SOFTWARE SERVICES LIFECYCLE PRIMER

CONTEXT SURVEY & RELATED WORK

Robert F. MacInnis, Ph.D.
June 2017

AetherWorks
501 Fifth Avenue
New York, NY 10017

E: info@activeaether.com

1 INTRODUCTION

This document begins by establishing a concrete vocabulary for the discussion of Web Services. It defines the actors, artifacts, activities, and objectives of each stage in the software service lifecycle and uses these terms to introduce the concepts, roles and standards of Web Services. It continues with a discussion of each stage in the Web Service lifecycle and finishes with a comprehensive survey of work addressing shortcomings of the traditional Web Services model.

2 SOFTWARE SERVICE LIFECYCLE

The software service lifecycle may be simplified into three macro stages of design & development, deployment, and management, each of which has multiple sub-stages [1]. Authors ascribe various names to these stages and provide inconsistent groupings of their sub-stages and tasks. The following sections address this ambiguity by defining a common vocabulary and a concrete set of lifecycle stages which will guide the description and discussion of work presented in this document.

2.1 Terms

The terminology of Web Services is insufficient for discussing work which addresses the shortcoming of the traditional Web Services interaction model. This section begins by introducing the broader terminology of component-based distributed systems and then proceeds to the actors, actions and objectives of each macro lifecycle stage. Later, these definitions and descriptions will be used to describe the lifecycle activities of Web Services, followed by work which addresses shortcoming of the traditional Web Services interaction model.

This document adopts the widely-used terminology of the Object Management Group (OMG) [2] specifications. The following definitions represent terms commonly used to describe distributed software systems together with a sub-set of OMG deployment terminology described in [3].

- **Artifact:** A physical piece of information that is used or produced by a deployment process. Examples of artifacts include models, source files, scripts, and binary executable files. An artifact may constitute the implementation of a deployable component.
- **Capability:** A feature offered by a component implementation.
- **Component:** A modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.
- **Domain:** A target environment composed of independent nodes and resources.

- **Installation:** The act of taking a published software package and bringing it into a repository.
- **Interface:** A named set of operations that characterize the behavior of an element.
- **Implementation Artifact:** An artifact used or produced as a result of an implementation (usually "executable code").
- **Launch:** The process of instantiating components on nodes in the target environment according to a deployment plan.
- **Metadata:** Information that characterizes data.
- **Node:** A run-time computational resource which generally has at least memory and often processing capability.
- **Package:** An implementation, or set of interchangeable implementations, contained in a set of artifacts and compiled code modules.
- **Repository:** A facility for storing metadata, and implementations.
- **Requirement:** A feature requested by a component implementation. Monolithic implementation requirements must be satisfied by node resources.

2.2 Lifecycle Stages

The following three sections introduce each stage and sub-stage of the software service lifecycle. These lifecycle stages consume portions of the generic lifecycle activities common amongst software services first characterized in [4] and later specified by the OMG in [3].

2.3 Design & Development

The design & development stage is composed of three sub-stages: *Specification*, *Implementation*, and *Publication*. As shown in Fig. 1, the completion of these stages yields an installable package that includes the code and metadata describing the software component. The design process begins with a *Specifier* actor defining an interface which describes the behavior of the component. This interface is called a *ComponentInterfaceDescription* and is passed to the actor responsible for implementation.

