**DMake: A Tool for Monitoring and Configuring Distributed Systems**

Theodoros Gkountouvas  
Cornell University  
Ithaca, NY, USA  
tedgoud@cs.cornell.edu

Kenneth Birman  
Cornell University  
Ithaca, NY, USA  
ken@cs.cornell.edu

David Anderson  
Washington State University  
Pullman, WA, USA  
daveanderson@wsu.edu

**ABSTRACT**

Designers of long-running distributed systems must define management policies to address the various runtime disruptions that can arise in distributed settings. In many applications, notably ones requiring high-assurance guarantees, this management infrastructure must be flexible enough to represent non-trivial requirements and dependencies, and poised to quickly adapt if changes in the behavior of the system require an intervention. Our belief is that with the emergence of cloud computing and a very large scale “migration” of applications towards cloud hosting, a growing community of non-experts will find themselves developing such applications, and that software tools oriented towards this sort of cloud user will be required.

Our paper describes DMake, a new system responding to this need. DMake is unusual in offering a very flexible range of management options through a familiar and widely popular model: that of the Unix make utility. DMake represents a significant generalization of Unix make, and its extended “makefile” format can express complex policies. The system has a rigorous semantics, including strong consistency and reliability properties that map to distributed consensus (state machine replication). We believe that DMake opens the door to much more sophisticated management policies by users with relatively little specialized training in distributed computing or system management.

**Keywords**

Distributed Systems, Management, Monitor, Configuration, Make, DMake

1. **INTRODUCTION**

Management of autonomous applications is classic problem for which a variety of solutions have been created, including some extremely sophisticated ones. For example, many readers will be familiar with dashboard interfaces for initial deployment and resource allocation for managing managed distributed applications on platforms like Amazon’s elastic compute cloud (EC2) [1], where the “Elastic Beanstalk” [19] service is used to automate load-balancing, fail-over and several other features: one selects the desired configuration and then fills in a parameters page. In other settings one finds far more elaborate domain-specific tools that provide users with the ability to employ highly customized policies, but these often have specialized models that the typical developer might find unfamiliar and hard to use. For example, cloud developers at companies like Google, Microsoft or Amazon have access to elaborate internal management infrastructures, with features going well beyond what Elastic Beanstalk or similar APIs provide. Our work begins with the observation that there is surprisingly little available in the “middle” of the spectrum: tools allowing customization of policies but that are easy to use and aimed at the large community of non-expert developers.

DMake is a new management tool that automates enforcement of user-defined management policies, mapping the management problem to a more familiar paradigm: that of constructing a system from components in a manner controlled by a dependency graph represented by a structure similar to a Makefile [2, 9]. The intent is to provide developers with much more control than is possible with tools like Elastic Beanstalk, yet without requiring them to become domain experts in the area of cloud-hosted distributed system management.

The policies provided to DMake take the form of application-specific dependency graphs, and the basic DMake model is one in which events trigger reactive actions, a structure that is also used in Makefiles. Predefined events include failures of computer nodes or failures of application processes which are detected from DMake, but the system is easily extended to sense any change in the state of an application process provided that there is some way to track that state (our preferred approach is to just have the application update status files, which use an XML data representation and can encode whatever data the developer wishes to track). DMake’s management cycle starts when some form of management event occurs. The system captures the associated information (or aggregates information from multiple locations), and then enforces the management policies according to the dependency graph specified by the current Makefile, which triggers the creation of an updated configuration for the application. This configuration is then employed by the system to adjust system functionality in any desired manner: appli-
Management of wide-area bulk power networks poses challenges. One must deploy instrumentation (so-called synchrophasor measurement units, or PMUs) on the power busses with adequate redundancy to properly capture grid state. Bulk power systems are operated by regional management entities called independent service operators (ISOs), and these need to collaborate and coordinate cross-regional power flow. ISOs increasingly need to integrate renewable power into their networks (large wind farms, solar, etc) and these sources can fluctuate, so the ISO must be prepared to balance any variability using other sources of power, perhaps imported at long distances. Disruptions can sweep through the grid much like a wave propagating on a water surface, and the ISO must watch for such events and take remedial actions in real-time to prevent damage or disruptive outages. Further, there is a need to plan ahead so that if power demand surges due to weather conditions or other unusual events, the generating capacity will be prescheduled and available to respond to that demand.

Not surprisingly, the machinery of the smart grid translates to a fairly large amount of software that involves many co-operating subsystems, each consisting of multiple programs and data sources. Overall, one has to manage the PMU deployment and the network connecting the PMUs to the data collectors (a network that would often include PMU data concentrators as a kind of relaying and control component). In our work on GridCloud, the network terminates within the Amazon EC2 cloud in a bank of data collectors which shard the data, so that any given PMU is redundantly monitored and archived. Then we can run applications such as state estimation on the data, and these applications also involve multiple programs running on multiple nodes. There are many such applications, some running continuously (if these are disrupted by some sort of EC2 elasticity event, we need to immediately repair the configuration to restore full service within a few seconds), and others on demand (i.e. if an operator poses a “what if?” question, or launches some sort of rarely needed program to investigate a new development).

