

A Service Management Facility for the Java™ Platform

Glenn Skinner, Grzegorz Czajkowski, David Hearnden, Mick Jordan, and Michal Wegiel

Sun Microsystems Laboratories
16 Network Circle
Menlo Park, CA 94025, USA
{firstname.lastname}@sun.com

Abstract

Managing operational and semantic interdependencies among software services is a relatively unexplored topic, despite its relevance to automating service deployments and to increased availability. In this paper we describe a framework for structured and programmatic dependency management among services written in the Java™ programming language. The framework's interface allows for defining an acyclic graph of dependencies. The graph's structure reflects the startup sequence of managed services. Upon a service failure or intentional termination, the dependencies are consulted to determine which services may be affected. If the dependencies so dictate, a service that has gone off-line will automatically be restarted. The requisite changes, e.g., restarting, will be propagated along the graph's edges to ensure that the required dependencies are satisfied for each service. We demonstrate the usefulness of this framework through real-life case studies.

1 Introduction

Software engineering and architectural trends, such as modularization, componentization, client-server separation, and the growing prominence of clusters of computers are driving forces behind systems organized as a set of interdependent services. Although structuring interfaces and interaction between services has been a topic of numerous research projects (e.g., [18, 1, 5]), managing dependencies among services has received little attention to date. However, the issue is rapidly emerging as of vital importance, due to its relationship to scalability and reliability of complex software systems.

We sketch the problem with a simple example of two services, *Logger* and *FTP*. The FTP service relies on the logger to record the history of incoming requests and their dispositions (Figure 1). The dependency stated above in prose as “FTP relies on Logger” lacks rigor, but conveys the idea that the first service cannot function (or, at best, can function sub-optimally) without the other. Further dissection of the statement leads to important operational and semantic questions: Can FTP be started without Logger being present? What should happen to FTP if

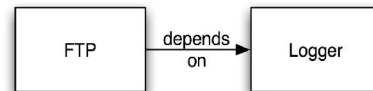


Figure 1: A dependency relationship

Logger terminates? If Logger should get re-started, then by whom and how?

This work attempts to provide programmatic, structured mechanisms to express answers to these and similar questions. The benefits of a systematic organization of inter-service dependencies include automated management of service startup sequencing, automated restart of services affected by a failure, and, consequently, higher availability. The design of the framework embraces distribution, as managed services can reside on different computers.

Type-safe platforms are becoming the environments of choice for a growing variety of programming tasks, and thus we chose to design and implement our service management framework in the Java programming language and for the Java platform. Our work builds on the abstraction of an (isolated) Java application and demonstrates that service management facilities, the need for which has been traditionally associated with system administration, are also needed in platforms based on safe languages and can be delivered in a principled and practical manner.

We now continue our example to show more details of the scope and mechanics of our framework. Beside FTP and Logger there may exist multiple services, some having multiple instances. For clarity of presentation we elide all the others from the output below. We run an administrative utility to examine the states of their instances:

```
% admin
admin> list
svc:/exp.smf.ftp:merlin Online
svc:/exp.smf.logger:port_6762 Online
admin>
```

We use a form of URI called a Fault Management Resource Identifier [16] (or FMRI for short) to name entities related to services, such as services themselves,

their instances, and properties concerning them. The output above shows that the FTP service (which we have identified by using its package name) has an instance named “merlin” and that that instance is in the Online state, indicating that it is up and ready to accept connections from FTP clients. Similarly, the logger service has an online instance named “port_6762”.

Our FTP service uses the logger service to record incoming requests. In particular, the “merlin” instance depends on the “port_6762” instance. We use an XML *manifest* document [15] to capture this dependency and other information describing the service. Figure 2 shows an abbreviated form of the FTP service's manifest.

```
<service name='exp.smf.ftp' type='service' version='1'>
  <dependency
    name="ftp_on_logger"
    grouping="require_all"
    restart_on="none"
    type="service">
    <service_fmri value="svc:/exp.smf.logger:port_6762"/>
  </dependency>
  <instance name='merlin' enabled='true'>
    <property_group name="configuration" type="application">
      <propval name="port" type="integer" value="2020"/>
      <propval name="logger_port" type="integer"
        value="6762"/>
    </property_group>
  </instance>
</service>
```

Figure 2: A manifest for the FTP service

The dependency element says that every instance of the FTP service depends on a specific instance of the logger. The manifest also defines a property that tells the corresponding service instance what port to log to and another that specifies what port to listen on.

Continuing the example, we decide to disable the logging service:

```
admin> disable svc:/exp.smf.logger:port_6762
admin> list
svc:/exp.smf.ftp:merlin Offline
svc:/exp.smf.logger:port_6762 Disabled
admin>
```

What has happened here? The *restarter* component of our implementation has acted on the disable request and told the logger instance to stop. Upon receiving the instance's response, it changed its state to Disabled. The state change induced events directed at each instance depending on the now-disabled logger instance, informing them that they should re-examine their dependencies. The restarter thread acting on behalf of the FTP instance processed the event, discovered that the instance's dependencies were no longer satisfied, and told the FTP instance to stop. The instance did so, and the restarter changed its state to Offline.