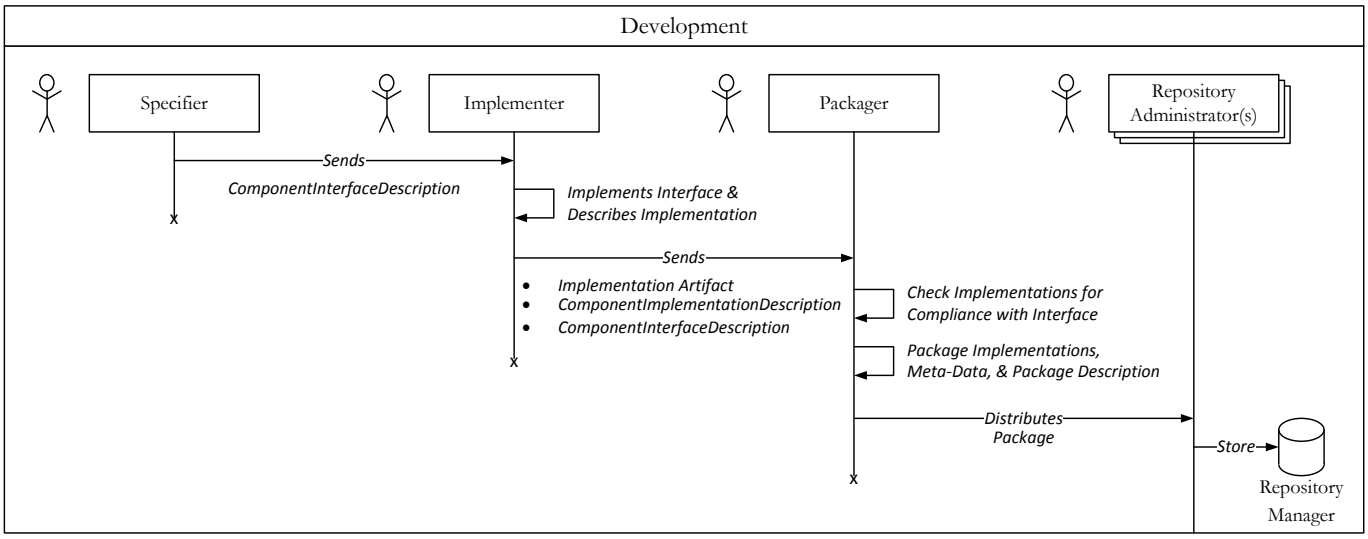

Fig. 1 The actors and actions of the Design & Development lifecycle stage.

The development process involves the *Implementer* actor writing the business logic of the component by creating an implementation artifact which implements the ComponentInterfaceDescription. This sub-stage can be completed by a *Developer*, who writes a monolithic implementation, or an *Assembler*, who uses existing components as building blocks. This sub-stage yields an implementation artifact which serves as the code module of the package.

The implementing actor must also describe the implementation with a *ComponentImplementationDescription*. An Assembler creates a *ComponentAssemblyDescription* which describes the component assembly in terms of its sub-components. A Developer creates both a *MonolithicImplementationDescription* describing the component implementation, as well as an *ImplementationArtifactDescription* describing the component's requirements of the target environment. These descriptions are combined with the original ComponentInterfaceDescription and serve as metadata of the package.

Packaging is the final step in the development process and is carried out by the *Packager* actor. The Packager begins by combining the implementations with their descriptor documents, ensuring that the component implementations conform to the component interface, and describing the package as a whole using a *ComponentPackageDescription*. The combined artifacts and compiled code modules comprise the final package which is ready for installation.

2.3.1 Installation & Configuration

The *Repository Administrator* actor is responsible for installing component packages – that is, storing them into a repository which is accessible to the entity responsible for deployment. The Repository Administrator is also responsible for setting and updating package configurations. Once a component package is created by the Packager it is distributed to Repository Administrators for installation and configuration.

Repository Administrators access repositories through a *RepositoryManager* interface which provides operations for installing, configuring, retrieving and removing component packages. While installation makes packages available for deployment, it does not involve moving them to the machines on which the components will be run. This task will be performed later during the 'Preparation' process of the Deployment lifecycle stage.

2.4 Deployment

Software deployment may be defined to be the process between the acquisition and execution of software [5]. Once a software component has been developed, packaged and installed, it is ready to be deployed. The deployment process consists of three steps: planning, preparation, and launch. Planning is carried out by the *Planner* actor and involves creating a *DeploymentPlan* dictating how and where software will be deployed. Preparation and launch are carried out by the *Executor* actor and involve executing the DeploymentPlan, preparing the target environment for launch, and orchestrating the 'activation' of each component in the deployment.

The following sections begin by defining the features and traits of target environments and the actors responsible for their administration. The Deployment sub-stages of planning, preparation, and launch are detailed next, followed by an overview of the techniques used to execute a DeploymentPlan.

2.4.1 Target Environments

Target environments – or 'Domains' – consist of a distributed system infrastructure comprised of nodes on which the software will ultimately run [3]. Target environments are administered by a *Domain Administrator* actor who describes the environment in terms of its nodes and shared resources. Each individual node in the domain is defined by its own resources – such as processing power, memory, and operating system – and domains may consist of nodes with varying resources. Nodes are managed by a *NodeManager* which is responsible for instantiating components on the node.

When the DeploymentPlan is executed in the following sub-stages, the deploying process does not control the target nodes directly. Instead, it interacts with them through their NodeManager interface, allowing for the software supporting component execution on the node to be functionally decoupled from the details of the deployment process as a whole. This separation allows a target environment to support heterogeneous nodes and component implementations without changing the implementation of the deployment system.

2.4.2 Planning

The Planner actor is responsible for deciding how and where software will be deployed. The decision making process involves matching software requirements with target environment resources and creating a DeploymentPlan to guide the 'Preparation' sub-stage. A valid DeploymentPlan describes a deployment of an application using concrete implementations that match requested selection properties, and an assignment of these implementations to nodes so that node resources match or exceed the requirements of component instances that are deployed on them [3]. The activities of the planning process are shown in Fig. 2.

