# Distributed Recovery for Enterprise Services

Shane S. Clark Raytheon BBN Technologies Cambridge, MA, USA 02138 Email: sclark@bbn.com Jacob Beal Raytheon BBN Technologies Cambridge, MA, USA 02138 Email: jakebeal@bbn.com

Abstract-Small- to medium-scale enterprise systems are typically complex and highly specialized, but lack the management resources that can be devoted to large-scale (e.g., cloud) systems, making them extremely challenging to manage. Here we present an adaptive algorithm for addressing a common management problem in enterprise service networks: safely and rapidly recovering from the failure of one or more services. Due to poorly documented and shifting dependencies, a typical industry practice for this situation is to bring the entire system down, then to restart services one at a time in a predefined order. We improve on this practice with the Dependency-Directed Recovery (DDR) algorithm, which senses dependencies by observing network interactions and recovers near-optimally from failures following a distributed graph algorithm. Our Java-based implementation of this system is suitable for deployment with a wide variety of networked enterprise services, and we validate its correct operation and advantage over fixed-order restart with emulation experiments on networks of up to 20 services.

#### I. INTRODUCTION

While a range of pragmatic approaches have been deployed for managing both individual machines and large-scale datacenters, management of small- to medium-scale enterprise systems has generally remained much more primitive (Figure 1). Such systems, supporting a variety of enterprise services, can often have quite complex architectures, but are often too small to justify the time and financial investment necessary to integrate and maintain the sort of sophisticated failover mechanisms used by large enterprises, such as enterprise virtualization or replication products. Similarly, they are at least as likely, if not more so, to contain hard-to-replace legacy components with peculiar requirements and to suffer from incomplete documentation or lack of understanding by system administrators. As a result, these smaller enterprise systems tend to fall back on process documentation, manual interventions, and patchwork do-it-yourself automation, and have a corresponding tendency to suffer much more disruptive and extensive outages than large enterprise systems: a recent survey [1] shows that small enterprises (<100 employees) suffer more than 50% more downtime and 5 times more relative financial impact than large enterprises, while medium enterprises (100 to 1000 employees) are even more badly impacted, with nearly 5 times the downtime and 25 times the financial impact.

One approach to addressing this challenge is through selfadaptation, by creating system-management tools that selfcustomize to work with ill-documented, poorly understood, and changing networks of services. This paper aims to address one critical aspect of system management: recovery from service failures. Recovery in enterprise service networks is Partha Pal Raytheon BBN Technologies Cambridge, MA, USA 02138 Email: ppal@bbn.com



Fig. 1. Small- to medium-sized enterprises often have complex networks with many services and servers, but are not large enough to have significant administrative resources to devote to customization or to benefit from economies of scale.

complicated by the dependencies (sometimes undocumented) between services. Restarting a service before its dependencies restart can often result in a new failure, so typical current practice for small- to medium-scale enterprise systems is to bring the entire system down and restart each service sequentially in a fixed "known safe" order of restarts.

To address this challenge, we have developed the Dependency-Directed Recovery (DDR) algorithm, which infers dependencies from network interactions and automatically coordinates service restarts based on these observed dependencies. While considerable prior work addresses the issue of dependency detection via network observations [2], [3], [4], this prior work focuses primarily on the mechanism for detection rather than the application of that information to service management. We thus begin with the assumption that one can observe network interactions for dependency information and build from that point to a self-stabilizing algorithm for distributed recovery that near-optimally coordinates service restarts without any need for specialization to a particular enterprise system.

We have implemented the DDR algorithm within the Java-based Protelis architecture [5], creating a self-adaptive management framework suitable for immediate use with many enterprise services, including legacy systems. Deploying this framework requires only connecting each existing service with Java or system hooks for observation and restart. For the small-to medium-sized enterprises we are targeting with this work, these requirements are likely modest compared to the time and financial costs of re-architecting applications or integration with large-scale enterprise solutions. Finally, we validate the efficacy of our distributed recovery framework by comparing it to fixed-order restart on three classes of emulated networks

with up to twenty services per network, finding that our approach recovers at near-optimal speed and with many fewer services affected.

## A. Related Work

While there are many monitoring tools to assist with detecting failed services and managing dependencies between services, to the best of our knowledge no prior method provides a practical end-to-end solution that includes both knowledge of dependencies and automation-assisted recovery.

On the simpler side of prior approaches, a monitoring tool such as mon [6] or Nagios [7] can be combined with custom logic to handle dependencies and restart, or a system administrator can build their own custom watchdog scripts. These "doit-yourself" approaches, however, do not provide dependency inference or adaptive execution, leaving the specification and maintenance of recovery scripts entirely in the hands of a system's human operators. A number of more sophisticated tools allow an administrator to manage a collection of machines by enforcement of set policies. Two popular examples are Puppet [8] and Chef [9], both of which use agents with optional server components to provision and manage hosts, including the states of services and the order in which those services are started. While these configuration management systems do provide per-host management, they assume that each host operates independently and do not offer any ability to coordinate management of interdependent hosts.

At large scales, modern datacenters frequently integrate open source or commercial offerings that use redundancy as a way to mitigate failures, rather than explicitly managing dependencies. For example, one straightforward approach for stateless web servers is to maintain multiple running copies behind a failure-aware load balancer, such as HAProxy [10]. Load balancers are more difficult to use with databases or other stateful services because they do not manage data consistency. Typical approaches for databases include multimaster or master-slave clustering, which are transparently tolerant to node failures but require multiple database hosts regardless of load (e.g., [11], [12]). Other commercial products include host-level mechanisms for failure recovery such as the fault tolerance and high availability mechanisms in VMware's vSphere [13], which maintain "shadow copies" of running VMs that can be promoted to active use in the event of failure, along with automatic VM monitoring and restart functions. All of these tools offer monitoring and/or restart functionality but none of them actually address the problem of dependency management. Instead, they treat each component in isolation and attempt to use replication and rapid failover between physical hosts to prevent the "virtual component" from ever failing. These large-scale solutions also generally require specific infrastructure support and spare hardware resources that are often not readily available for smaller enterprises.