Now we re-enable the logger instance:

```
admin> enable svc:/exp.smf.logger:port_6762
admin> list
svc:/exp.smf.ftp:merlin Online
svc:/exp.smf.logger:port_6762 Online
```

```
admin> quit
%
```

This request has induced a similar chain of events. The restarter invoked the logger instance's `start()` method and subsequently changed its state to Online. The state change generated dependency events, one of which was directed to the FTP instance. The restarter re-evaluated FTP's dependencies, noted that they were now satisfied, (re) started it, and changed its state to Online.

In the remainder of this paper we explore the concepts introduced here in greater detail. The next three sections lay out the basic concepts behind our framework, the architecture we devised from these concepts, and our implementation of that architecture. Section 5 presents case studies that demonstrate our framework in use. The final two sections discuss related work and conclude with a summary and our plans for future work.

2 Key concepts

When we first started developing our service management facility, we wrestled with devising a definition of the term “service”. After several false starts, we realized that a very operational definition would suffice for our purposes: a service is anything that one can describe with a service manifest and that conforms to our management API.

A service's manifest contains information describing it and its instances, captured as an XML document. The central element in a manifest is its service element, which defines the service's name and contains elements describing its instances, dependencies, management methods, and so on. Each of these elements can contain list-valued properties, organized into groups, that provide additional information about that element. For example, an instance element can include properties that describe how that instance is to be configured.

A given instance element stands in an inheritance relationship to its parent service. Elements present in the parent contribute to the instance's behavior unless the instance contains overriding elements. This arrangement enables increased descriptive succinctness and precision by factoring information common to all instances of a service into their parent. As a convenient abbreviation, a service can appear in a context that requires an instance; in this case, the service stands for all of its instances.

Dependency elements describe relationships between one instance and other instances, which are named by their FMRIs. The grouping attribute describes the nature of the relationship, which can range through stating that the designated instance depends on all of the others, on one of them, or on none of them (i.e., is mutually exclusive with them).

Other elements fill in additional details of the ones described above. For example, the `exec_method` element describes how to find the classes containing the methods of the instance management API that we now describe.

2.1 MBean interfaces

The management API consists of two parts, each expressed using the JMX [9] MBean design pattern: a set of methods that the framework uses to request a service instance to perform management actions, and a set of methods that instances (and the framework itself) can call to manipulate their properties.

For a service to become manageable under our framework, it must implement the methods of the ServiceableMBean interface. The framework calls methods of this interface to cause an instance do its part of a management operation. All services must implement the start() and stop() methods, which, as their names indicate, instruct the target service instance to start offering its service and to cease offering service. Other methods request the instance to enable itself, to disable itself, to re-check its configuration information and adjust its behavior accordingly, and to affirm that it's still healthy. All of these latter methods are optional and a service can implement any of them to throw an exception if there is nothing it need or can do for that operation.

Most services use the *repository* component of our framework to store and retrieve configuration information. To access the repository, they can use support code from our framework to create a proxy for it and then invoke methods of the RepositoryMBean interface to access it. These methods rely on FMRI to name properties. For example, the FTP instance from the example in Section 1 could retrieve the port number to listen on with the invocation:

```
getProperty(myFMRI + "://properties/configuration/port")
```

Here “configuration” names the *property group* containing the property and “port” names the property itself.

The RepositoryMBean interface includes methods for getting or setting multiple properties in a single call. Each such method invocation is guaranteed to act consistently with some serial ordering of all invocations made to the same receiver through the interface. This characteristic of the interface allows clients to ensure that they can get and set consistent views of properties that pertain to multiple instances. As discussed below, the restarter component of our framework depends on this behavior to orchestrate dependency management.

2.2 The service instance life cycle

As the example in the previous section illustrated, each service instance has a state, and that state can change over the course of the instance's lifetime. From the perspective of our management interfaces, the set of possible states forms the nodes of a finite state machine, and it is our framework's job to manage each instance's transitions among these nodes during its lifetime. Indeed, this is the central responsibility of our framework's restarter component.

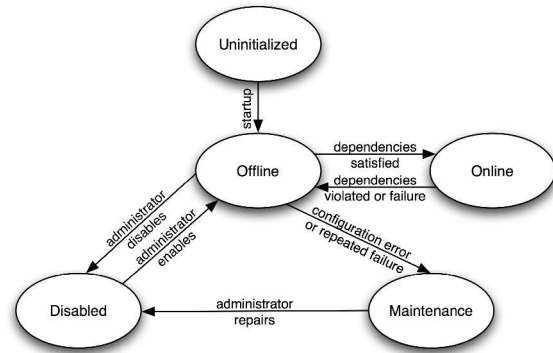


Figure 3: The life cycle state machine

As shown in Figure 3, a given service instance starts in the Uninitialized state and moves from there to the Offline state. When the restarter determines that its dependencies have been satisfied, the restarter starts it up and moves it to the Online state. Conversely, if a state change in some other instance results in unsatisfied dependencies or if the instance itself fails, the restarter stops the victimized instance and moves it back offline. An administrator may choose to disable the service instance, which causes it to move to the Disabled state, making an intermediate stop in the Offline state if it was online when the request to disable occurred. If something goes badly awry with the instance, it moves to the Maintenance state so that an administrator can diagnose and fix the problem and subsequently request that it go to Offline or Disabled after repairs are effected.