![Figure 1: GridCloud as Motivation for DMake](image)

Cations can be stopped or started, entire nodes shut down, new parameter files can be written, and many more changes are supported. Interaction with active applications is accomplished by writing XML files that the application reads (an interrupt or notification event can also be sent to the application, if desired).

The structure of our paper is as follows. We start by briefly motivating the need for a new kind of management solution in the sections that immediately follow. The architecture of DMake is described in Section 4 and Section 5 includes the optimizations made to improve the performance of DMake. We report the evaluation results of DMake at Section 6. We summarize topics for future research in Section 7 and in Section 8, we give more details on prior work related to DMake.

2. CLOUD MANAGEMENT

2.1 Motivation: GridCloud

Our work started as a side-effect of setting out to create a new kind of cloud computing platform: GridCloud, which is a hosted environment for the so-called Smart Power Grid [13]. Very briefly, this term refers to a new class of computing systems that capture information about the state of a power grid and then use machine learning or optimization techniques to efficiently deliver electric power to the home, industry, and other kinds of infrastructures like urban street lighting systems. The power grid is hierarchical, with a bulk power infrastructure that handles high-tension lines at long distances and a collection of regional distribution systems that deliver power to the home over lower tension networks. We’re looking at transmission and distribution, but the work that led us to develop DMake is concerned with the bulk power network (Figure 1).

The critical nature of bulk power delivery brings the obvious requirements: extremely strong security (all stored or transmitted data is encrypted), tightly controlled sharing for information moved from ISO to ISO or to the various distribution network operators, real-time data delivery obligations, consistency, redundancy for archived historical records, geographic redundancy for the system as a whole, etc. The main elements of these applications are software systems created by the same experts who have operated the ISO’s control systems for decades: normal developers who use normal computing tools and have extensive experience reasoning centers on economics and is out of our scope here, but in short, the cloud offers the scaling these systems require (PMU deployments will someday be national in scale: a full US deployment could someday include 100,000 PMUs, each generating 44 bytes of data 30 times per second). Obviously any single ISO would need just some of this data, but the volume would be considerable; the cloud fits this model today, while other approaches might require substantial and costly new data centers. Thus it is appealing to at least consider using existing cloud systems, operated by vendors that the power grid community considers adequately trustworthy.

Why host a smart-grid control system on the cloud? The reasoning centers on economics and is out of our scope here, but in short, the cloud offers the scaling these systems require (PMU deployments will someday be national in scale: a full US deployment could someday include 100,000 PMUs, each generating 44 bytes of data 30 times per second). Obviously any single ISO would need just some of this data, but the volume would be considerable; the cloud fits this model today, while other approaches might require substantial and costly new data centers. Thus it is appealing to at least consider using existing cloud systems, operated by vendors that the power grid community considers adequately trustworthy.
In effect, one can then leverage the huge existing investment that went into creating the cloud and today sustains it.

2.2 The management challenge
As this background should make clear, our work is motivated by the emergence of a new kind of cloud computing application. In the past, most cloud applications were built using platform as a service (PaaS) solutions: customizable platforms that support some style of cloud computing popular (and highly monetized) by the Web. Popular examples include Google AppEngine [24], Microsoft Azure [3], etc. Such systems allow the user to build a wide variety of self-managed cloud-hosted functionality, but they also constrain the developer to a certain style of computing that on the whole rejects strong guarantees in favor of weaker forms of eventual consistency. Further, these kinds of assumptions are often baked deeply into not just the application development tools, but also the associated management infrastructures.

But GridCloud is just one of many platforms that departs from those PaaS models by using the cloud as an infrastructure (IaaS), importing potentially complex functionality that brings an application into a cloud setting for which much stronger requirements arise. GridCloud illustrates many of these: it must operate in a highly autonomous manner, and because of the EC2 sharing elasticity model, events such as failures, activity migration or transient load spikes might be common. For example, our effort emerges from a a community developing cloud-hosted smart power grid applications. In the past, power grid state estimation (SCADA) and analytic systems ran mostly on dedicated HPC clusters. The cloud is not a particularly good match for these kinds of systems at present, but has such strong cost advantages, and brings such exciting new capabilities (particularly with respect to elastic computing and hosting large data sets) that work aimed at overcoming barriers to adoption seems appealing. DMake is a contribution aimed at this need.

Our assumption is that the DMake user community will be united by a number of features. First, just as in the power grid example, they will be a community with extensive existing code bases but motivated to migrate toward the cloud because of a new need to leverage elasticity (pay for what you use, when you need it), to reduce cost of ownership, and to leverage cloud-scale data collection and machine-learning algorithms to optimize power delivery. Indeed, for certain kinds of large-scale uses, the cloud seems to be the only computing model that one could reasonably consider. Yet such communities also bring a very real form of inertia. The very nature of the process forces developers who have long been comfortable with a dedicated Linux server model of computing to suddenly confront platforms like EC2 or Eucalyptus, which is a situation very different from the experience of a developer working at a company like Google or Amazon on applications created with the cloud in mind from the outset. Moreover, again using GridCloud as our example, because the existing power grid still centers on human control, these developers are not merely moving applications into an unfamiliar setting, but are interconnecting them to control centers staffed by operators who bring all sorts of human-factor considerations into the mix, while also extending applications that in the past might have been used mostly in offline settings to play real-time, infrastructure-critical roles.