With regards to adaptively identifying dependencies between services, there are currently no commonly deployed administrative tools, but there is significant prior research in the area [3], [4]. We do not attempt to advance the state of the art for dependency detection in this work, given the variety and effectiveness of systems already proposed and evaluated. Published approaches include network traffic monitoring [3],



Fig. 2. Diagram of a typical enterprise service network, inspired by TBMCS, featuring multiple layers of interwoven dependencies. Some hosts effectively act as a single service (green servers), while others (grey servers) host multiple services (green ovals below server).

log analysis [14], and hybrid approaches [15]. A number of these approaches offer high accuracy in realistic environments: for example, the Macroscope system correctly identified 95% of dependencies in evaluation, with a false positive rate of only approximately 18% of the number of dependencies [15]. Furthermore, the authors of the Orion system [3] claim that such false positives are relatively easy to correct, given some time for directed testing after initial dependency detection. We thus take accurate dependency detection as a given (using a simple such mechanism in our emulations) and focus instead on applying information about service dependencies to achieve efficient recovery coordination.

Finally, in the contexts of autonomic computing [16], organic computing [17], and other approaches to self-adaptation, there have been a number of investigations of various selfadaptive frameworks for managing multi-service environments, including a number that take a distributed agent / distributed control perspective such as we use in this paper (e.g., [18], [19], [20], [21], [22], [23], [24]). To the best of our knowledge, however, none of these publications to date have specifically addressed the problem of dependency-aware recovery of failed services. Most such frameworks, however, should readily be able to incorporate the algorithm and results presented in this paper for use in the more comprehensive coordination schemes that such frameworks typically aim to support.

#### II. MOTIVATING EXAMPLE: TBMCS

Throughout this paper, our discussion will make use of a motivating example based on a real system with properties typical of small- to medium-scale enterprise services. This example is based on the Theater Battle Management Core Systems (TBMCS), a set of software systems used by the United States military to plan and execute air campaigns [25]. We use TBMCS as a motivating scenario because it is a real-world example of a long-lived complex deployed enterprise-level system that spans multiple hosts and operating systems, and whose operators actually encounter management challenges of the type described in this paper. In a life cycle typical of such systems, TBMCS was first released in 2000 and has undergone a number of releases since that time, accumulating complexity and legacy requirements, resulting in internal dependencies that are difficult for users or administrators to observe. At this point, a TBMCS deployment may be comprised of many different components, including a legacy UNIX server, a database server, a web application server, an email server, and email and browser-hosted clients Each of these components may in turn be made up of semi-independent sub-components, with multiple dependencies between them.

This type of complexity of components and dependencies is illustrated in Figure 2, which shows a network embodying key relevant challenges faced by TBMCS<sup>1</sup> that we will use as a running example throughout the rest of this paper. This network contains 10 services distributed across 6 physical servers, and features multiple layers of interwoven dependencies.

Managing such a network often presents difficult and poorly defined tasks. When components fail, they may fail partially or cryptically, making it difficult to identify which components, exactly, have failed. For example, a common failure mode might be for a user to get a timeout message from a mail client or a page not found error from a web browser. Neither users nor administrators have a clear indication of which component has failed in such a case. The failure could be at the client, the server that the client is attempting to contact, a particular service within that server, or some unseen (and possibly undocumented) dependency farther "upstream" from that server. Recovering from failures can also be difficult. Individual services or machines can typically be easily restarted, but poorly managed (and again possibly undocumented) dependencies between services can cause new failures if a service is restarted before other services that it depends on.

Managing enterprise services like TBMCS is thus often a matter of applying either administrative experience (i.e., an administrator is available, has seen the specific error, and knows the likely culprit) or applying a conservative recovery strategy such as bringing down all services and then restarting them sequentially in a known-safe order. The former strategy requires action by a system administrator intimately familiar with all of the components and dependencies, and the latter guarantees maximal disruption for all users before the system is brought completely back on-line. The goal for the automated recovery system presented in this paper is to improve this situation by decreasing the need for intervention by a system administrator, while simultaneously minimizing service disruptions.

## III. SERVICE MANAGEMENT ARCHITECTURE

The work presented in this paper builds upon on an architecture designed to support retrofitting of new service management mechanisms onto existing services without modification



Fig. 3. Server management architecture, adapted from [5]: within an enterprise server, each managed service (red) is monitored and controlled by a management daemon (purple), one daemon per service. Management daemons maintain connections to one another that parallel the connections they observe between their respective managed services. Management is then coordinated using a distributed algorithm, in this case implemented using the Protelis aggregate programming framework.

of those services. Such a retrofitting approach is important, particularly given the motivation of supporting enterprises with few administrative resources, since modifying existing services can be extremely costly in time and resources. In fact, in many cases, modifying existing services is nearly impossible due to certification requirements or to use of effectively immutable vendor-supplied or legacy binaries.

The architecture, first presented in [5] is illustrated in Figure 3. For each service to be managed, an associated management daemon is created (e.g., a server with four managed services will have four management daemons). Each management daemon needs to be able to do three things:

- Detect its managed service's interactions with other managed services.
- Detect current service state, including whether the service is stuck in some form of failure state.
- Start and stop the managed service, even if it is currently in a non-functional state.