2.3 The service graph

The *service graph* combines individual descriptions of services and their instances, as described by their manifests, together into a single representation that captures all of the dependency relationships among them.

The graph is a straightforward translation of the XML manifests into Java objects. The subgraph for each service and its instances has direct object references connecting its nodes. However, references from one such subgraph to another are indirect; one instance refers to another by mentioning the other's FMRI, and the class representing the service graph provides methods for iterating over all its contained instances and for looking them up by name. This decoupling into subgraphs allows the structure of the graph to be easily updated as services and their instances are configured into and out of the system.

Figure 4 illustrates how these relationships apply to the service graph derived from the example in Section 1. The ftp and logger services each have a single instance. The ftp service has a dependency on logger's sole instance. It expresses this dependency as a symbolic reference to the instance's FMRI rather than as a direct object reference. logger's “port_6762” instance has a property group that holds its configuration properties.

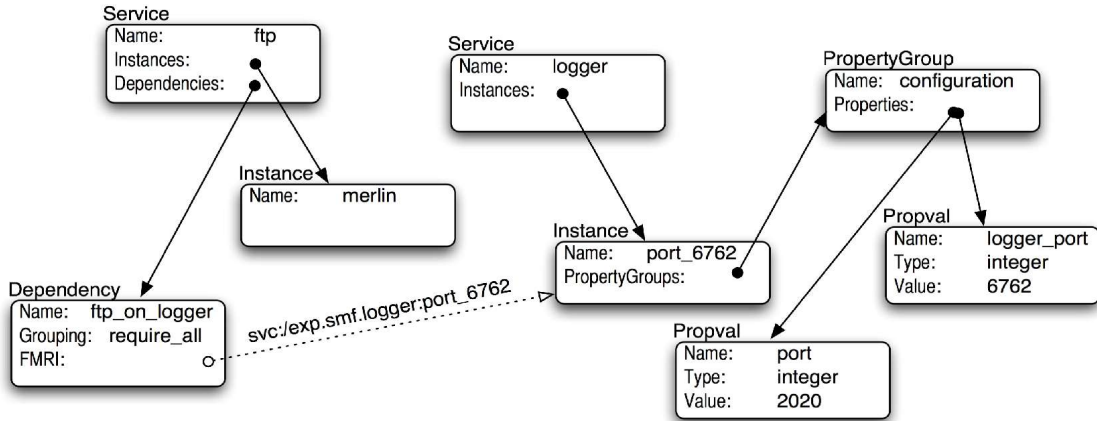


Figure 4: A simple service graph

3 Architecture

In our design we have paid particular attention to providing for fault tolerance. An overarching goal of the framework is to allow individual service instances to fail with minimal impact on other instances and with minimal (typically no) requirement for administrator intervention to recover from such failures.

However, we also want our framework itself to be resilient from failure. As discussed below, this requirement has had a pervasive effect on our design, leading us to factor it into multiple event-driven components.

A successful factoring into components goes hand in hand with well-defined interfaces among the components. Our primary interfaces are the *ServiceableMBean* and *RepositoryMBean* interfaces discussed above. Both interfaces are quite narrow; they enforce an arm's length relationship between a component calling one of their methods and the component implementing the method. The *RepositoryMBean* methods center around property manipulation, which encourages a loosely coupled publish-subscribe design. Our design follows this style by consistently setting properties that record intent to take some action before taking that action. If the component that registered its intent fails before acting on that intent, it can recover by reading the property it wrote to pick up from where it left off.

We house each framework component and each service instance in its own *isolate*, to ensure that any failure it might suffer is contained and does not affect other components or service instances.¹ Since isolates by definition cannot share objects, using narrow interfaces between them, as described above, becomes doubly important.

¹ The *Isolate* class of the *Isolate* API [8] provides the analog of the address space abstraction of traditional operating systems. Running code in an isolate provides confinement equivalent to running it in a separate JVM™.

The design considerations described above led us to factor our management framework into three major components, as illustrated in Figure 5. Arrows between components show the interfaces that each component uses to communicate with another.

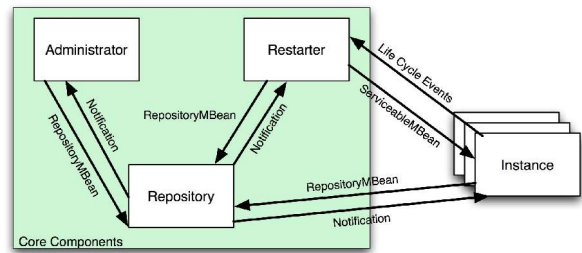


Figure 5: Component relationships

The first component is the *repository*. Its focus is providing persistent storage for the other components to use. By implementing the *RepositoryMBean* interface discussed above, it allows the other components to save and retrieve causally consistent views of state information pertinent to instances that have a dependency relationship with each other. The repository also supports JMX-style *notifications* [9], allowing other components to be informed when specific properties of interest are updated. The repository's final responsibility is to manage a persistent copy of the service graph.

The *restarter* centers around managing the service instances and relationships embodied in the service graph. As its name suggests, it starts service instances when their dependencies are first satisfied, and restarts them after failures or after violated dependencies have been restored. More abstractly, it shepherds service instances through their transitions between states in the life cycle state machine.

