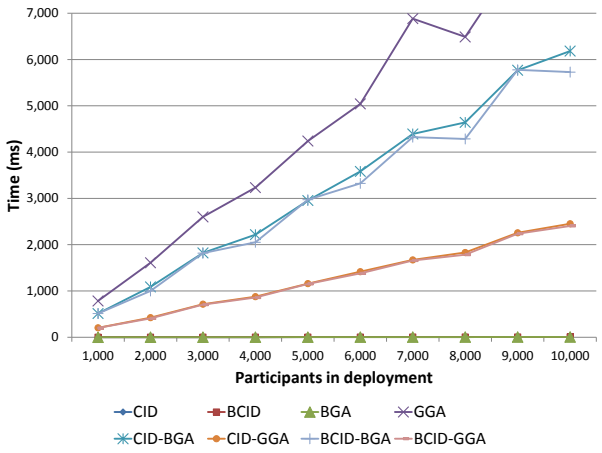

Fig. 4. Runtime Required Under the First Experiment

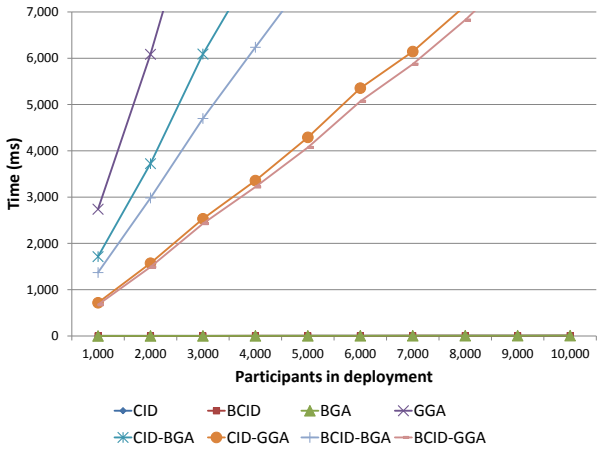

Fig. 5. Runtime Required Under the Second Experiment

Running CID and BCID alone requires roughly 18-20ms to approximate dataflows of 10,000 participants in the various configurations noted in Section IV-A. The genetic algorithms are anytime algorithms that can be given a timed run. To show the runtime differences between these algorithms, we allowed them both to perform 500 mutations of up to 10 chromosome changes. The guided genetic algorithm has more intelligence, so it takes longer to perform these mutations. As show below, however, it can find better solutions more quickly than the blind genetic algorithm.

It takes the genetic algorithms much longer to perform the same number of mutations in experiment 2, shown in

Figure 5. This increase in time required to approximate is caused by the increased frequency of finding better solutions and performing deep copies of these new deployments (*e.g.*, line 19 of Algorithm 4). When there are fewer good solutions to find, the genetic algorithms tends to perform mutations much faster because it rarely has to perform these expensive deep copies of the current best deployment.

The final analysis we make for runtime performance concerns the latency preparation defined in Algorithm 1. The Prepare algorithm is required by each heuristic to sum latencies by degree before the heuristics can approximate deployments from the underlying cloud. Figure 6 shows the runtime required on each host that runs this algorithm.

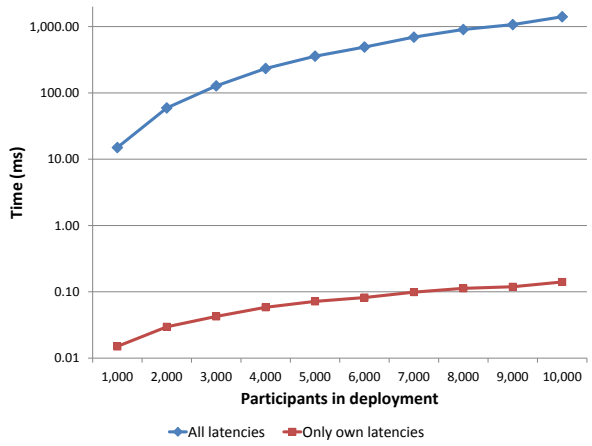

Fig. 6. Preparation Runtime for 4 Specialized Drones in a Noisy Environment

As the number of participants (*i.e.*, along the x-axis) increases, the runtime also increases, but more importantly, the runtime to prepare all latencies on each node becomes prohibitive. By requiring each host to aggregate its own latency information by the degrees in the user-provided dataflows, we reduce the runtime of the Prepare algorithm from seconds to microseconds for a 10,000 participant deployment.

System slowdown. System slowdown is a factor of worse overall performance within the distributed application according to the important paths defined in the dataflow description. The lower the slowdown, the longer the battery life of the mobile DRE cloud and the better latency and throughput will be along important paths. We measure system slowdown as the total latency along important links, as this maps to the conditions we outlined in the motivating scenario—namely that no important path is being used more frequently than any other and each are weighted equally important.

In the experiments described below, a random deployment exists for the distributed application before heuristics are applied. The system slowdown is the resulting performance of the deployment as a factor of the optimal performance available in the underlying cloud environment.

