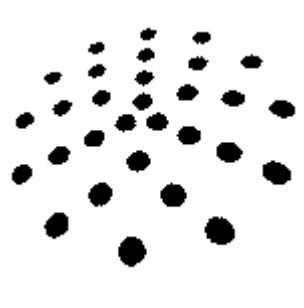

Master's Thesis

Quality Attributes and Reference Architectures for Software Distribution Environments

carried out at the

Information Systems Institute
Distributed Systems Group
Technical University of Vienna

under the guidance of

o.Univ.Prof. Dipl.-Ing. Dr. techn. Mehdi Jazayeri
and
Univ.Ass. Mag. Dr. Schahram Dustdar
as the contributing advisor responsible

by

Rainer Anzböck
Klopsteinplatz 4/11, 1030 Vienna
Matr.Nr. 9326149

Vienna, 11. February 2002

Abstract

The motivation to find quality attributes in architectures of Software Distribution Environments (SDE) arises from two different aspects, the evolution of Configuration Management (CM) systems on one hand, and the change in structure of software products and PC networks over the last years on the other.

Software distribution has changed through the Internet evolution from simple e-mail distribution of software (e.g. Xnetlib) to sophisticated distribution and configuration portals for software. Such Web portals are built to provide the newest releases of a vendor's software: product updates, service packages, or complete software packages. Internet users are provided easy and fast access to large collections of software across many different software and hardware platforms.

In the early stages, Configuration Management aspects were not part of SDEs, but nowadays they cover Software Distribution and Configuration Management tasks for the development, deployment and maintenance of software systems as described in [1].

Many concepts covering Software Distribution are also related to Configuration Management. Producers require software Configuration Management (SCM) to be integrated in their development environments. Software distribution should be based on SCM information to ship or offer particular releases (or configurations) to customers. For that, many SDEs also cover some software configuration tasks such as management of development artifacts, product and release management, software description (languages), or software packaging.

Many commercial tools and environments have been developed and are in use in such portals: System Management Server, Marimba Management Solution, Webstart, InstallShield, Perforce, CM Synergy, Tivoli Software Distribution, and many others. A detailed evaluation of 12 products in the area of Software Distribution (and configuration) environments is performed that defines the basis for architectural considerations.

The analysis covers architectural properties, common components and relationships across particular tools and products. Further, quality attributes of a reference architecture for SDEs are derived. The results are based on the aforementioned product evaluation and three typical case study scenarios that function as means to distill common architectural elements. The contribution of this thesis is targeted towards a generalized software architecture of SDEs.

Table of Contents

1 Introduction

I describe in short the domain of Configuration Management and Software Distribution and why it is necessary to evaluate current commercial products and to define reference architectures.

First I will clarify the main terms used throughout the thesis. Configuration Management (CM) as described in [1] consists primarily of the following tasks: a scheme to identify components, their type and structure is necessary to make them unique and accessible in some form. Controlling releases of software products and changes over their lifecycle ensure a consistent product baseline. A key part is validating completeness of the product and maintaining consistency among components to extract a well-defined product out of its components. Recording and reporting status information and collecting statistical data of software components is another explicitly defined task.

The meaning of Software Distribution has increased with the expansion of the Internet. Software Distribution was not part of Configuration Management in its initial definition. Now it covers installation, deletion, and re-configuration and is related to software configuration in the process of software development and maintenance. Most concepts covering distribution are also attributed to Configuration Management concepts.

1.1 Evolution of Configuration Management

A lot of theoretical work in the 1980s has been performed by the SEI (Software Engineering Institute) at the Carnegie Mellon University in Pittsburgh [2], which is affiliated to the DOD (Department of Defense) of the United States. Configuration Management (CM) is seen as a process in software development. It consists of constructive tasks, as there are product customization, testing, using software component repositories, system maintenance, and software deployment. And it consists of process related tasks like auditing, control, and status accounting. Therefore CM starts with Version Management at the producer and ends with installation and maintenance of software at the end user's machine. Additionally different roles like Customer, Project manager, Developer, Tester, and Reuse Librarians participate in CM tasks. They demand system support in very different ways. I call these concepts "Classic CM". Major parts of their work are still relevant, but I will point out where I suggest modifications and extensions.