For purposes of this paper, we assume that such software instrumentation and controls are possible and available. As discussed in Section I-A, there are multiple options for dependency detection and service monitoring.

When a management daemon observes its associated service interacting with another service, it attempts to create a parallel communication link with the other service's corresponding management daemon. The management daemons then use these links to coordinate their actions, maintaining the link as long as either the services are interacting within some timeout or else the management daemon has some other reason to believe the coordination link should be preserved. In the example systems used in this paper, all services use network sockets to communicate (even if they are communicating locally within the same machine), and their management daemons thus open a parallel socket connection with one

<sup>&</sup>lt;sup>1</sup>Actual system specifics are not used since TBMCS is a deployed operational system.

Symbol	Definition
$G = \{V, E\}$	Graph defining a service network in terms of a set of daemon nodes $V$ and communication edges $E$ between them
N(v)	Neighbors of daemon $v$ in service network
t	Update period for daemons
δ	Network delay in communication between daemons
F	Set of daemons that have either failed or whose service has failed
D(v)	Dependencies $D(v) \subseteq N(v)$ of service for daemon v
$D^{-1}(X)$	Inverse dependencies of daemons: $\{v   \exists_{x \in X} \text{ s.t. } x \in D(v)\}$
$\overline{D^{-1}(X)}$	Closure of a set X via inverse dependencies: $X \cup D^{-1}(X) \cup D^{-1}(D^{-1}(X)) \cup D^{-1}(D^{-1}(D^{-1}(X))) \cup \dots$
$r_v$	Time required to successfully restart daemon $v$ and/or its associ-
	ated service, given no missing dependencies
$R_{v F}$	Minimum time to safely restart daemon $v$ and/or its associated service, given a starting failure set $F$

 
 TABLE I.
 Symbols used in description and proofs for Dependency-Directed Recovery algorithm

## another.

For simpler development of coordination mechanisms, the architecture that we use separates the coordination logic from instrumentation and controls. In particular, we implement coordination using Protelis [5], a Java-based aggregate programming framework based on field calculus [26], [27]. This framework allows a programmer to specify an application in terms of the collective behavior of a network rather than the behavior of individual devices, raising the level of abstraction for application development and making many implementation details implicit. This allows coordinated services (such as we discuss in this paper) to be implemented more cleanly and succinctly than with conventional networking APIs, as well as enabling them to be safely composed in an applications environment. It is thus a useful enabling technology for lightweight development of enterprises services, but not required for implementing or understanding the particular distributed recovery mechanisms that we discuss in this paper.

## IV. DEPENDENCY-DIRECTED RECOVERY ALGORITHM

We now present our Dependency-Directed Recovery (DDR) algorithm. We first formalize the recovery problem in the context of the service management architecture presented in the previous section, then present our simple reactive algorithm for solving this formalized problem.

## A. Problem Formulation

We formalize the operating environment for enterprise recovery as follows:

- The service network is a graph  $G = \{V, E\}$ , where the nodes V are daemons (each associated with a service) and undirected edges E indicate communication between services and their corresponding daemons. Each service is also assumed to have a unique identifier associated with it.
- Network communication is assumed to be reliable, with a delay of up to  $\delta$  to send a coordination message from one daemon to another. Note, however, that the algorithm can also assist in recovering from network failures, and extension to this case is discussed in Section V.

- Each daemon executes asynchronously, independent of its associated service and of other daemons. Every daemon  $v \in V$  executes, updating its state and possibly attempting to act, at least once every t seconds, and shares state with its neighbors N(v) in the service network on an as-needed basis (described in more detail below).
- Every daemon has a reliable status detector that indicates which of three states its associated service is in: *run* (normal operation), *stop* (a safe shutdown state), or *hung* (partial or complete failure).
- Every daemon v has a reliable dependency detector that reports a set of dependencies D(v) ⊆ N(v), indicating the set of neighbors that its associated service needs correct responses from in order to operate correctly. Such correct operation is only guaranteed when all services of v and D(v) have status run. If some services of D(v) are not running, then the service of v is likely to fail and become hung.
- Daemons can take two actions to affect service state: *stopService* attempts to move a service from *run* or *hung* to *stop*; *restartService* attempts to move a service from *stop* or *hung* to *run*. Either action may fail, leaving the process in a *hung* state. Such a transition takes some amount of time that may or may not be predictable. We further assume that the *stopService* and *restartService* actions are assumed to be idempotent (meaning in this case that if they are invoked again while executing, the duplicate invocation is ignored) and that invocation of *restartService* preempts *stopService*.
- Both services and daemons may fail arbitrarily. When a service fails, it is detected as *hung* by its daemon within its update period of *t* seconds. When a daemon fails, its failure is detected by its neighbors within *t* seconds, which discard any state information they have from that daemon. A failed daemon will automatically attempt to restart, but this attempt may fail.

The goal of a distributed recovery algorithm is thus as follows: given a service network with some set of failed daemons and services, coordinate daemon actions in order to return as rapidly as possible to a state in which all services are running correctly. Restarting in an inappropriate order, on the other hand, can significantly extend the recovery time due to failures introduced while dependencies are restarting, as can restarting services that do not need to be restarted.

Note that this problem formulation can model both process and server failures, as well as failures that require human intervention to resolve. Process failures are independent, while server failures will cause correlated failures in both daemon and service, possibly multiple pairs if there are multiple services running on a given server (e.g., the application server in Figure 2). The difference between failures that can be handled with automation versus failures requiring human intervention (e.g., a hardware failure that requires a component be replaced) is modeled by the potential for *stopService*, *restartService*, and daemon restarts to fail; in this case, the goal remains the same, but recovery is inescapably delayed until the restart actions are able to succeed.