The *administrator* is the bridge to the outside world. It provides the foundation for user interfaces to our facility, such as the simple admin utility illustrated in the example in Section 1, by accepting external requests over a socket and translating them into property updates that it hands to

the repository. Other components then react to these updates to carry out the original requests.

In addition to the core framework components, Figure 5 also shows how service instances relate to those components. Each instance receives administrative requests from the restarter through its implementation of the `ServiceableMBean` interface. When an instance suffers a failure, it notifies the restarter by sending it a termination event. Service instances can also use the repository to store configuration information, using the same interfaces to the repository that the other components do.

4 Implementation

4.1 Repository

The repository has two main tasks: to provide access to the service graph and to properties named by FMRIs. To simplify its life it internally combines these into one, by storing properties as nodes within the service graph. Since the DTD describing the service graph as an XML document allows elements such as service and instance to have properties, this unified treatment makes sense. But it does introduce a small complication: the repository implementation must translate back and forth between property names given as FMRIs and objects corresponding to property elements in the DTD. For example, to fetch a property named `“svc:/exp.smf.ftp:merlin:/properties/restarter/next_state”`, the repository traverses the graph starting at the Service object named `“exp.smf.ftp”` through the Instance object named `“merlin”`, arriving at a PropertyGroup named `“restarter”`, where it accesses a map containing the `“next_state”` property.

Our current implementation of the repository is quite simple. It achieves the consistency properties that the `RepositoryMBean` interface requires by enqueueing incoming requests and serializing them by handling them one at a time. It achieves persistence by writing out the entire service graph as required, in XML form. This implementation has adequate performance for configurations involving a few dozen service instances. Ultimately, we plan to redo the implementation to use a database and to rely on the database for method invocation consistency and for persistence.

4.2 Restarter

The restarter's primary job is to manage service instances. To do so, it must maintain state information for each service instance. To meet its fault resilience requirements, this state must be recoverable after a restarter failure. Thus, the restarter uses the repository to record state information pertinent to each instance, storing each datum for the instance as a property in the `“restarter”` property group. It follows the `“record intent, then act”` pattern described above. For example, before invoking an instance's `start()` method to bring the instance on line, it

writes `Online` into the instance's `“restarter/next_state”` property, and subsequently uses the repository's `setProperty()` method to simultaneously clear that property and set `“restarter/state”` to `Online` after `start()` returns success.

When a given service instance changes state, every instance that depends on it is potentially affected, and the impacts on these instances can result in additional state changes that ripple outward from the original change. The restarter handles these cascades by associating an event queue and a thread to process that queue's events with each instance. When it changes a given instance's state, the restarter places a `Dependency` event on the queue for each instance that depends on the first. The threads for each of these instances process the events concurrently. When one of these threads determines that its corresponding instance needs to change state, it writes its intent to the repository as described above and then waits for the next event.

Another restarter thread waits for notifications of property changes from the repository. When one arrives, the thread determines which instance the property belongs to, generates an event describing the new property value, and places it on the instance's queue. In particular, the notification for the `“restarter/next_state”` property results in a `StateTransition` event, and the thread for the target instance responds by invoking one of the `ServiceableMBean` methods on the instance, changing the instance's state according to the result, and generating new `Dependency` events as needed.

This event-driven design with a thread per service instance eliminates the need for locking throughout much of the restarter's code, reducing it to ensuring that each event queue is properly synchronized as events are added and removed.

4.3 Administrator

The administrator provides support for external administrative utilities, and mediates between them and the rest of our framework. It reads command lines over a socket and interprets them, consulting with the repository to read properties for commands that request status information and to write properties for commands that request action. For example, its commands to get and set properties invoke the corresponding `RepositoryMBean` methods and relay the results back to the invoking utility.

Other commands, such as `enable` and `disable`, require interaction with the restarter. However, in keeping with our philosophy of using the repository to record intent, the administrator does not interact directly with the restarter. Instead, it writes properties in the `“restarter_actions”` property group that describe the action desired of the restarter. The restarter subscribes to these properties and handles notifications that they have changed by reading them and reacting accordingly. If such a command generates results, the restarter writes them to the

repository as properties and the administrator waits for property change notifications on these properties.

4.4 Service instances

Our framework includes support classes that service implementers can use as proxies and dispatchers for the `RepositoryMBean` and `ServiceableMBean` interfaces that connect the service to the components of our framework. To port a service to our framework, one must implement the methods of the latter interface. Since the methods of `ServiceableMBean` mirror common administrative tasks, this effort often amounts to simply writing the methods to forward calls to pre-existing methods.

5 Case studies

5.1 Turning a program into a service

As learning exercises, we wrote several simple services; the example in Section 1 shows two of them, `ftp` and `logger`, in action. Another, `hello`, is trivially simple; it writes a message given as a command line argument to a UDP port specified as a property. Since `hello`'s service consists solely of emitting its message, its `start()` method simply writes its message and returns, and its `stop()` method does nothing.

`ftp`'s methods have more substance; `start()` enables listening for new client connections, and `stop()` disables the listener and terminates existing connections. `logger` is similar; its `start()` method sets up a listening socket, and its `stop()` method disables new logging requests and then allows any in-progress requests to complete before returning.