A lot of today's CM systems are built upon and related to their concepts. All suffer from non-standardized application protocols, minimal interoperability, and an undefined coverage of CM systems' functionality. Changing requirements and the complexity of software infrastructure in future environments cannot be dealt with satisfactory. The following sections describe what has changed since the 1980s and why classic CM cannot meet most of current requirements.

1.1.1 Change in the distribution process

The distribution process has changed for the producer, distributor and customer. The producers require integrated Software Configuration Management (SCM) components to fit in their existing environments to keep consistency and to cooperate. New product versions are created and customized for individual customers (see 1.2.4 for an overview of interfaces with CM). Additionally the software structure changes (see 1.2.3). Often software is

distributed as compressed packages in different versions. Producer and customer information has not been integrated in the product description or into the ordering and sales logic. Different distribution and sales mechanisms should be supported. CM systems are developer- and not customer-related. Even if distribution of software is part of the covered processes, there is no ability for possibly anonymous customers to get informed, gain access, buy, and install products. CM repositories also contain secret information not intended for customers.

1.1.2 Networking environments

The following requirements can be concluded from the use of networking environments for CM systems. To guarantee that a product is available in the same version in all locations the distribution process has to support parallel activation of a product in different locations. Products have to be distributed reliably using transactional semantics. I call this requirement "Distribution Consistency".

Certificates should be used for the origin of products and participants and encryption for the secure transfer of products. Different roles should be defined to interact in the distribution process. Additional security requirements exist (e.g. producers have to be secured from other producers or customers have to be granted access to defined product resources).

Availability and Performance are key factors, especially because they are the weakness of today's Internet services. Immediate payments and downloads make a solution more successful. The producers will select the distribution channels depending on these factors.

The costs of producers and customers can be reduced by integrated CM tools. A networking environment connects participants in Software Distribution, making updates easier. Additionally a distribution process is performed more efficiently and less staff is occupied.

1.1.3 Collaboration

A networking environment influences type and intensity of collaboration between all participants. Because the participants that can be networked with a CM solution increase, the following interactions are possible: The customer can get sales, update, and service support. Further he can provide product satisfaction feedback to the producer. Additionally the cooperation of producer and distributor in the process of releasing a product and defining a distribution plan is necessary.

1.1.4 Frequent content change

Customers demand a high quality software. To satisfy them it is necessary to frequently update the software. In my own company we are trying to keep customers' software installation as recent as possible to reduce support requirements. This requirement is currently addressed in Content Management systems, but also relevant to Configuration Management [3]. Content Management systems therefore lack most of the established CM functionality.

1.1.5 Web-based applications

With the introduction of Web-based applications additional technical requirements for CM systems have to be met. Theoretical work on the implementation of the HTTP protocol has been done by the IETF (Internet Engineering Task Force) WebDAV working group [4], which is responsible for standardization of the Internet environment. They concentrate on HTTP extensions to perform Version Management and have released recent drafts. As in my company a lot of environments have to support Web-based and conventional applications in parallel. Therefore the distribution of software cannot be restricted to HTTP transport in any form.

1.2 Problem Statement

This section describes the problem domain for Software Distribution Environments. Beside Configuration Management, Software Distribution and deployment as well as software structures are key issues.

1.2.1 Software Distribution

Software Distribution is currently supported by Software Distribution systems like System Management Server [5] or InstallShield [6]. The processes covered are the installation, deletion, repair and re-installation of software.

Currently the support for software configuration tasks (on a release or component level) in such systems is minimal. The Software Distribution systems make no assumptions whether version control systems are used. They maintain their own semantic model for software versions. The target client is installed through external setup procedures that are not part of the distribution process itself. These systems have no abstractions for client configurations (see customer abstraction 4.4).

The products support a software description language where attributes of a product can be stored and used for the installation. Some others define distribution templates that can be customized for corporate use. The distribution itself can be based upon computer and user information. The organization of users and computers in groups requires dynamic changes in the distribution plan. The distribution process allows different sources and executes reliable regarding network connection timeouts and server failures. Dynamic reconnection to servers, error recovery and file verification features are implemented.

Some systems support conventional software; some are well suited for Web-based applications too. The systems provide a server component for the distribution process implementation and a client component for the target machine and for configuration of the services. Certification and encryption are supported partially for client and server communication.