Notice the complex mix of needs that now arises. Our goals include robustly capturing data from a wide area in real-time, performing an optimization task that might lead to actions such as directly controlling switchable bulk power lines, or responding to a transient disruption, and we need to carry out these tasks on a cloud infrastructure that might be reconfigured dynamically. The system will need to react to these events, yet was built by a developer who is familiar with a standard style of legacy “owned” HPC computing, and who needs to reuse the large body of pre-existing code and solutions used in that setting. Thus, control actions that in the past would have occurred offline and with ample time will now need to be performed from the cloud and in real-time, using automated solutions that must remain continuously active. Control mishaps could cause costly damage to physical infrastructures, like large transforms or wind generating systems, which could take years to repair or replace. In summary, we see a mix of needs that diverge in many ways from those of the more typical cloud services operated in support of mobile apps or social networking. Further, this need is not unique to the smart power grid community. Similar trends are occurring in health-care, transportation control (especially smart vehicles that use cloud-hosted helper services), smart buildings and cities, and many more settings.

3. MANAGEMENT AS MIDDLEWARE
Our DMake solution operates as a middleware layer, resident between the application itself and the management hooks that can be used to directly access application state or modify application configuration (see figure 2), representing policies as dependency graphs, an approach very similar to Makefile. As seen in the figure, management of a distributed system using DMake involves a) monitoring of the state of the system b) taking management decisions that depend on the monitored state and c) configuring the system in order to comply with these decisions. DMake presumes that this procedure should be automatic.

Figure 2: DMake as Middleware

There are many management tools in the same domain as DMake, but we believe that DMake is the first to combine dynamic management of applications with strong guarantees, the ability to define rich set of management policies and the ease of usage afforded by our decision to mimic the
make utility. More specifically, some of these tools [15, 23] focus on deploying the application to a distributed environment and on allocating resources to application processes and thus, they offer limited management choices. Other solutions [4, 5], provide dynamic management but require human intervention to manage the distributed system when something unexpected happens, rendering them more suitable for debugging. Several tools [11] are designed for management of specific types of systems and do not target a wide range of distributed systems applications. Finally, there are tools [11, 18] designed for only monitoring, and that do not link monitoring and management of the system.

We believe that implementation of user-defined management policies is significantly simplified by separating it from the managed application code. In this we are inspired by trends in networking, where researchers are demonstrating major advantages to Software Defined Networks [21] in which the control plane (where management decisions are made) is decoupled from the data plane. Indeed, we increasingly view DMake as a tool that might, with further evolution, be usable in SDN settings; they are, after all, large distributed systems with distributed control needs, triggered by events, and subject to a variety of constraints. But the SDN community is not the only one to have successfully separated management from the application itself. For example, the Hadoop platform [20] has been hugely successful. In this system, a master node orchestrates the worker nodes according to management policies provided by the user. And the SWIFT scripting technology [22, 25] is similarly popular for control of semi-autonomous HPC applications, which are often developed on dedicated clusters but then operated in very large scale shared ones.

In contrast, DMake targets applications that will be wired directly to the real world, capturing data in real-time, processing it rapidly and delivering results to human operators who need to see system state within seconds, and need strong guarantees of security and consistency when they share information across regional boundaries. The requirements thus take us in a direction very different from those of these prior solutions, which generally have a strongly “batch computing” feel to them, or in the case of today’s cloud computing control systems, are mostly aimed at servers that are far less interdependent and interconnected than in a system like GridCloud. For example, with Elastic Beanstalk, Amazon will dynamically vary the pool of first-tier servers, and then encourages a style of coding in which those servers can be launched or killed as needed, without loss of information (they store data in S3, Dynamo or Dynamo-DB [16], or even Zookeeper [10]). This is very different from what a community like our target one would be used to. DMake aims at a developer who has absolutely no intention of reimplementing something unexpected happens, rendering them more suitable for debugging. Several tools [11] are designed for only monitoring, and that do not link monitoring and management of the system.

Thus, application state should be able to include anything which this is done by writing information into an application-state file. Much as for the application-state file from some PMU sensor to a different data collector node, thus could avoid much of the associated overhead. Consequently review network latencies and sense process failures, which this is done by writing information into an application-state file. Much as for the application-state file. For example, data-streaming applications may need to frequently review network latencies and sense process failures, reacting by quickly rerouting data to a different data collection node so as to avoid loss of data. In this case, the frequency at which DMake checks for updates should be set to a relatively small number in order to avoid disruptions to the application itself. However, rapid polling consumes CPU time and frequent checks (<200ms) can have a significant impact in the performance of the application (especially in computer nodes with a low number of CPUS). In contrast, an application that does not need such active management can afford a more relaxed frequency of status checking, and thus could avoid much of the associated overhead. Continuing with our example: clearly if we decide to reroute traffic from some PMU sensor to a different data collector node, that data collecting process needs to be told of its new role. This is an example of a dynamic configuration task. We address it by having DMake update the relevant configuration files, again by writing information in an xml format that the application can scan. Much as for the application-state file, but now with information flowing in the opposite direction.