Many existing programs can be turned into services as simply as in the FTP server example. However, in some cases one may have to depart from this pattern. Our experience with the Cloudscape database is a case in point. Cloudscape [6] is a pure Java relational database management system that can be embedded in Java programs. The most recent version of Cloudscape is open-source but we chose an older distribution that we had been using in other projects.

As modifying the sources was not an option, we had to consider various execution paths to ensure that none of them left the database in an inconsistent state or compromised service management integrity. The critical issue turned out to be database shutdown, which is accomplished by a ping program, typically run in a separate virtual machine. Just like any other interaction with cloudscape (e.g., executing SQL statements), ping creates a socket connection to it and sends an appropriate request. Both ping and cloudscape can internally call `System.exit()`. Co-locating them in a single isolate can lead to inconsistencies and races: if ping returns before cloudscape manages to cleanly shut down, the database may get corrupted; and if cloudscape is the first to exit, ping may be unable to report successful shutdown.

Moreover, cloudscape can exit asynchronously due to an internal problem or due to a lack of resources (e.g, disk space). Co-locating it with the service management code can thus cause the death of both.

These considerations led us to the following architecture. The framework sees the Cloudscape service as a `CloudscapeSvc` isolate. When switching its state to `Online`, `CloudscapeSvc` creates another isolate, which executes the database. When requested to stop, `CloudscapeSvc` creates an auxiliary ping isolate that properly terminates cloudscape. Thus, `CloudscapeSvc` is always accessible, while the actual service it manages can be turned on and off as requested. Users of the service are unaware of the implementation details described above.

Arguably these steps are overly defensive and not necessary in most situations. However, they nicely illustrate the kinds of issues one must consider when attempting to transform existing code into a service.

5.2 A load-balanced application server

A clustered Java 2 Enterprise Edition (J2EE™) Application Server is an example of a complex real-world system that typically consists of multiple services. A common configuration might be an administration server, one or more servers hosting J2EE applications, a load balancer that distributes requests to these servers, and a database server. Current J2EE servers manage these services through ad hoc combinations of vendor-specific scripts and custom code.

In previous work [7] we described refactoring the J2EE 1.3.1 Reference Implementation (J2EERI) into a collection of components, each residing in an isolate. For example, each application is executed in an *application domain*, which is an isolate executing the application (e.g., its `.ear` file) as well as the necessary middleware code. The resulting component structure, shown in Figure 6, led to a much cleaner design and performance benefits but still suffered from the inadequacies of life-cycle management support offered by current application servers.

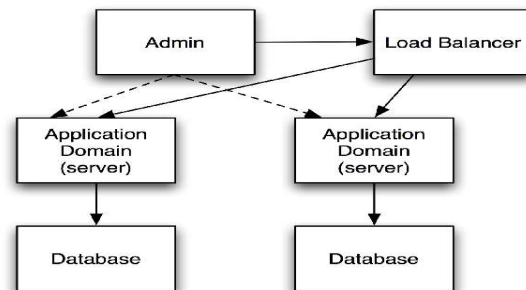


Figure 6: Application server components

Modifying the isolate-based J2EERI to utilize the service management framework was straightforward and yielded immediate benefits. Since each server component was already available as an isolate, it was easy to

construct a wrapper that implemented the start() and stop() methods.

One virtue of using the framework was the opportunity to consolidate the various mechanisms for configuring the components, e.g., Java properties, configuration files, and command line arguments, into one place, namely the service manifest file.

Each application domain instance depends on a Cloudscape instance as the database server. In current application servers, this dependency is specified implicitly in the JDBC connection pool configuration. If the database server is not available, the usual result is an application failure when trying to connect. In our environment, the dependency is specified explicitly using the dependency element of the application domain's manifest. If the database server is unavailable, the restarter will take the application domain instance offline until it becomes available. One might argue that this is an overly aggressive response, as the application domain may offer other services that do not require the database server to be running. Refactoring the system into finer grain services to increase the dependency specifications' precision would address this problem.

The load balancer uses the repository's state-change notification mechanisms, since it needs to keep an accurate record of which application domain instances are available. Unlike the application domain's dependency on the database, which is one-to-one, the load balancer's dependency on the domains is one-to-many. This type of dependency is supported in our service management framework by the "require_any" annotation in the appropriate manifest file. It is not currently possible to state a "require-N" dependency, although such an extension would be straightforward.

5.3 SEDA

The Staged Event Driven Architecture (SEDA) [17] is a framework designed for developing server applications that perform optimally under the heavy and fluctuating loads experienced by many internet-based services.

SEDA applications are built from a set of stages, each of which has an associated event queue, an event handler, and one or more controllers. Event handlers contain the application logic and, after processing events from their queue, typically queue additional events at other stages. The controllers manage allocation of resources to their stages, for example, the number of threads allocated to event handling in a stage.

Stages in SEDA are conceptually isolated from each other and events are recommended to be immutable. SEDA is therefore a good candidate for our framework, which can enforce these architectural constraints by factoring stages out as separate isolated service instances. The standard implementation, Sandstorm [17], restricts stages to a single Java™ Virtual Machine, forcing them

into a single service instance. We modified it to allow stages to be separated into distinct service instances.