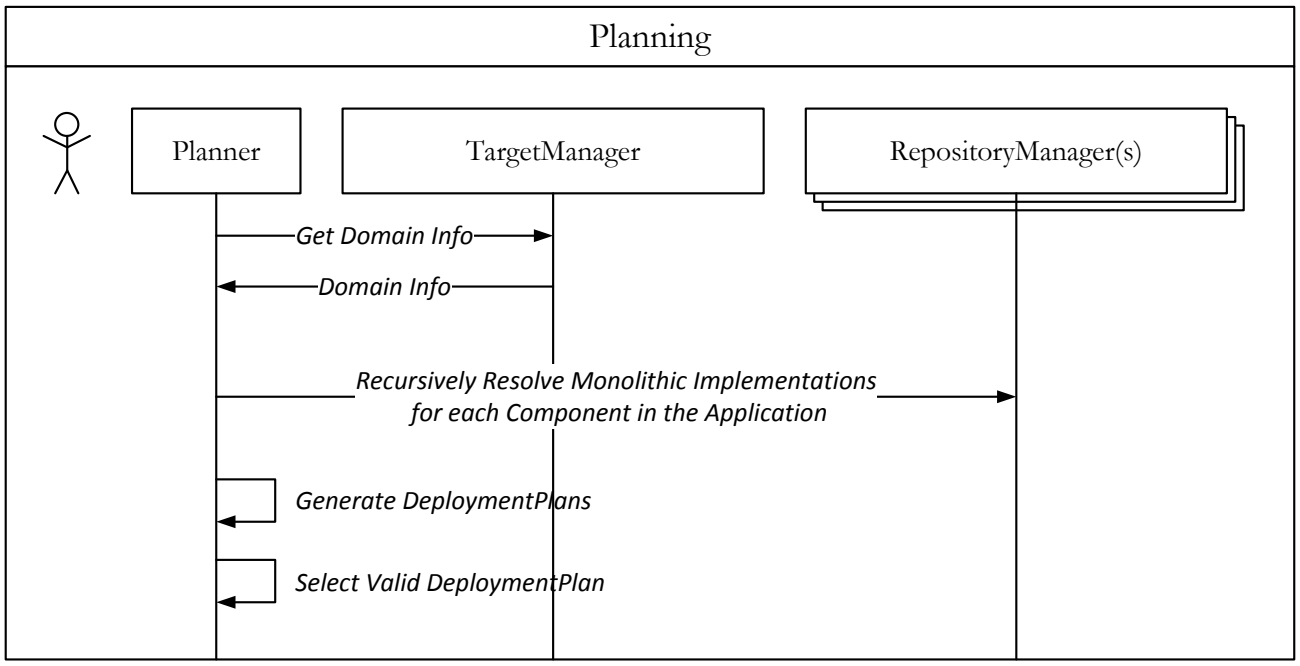

Fig. 2 The processes of the Planning sub-stage, including information retrieval, generation of candidate DeploymentPlans, and selection of a valid DeploymentPlan to guide application Preparation and Launch.

The Planner begins the planning process by retrieving information about the target environment from a *TargetManager* controlled by the Domain Administrator. The Planner then contacts one of potentially many RepositoryManagers to find a package and ComponentPackageDescription for the application to be

deployed. If the package contains an implementation which is an assembly then the Planner must retrieve packages for each sub-component in the assembly until it has monolithic implementation artifacts for all components in the application.

The next stage in the planning process involves generating candidate DeploymentPlans by matching node resources to implementation requirements, then selecting a plan for use. Deciding which plan to use, and when the choice is made, depends largely on the resource behavior of the target execution environment. In environments with static resource availability, DeploymentPlans may be pre-generated, reducing the time needed for deployment. Environments with dynamic resource availability will need to generate and select plans on demand in order to ensure that the DeploymentPlan is valid.

The process of generating and selecting DeploymentPlans is implementation-specific and can be one of the most complex lifecycle tasks. Generating all possible configurations for a low-complexity deployment on even a small set of nodes can yield an astronomical number of equally valid DeploymentPlans. Further, the process and criteria for comparing and ranking DeploymentPlans can be very complex and time-consuming. Choosing the right approach to planning can have a large impact on the timeliness and effectiveness of the deployment process; the various approaches to planning and their implications for deployment are discussed further in section 2.4.5 'Deployment Techniques'.

2.4.3 Preparation

A valid DeploymentPlan is used by the Executor actor during the Preparation stage in order to prepare the target environment for software launch. To prepare the deployment, the Executor sends a DeploymentPlan to the *ExecutionManager* who is responsible for managing the execution of the application into the domain.

The Preparation process, shown in Fig. 3, begins with the creation of *ApplicationManagers* at both the domain and individual node levels which are capable of enacting a particular DeploymentPlan, possibly multiple times. The ExecutionManager first creates a domain-level manager called a *DomainApplicationManager.* For each node in the DeploymentPlan the DomainApplicationManager creates a partial DeploymentPlan representing the single node's deployment responsibilities. Each node's NodeManager is sent their partial DeploymentPlan and creates a *NodeApplicationManager* capable of enacting it on the node. These domain- and node-level managers will be used later during the 'Launch' process to orchestrate and enact the execution of the application.