4. ARCHITECTURE
The deployment of DMake is conceptually simple, as seen in Figure 3. A DMake process should be instantiated on every computer node in the system, along with the executables for whatever application processes the user wishes to deploy. DMake itself is launched first, and as will be explained in more detailed shortly, it will then launch the application processes that the current configuration requires on the node in question. Because they are children of DMake, these application processes can be monitored and configured by the DMake daemon.

We needed a simple way for application processes to share their states with DMake, and settled on an approach in which this is done by writing information into an xml file. Thus, application state should be able to include anything the user needs to monitor for configuring the whole application. DMake augments application-supplied state with additional state it captures from the operating system. DMake periodically checks if the state has changed, under a refresh policy the developer can control (frequency of checking, event-driven or time-driven, etc).

For example, data-streaming applications may need to frequently review network latencies and sense process failures, reacting by quickly rerouting data to a different data collection node so as to avoid loss of data. In this case, the frequency at which DMake checks for updates should be set to a relatively small number in order to avoid disruptions to the application itself. However, rapid polling consumes CPU time and frequent checks (<200ms) can have a significant impact in the performance of the application (especially in computer nodes with a low number of CPUS). In contrast, an application that does not need such active management can afford a more relaxed frequency of status checking, and thus could avoid much of the associated overhead. Continuing with our example: clearly if we decide to reroute traffic from some PMU sensor to a different data collector node, that data collecting process needs to be told of its new role. This is an example of a dynamic configuration task. We address it by having DMake update the relevant configuration files, again by writing information in an xml format that the application can scan. Much as for the application-state file, but now with information flowing in the opposite direction,
We see this pattern in Figure 4. For each managed subsystem, information captured from the local nodes where that subsystem has components is transferred to the state machine of leader replicas. The replicas share state information periodically, we see these as acceptable obligations to impose. DMake doesn’t require any particular coding style, language, or even any particular speed of reaction, and because there are powerful existing xml file I/O tools for every language, the developer shouldn’t find it particularly difficult to carry out these new tasks. In contrast the developer who decides to build an application using an existing cloud platform would be urged to work in a completely new style that would typically be best suited to creation of a completely new style application, from the ground up.

As each cloud node is launched, its local DMake daemon connects itself to other active DMake processes form a DMake group, using a software library that addresses these group formation and messaging tasks (see Figure 4). They also elect a DMake leader that takes the role of the coordinator in the system. The DMake leader collects all the state and initiates management actions according to the user’s policies. The DMake leader itself is a replicated state machine: one should understand it as a virtual process, operating in a subgroup of the full DMake deployment with a size that can be customized by the user depending on how critical the application is. For applications that do not require high assurance and are mostly concerned with performance, the size can be one. Loss of the leader need not be catastrophic: if it fails, the system remains unmanaged for a short period of time, after which DMake would detect that it has lost its leader (consensus) and automatically launch a new leader. The new leader queries all the DMake processes for the current state and after it pulls all the state it starts functioning and the system is no longer unmanaged.

On the other hand, for applications like GridCloud that need continuous management and cannot tolerate even brief loss of the leader role, we simply replicate the leader. The group acts as a single entity, and if one of its members fails, the others continue to act without even a brief loss of continuity. The group management library we’ve selected, Isis$^2$ [6, 7], was explicitly designed for cloud settings; it handles group formation and provides the communication and dynamic reconfiguration mechanisms described above, offering strong consistency and reliability guarantees.

The use of Isis$^2$ (isis2.codeplex.com) simplifies DMake in many ways, bringing fault-tolerance and consistency in the form of a strong execution model: virtual synchrony [8]. In this model, members of a process group see the identical events in the identical order, including membership changes triggered by process or node failures, which are automatically sensed by Isis$^2$. The library also handles bootstrapping, and will automatically configure itself to use the best available communication option (UDP only, UDP + IPMC, RDMA over Infiniband, etc), provides multicast flow control and ordering, and includes features for high speed “out of band” file transfer. The consistency features of the platform help ensure that DMake’s management actions are consistent, as well.
mation: files, stored within folders (one copy per replica) with naming conventions used to distinguish the managed entity from which the state was captured, and one file per instance in the case of managed entities that can themselves be replicated over a set of nodes. An application consisting of multiple subsystems might instantiate this pattern multiple times, once per subsystem, or it could use a single overall configuration management infrastructure covering all the subsystems at once; as we will see, the choice would generally depend upon the kinds of configuration policies desired for the application (e.g. in an application that needs one B server for every N members of the A service, one would want both managed by a single DMak instance; if A and B were completely independent, one might prefer to also manage them independently).

Figure 5 explains how the files are stored in the leader replicas. DMak creates a root directory directly under the directory it operates. Additionally, under the root directory, folders with the names of the nodes inside the DMak group are created. These names can be customized by the user as well during the start-up of DMak or during the execution (changes in node names). If no name is given, DMak employs the DNS hostname or IP address. Below them, the application processes folders are created along with the corresponding state and configuration files.