1.2.2 Software Deployment

Rick Hall defines activities of Software Distribution in his thesis [7] and calls them the deployment process. Throughout this thesis I will use the term Software Distribution. The theoretical base of his thesis is the definition of a software deployment (distribution) process. This process is not completely implemented in any of the products evaluated in [8]. It consists of producer-side processes (release and retire) and consumer-side processes install, (de)activate, (re)configure, update, adapt, and remove.

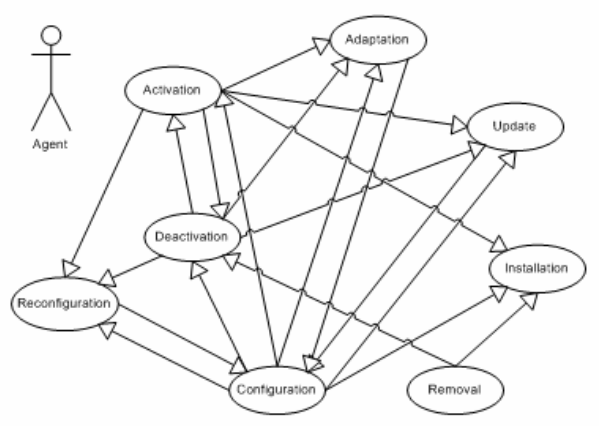

Figure 1: consumer-side use cases

A use-case diagram is shown in Figure 1. The arrows symbolize the <<extends>> stereotype, the activities are therefore very flexible but interdependent. Installation, Update, and Removal are base processes, activation, adaptation, and configuration are more advanced and not implemented yet in most products.

Compared to the requirements of Configuration Management systems it has to be stated that the deployment itself is beneath the version- and Configuration Management one part of a distribution environment.

1.2.3 Distributed component software

Structure of software systems change. Isolated small software solutions in local networks decrease, while complex software components distributed over larger networks and different locations will dramatically increase in the near future as a result of globalization and the evolution of the Internet. Heterogeneous environments in hard- and software have been covered with projects like CORBA [9], Enterprise Java Beans [10], DCOM [11] and .NET [12], as an overall communication backbone in such systems, and JAVA [13] as a platform-independent programming language. Furthermore component based applications reuse existing infrastructure code for user interfaces or database access. Most applications are systems of systems [7] as they have dependencies for installing some ODBC or JDK version prior to software installation. The distribution of complex software systems demands a high quality distribution process and defines server machines as part of the software environment. Individual installation procedures for component registration and registration of remote server processes have to be supported. Even distribution environments are distributed systems too.

1.2.4 Related systems

There are systems that can be part of a distribution environment and systems that will have interfaces. The first group of Configuration Management and Software Distribution have been introduced, other related domains are described in this section.

Version management is a producer-related process to manage the creation and modification of software products. The database with the current released software version is part (starting point) of the distribution process. Change management systems cover tasks between configuration and Version Management and the Software Distribution. Changes are defined through change management systems and implemented with Version Management support, creating new releases for distribution. E-commerce products can profit from data about users and product usage gathered throughout the distribution process. Content Management systems are more related to specifying content and to upload a set of content to a Web-server. A CM system covering Web-based software will succeed current Content Management solutions. Directory services function as global name-space holder and object stores. Access to a CM database for directory services strengthens the integration of information systems. Of course Inventorying systems provide detailed hardware and software information about target systems.

1.3 Case studies

The goal of this section is to present real-world scenarios regarding software distribution in organizations in form of case studies. In the next step commonalities and differences among them will be discussed. I present three case studies, which are more or less typical for most organizations, and used throughout this thesis. The case studies will be analyzed according to requirements and quality attributes discussed in Chapters 0 and 0 Those quality attributes and their use in real world scenarios build a foundation to further investigate general architectural principles of software distribution environments.