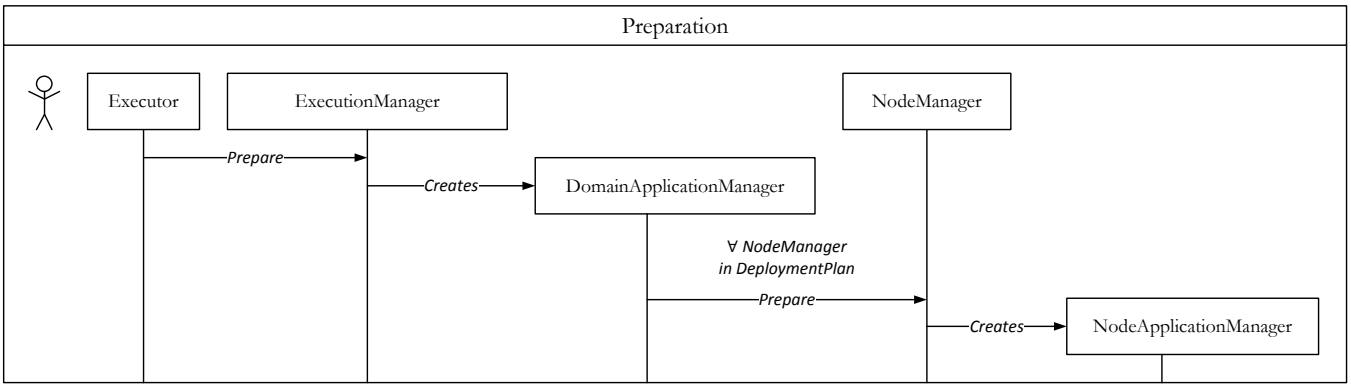

Fig. 3 The Executor is responsible for preparing the target environment for application launch.

As the semantics of the Preparation process are undefined, this phase can be a source of features which differentiate deployment system implementations. While typical Preparation tasks include retrieving component packages from a RepositoryManager and moving them to the nodes where they will be executed, the coordination and timing of these and various other tasks can be tailored to exhibit vastly different behavior. Synergies with other Deployment sub-tasks can also be realized during Preparation, affecting exhibited behavior even further. Pre-loading machines with artifacts, for example, may increase launch speed but will also reduce available resources. The Preparation phrase could be intelligently coordinated with the Planning phase, however, by using pre-generated DeploymentPlans to *selectively* pre-load artifacts, reducing the original scheme's resource consumption while retaining the gained launch speed.

The Preparation process is finished when the target environment is prepared for launch according to the implementation-specific criteria of the deployment system. The stage is signaled as complete by the ExecutionManager returning the DomainApplicationManager reference to the Executor. Once returned, the Executor can use the launch operations of the DomainApplicationManager to instantiate the application and conclude deployment.

2.4.4 Launch

The Executor uses the DomainApplicationManager produced by the Preparation stage to finalize the software deployment process during the Launch lifecycle stage. A successful launch brings the application into an executing state and results in an object that satisfies the ComponentInterfaceDescription described by the Specifier in the Design & Development stage.

The Executor launches an application in two separate steps called 'start launch', shown in Fig. 4, and 'finish launch', shown in Fig. 5. The process begins with the Executor calling a 'start launch' operation on the

DomainApplicationManager which creates a *DomainApplication* representing a single instance of the deployed application. This DomainApplication is responsible for enacting the DeploymentPlan by calling the 'start launch' operation of the NodeApplicationManager on each node of the deployment. The 'start launch' operation signals each NodeApplicationManager to instantiate a *NodeApplication* representing a single instance of the node-level partial deployment of the application.

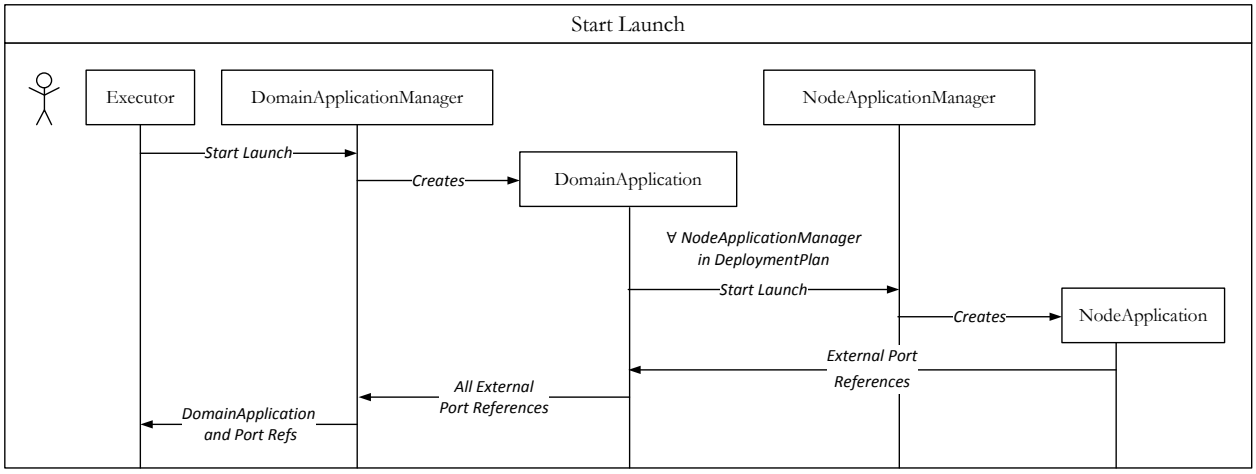

Fig. 4 The Launch stage is initiated through a series of 'start launch' calls resulting in the creation of domain- and node-level component deployments.