In addition to the the application status files, DMak uses files to report other kinds of management events. When a new node joins the DMak group a folder is created to represent its state, and the event is reported under it. Similarly, when a failed node is detected the event is reported under the corresponding folder and later it is deleted along with the folder. A failed node event is triggered when a DMak process fails. Notice that because of network partitioning events that can arise even within a cloud computing system (we have seen such events within Amazon EC2), it might be the case that the node has not failed and the application processes is actually still functioning, although DMak has lost the ability to monitor and configure it. We adopt the view that an uncontrollable application is a failed application, because we view active control as a part of application health. Accordingly, a DMak agent that loses connectivity with the rest of the DMak group shuts the node down. DMak can also track other node attributes such as CPU loads, paging rates, etc.

DMak generates a per-node configuration file that it passes to the daemon on the node, which reads the file to learn the processes it should run (or stop), how to find the executables for them, what file names it should monitor to capture application state, application, and how to configure node-level parameters such as environment variables.

The user defines the policies for the managed application using a dependency graph (see Figure 6). This dependency graph turns out to be a valuable source of information; as shown later in Section 5 it can be used for a application optimization and to automate self-repair after a disruptive episode, and even lends itself to a more formal style of analysis that permits us to explore reachable states and to reason rigorously about management semantics. Thus the dependency graph is in a very real sense the core of DMak. We use the term DMakfile to refer to this graph.

The DMakfile scripting language is the same as that used by the Linux make utility [9], and hence can encode simple management logic. However, far more sophisticated management actions can also be defined, by providing user-defined programs that might maintain an internal application model and in the event that the application state changes, initiate reconfiguration or repair actions. Such user-defined management code would be triggered by changes to processes’ state files, events (new node, failed node, failed processes), initial configuration files for the application (instructions about initial setup of the application) or intermediate files that aggregate the state from multiple processes (probably output files for some other policy). Notice that because DMak uses files to represent events, there is a natural match to the Makefile model, in which changes to files trigger rebuilding of dependent files: in DMak, changes in the environment or application state trigger the recomputation of configuration files. A notational extension to the Makefile syntax is used to access information within a file: if X is a file, X{tag} represents the value associated with some xml-tagged element within that file (or the set of values, if the tag is used multiple times).

Since the DMak leader is elected from the DMak group, it is important that any node in the DMak group eligible for this leadership role have a way to access the needed management programs, system state files and other data files. Thus, we need to replicate the associated data across the nodes that might participate in the leader replica group, and this could be costly. Accordingly, DMak includes an option that can be used to restrict the set of nodes that will be considered as candidates for leadership. This narrows the choices for DMak but limits the need for replication to a smaller group.

Outputs of the management computation again take the form of files: these may contain some form of intermediate information, or could be configuration files for delivery to application or DMak processes. Again, DMak actions can
either generate entirely new versions of the needed files, or can update fields within them: $X\{\text{tag}\} = Y$ creates an empty file $X$ if the file doesn’t already exist, then updates the file, setting the xml field with the specified tag to the given value.

The procedure followed when DMake applies the policies in the DMakefile is very similar to the behavior of make. DMake parses the DMakefile to find input files that changed and triggers the corresponding policies. If output files are changed then DMake triggers the policies that contain these files as input files and this action continues until there are no new policies that can be triggered. Indeed, rather than replicate the functionality of Make, DMake actually runs Make: our code is basically a wrapper that creates a context within which the normal functionality of Make can be leveraged to perform distributed management tasks. DMake runs Make whenever it might have an action to trigger: for example, when the state of application processes changes, or when a DMake daemon reports an event (such as new node, failed node, failed process, etc).

Make is not designed for concurrent execution, hence we impose a simple form of locking: While a management policy is actively computing, new state or event updates are not processed. This avoids the risk of erroneous results. When Make finishes, the DMake leader replicates process any updates that were generated during its execution, which could trigger a further execution of Make. Under the state machine replication model we use, leader replicates always process updates in the order they were received: events occur in a total order that also respects the sequencing in which they initially occurred (that is, a total order that is also causal and FIFO). We are thus able to ensure that an old state will never somehow overwrite a new one. Further, since policies can cause changes to the configuration of the application processes being managed, DMake distributes to each of the managed local nodes the corresponding configuration file (if it has changed) in the end of every execution of Make.

Notice that although the DMake leaders perform the same actions in the same order, they might not run in perfect temporal lock-step. Accordingly, when we have more than one leader replica, an update can be initiated earlier (in clock time) by one replica than by some other replica. In order to disambiguate messages from the leaders to the local nodes, we maintain a vector clock structure where the fields correspond to the version of the state file the leader replicas receive from the local nodes. When we run Make, we mark the output configuration files with the value of the vector clock. We piggyback this value to the message that delivers the configuration files to local nodes and they can tell by comparing the values from two different replicas which one is the configuration with the most recent information. The values would be always comparable since replicas process state updates in the same order.

\[
\text{output: input}_1 \text{ input}_2 \ldots \text{ input}_n
\]

\[
\text{policy}
\]