Fig. 4. Example of how dependency management proceeds with recovery in a service network, showing status *run* as green, *stop* as blue, and *hung* as red. Following failure of some set of services (a), other services that depend on them shut themselves down (b). When the initial failures have been able to restart (c), the services that depend on them restart incrementally (d), until the entire service network has recovered (e).

# B. Algorithm

Given this problem formulation, Dependency-Directed Recovery can be realized using the following reactive algorithm, (presented here as implemented in Protelis [5]):

```
// Collect state of monitored service from service manager daemon
let status = self.getEnvironmentVariable("serviceStatus");
let serviceID = self.getEnvironmentVariable("serviceID");
let depends = self.getEnvironmentVariable("dependencies");
let serviceDown = status=="hung" || status=="stop";
// Compute whether service can safely be run (i.e. dependencies are satisfied)
let liveSet = if(serviceDown) { [] } else { [serviceID] };
let nbrsLive = unionHood(nbr(liveSet));
let liveDependencies = nbrsLive.intersection(depends);
let safeToRun = liveDependencies.equals(depends);
// Act based on service state and whether it is safe to run
if(!safeToRun) {
  if(!serviceDown) {
    self.stopService() // Take service down to avoid misbehavior
  } else { false } // Wait for dependencies to recover before restarting
}
 else {
  if(serviceDown) {
    self.restartService() // Safe to restart
    else { false } // Everything fine; no action needed
  }
```

In essence, the code is relatively straightforward. Each daemon executes the whole sequence at each of its periodic updates. In the first block the DDR algorithm simply accesses the service identifier, status, and dependencies detected by the daemon. In the second block, each daemon computes whether it is safe for its service to run. The liveSet variable is a set that contains the local service if it is running, is empty for a failed service, and is non-existent for a failed daemon. This set is then shared with neighbors via the Protelis function nbr,<sup>2</sup> and combined with other neighboring values via unionHood, which takes the union of the sets shared from neighbors, to produce at each daemon a set of the services currently running on its neighbors. Intersecting this with the local service's dependencies produces a set of live dependencies, and if this set is equal to the set of dependencies, then it is safe for the service to run. Finally, the daemon uses this judgement to adjust service state: if the service is running but it is not safe for it to do so, the algorithm tries to stop it; if it isn't running but it is safe to do so, the algorithm tries to restart it.

The effect of this algorithm is to follow chains of dependencies, neighbor by neighbor, from failed services to the services that depend on them. When a service fails, all of the services that depend on it attempt to shut down in an orderly fashion. Services then wait to restart until nothing they depend on is down, restarting in an orderly and incremental fashion following the partial order established by dependencies.

Figure 4 shows an example of DDR in operation on the service network from Figure 2. The algorithm, quiescent when all services are running, is triggered into action by the failure of services App 2 and Core 1 (Figure 4(a)). Their corresponding daemons report this change in status to all of their neighbors, and the three that depend directly on these services (Gateway A, Legacy Unix, and App 1 on App Server) transition to a safe stop state, while the other two neighbors, that have no such dependency, do not (Figure 4(b)). In this case, there are no further "upstream" dependencies, but if there were the safe stopping of processes would continue chaining up through dependencies, stopping each service affected by the failures. In parallel with the safe shutdown of upstream dependencies, the failed services, having no unsatisfied dependencies themselves, begin trying to restart. When they succeed (Figure 4(c)), this means that Legacy Unix and App 1 have all of their dependencies once again satisfied and are able to safely restart (Figure 4(d)). Gateway A, however, also depends on Legacy Unix and App 1 and is not able to safely restart until these two services have also finished restarting. Finally, when Gateway A completes its restart, all services are running again and the system is fully recovered (Figure 4(e)). Importantly, note that throughout the whole recovery process, the operation of Gateway B and all of the services that it depends on was unaffected, even though these services share the Portal and Core Logic servers with services affected by the failure.

#### V. ANALYSIS OF DEPENDENCY-DIRECTED RECOVERY

We now prove that Dependency-Directed Recovery provides near-optimal recovery time for a service network. After first analyzing the case where only services fail, we then extend to the case where daemons and network connections may fail as well. Finally, we analyze the resource costs associated with the DDR algorithm.

<sup>&</sup>lt;sup>2</sup>Note that this does not mean that a message is sent each round: caching and reliable communication via sockets means that the Protelis infrastructure only needs to actually send a message on initial connection with each neighbor or when the shared set changes.

## A. Near-Optimal Recovery of Failed Services

First, we will compute the optimal recovery time that can be achieved. To do this, we will make one further assumption, that, after any given failure, the time required to restart a service (assuming its dependencies are satisfied) is independent of when the attempt to restart is initiated. In other words, each daemon v is assigned a fixed restart duration  $r_v$  (e.g., very fast for a lightweight web server, much slower for a large database that performs thorough integrity checks as part of its startup). This independence assumption also subsumes the possibility that a *restartService* action will independently fail: this essentially just results in a higher value for  $r_v$  than would otherwise be the case.