The NodeApplication can carry out component- and node-specific actions such as loading software into memory or opening ports in preparation for use – after which the application is deemed to have been 'executed' but not yet 'started'. Once each NodeApplication is ready to start it returns the DomainApplication a list of the external ports provided by newly instantiated application components on the node. The DomainApplication waits until all nodes are ready to start before returning the NodeApplication port references to the DomainApplicationManager. The DomainApplication reference and the port list are both returned to the Executor who will use them during the 'finish launch' step.

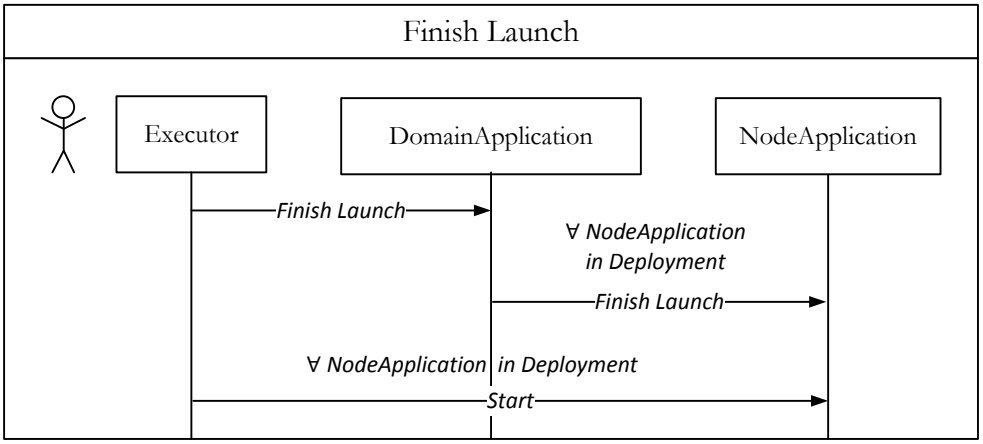

Fig. 5 The Deployment process is completed by calls to 'finish launch' on each deployed component; the timing and order of their 'activation' is controlled by the Executor issuing explicit commands to 'start'.

The Executor finishes deployment by calling the DomainApplication's 'finish launch' operation with the list of external ports. The DomainApplication coordinates the dissemination of the port references, calling 'finish launch' on each NodeApplication in the deployment and sending them the list of external ports so that they can communicate with other components in the deployment.

Deployment is concluded by the Executor sending each NodeApplication a command to 'start'. At this point the entire deployed application is deemed to be 'started' and the launch stage is complete; the application has been deployed.

2.4.5 Deployment Techniques

While a successful deployment always results in the instantiation of one or more objects that satisfies their ComponentInterfaceDescription, there are a great variety of techniques used to carry out the individual deployment tasks. All of the roles in Deployment can be carried out by different entities and through different means, such as manually by a system administrator or automatically by a computer-based deployment system. The ability of an application to adapt to changes in the target environment – such as rising usage levels or node failure – is largely dictated by the techniques used to develop and deploy the application.

The techniques utilized for deployment can be categorized as manual-, script-, language-, or model-based, and are selected based on system scale, complexity, and the expectation of change. The effectiveness of the chosen approach is a function of its suitability to the deployment environment, the availability of

management and monitoring facilities (if any), the skill of the developers, and a willingness to invest time and resources [6].

Using manual deployment, the Planner and Executor roles of the Deployment lifecycle stage are assumed by a person (such as a system administrator) who is responsible for performing all of the tasks by hand. The individual must acquire the software from a repository, resolve all component package references, gather and analyze target environment resources, then manually prepare and execute a DeploymentPlan. This approach is practical for testing individual components – monolithic implementations in particular – but is clearly unsuitable to any but the smallest and least complex deployments.

Executors may introduce automation into the process of executing a DeploymentPlan through the use of shell scripts. By formalizing the manual deployment tasks and chaining them together into a programmatic deployment script, individuals may repeatedly execute a single DeploymentPlan. Writing deployment scripts requires a larger initial investment of time during the development phase, but for low-complexity systems and highly homogeneous target environments it can provide a cheap means of performing large-scale deployments.

Manual- and script-based techniques are useful for the deployment of simple applications either singularly (by hand) or potentially multiple times (with shell scripts). When it is necessary to express highly complex and inter-dependent deployments, however, these approaches become limiting and other techniques must be considered.

In language-based deployment, constructs are provided in the programming languages which can be used to specify complex interdependencies between components in a distributed system. Frameworks such as SmartFrog [7] and eFlow [8] use language-based constructs to define the static and dynamic bindings between application components, which can then be combined with executable code in the same implementation (combining the Developer and Assembler roles).