Figure 6: Management Policies Format

There are some inherent limitations with the architecture of DMake. First, while our model is simple, our centralized approach to management implies that if the application being managed is very large or generates very frequent management events, the system could suffer from performance bottlenecks in the leader replicas. The DMake leader replicas have to maintain the state of the whole system, run Make and distribute back the configuration files that changed. As a result, they incur potentially substantial performance overheads, and in a heavily loaded setting the reconfiguration of the nodes will reflect these delays. However, even in Grid-Cloud, a fairly demanding use, our experience suggests that reconfiguration of the system is not very frequent and management generally does not require a level of state or computation that would cause significant delay. Another limitation is that a complex reconfiguration of a system can take a significant amount of time (response time as we refer to it from now on). Applications that require quick reconfiguration (within milliseconds) or applications where the state changes frequently (receive configuration resulting from a different state than the current one) might not be a good match for our approach, since DMake requires a round trip to the leader replicas and a certain amount of delay is thus incurred. The next Section explains how DMake tries to alleviate these limitations.

5. OPTIMIZATIONS

The optimizations in DMake have the purpose of relieving the burden in the leader replicas. First, we batch messages from the local nodes to leader replicas and from leader replicas to local nodes. Messages that originate or target application processes that reside in the same node are similarly batched. Thus, DMake ends up using significantly less network bandwidth, especially for applications that require frequent updates. Additionally, the users may annotate their policies in the DMakefile with a “local” tag. A local policy is one that can be performed without obtaining state from any other node. In such cases, DMake can perform the necessary action without involving the central leader. For example, if the user desires to relaunch an application that fails, DMake can do so with no delay and no communication. Notice, however, that this assumes that the node configuration files are not in any way dependent upon global information that might need to be accessed as a consequence of the failure event. The local tag allows the developer to control the use of this feature, which can be useful in situations where the DMakefile doesn’t encode enough information to correctly sense the presence of a global dependency. When the local optimization can be used, we prevent extra overhead in the leader instances and achieve improved response time, since we avoid the round trip to the leader.

Finally, by annotating policies with different tags the users can instruct DMake to employ distinct leaders for the different policies. The idea is as follows: when tags are used, DMake will check the dependency graph for disjoint management rules, treating the tags as hints but then verifying that there are no cross-tag dependencies. If two policies are tagged using distinct tags and the associated event files are confirmed to be independent (no file is created using information from the other), DMake will employ multiple different leader groups (corresponding to different Isis process groups), one for each distinct tag. This optimization can thus yield a decentralized management behavior, where the overhead of management is spread over multiple replication groups. In effect, disjoint subsystems can thus be managed...
DMake: Dynamic Scheduler
We made a static and a dynamic scheduler for memory intensive workloads. All the tasks require approximately the same amount of resources and require similar time for completion. For the static scheduler we assume that the number of available nodes is known beforehand. Thus, we simply assign to every node its share from the pool of tasks. Since there is no heterogeneity in the jobs or in the hardware we use, this scheduling is highly efficient. Next, we designed a dynamic scheduler with the help of DMake that can perform quite well even in unfavorable conditions. The dynamic scheduler initially assigns a batch of jobs to the nodes that join the group. Then, it monitors them to determine when it should assign to them another batch of works and it continues until all the tasks have finished. The dynamic scheduler has the potential to outperform the static one for cases when the outcome of the execution cannot be easily predicted; our experiment undertook to evaluate this hypothesis.

Figure 7: Normal Operation
In the first experiment (see Figure 7), we run the static and the dynamic scheduler with 30 available nodes and 9000 different jobs. Although, they ended up making very similar decisions (the same number of tasks were assigned to each node), static scheduling slightly outperforms the dynamic one. This happens because the dynamic scheduler needs to keep track of the current state in order to take further decisions while the static one is able to distribute all the tasks at the outset. However, the difference is very small: the number of completed jobs in the dynamic case is within 5% of the number completed in the static system. In effect, although DMake constantly monitors and re-configures the system, this background activity added only a small overhead to the system.

Next we repeated the experiment with 1/3 of the nodes significantly slowed down (artificially) to see how the dynamic scheduler adapts when compared with the static scheduler. Figure 8 shows that the dynamic scheduler maintains a constant rate of tasks completion over time while the static scheduler does not. The dynamic scheduler is able to detect the slow nodes and distributes tasks accordingly. In contrast, the static scheduler distributes tasks evenly over both slow and fast nodes. As a result, when the fast nodes complete their tasks (around 240 seconds from the beginning of experiment), the slow ones still have to do more than half of their tasks (they end up finishing at around 550 seconds). In this case, the dynamic scheduler thus outperforms the static one (327 versus 548 seconds). We would argue that there are actually many situations in which the dynamic scheduler would be a better choice. For example, integration or failure of worker nodes during the execution would be impossible with the naive static approach, since it has no way to recompute its schedule; the dynamic scheduler would adapt and perform well under these conditions. Failure raises similar issues, as do situations in which the nodes are heterogeneous hardware or where it is difficult to predict the completion time of a task, or where tasks will need different resources and their execution time will vary accordingly.