Assembling a SEDA application requires the constituent stages to be wired together by connecting the outputs of one stage to the event queues of other stages. The set of stages to be used in a given application is specified in a configuration file, and application logic is inserted by specifying the event handler for each stage. Stage wiring is managed explicitly in each stage's initialization code by looking up downstream stages and acquiring handles to their event queues. Initialization is multi-pass; first all the stages are created, then the stages are connected, and finally events can flow.

Running stages as cooperating service instances requires careful management of the interaction between stages grouped in different instances. In Sandstorm, stages obtain handles for other stages through a stage manager that acts as a naming service. In our modified version, we rewrote the stage manager as a pair of services. The localManager service groups together a specified set of stages and the globalManager service coordinates the local managers. Application logic only interacts directly with its local manager. If a local manager is asked for a stage it cannot find locally, it delegates to the global manager, which then finds the service instance containing the stage. Thus the overall SEDA service consists of a single instance of a global manager service and several instances of a local manager service. The dependencies between these services and their instances are straightforward: every local manager depends on the global manager, and one local manager depends on another if a stage in the former needs to send an event to a stage in the latter.

Sandstorm is typically demonstrated with one of its sample SEDA applications. The Haboob web server [17] is a staged web server that illustrates SEDA's scalability under heavy load conditions. We have modified it to factor its stages into services. This modified version of Haboob decomposes into four distinctly configured instances of localManager: HttpServer, which manages HTTP connections and HTTP request parsing; receiver, which handles GET requests; cache, which manages a file cache; and sender, which handles GET replies.

HttpServer manages incoming HTTP connections and passes GET requests on to receiver, which passes file requests to cache, which in turn obtains the contents of that file and hands them to sender. Finally, sender sends an HTTP response back to HttpServer. These instances each use one or more stages to provide their service.

The stages described above have a cyclic dependency; to make them manageable in our framework, we had to break the cycle. We chose to reverse the dependency of HttpServer on receiver, which has the consequence of requiring HttpServer to be able to run without receiver. When receiver is available, HttpServer sends incoming requests to it; otherwise it drops them. Figure 7 shows the resulting dependencies.

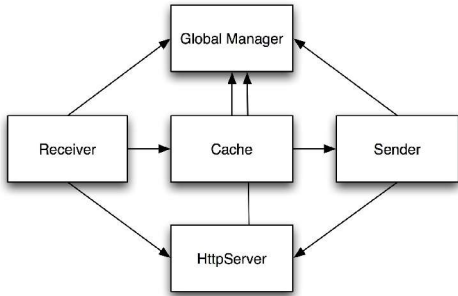


Figure 7: Haboob dependencies

This configuration of Haboob exhibits the benefits of our framework: its components configure themselves properly from their manifests, start automatically as their dependencies are satisfied, fail independently, and restart without manual intervention after failure.

5.4 Monitoring framework performance

The preceding case studies have focused outward, demonstrating our framework's design center of managing non-trivial service configurations. In this subsection, the focus turns inward, to using simple sleep and tracker services to illuminate aspects of the framework's performance.

When an instance of sleep starts, it delays by a configurable duration before returning from its start() method. By default, sleep acts as a *ticker*; after a given instance's delay period expires, the instance exits, the restarter notices that the instance has exited, and it restarts the instance, causing the cycle to repeat. However, one can also configure sleep as a *transient* service; doing so instructs the restarter to stop monitoring such an instance after its start() method returns successfully, thereby turning the instance into a one-shot until it is disabled and re-enabled.

By making another service instance depend on a transient sleep instance, one can influence the sequence in which services start up. For our purposes here, we configured sleep's delay to zero, thereby making it act as a *null* service that does nothing other than return success and exit.

The tracker service registers for notifications from the repository of changes to all properties whose names match "restarter/state", thereby arranging to learn of every state transition for every service instance. When it receives a notification, each tracker instance uses logger to extend a log file with a new record. The overall effect is to capture a timestamped history of our framework's state transitions.

We ran the experiments described below on a SunBlade 2500 workstation with two 1.28GHz SPARC® processors and 2Gbytes of memory. The machine was running a Solaris™ 10 desktop environment during the experiments, but was otherwise unloaded. We ran our framework and the service instances it managed under MVM [2], a research derivative of version 1.3.1 of the

Java HotSpot™ virtual machine [14], that implements the Isolate API.

Our first experiment examined the question of what our framework's raw performance is in starting a service instance. We configured a single non-transient instance of the null sleep service. In this configuration, the instance exits immediately after coming online, causing the restarter to mark it offline and then restart it. Measuring the rate at which the instance enters the Offline (or, equivalently for this configuration, Online) state gives a rough measure of overhead. The time from one entry to the next includes communication between the restarter and sleep to start it and observe its death, and sleep's execution time itself, which in turn includes communication from sleep to the repository to obtain the desired sleep time.

On a run that included over 1000 restarts we measured a restart rate of 13.85 restarts per second, which corresponds to a full cycle time of 72.15 milliseconds. We regard this as acceptable for our untuned implementation.

Our second experiment examined the restarter's efficiency in propagating dependency events from one service instance to another that depended on the first. We set up a configuration of ten instances of the null sleep service, each specified as a transient. Each instance depends on the one before it, so that before the tenth instance can start, all nine of its predecessors must be online. Similarly, disabling the first instance causes the second through tenth to successively go offline, as dependency violations propagate down the chain.