Using a language-based deployment technique requires the Implementer to learn and understand the complexities of the language, but once implemented the remainder of the application deployment process can be fully automated. A deployment engine can first assume the role of Planner, generating and selecting valid DeploymentPlans for target environments with both static and dynamic resource availability. It can also perform the preparation and launch roles of the Executor, preparing the target environment for

execution and coordinating launch according to the plan. Further, by analyzing the inter-component dependencies specified by the implementation artifact (before a specific DeploymentPlan is developed), change-management systems such as [8] can detect component failure and automatically reconfigure the deployed system – an activity detailed further in section 2.5 'Management'.

Model-based techniques take a higher-level approach by describing the business functionality and behavior of an application separately from the technology-specific code that implements it [9]. Applications are described as high-level models representing valid deployment configurations together with a set of transformation rules. By statically specifying all of the valid deployment configurations ahead of time, the use of models reduces the effort required during the Planning stage of Deployment. The generation of DeploymentPlans only involves matching requirements to resources, after which the Planner can decide which configuration to choose. Model-based approaches from IBM [10] and Radia [11] (absorbed as part of HP OpenView Application Manager [12]) allow developers to use desired-state modeling to define the ideal state of the system with regard to some set of constraints. These constraints guide the selection of an optimal DeploymentPlan for both the initial system deployment and any subsequent reconfigurations due to changes in the target environment.

Application deployment using models requires a comparatively large application of development effort before the actual business logic of the application can be written, and sophisticated model execution engines are necessary in order to leverage this effort. However, once the system is modeled and the required components implemented, all monitoring, management and change-enactment can be taken care of autonomically and intervention by the Executor or DomainAdministrator should be limited unless the system goals or requirements change.

2.4.5.1 Comparing Deployment Approaches

Each of the preceding deployment approaches has its strengths and weaknesses and no single approach is appropriate for all scenarios. The choice of approach depends primarily on the target deployment environment and the amount of time invested during Development in order to save time adapting to change after Deployment. The burden of selecting an approach rests heavily on the Specifier of the application, while the repercussions for selecting the wrong approach are felt by the Executor, the DomainAdministrator, and the users of the application. Foreknowledge of the scale, complexity, and anticipated frequency of change in the system are required in order to make an informed decision. A

quantitative evaluation of the preceding deployment approaches is presented in [13] and a qualitative comparison is presented in [4].

2.5 Management

The W3C defines management as "a set of capabilities for discovering the existence, availability, health, and usage — as well as the control and configuration — of manageable elements, where these elements are defined as services, descriptions, agents of the service architecture, and roles undertaken in the architecture" [14]. This definition identifies the two requirements of management: information about a system, and the tools to enact change in the system. A functioning management system can gather information about manageable elements, analyze and derive meaning from this information, and change the system to better address a set of pre-defined management goals.

Although one may think of management as only happening at one stage in the lifecycle – the Management stage – it is in fact an important concept at each stage [1]. As identified in [13] the choice of application deployment technique is directly linked to a.) the effort required before deployment, and b.) the ease of managing system changes after deployment. A model-based approach, for example, requires a significant application of developer effort before the application can be made available. It requires a sophisticated Planner implementation and a thoughtful specification of the acceptable range of operational parameters (such as system response time). This additional up-front effort and complexity, however, can greatly reduce the amount of time spent reacting to change: when the operational parameters leave the acceptable bounds, the current state of the system can be used by the same Planner actor to create a DeploymentPlan suitable for the current conditions of the target environment, possibly utilizing existing deployed components. This plan can then be executed (by the Executor) after which monitoring of the operational parameters can recommence.

2.5.1 Autonomic Management

A common scenario for the management of a distributed system involves one or more system administrators actively monitoring the pre-defined measure of 'health' in a deployed application, identifying when something has gone wrong, devising a plan to bring the system into a healthy state, and executing that plan. This process can be termed 'manual troubleshooting' and is directly linked to the 'manual deployment' tasks identified in the previous section.

In a closed-world view it is conceivable that, up to a certain size and complexity, a complex software system may be effectively managed by a set of humans. Beyond a point this is unrealistic, and it becomes necessary

to employ a system which can make decisions in lieu of human guidance. This type of management is called 'autonomic management' and is part of a broader field called 'autonomic computing' [15] [16].

The term autonomic computing applies to a system that can monitor itself and adjust to changing demands [17]. There are four distinct characteristics of an autonomic computing system:

- Self-configuring
- Self-healing
- Self-optimizing
- Self-protecting

The goal of autonomic computing is the design and development of systems which are able to run themselves with little to no human intervention. In the context of autonomic management, these goals are achieved through a formalization of the above 'manual troubleshooting' steps into what is termed the 'autonomic control loop'. Fig. 6 shows the four stages in the autonomic control loop: monitor, analyze, plan, and execute.