Figure 8: With 33% Slow Nodes
Figure 9 shows the DMake response times for the same experiments, again looking at two scenarios. Here, the term response time denotes the interval between the moment a local DMake process detected a change in the state file of an application processes and the moment it wrote to the corresponding configuration file after receiving a new configuration from the relevant Leader. Notice that state changes do not necessarily lead to a new configuration and we count only the ones that do. The application response time is the time between the actual change of the process state and the enforcement of the configuration that "fixes" the issue caused by it.

As it can be seen in Figure 9, the majority of the DMake
responses are within 0.5 seconds. Given that the average round trip from the DMake Leader to local nodes at the time of the experiment was 0.25 seconds these results seem very reasonable to us (EC2 communication can be surprisingly slow because the nodes are virtualized and often run on shared physical hardware, so there can be long scheduling delays). There are a few cases in which the response time spikes to more than 1 second. Unfortunately, EC2 has poor clock synchronization and offers limited monitoring options, so we are not able to fully isolate the causes. Our belief is that these outliers are caused either by transient network bottlenecks (again, recall that our nodes are shared), or by spikes of state updates that cause delays in the execution of the DMake Leader. However, for this experiment we use a single leader. If we had used a replica group consisting of more than one, our solution would have been more tolerant of late responses since all replicas would initiate the same action, and we would end up measuring the the delay until the first response out of many instead of just one. On the other hand, we would then see slightly increased overhead within the replica group itself, an effect that could slightly increase the response delay. We use the term ‘slight’ because in our experience, Isis² has been quite fast on this platform.

The application response time depends on the frequency with which DMake checks state and configuration files (see Section 4). In this particular experiment, we set the rate to one second. Thus, the application and DMake response times for the same event should not differ by more than two seconds (at most one second for DMake to get the state and at most one second for the process to retrieve the configuration). As seen in Figure 9 this is indeed the case. We can increase the rate to obtain better response times for the processes, paying a mild penalty for the increased demand on CPU resources. However, for this application the 1 second rate is adequate, yielding high performance matched to the experimental setup.

Figure 9: DMake and Application Response Times

It should be stressed that the dynamic scheduler is a dummy application, designed purely to evaluate the performance of DMake. In our view, the experiment represents a good stress test for DMake since it requires constant reconfiguration and has frequent state updates. However, it is atypical in the sense that few real applications would require so demanding and active a form of management: DMake targets a class of uses that would normally involve long-running systems supporting users who need responses within seconds, but not milliseconds.

**DMake: GridCloud**

Next, we experimented using our GridCloud platform, described in Section 2. The experiment used a real deployment of the current version of the system, in which we collect data from simulated Phasor Management Units (PMUs) that are deployed in the Internet so as to replicate conditions seen in real power grids (security concerns that make it difficult to experiment on the actual US power grid, hence these sorts of emulation or simulation studies are the best one can do). Our configuration, however, was extremely realistic, and enabled us to capture data at the proper rates, and then to run a variety of power grid applications on top of that data. After the data is acquired in the front end (Collectors), it is forwarded to the back-end (state estimators). The state estimators collectively calculate the global state of the electric power grid, and forward the data to a final node, where it is rendered for display to the grid operator.

As was mentioned before, GridCloud is the application that motivated DMake, and typical of a larger class of streaming applications where data flows from sensors deployed in the real world into the cloud, is handled by multiple systems on multiple nodes, and finally delivered in real-time to a human user. GridCloud needs strong guarantees about the data (no loss of streams is permitted) and thus, all the processing nodes within our cloud configuration are replicated in order so as to compensate for any possible node failures. Furthermore, GridCloud requires liveness guarantees (the data visualized should correspond to a recent state of the system). The system is thus a real and serious test for DMake. Further, it requires that a variety of management policies be enforced.

The experiment shown below starts with a demanding initial configuration in which we use 24 computer nodes for 381 state estimator processes and 42 computer nodes for 4731 collector processes (a configuration that might correspond to a monitoring setup for the entire Pacific NW). In total we use DMake to configure 72 nodes that run more than 5000 processes. The state estimator configurations here are relatively simple and they do not require sophisticated policies. On the other hand, for the collector processes things get more complicated. We have to enforce application-specific constraints about the mapping of data streams to the appropriate collectors, and we need to simultaneously balance the workload between collector nodes. Our goal is to assign approximately the same number of streams (or collector processes since we have one process per stream) to each of the collector nodes.

Figure 10 shows the amount of time DMake needs to configure the entire GridCloud platform. More specifically, the figure shows what percentage of processes has been configured (and also have provided an acknowledgement to the DMake Leader) after the formation of the group. The first 44 seconds are needed for some initial DMake configuration after the group formation and for enforcing all these sophis-
ticated policies described in the previous paragraph for the 4731 collector processes (Make). The whole platform is configured in approximately 60 seconds resulting in more than 80 processes configurations per second in average. The effects of batching are clear: one can see bursts of actions throughout the whole timeline.

![Figure 10: DMak](image)

We reran the same experiment with the injection of collector process failures (1/4 of the processes fail shortly after they are configured). In this experiment, our policy for handling these failures is to rerun the collector with the same configuration, a policy that can be carried out locally in each node (see Section 5). Thus, process failures are handled within 200ms, which is the monitoring cycle (every 200ms DMak locally searches for failed processes). This permits us to completely mask the failures: in the eyes of our human operators, the resulting run is not distinguishable from the failure-free one. Note, however, that this class of failures is easy to detect, because the node itself remains healthy and hence the DMak daemon can directly sense the crashes.