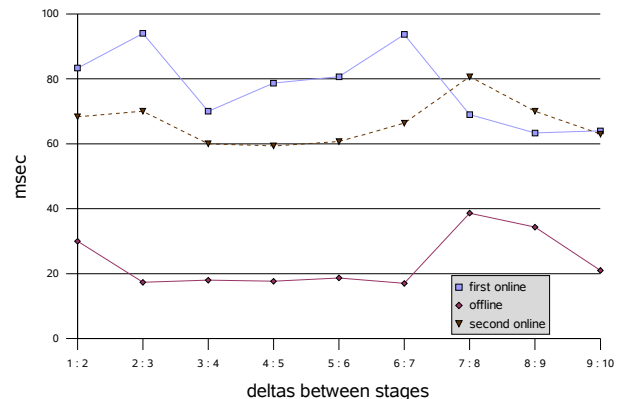


Figure 8: A Chained Dependency Experiment

Figure 8 displays the results of starting the configuration, disabling the first instance, and then re-enabling the first instance. Each line shows one set of transitions, with the value at each column giving the incremental time taken for a successor stage to react to a state change in its predecessor. The lines are roughly horizontal, indicating that the time required to act on an instance does not depend on its position in a dependency chain, but there are substantial deviations that we suspect are due to the internal operation of the virtual machine. The time required for the restarter to bring each sleep instance

offline is consistently lower than the time required to start or restart that instance. This difference arises from the fact that the restarter has to do less work; since the sleep instances are transient, they have already exited when the restarter attempts to bring them offline by calling the stop () method, causing communication to short-circuit with an immediate exception.

6 Related work

Our work was largely inspired by the Solaris Service Manager [13, 16], which defines an architecture as well as a specific implementation for the Solaris Operating Environment. Its primary design center is the management and restart of Unix™ process-based services, but it also offers support for multiple application models through the concept of delegated responsibility for a service. For example, the Solaris implementation delegates responsibility for managing network services to a revised version of the inetd daemon. A service is defined by the set of attributes appropriate to its application model, but the framework encourages a minimal common set, so that widely varying software components may be treated as manageable services. As a result, service definition requires an understanding of the relevant application model, as well as use of a facility-specific set of interfaces.

Viewed narrowly, our work is an instantiation of this architecture for the Java platform, defining a precise service object model around the Isolate abstraction, expressed by JMX MBean interfaces. Rather than fitting directly into the Solaris implementation by using its delegation support, our implementation is a free-standing realization of the architecture that is adapted to the characteristics of the Java platform. As that platform's capabilities expand, through additions such as the Isolate API and the Resource Management API we developed in previous work [3], we have the opportunity to expand the capabilities of our framework beyond what the Solaris architecture can easily contemplate, for example by extending our management scope to clusters of machines with dependencies extended to include statements of resource requirements.

The Java Management Extensions (or JMX) [9] specify an architecture and design patterns for managing Java applications. The central JMX design pattern is the MBean, which specifies how to define management operations and attributes. Instantiating this pattern for a given application yields an interface for managing that application. JMX also provides a set of extensible classes for producing and consuming notifications of events of management interest. The JMX architecture includes the concept of an agent, which is an entity that exerts control over an application. As most applications execute separately from their management agents, the architecture includes a notion of adapters that convert between

external protocols and invocations through JMX interfaces.

Our management framework can be regarded as an instantiation of the JMX architecture that provides an agent specialized to managing the life cycles of applications that deliver services. The requirement that services implement the ServiceableMBean interface allows for the possibility that they can extend it with other management operations relevant to other JMX agents. Our proxy and dispatcher classes act as JMX adapters to the Isolate API's Link and LinkMessage classes for inter-isolate communication [8].

The OSGi Service Platform [10] provides service life cycle management and includes a notion of dependency; refreshing a given service causes services depending on that one to stop, re-resolve their dependencies, and restart. In this respect the OSGi platform is very similar to our work. However, it is focused more toward treating services as components that integrate together within a single JVM to build up applications. Although it includes a strong security model that can be used to forestall malicious interference among services, a given service is still vulnerable to denial of service attacks or to crashes. In contrast, our framework's foundation on the Isolate API allows us to tolerate faults and malicious behavior in individual services that would cripple a similar OSGi configuration. However, this foundation also encourages a more arm's length relationship style among services and thus discourages the closely coupled service relationships that the OSGi platform fosters.

Project Rio [12] provides a model to dynamically instantiate, monitor, and manage service components as described in an architectural meta-model. The meta-model provides context on service requirements, dependencies, associations, and operational parameters. Key to the dynamic service provisioning model is the recognition that the network is composed of heterogeneous compute resources with multiple architectures and operating systems, all with different capabilities. A lightweight dynamic "agent" called a *Cybernode* turns heterogeneous resources into services available through the network. Cybernode instances dynamically discover and enlist with dynamic provisioning agents, and provide a lightweight container to instantiate mobile services.

Rio's focus is on managing services in a dynamically changing networked environment. Where our framework is oriented toward specifying precise relationships among a known set of components, Rio relies much more on discovery. Where our framework provides strong fault confinement and detection, Rio relies on whatever capabilities happen to be present where a given Cybernode is hosted. Both systems address the same general problem space and each provides features that the other lacks. An interesting topic for future research, and one we hope to pursue, is to investigate ways of blending them together.