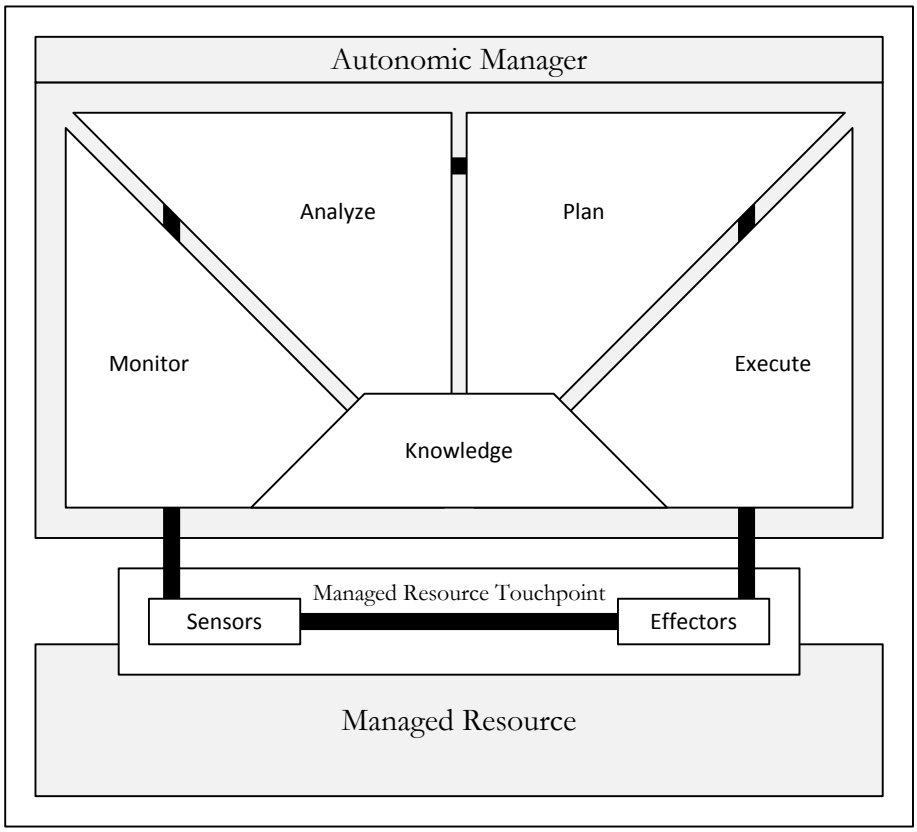

Fig. 6 The Autonomic Control Loop [25].

The monitoring and execution stages of the autonomic control loop require suitable sensors and effectors for each resource being managed – in other words, there must be ways to derive information about and exert control over manageable resources, where a manageable resource is a target environment, application, application component, client using the application, or an actor in the architecture [14]. In a distributed management architecture, the sensors and effectors linked to the manageable resource through a 'managed resource touchpoint', as shown in Fig. 6, may also be linked to a 'probe' which remotely exposes their functionality for external examination and invocation.

The two most common distributed monitoring models are called the 'proxy' model and the 'agent' model. In the proxy model, shown in Fig. 7, requests to an application are routed through an intermediary who records usage and performance statistics using a 'collector'. This intermediary can be linked to a single application ('Proxy 1' in Fig. 7) or can serve as the proxy for a number of deployed applications ('Proxy 2' in Fig. 7). From an architectural perspective, the proxy model is vulnerable to a single point of failure if there is no infrastructure in place to cluster the proxies [1]. In the agent model, shown in Fig. 8, statistics are collected by 'collector' software installed at each node in the target environment and information is recorded for all components deployed on the individual node. The agent model removes the central point of failure inherent to the proxy model, but running extra software on the same nodes as deployed application components may have an unpredictable impact on their performance.

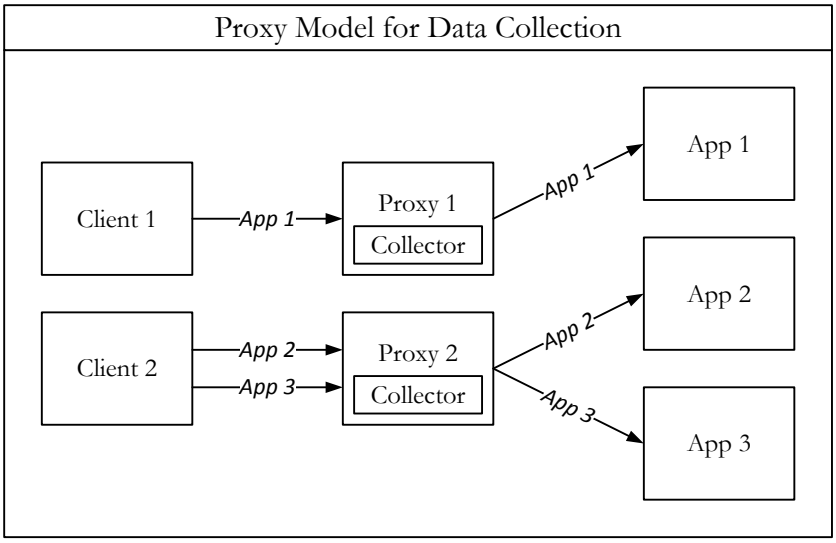

Fig. 7 The proxy model for collecting data about managed elements.

After information is collected it must be transferred to a manager for analysis. This transfer can be initiated by the collector (called reporting or 'push') or by a manager (called polling or 'pull'). Polling requires a

managing entity to periodically interact with the collector via an externally exposed interface on the node (i.e. probes). Reporting requires the collector to periodically contact one or more managing entities at a well-known location. The choice of approach has implications for scalability, reliability, and bandwidth consumption. It is argued in [19] that the reporting approach is better suited to large distributed systems as it conserves bandwidth and CPU time on the management systems.