Given an arbitrary set  $F \subseteq V$  of daemons with a failed service, the set of services affected by this failure can be computed using the dependency function D(v). The inverse of this function can compute the set of daemons whose services depend directly on the service of any daemon in a set  $X \subseteq V$ , namely:

$$D^{-1}(X) = \{ v | \exists_{x \in X} \text{ s.t. } x \in D(v) \}$$
(1)

The closure of this function then determines the set of all daemons with services dependent through any chain of dependencies:

$$\overline{D^{-1}(X)} = D^{-1}(X) 
\cup D^{-1}(D^{-1}(X)) 
\cup D^{-1}(D^{-1}(D^{-1}(X))) 
\cup \dots$$
(2)

The minimum time to safely restart the service at daemon v, given failure set F can then be computed by induction as:

$$R_{v|F} = \begin{cases} 0 & : v \notin \overline{D^{-1}(F)} \\ r_v & : v \in \overline{D^{-1}(F)}, D(v) = \emptyset \\ r_v + \max_{v' \in D(v)} (R_{v'|F}) & : v \in \overline{D^{-1}(F)}, D(v) \neq \emptyset \end{cases}$$
(3)

Two important notes about the values thus computed: first, note that this value only converges when  $G|\overline{D^{-1}(X)}$  (i.e., the sub-network formed by considering only nodes and edges in the region of the failures and their dependencies) is a directed acyclic graph. If there are cycles,  $R_{v|F}$  is ill-defined for any daemon participating in a cycle, indicating the intuitive fact that the service network cannot be safely restarted when there are dependency cycles. Second, note that if all  $r_v$  are equal, then the maximum  $R_{v|F}$  is equal to  $c \cdot r_v$ , where c is the length of the longest chain of dependencies from the failure, i.e., the diameter of the graph  $G|\overline{D^{-1}(X)}$ . In other words: restart time is proportional to the length of dependency chains, unless some services are much faster or slower to restart than others.

We can now prove that DDR self-stabilizes [28], meaning that the service network will return to a correct state (i.e., all services in the *run* state) from any arbitrary initial state in a finite number of steps. Moreover, when recovering from failed services it performs near-optimally, slowed only by communication delays and the update period of the daemons:

**Theorem 1.** Given a set F of failed services, DDR selfstabilizes on any acyclic dependency graph G within time  $(\max_{v \in V} R_{v|F} + (2t + \delta) \cdot c)$  where c is the diameter of  $G|\overline{D^{-1}(F)}$  (i.e., the longest chain of dependencies).

**Proof:** Consider the subgraph  $G|\overline{D^{-1}(F)}$ , which contains all daemons for failed services, daemons with services dependent on those failed, and dependencies between elements of this set. Any daemon outside of this subgraph must, by definition, have its service in the *run* state, and will remain there since by definition it has no dependency relationship, direct or indirect, with any daemon with a service not in the *run* state. These services thus start in and remain in a correct state.

Returning to services in the subgraph: since G is acyclic, this subgraph must be acyclic as well. Any directed acyclic graph may be interpreted as a partial order, so let us choose the interpretation where the minima M are those daemons that have no dependencies contained within  $G|\overline{D^{-1}(F)}|$ . This set M of minima is non-empty because the graph is finite and acyclic.

For any daemon v in the minima M, its variable nbrsLive must contain all of the dependencies D(v), as otherwise it would not be a minimum, and thus its safeToRun variable is true. If the daemon's service is currently in the *run* state, then no action need be taken; if it is not running, then the *restartService* action will be asserted and it will take up to  $r_v + t$  for the associated service to restart, return to the *run* state, and have this recognized by the daemon during an update (note that since *restartService*, we can ignore the question of whether any action is currently executing on the service).

After the daemon returns to the *run* state, it can take up to  $\delta$  time for this information to be sent to its neighbors, such that their nbrsLive variables can update, and up to another t time for that update to actually occur. At this point, all of the daemons in the minima M have restarted (their services are in the *run* state) and their neighbors information updated, such that we may consider instead a reduced graph  $G|\overline{D^{-1}(F)} - M$ . By the same reasoning as before, this graph is also acyclic and has a non-empty set of minima that can restart, but has a diameter that has decreased by one.

This cycle may thus repeat up to c times, where c is the diameter of  $G|\overline{D^{-1}(F)}$ , before the maximum diameter reaches zero, meaning the subgraph of daemons that may have a service not in the *run* state is guaranteed to be empty. The sum of the  $r_v$  components along any path is equal to  $R_{v|F}$ , and there can be up to c steps of  $(2t + \delta)$  additional delay per step, so therefore the total restart time must be bounded by  $\max_{v \in V} R_{v|F} + (2t + \delta) \cdot c$ .

A key implication of this result is that the time for DDR to recover from failures will generally be dominated by service restarts, unless those restarts are very fast. Another corollary implication of this proof is that DDR will also self-stabilize on cyclic graphs, as long as  $G|\overline{D^{-1}(F)}$  is acyclic. Since a failure might potentially occur anywhere, however, this distinction is not particularly useful.

# B. Failed Daemons and Communication Links

The same basic principles for failed services hold as well for extension of the proof to the case where daemons and services fail together (e.g., hardware failures), since the DDR algorithm treats a missing daemon the same as a daemon reporting that its service is failed. In this case, the restart time may be further extended by the time required to restart a daemon and to detect a missing neighboring daemon (it is not affected by diameter since daemons have no dependencies and can restart in parallel).

Likewise, failure of a network connection is equivalent to service failure for any dependency that requires that connection; the same directed subgraph reasoning can thus be applied for any set  $F_E$  of failed edges in E, as well as to mixtures of failed edges and services.

Treating a missing daemon as equivalent to a failed service, however, does mean that the DDR algorithm will perform suboptimally when a daemon fails but its associated service does not. In this case, any daemon with the failed daemon as a dependency will stop its service and wait to restart, when in fact the service could have continued safely running. This is an inherent limitation of instrumenting services, however, and cannot be addressed at the algorithmic level, but rather only by improving instrumentation, e.g., to infer that a neighboring service is still operating correctly, even though its daemon is down, by examining incoming traffic from it.