The key insight of the Recovery-Oriented Computing project [11] is that availability depends both on the mean time to failure and the mean time to recover from a failure. For a fixed recovery time, successive improvements to the mean time between failures become increasingly hard to achieve and yield diminishing returns. Thus, improving recovery time becomes an easier and more fruitful way of improving availability. Our facility fits nicely with this philosophy; by automating recovery from failures, we eliminate the need for human intervention and significantly improve recovery times. More subtly, our framework's dependency tracking helps prevent cascading failures and corresponding availability losses. By noticing that a service instance has failed and by bringing instances that depend on it offline, our framework reduces secondary availability losses from the time it would take the dependent instances to recover and then restart to just the time to stop and then restart.

7 Conclusions

This paper has described a service management facility tailored to the Java platform. The facility's major benefit is automated reaction to a failure of a service: it promptly restarted, along with other impacted services. In addition to the potential improvement to such availability metrics as mean time to repair, the framework rationalizes the ordering of service creation during boot time, preventing a class of hard-to-detect configuration errors. It also facilitates routine aspects of service management, such as taking a service offline for an upgrade.

Incorporating a service into the framework is a two-step process. First, one has to analyze what dependencies the service has, either as a dependent or a dependee, with other services already in the framework. Second, one must write connector code to control the service's life-cycle. Our experience, based on several small examples and two larger ones (a J2EE application server and a SEDA-based Web server) demonstrates that both steps are easy and that the framework is attractive from a practical standpoint.

The architecture and results presented in this paper are by no means the end of the road; rather, we view them as a solid foundation for further exploration. We are planning to realize the full potential of fault tolerance by segregating key components into isolates and implementing recovery code. Progressing towards other services may require re-examining and refining the notions of dependencies and restart conditions. We are also planning to enhance the expressiveness of dependency definitions to allow for requesting resources necessary for a service to thrive. Finally, even though the framework and described experiments execute on a cluster (as immediate fallout from cluster extensions to our underlying virtual machine [4]) the relationship between inter-service dependencies, placement on cluster nodes, and state replication has not yet been examined and is on our short term research agenda.

8 References

- [1] Cardelli, L. *Program Fragments, Linking, and Modularization*. In 24th ACM POPL, Paris, France, January, 1997.
- [2] Czajkowski, G., and Daynes, L. Multitasking Without Compromise: A Virtual Machine Evolution. 17th ACM OOPSLA'01, Tampa, FL, October, 2001.
- [3] Czajkowski, G., Hahn, S., Skinner, G., Soper, P. and Bryce, C. *A resource management Interface for the Java platform. Software—Practice and Experience*, 2005; **35**: 123-157.
- [4] Czajkowski, G., Wegiel, M., Daynes, L., Palacz, K., Jordan, M., Skinner, G., Bryce, C. *Resource Management for Clusters of Virtual Machines*. Submitted for publication.
- [5] Drossopoulou, S., Lagorio, S., and Eisenbach, S. *Flexible Models for Dynamic Linking*. In 12th European Symposium on Programming, Warsaw, Poland, April, 2003.
- [6] IBM Cooperation. *IBM Cloudscape*. <http://www-306.ibm.com/software/data/cloudscape>.
- [7] Jordan, M., Daynes, L., Czajkowski, G., Jarzab, M., and Bryce, C. *Scaling J2EE(TM) Application Servers with the Multi-Tasking Virtual Machine*. Sun Microsystems Laboratories Tech. Report SMLI TR-2004-135, June, 2004.
- [8] Java Community Process. JSR 121: Application Isolation API. <http://jcp.org/jsr/detail/121.jsp>.
- [9] *Java™ Management Extensions Instrumentation and Agent Specification, v1.2*. Downloadable from <http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html>.
- [10] OSGi Alliance. *About the OSGi Service Platform*. http://www.osgi.org/documents/osgi_technology/osgi-sp-overview.pdf.
- [11] Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., Treuhaf, N. *Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*. UC Berkeley Computer Science Technical Report UCB-CSD-02-1175, March, 2002.
- [12] Reedy, D. *Project Rio: A Dynamic Adaptive Network Architecture*. Available from <http://rio.jini.org/>.
- [13] Shapiro, Michael W. "Self-Healing in Modern Operating Systems". *ACM Queue*, vol. 2, no. 9, 2004, pp. 67-75.
- [14] Sun Microsystems, Inc. Java HotSpot Technology™. <http://java.sun.com/products/hotspot>.
- [15] Sun Microsystems, *Solaris 10 Reference Manual, smf_manifest(5)* entry. Available as <http://docs.sun.com/app/docs/doc/816-5174/6mbb98ujl?a=view>.
- [16] Sun Microsystems, *Solaris 10 Reference Manual, smf(5)* entry. Available as <http://docs.sun.com/app/docs/doc/816-5175/6mbba7f3m?a=view>.
- [17] Welsh, M., Culler, D., and Brewer, E. *SEDA: An Architecture for Well-Conditioned, Scalable Internet Services*. Proceedings of the 18th Symposium on Operating Systems Principles, Banff, Canada, October, 2001.
- [18] Liu, Y., and Smith, S. *Modules with Interfaces for Dynamic Linking and Communication*. European Conference on Object-Oriented Programming, Oslo, Norway, June, 2004.