In the first experiment only one optimal deployment has microsecond latencies within the cloud environment that mimic pathways in the defined dataflow of four collector drones communicating with four equally sized groups of participants, as shown in Figure 7. This figure shows how CID finds the

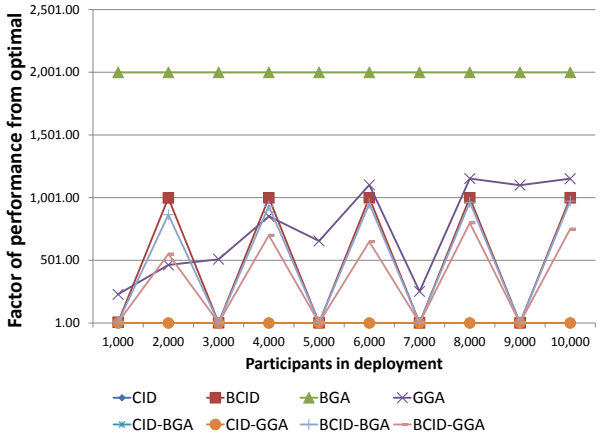

Fig. 7. System Slowdown under the First Experiment

optimal deployment every time, whereas the less-informed BCID heuristic sometimes finds the optimal solution, but often does not. The guided genetic algorithm (GGA) also finds the optimal solution occasionally, but the blind genetic algorithm (BGA) never finds an optimal solution and the resulting system deployment performance is poor. When GGA and BGA are seeded with the results of CID and BCID, they quickly find much better solutions.

Figures 8, 9, and 10 highlight system slowdown performance in the second experiment. When many good approx-

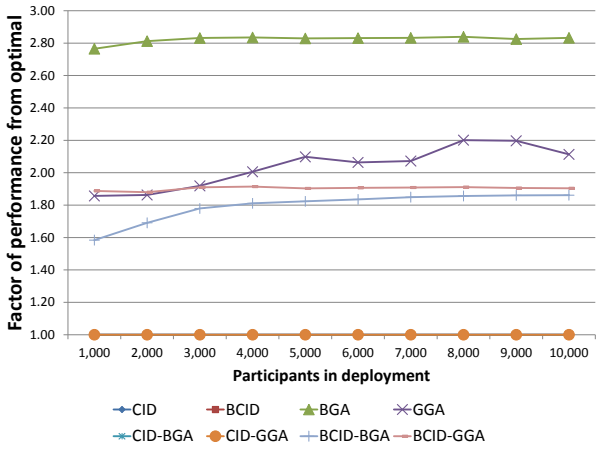

Fig. 8. System Slowdown with 2 Specialized Drones in a Noisy Environment

imations exists, the CID heuristic still always finds optimal solutions. BCID and GGA both find excellent alternatives, which is an important point because only the BCID heuristic requires O(M) memory. All other heuristics require fully-informed hosts of each other's latencies. If cloud infrastructure providers can be reasonably certain that many thousands of

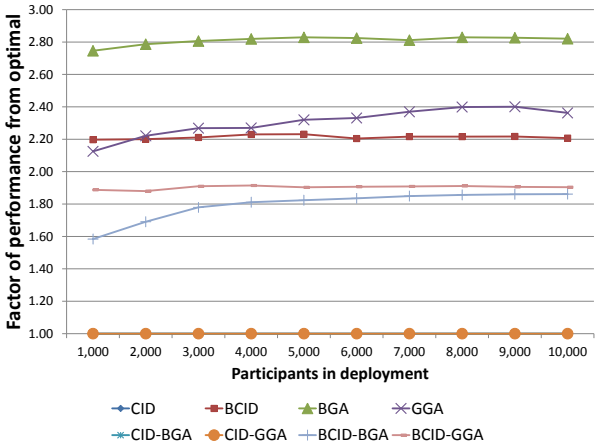

Fig. 9. System Slowdown with 3 Specialized Drones in a Noisy Environment

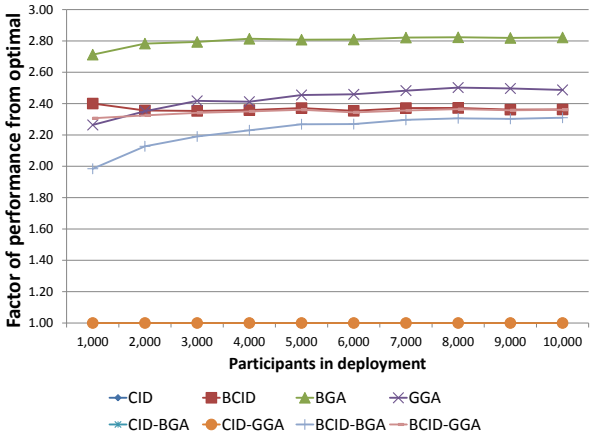

Fig. 10. System Slowdown with 4 Specialized Drones in a Noisy Environment

good solutions exist, then BCID can be a powerful approximator.

The BCID-BGA hybrid performed well in this second experiment and benefited from the selection of good generic candidates for the collector drones and blind mutations of the other lower-degreed nodes. GGA mimics the intelligence of the CID and BCID algorithms, so its performance in combination with them is lower because it continues to try to change the highly-degreed nodes, even though CID and BCID already made excellent guesses concerning the appropriate participant-to-host mappings within the DRE cloud. This result helps create better guided genetic algorithms and random search techniques that are specifically tailored for being seeded with excellent approximations from CID and BCID.