# C. Resource Requirements

Dependency-Directed Recovery has relatively low resource requirements. Statically, each daemon requires |N(d)| + 1 sockets: one dedicated socket for each live connection to a neighbor, plus a server socket listening for new connections. This is unlikely to pose a significant burden on system resources unless there are either a very large number of neighbors or a very large number of services (and associated daemons) on a single machine.

The actual DDR algorithm is very simple, so as long as t is non-trivial, executing the computational components of the algorithm should not pose any significant computational burden on a machine. In terms of its communication: the only signals that need to be sent between daemons (by a smart infrastructure) are announcements of changes in the liveSet variable, plus infrequent "heartbeat" messages to let neighboring daemons know a daemon is still alive when this variable is not changing. The number of change messages required for a recovery is bounded by  $2N(\overline{D^{-1}(F)})$ : one to each neighbor as a daemon's service goes down (failing or preemptively stopped) and another as it returns to the run state. Heartbeat messages may be much more infrequent, though their frequency does determine how quickly a daemon failure can be detected. With such small and infrequent messages, this is again unlikely to pose a significant burden on system resources unless the number of neighbors is very large.

Taken together, what this all means is that, for most reasonable deployments, the main cost of Dependency-Directed Recovery is likely to not be the algorithm itself, but the instrumentation for detecting dependencies and for monitoring service state.

# VI. EXPERIMENTAL VALIDATION

We now turn to validation of the DDR algorithm and the theoretical results presented in the previous section. We first provide additional details of the implementation and emulation environment, then present a set of distributed recovery experiments that provide experimental validation of the analytical results presented in the previous section.

## A. Emulation Environment

Building on the architecture presented in Section III, we developed a Java-based enterprise service emulation environment to evaluate our dependency-directed recovery algorithm. The environment is an *emulation* rather than a *simulation*. Using a JSON specification file, our emulation system can remotely launch and manage an arbitrary network of "query-response" services and associated daemons. By using actual distributed services, potentially across many different machines, this emulation environment allows a more realistic approximation of actual enterprise service environments. The emulation environment also includes an optional management GUI for visual monitoring of the running network of services and daemons.

For simplicity, the query-response services are implemented in Java and offer hooks for observation and control by the daemon implementation, which is executed in the same JVM. Although these services are very lightweight, they emulate the longer initialization time of more complex services by sleeping for a fixed delay during a restartService action. Each query-response service has a set of other queryresponse services as dependencies; when it receives a query, it passes it onto all of its dependencies and waits for a response. Once all dependencies have responded (or if there are no dependencies), it sends its own response to the query it received. If a fixed timeout passes without dependencies being satisfied, the service drops the query, and if it drops queries too frequently, it hangs (this creates persistent state that can cause failures to propagate and recur when restarts occur in the wrong order). Finally, a subset of services are designated as query originators, representing points of external interaction with users of the service network: these behave identically except that they also spontaneously originate queries (which then are passed on to their dependencies). Both query-response services and daemons communicate with one another via standard TCP network sockets; the query-response services have their relationships hard-wired by the JSON specification file, while the daemons start with no such knowledge. Instead, daemons rendezvous by observing the connections made by their service, and serving a socket of their own at a port with fixed offset from that of their service.

Given the strength of prior dependency detection work (as discussed in Section I-A) and our focus on the use of dependency information rather than its acquisition, we chose not to attempt to integrate a dependency detection system like PinPoint [2] or Orion [3], but instead simply to use emulated services whose dependencies are easily extracted from their traffic. In particular, the emulated services 1) initiate socket communication with all other services that they depend on at least once every T seconds and 2) do not initiate communication with any service that they do not depend on. Each

management daemon then embeds a sensor in its associated service that reports an <IP address,port,timestamp> tuple for each client socket creation, and takes the set of dependencies to be the set of IP-address/port pairs that it has observed within the last T seconds. When combined with the constraints placed on services, this guarantees every daemon will have correct dependency information that is no more than  $2T + \delta$  seconds out of date.

This emulated implementation is thus fairly close to a real deployable system. Modification for a real enterprise service network deployment would require that the implementation of monitoring, dependency detection, control, and rendezvous be changed to a "wrapper" model that does not require cooperation from the managed service. The naive dependency detection just described would also need to be replaced with a real state-of-the-art dependency detection mechanism, such as the ones discussed in Section I-A, and distinctions might also need to be drawn between startup dependencies and runtime dependencies.

# B. Experimental Setup

We evaluate the performance of the DDR algorithm by emulating failure recovery on a variety of service networks, comparing against the common current strategy of sequential restart in a fixed "known-safe" order.