We also have policies for node failures. In case a collector node fails, we look for other available collector nodes that are not fully loaded and distribute the workload to them according to our load balancing policies, somewhat like the dynamic scheduler experiment. If there are no available collectors which also satisfy the constraints GridCloud imposes, our current policy is to wait until some node becomes available (it would not be hard to build a fancier one in which we would ask EC2 to launch extra nodes). If a state estimator node fails, we stop collector processes that transfer data to it and then wait for an available state estimator node to take on the unassigned workload. But here failure detection is a much harder problem: the DMak daemon becomes unresponsive, but on EC2, such events are common and Isis\(^2\) has a long built-in hysterisis to reduce the risk of false detections. Once failure detection actually occurs, the DMak response time is very short. We did not include data for this case (although we have tested it): the resulting graph is dominated by the failure detector delay parameter used by Isis\(^2\) and hence is not a very interesting case for evaluation of DMak.

7. FUTURE WORK

As future work, we would like to explore the idea of integrating formal methods in distributed computing management. By doing so, we hope to be able to describe desired application properties (for example, that a system always creates one data collector for each data stream), and use formal methods to confirm that the policies applied to the underlying application satisfy the objectives. Similarly, it may be of interest to explore the space of states the application could enter, and the trajectories employed to recover from undesired states; one could flag as incorrect a policy that might leave the system permanently in an undesired state.

Another interesting topic is to create a management framework for multiple users. Recall that GridCloud is intended as a shared management and monitoring infrastructure that might be used by many operators, each of whom might launch applications as needed. This creates the need for a collaboration platform within which we may need to confirm that different users use compatible policies, and more generally to explore the question of whether policies might interact with each other in an undesired manner. Indeed, we see the lack of a framework like ours as one major barrier to the emergence of solutions for the smart grid and similar applications.

8. RELATED WORK

Early work\[12, 14, 17\] focuses on management policies, how they should be defined by users and how conflicts are solved when these policies are provided by different entities. This work emphasizes the need for dynamic management and automatic enforcement of the policies to the underlying systems. They envision policies as hierarchies where the low-level policies are generated automatically from high-level user defined policies, sharing the same perspective as DMak that policies should be easily defined by users. Furthermore, they discuss about conflicts in management policies posed by different users and they propose an authorization scheme in order to solve them.

Rhizoma\[23\] is a more recent work with very similar architecture as ours. In Rhizoma, the computer nodes form a communication group and they elect a coordinator amongst themselves. Similar to DMak, the coordinator is not a single point of failure since if it fails a new coordinator would be elected from the rest of the group. The coordinator collects all the monitored data and applies the corresponding actions to the rest of the system. Actions in Rhizoma are defined by the users in the form of a constraint program (using constraint logic programming). However, Rhizoma is more focused on deployment of the application and resource allocation. Thus, the only actions that are allowed is adding or removing new nodes, limiting significantly the management options of the user.

Plush\[4, 5\] is another tool focused more on the deployment and the debugging of a distributed systems application. Plush uses a primary-backup scheme for controllers following a slightly different approach than Rhizoma and DMak. Again, all the monitored data is collected to the controllers and Plush can take actions according to the users input. However, Plush requires human intervention to take actions and reconfigure the system, rendering it inappropri-
ate for applications where reaction to certain events in a minimum amount of time is critical. Thus, it is more appropriate for debugging or for applications where a quick response to state changes is not crucial. Also, users need to become familiar with the concept of blocks that Plush utilizes in order to define their policies, rendering Plush difficult to use.

Other works in this domain [11, 18] aim to monitor efficiently the underlying applications. Monitor of the applications enables dynamic management of the system, but these tools deal solely with monitoring and they treat the management of the system as an independent procedure that needs to be done.

9. CONCLUSIONS
Our work on the smart grid led us to recognize that existing cloud management tools are inadequate in many ways. While very suited to the control of highly scalable, stateless, first-tier cloud applications that store data deeper in the cloud, such tools offer very little to developers who are faced with migrating applications from dedicated cluster environments onto cloud platforms. Such developers will often have existing code that needs to run in new settings, and that must be closely monitored and managed to maintain availability. The DMake system addresses this requirement, using a familiar paradigm (Make and the associated Makefile model), but augmenting it to play a management role in which distributed events are mapped to file updates.

Our evaluation shows that DMake can handle some very demanding, large-scale use cases. While we see a number of topics for future work, the existing system is already capable of managing GridCloud, a complete system for management of the bulk power grid, and could easily be adapted to other applications with similar needs. The solution is open source and will be available for no-fee download from our development website by mid 2014.

10. ACKNOWLEDGEMENTS
The authors are grateful to Robbert van Renesse, Carl Hauser, Dave Bakken and Nate Foster for their help and suggestions. We also wish to acknowledge our funding sources: the DOE/ARPA-E GENI program, DARPA’s MRC program, the NSF Smart Grid program and Amazon EC2.

11. REFERENCES