V. Related Work

This section compares our work on MADARA with related work on deployment problems based on constraint satisfaction problem solving, genetic algorithms, and heuristics.

Constraint satisfaction problem solving. Haldik et. al. [11] presents a constraint programming technique to solve static allocation problems in real-time tasks. Cucu-Grosjean et. al. [4] propose two approaches to addressing real-time periodic scheduling on heterogeneous platforms, which is a constraint satisfaction problem (CSP). 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 II that exhibits thousands of constraints defined on the deployment between the collection drone and its group.

White et. al. [21] 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 II requires runtime solutions within milliseconds or at most seconds. Section III 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 an optimal match through mutations, fitness functions, and crossbreeding best candidates according to the fitness criteria. Heward et. al. [10] 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. [22] use a genetic algorithm to schedule scientific dataflows 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 [15] and even knowledge and reasoning [12] 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 [13], [5], [18] and our on experience with the MADARA guided genetic algorithms (via heuristics) described in Section III, 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 [10] in grids and dataflow solutions [3] for real-time solutions. The latter is of interest to us since the heuristic approximates a constraint problem involving a set

of dataflows within milliseconds. The solution [3], 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 [9] 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.

VI. Concluding Remarks

Enterprise DRE systems are increasingly essential in mission-critical domains, such as aerospace, defense, telecommunications, health care, and financial service. This paper presented two heuristics provided in MADARA to approximate user-provided dataflows in next-generation DRE clouds. We also presented MADARA's genetic algorithms and hybrids of the heuristics and genetic algorithms to improve the solutions generated by the heuristics. We analyzed the results of experiments to validate the MADARA heuristics and genetic algorithms, as well as highlighted issues with unguided genetic algorithms in a representative DRE application context.

The following are a summary of lessons learned from our work on MADARA :

- The CID heuristic can produce optimal results (lowest possible aggregate latency for a DRE application dataflow) in a variety of useful scenarios like a centralized or common broker, failover servers, collectors, and broadcasters. This heuristic can be used to produce effective solutions within microseconds for developers who need to optimize these types of deployments online for enterprise DRE systems.
- Pairing an unguided genetic algorithm with breeding candidates from fast heuristic approaches shows promise to create targeted guided heuristics. If a purely random mutating scheme can improve results by 10-20% under noisy conditions, a more intelligent version may produce even better system utility in fewer generations, which is an important goal for enterprise DRE system developers.
- Our experiment results showed that degree-based heuristics (such as CID and BCID) are less effective at generating solutions for hierchical tree graphs and complex dataflows. To address this issue, our future work will develop better guided genetic algorithms and local searches that cater to these types of DRE application dataflows. We will release these techniques within the MADARA project to aid enterprise DRE system developers.

C++ code for the MADARA heuristics and algorithms is available in open-source form from madara.googlecode.com.

References

[1] K. Arisha, M. Youssef, and M. Younis. Energy-aware TDMA-based MAC for sensor networks. *IEEE IMPACCT*, pages 21–40, 2002.

[2] J. Balasubramanian, S. Tambe, B. Dasarathy, S. Gadgil, F. Porter, A. Gokhale, and D. 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.

[3] T. Cucinotta and G. 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.

[4] 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.

[5] 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.

[6] J. Edmondson and A. 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)*.

[7] J. Edmondson, A. Gokhale, and S. Neema. Automating testing of service-oriented mobile applications with distributed knowledge and reasoning. In *Proceedings of the Service-Oriented Computing and Applications (SOCA)*.

[8] J. Elson and D. Estrin. Sensor networks: a bridge to the physical world. pages 3–20, 2004.

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

[10] G. Heward, J. Han, J.-G. Schneider, and S. 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.

[11] P.-E. Hladik, H. Cambazard, A.-M. Daplanche, and N. Jussien. Solving a real-time allocation problem with constraint programming. *Journal of Systems and Software*, 81(1):132–149, 2008.

[12] Y. Hu and S. 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.

[13] L. Ingber and B. Rosen. Genetic algorithms and very fast simulated reannealing: A comparison. *Mathematical and Computer Modelling*, 16(11):87 – 100, 1992.

[14] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

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

[16] J. S. Kinnebrew, W. R. Otte, N. Shankaran, G. Biswas, and D. 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, Mar. 2009.

[17] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Real-time Systems. Springer, 2011.

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

[19] A. Sinha and A. Chandrakasan. Dynamic Power Management in Sensor Networks. *Smart dust: sensor network applications, architecture, and design*, page 1, 2006.

[20] V. Subramonian, G. Deng, C. Gill, J. Balasubramanian, L. Shen, W. Otte, D. 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.

[21] J. White, D. 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.

[22] M. Wieczorek, R. Prodan, and T. Fahringer. Scheduling of scientific workflows in the askalon grid environment. *SIGMOD Rec.*, 34:56–62, September 2005.