To obtain a sufficient variety of service network topologies, we evaluate both of these algorithms on random service networks of n devices. Services are numbered sequentially, from 1 to n; graphs are generated with undirected edges, which are then interpreted as dependencies from the higher numbered service on the lower numbered service (thereby guaranteeing graphs are acyclic). We generated these networks from three classes of random graph selected to be representative of typical service architectures (examples shown in Figure 5), with either sparse or dense dependencies for each graph class:

- Erdos-Renyi: The network is generated using the Erdos-Renyi process [29], where there is probability p of each possible edge existing, using p = 2/n for sparse and p = 0.3 for dense. These networks have high variability in dependency chain length and the number of dependencies per service. Queries originate from the three highest-numbered services.
- Layered: The network is organized into  $\lceil p\sqrt{n} \rceil$  equally sized layers, where  $p = 10^{U[-0.5,0.5]}$ , i.e., a uniform geometric distribution from "short and wide" to "thin and deep" distributions. Services in each layer are numbered sequentially, and each layer is connected to the prior layer by a set of random edges, using edge probability 1/n for sparse and 0.5 for dense. When the number of total services did not divide evenly by the number of services per layer, lower layers were filled first. Queries originate from all services in the highest layer.
- **Clustered:** The network is formed as an Erdos-Renyi graph of a set of "clusters," where each cluster is itself an Erdos-Renyi graph of size  $\lfloor \sqrt{n} \rfloor$  Services in each cluster are numbered sequentially, and connections between clusters are realized by randomly selecting



Fig. 5. Typical examples of the three classes of random dependency graphs used for evaluation of dependency-directed recovery: Erdos-Renyi (a), layered (b), and clustered (c).

one service in each cluster. When the number of services did not divide evenly by the number of services per cluster, the lower-numbered clusters were filled completely. These networks have the shortest dependency chains on average, but each service may have many dependencies within its own cluster. Queries originate from the three highest-numbered services in each cluster.

For each network class and density, we generated 50 random samples each for n equal to 2, 4, 8, 12, 16, and 20. In total, this is 1800 service networks<sup>3</sup>, with a wide variety of topologies and spanning a range from very simple to rather complex service networks. On each of these service networks, we executed both DDR and fixed-order restart, using a t = 0.5 second update period and setting the restart delay to 3 seconds (producing an  $r_v$  of the actual restart time plus 3 seconds).

For each experimental run, the service network was first initialized and allowed to stabilize in operation. One service was then randomly selected to fail. Following the injection of the failure, the service network was then run until at least 30 seconds elapsed with every service staying in the *run* state.

On fixed-order restart runs, a central controller executed the recovery by first invoking *stopService* on all services, then invoking *restartService* on each service sequentially, starting with service 1 and moving to each subsequent service as the prior one returns to the *run* state. For DDR runs, the algorithm executed as described in Section IV.

# C. Results

As predicted, the time required for recovery from failure is dramatically lower for DDR than for fixed-order restart. Figure 6(a) shows restart time as a function of the number of services. The recovery time for fixed-order restart is, of course, directly proportional to the number of services, and shows minimal variation since it does not adapt to either failure location or network structure. DDR, on the other hand, grows

 $<sup>^{3}</sup>$ A small number of experimental runs failed to execute properly for unrelated reasons, so the actual number of experimental samples for which data was collected is 1766 for DDR and 1756 for fixed order.



Fig. 6. (a) Fixed-order restart (red) always requires recovery time proportional to the number of services in the network, while the time for Dependency-Directed Recovery (all others, with solid lines being sparse and dashed lines being dense) tends to grow much more slowly. (b) The time for DDR is instead proportional to the depth of the longest dependency chain stemming from the set of failures ('+' markers indicate sparse edges and 'o' markers indicate dense edges). For visualization purposes, depth and time have been randomly perturbed in the range  $\pm 0.1$  to avoid aliasing.

much more slowly with respect to the size of the service network, and with a great degree of variation. This is because, as analyzed in Section V-A, the recovery time is proportional to the longest dependency chain c. This is verified by graphing recovery time against c (Figure 6(b)), producing a clearly linear relation. Given t = 0.5 and the 3-second restart delay, the estimated mean recovery time is  $3.75 \cdot c$  (for each service in the chain: restart time, plus one update to notice, plus half an update average asynchrony between daemons, omitting  $\delta$ and the unknown portion of  $d_v$ ). Linear regression against the experimental data finds an observed mean recovery time of  $4.9 \cdot c$ , implying that some combination of  $\delta$  and variation in  $d_v$  is adding about one second per restart—a reasonable degree of additional overhead to observe in this emulation experiment.

The other predicted advantage of DDR is that some services can remain operational throughout the recovery, if they are not affected by the failure. Fixed-order restart, by definition, always shuts down all services. With DDR, on the other hand, only those devices directly dependent on the failure will be affected, which means that, for the most part, larger



Fig. 7. Dependency-Directed Recovery reduces the disruption caused by recovery by ensuring that services unaffected by a failure remain undisturbed. Whereas fixed-order restart always restarts all devices, causing 100% disruption (not shown), with DDR more complex networks tend to be proportionally less disrupted by any given failure (except for dense Erdos-Renyi). Solid lines indicate sparse graphs and dashed lines indicate dense graphs.

networks should have a smaller percentage of their services affected. Figure 7(a) shows that this indeed holds for all of the service networks except for dense Erdos-Renyi networks, where there are so many dependencies that most failures affect many services.

Finally, shifting from a system administrator perspective to a user perspective, disruption of services in the service network would only be apparent to an external user when queryoriginator services are affected. Here as well, the same trend of lower impact in larger networks should hold. Figure 7(b) shows that this does indeed hold as expected, in this case even for dense Erdos-Renyi (where there are many dependency paths, but most do not make it all the way to all of the originators). We thus see that DDR provides a major improvement not only in the time for recovery, but also the number of services affected and the degree of service disruption that affects users.

#### VII. CONTRIBUTIONS

In this paper, we have demonstrated a lightweight adaptive capability for managing recovery from failures in enterprise service networks, based on dependency detection and a reactive Dependency-Directed Recovery algorithm. This mechanism guarantees an optimal automation-assisted recovery time, providing a large gain in performance over the typical current practice of fixed-order restart, and we have further validated this performance experimentally on emulated service networks.