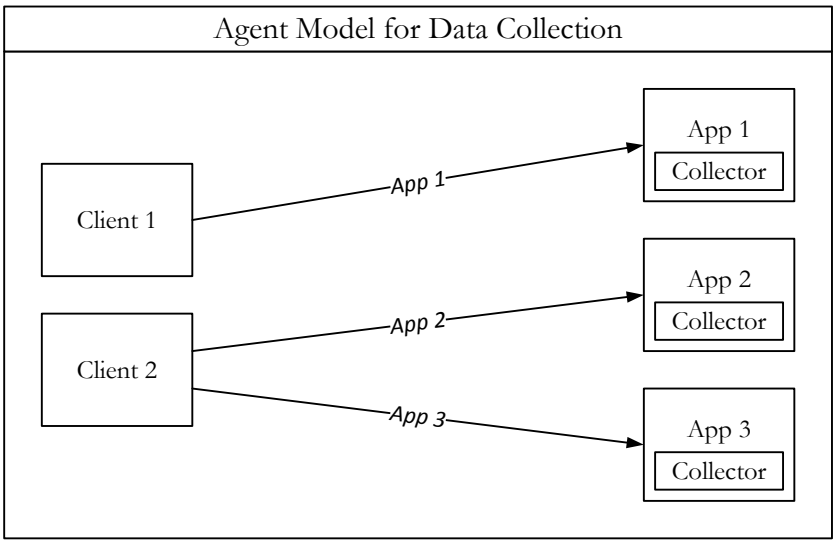

Fig. 8 The agent model for collecting data about managed elements.

Once autonomic management is introduced into a system it can be difficult to identify where the management chain ends as the introduction of each new management entity also introduces a new entity to be managed. Further, having both a deployed application and a peered set of deployed managers can be a waste of resources if the application is not being used, leading to situations where the cost of management overshadows the cost of running an application. The careful design of management architectures and the interactions between managers and managed resources is vital if system management is to be effective.

REFERENCES

[1]Sun Microsystems. (2003). Web services life cycle: Managing enterprise web services. [http://www.sun.com/software/whitepapers/webservices/wp_mngwebsvcs.pdf]2008).

[2] Object Management Group. [http://www.omg.org/].

[3] Object Management Group, "Deployment and configuration of component-based distributed applications specification - version 4.0 - OMG documents formal/06-04-02," April 2006. 2006.

[4] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. v. D. Hoek and A. L. Wolf, "A characterization framework for software deployment technologies," Department of Computer Science, University of Colorado, Boulder, Colorado, April 1998.

[5] A. Dearle, "Software deployment, past, present and future," in *International Conference on Software Engineering,* 2007, pp. 269-284.

[6] T. Vanish, M. Dejan, W. Qinyi, P. Calton, Y. Wenchang and J. Gueyoung, "Approaches for Service Deployment," *IEEE Internet Computing,* vol. 9, pp. 70-80, 2005.

[7] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray and P. Toft, "SmartFrog: Configuration and automatic ignition of distributed applications," in *HP OpenView University Association (OVUA),* 2003.

[8] F. Casati, S. Ilnicki, L. J. Jin, V. Krishnamoorthy and M. C. Shan, "Adaptive and dynamic service composition in eFlow," *Advanced Information Systems Engineering,* vol. 1789, pp. 13-31, 2000.

[9] Object Management Group. (2002). Model driven architecture. [http://www.omg.org/mda].

[10] T. Eilam, M. Kalantar, A. Konstantinou and G. Pacifici, "Model-based automation of service deployment in a constrained environment," IBM, Tech. Rep. RC23382, 2004. REFERENCES

[11] NovaDigm. Radia. [http://www.novadigm.com/].

[12] Hewlett Packard (2008). HP OpenView application manager using radia. [http://www.openview.hp.com/products/radia_appm/ds/radia_appm_ds.pdf].

[13] T. Vanish, W. Qinyi, P. Calton, Y. Wenchang, J. Gueyoung and M. Dejan, "Comparison of approaches to service deployment," in *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems,* 2005, .

[14] W3C. (2002). Web services management concern. [http://www.w3c.org/].

[15] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer,* vol. 36, pp. 41-50, 2003.

[16] IBM, "Autonomic computing: IBM's perspective on the state of information technology," IBM, 2002.

[17] IBM. (2008). Autonomic computing @ developerWorks: Self-managing autonomic technology. [http://www.ibm.com/developerworks/autonomic/]2008).

[18] IBM. (2006, June 2006). An architectural blueprint for autonomic computing. IBM. [http://www-01.ibm.com/software/tivoli/autonomic/].

[19] J. Martin-Flatin, "Push vs. pull in web-based network management," Eidgenössische Technische Hochschule Lausanne, Lausanne, Switzerland, Tech. Rep. SSC/1998/022, 1998.