From these results stem two clear directions for future work: the first is maturation and transitioning of these mechanisms for use in actual enterprise management environments. This will require integration with state-of-the-art dependency detection mechanisms and replacement of the service status and action mechanisms, as well as general hardening and interface improvement. At the same time, this work has also demonstrated the potential ease with which self-adaptive enterprise management tools can be developed using an aggregate programming architecture like that described in Section III. Distributed recovery is only one of many management challenges faced by small- to medium-scale enterprise systems, and this same approach is likely to be able to produce useful self-adaptive management tools for addressing others of these challenges as well.

#### **ACKNOWLEDGMENTS**

This work has been supported by the United States Air Force and the Defense Advanced Research Projects Agency under Contract No. FA8750-10-C-0242. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views, opinions, and/or findings contained in this article are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for public release; distribution is unlimited.

## REFERENCES

- Aberdeen Group, "Why mid-sized enterprises should consider using disaster recovery-as-a-service," http://www.aberdeen.com/Aberdeen-Library/7873/AI-disaster-recovery-downtime.aspx, April 2012, retrieved July 13, 2015.
- [2] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on.* IEEE, 2002, pp. 595–604.
- [3] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating network application dependency discovery: Experiences, limitations, and new solutions," in OSDI, vol. 8, 2008, pp. 117–130.
- [4] M. łgorzata Steinder and A. S. Sethi, "A survey of fault localization techniques in computer networks," *Science of computer programming*, vol. 53, no. 2, pp. 165–194, 2004.
- [5] D. Pianini, M. Viroli, and J. Beal, "Protelis: Practical aggregate programming," in ACM Symposium on Applied Computing, 2015.
- [6] J. Trocki, "mon(8) linux man page," http://linux.die.net/man/8/mon, Accessed May 2015.
- [7] Nagios Enterprises, "Nagios The Industry Standard in IT Infrastructure Monitoring," http://www.nagios.org, Accessed May 2015.
- [8] Puppet Labs, "Puppet labs: It automation software for system administrators," https://puppetlabs.com/, Accessed May 2015.
- [9] Chef, "An overview of chef chef docs," http://docs.chef.io/chef\_overview.html/, Accessed May 2015.
- [10] W. Tarreau, "HAProxy-the reliable, high-performance TCP/HTTP load balancer," http://www.haproxy.org/, 2012.
- [11] Oracle Corporation, "Mysql 5.1 reference manual :: Multi-master and circular replication," https://dev.mysql.com/doc/refman/5.1/en/mysqlcluster-replication-multi-master.html, Accessed May 2015.

- [12] —, "Mysql 5.1 reference manual :: Replication," https://dev.mysql.com/doc/refman/5.1/en/replication.html, Accessed May 2015.
- [13] VMware, "vSphere 6.0," https://www.vmware.com/products/vsphere, Accessed May 2015.
- [14] J.-G. Lou, Q. Fu, Y. Wang, and J. Li, "Mining dependency in distributed systems through unstructured logs analysis," ACM SIGOPS Operating Systems Review, vol. 44, no. 1, pp. 91–96, 2010.
- [15] L. Popa, B.-G. Chun, I. Stoica, J. Chandrashekar, and N. Taft, "Macroscope: end-point approach to networked application dependency discovery," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM, 2009, pp. 229–240.
- [16] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.
- [17] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, and H. Schmeck, "Towards a generic observer/controller architecture for organic computing." *GI Jahrestagung* (1), vol. 93, pp. 112–119, 2006.
- [18] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka, "On patterns for decentralized control in self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 76–107.
- [19] F. Nafz, H. Seebach, J.-P. Steghöfer, G. Anders, and W. Reif, "Constraining self-organisation through corridors of correct behaviour: The restore invariant approach," in *Organic ComputingA Paradigm Shift for Complex Systems*. Springer, 2011, pp. 79–93.
- [20] F. de Oliveira, T. Ledoux, and R. Sharrock, "A framework for the coordination of multiple autonomic managers in cloud environments," in *Self-Adaptive and Self-Organizing Systems (SASO)*, 2013 IEEE 7th International Conference on, Sept 2013, pp. 179–188.
- [21] N. Gui and V. De Florio, "Towards meta-adaptation support with reusable and composable adaptation components," in *Self-Adaptive* and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference on. IEEE, 2012, pp. 49–58.
- [22] H. Nakagawa, A. Ohsuga, and S. Honiden, "Towards dynamic evolution of self-adaptive systems based on dynamic updating of control loops," in *Self-Adaptive and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 59–68.
- [23] B. Debbabi, A. Diaconescu, and P. Lalanda, "Controlling self-organising software applications with archetypes," in *Self-Adaptive and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference* on. IEEE, 2012, pp. 69–78.
- [24] P. Vromant, D. Weyns, S. Malek, and J. Andersson, "On interacting control loops in self-adaptive systems," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems.* ACM, 2011, pp. 202–207.
- [25] T. Gorman and L. Muir, "Theater Battle Management Core Systems (TBMCS) executive overview," ESC/AFC2TIG/LMMS, Hurlburt Field FL, vol. 23, 2003.
- [26] M. Viroli, F. Damiani, and J. Beal, "A calculus of computational fields," in Advances in Service-Oriented and Cloud Computing, ser. Communications in Computer and Information Sci., C. Canal and M. Villari, Eds. Springer Berlin Heidelberg, 2013, vol. 393, pp. 114– 128.
- [27] J. Beal and M. Viroli, "Space-time programming," *Proceedings of the Royal Society A*, to appear.
- [28] S. Dolev, Self-Stabilization. MIT Press, 2000.
- [29] P. Erdos and A. Renyi, "On random graphs I," *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.