1.3.1 Case study "Workflow"

As the first case study a fictitious company is covered that implements the IBM MQ Series Workflow product in their infrastructure. The workflow product is a client/server architecture with a hierarchical system structure that can span multiple geographical locations. A workflow domain is created that spans the whole company and holds shared workflow models in a database. These models are defined at build time. The domain is structured into three system groups covering different company locations. System groups optimize MQ series communication with a queuing mechanism and can also be used for clustered systems. Within these groups, systems are defined that consist of a three tier client/server model and share a database for the workflow at runtime. The first tier consists of an implementation of the workflow client API. A sample implementation ships with the product that consists of a Web-based workflow client and is used in this case study. The second tier consists of the workflow server. The server covers functionality for process execution, system administration, scheduling, and clean-up. It has a Java component-based architecture, provides the workflow interface to clients and uses a database client for the third tier. The third tier consists mainly of the runtime database. To implement the workflow product the company has to set up a domain, three system groups with one system each. The build time database has to be configured for the whole domain and three runtime databases have to maintained. The server components run on multiple servers at each location and all clients of the sites have to be integrated. Installation of all components has to be automated and the necessary inter-site communication of the queuing facility has to be considered.

1.3.2 Case study "Virus Scan"

As a second case study I introduce a fictitious company ImmuneWare that distributes a virus scanner as a standard software product. The product consists only of client components (executives and shared libraries) and a connection feature that allows to request status information from an internet server about new product releases. The application has a broad customer base and should be available from the Internet. The company maintains multiple Web-servers in different locations. The product has to be available in parallel but with a different range of versions at each location. They provide different versions of the product and the configuration is further standardized for different platforms and languages. On the development site all versions are maintained in a version control system. Each time over 40 versions are active working items. The rest of the configuration takes place on the client machine upon installation through a wizard that requires parameters from the end-user. Software component registration is necessary during installation on the target system. The user decides for installation and updates with new versions of products. Updates are performed manually with a Web-server interface or through auto-update functionality. Additionally the distribution environment should support an interface for end-users to get information of new releases and product features. Beside deployment issues, licensing functionality should be integrated into the distribution process. Because of the large number of users all tasks have to be automated and allow intuitive user interaction. Most of these tasks are performed with pull semantics.

1.3.3 Case study "Custom Web-service"

In my company we are constructing a complex product family that consists of conventional client/server products and Web-based applications. Our configuration and distribution tasks will be covered as the third case study.

All applications are being developed in one physical location from one producer with some members having external access. Further all applications are under version control. Versions are accessed on a standard file system for the distribution process. A Software Distribution Environment (SDE) should consider these development tasks. The client/server applications consist of Windows client executives and a database server system. The Web-based applications consist of a database tier, an application tier on the Web-server that provides a

Web-based client interface for end-users, and a Web-service interface for the client/server applications through the SOAP protocol [14]. Therefore a Web-service interface for these central services is also implemented on the database servers of the client/server products.

The Web-server based configuration takes place on the company site where the web and application servers are hosted. The client/server products need an initial setup and further reconfiguration at the customer site. It is necessary to maintain a network infrastructure to deploy configurations to the customer via ISDN or VPN connections. The configuration and installation of the client/server products is comparable to the distribution of the standard workflow system and is not covered in detail. The main focus of this case study are the internet portal, the Web-based applications provided directly, and the Web-service-based functionalities that extend the client/server products. Upgrades, deletions, and re-installations of the server-side components are currently performed manually using the network infrastructure. Most tasks are performed with push semantics. The company decides which customer-site gets which product version and when upgrades have to be performed.

1.4 Goals

At the end of this introduction I will follow up with the goals of this thesis:

- An overview of concepts should provide the background for a Software Distribution Environment. Different concepts have to be considered for evaluating products.
- The product evaluation should give an overview of the products and extract common functionality and architectural styles. The products have to be chosen to cover all relevant aspects of the domain.
- The reference architecture should abstract the product functionality, define a cover of operations that a Software Distribution architecture should support, and incorporate the architectural styles.
- The reference architecture should consider the different scenarios presented in the case studies and key features should be evaluated.

All parts should take into consideration current changes in software and network infrastructure.

1.5 Structure of this thesis

This thesis is structured as follows:

Chapter 1 provides an overview, relevant considerable background, and discusses what is part of the thesis and what is not. Chapter 2 covers State-of-the-Art in Configuration Management and Software Distribution concepts. Chapter 3 covers requirements, definitions of evaluation factors, product evaluations, and a comparison of the products. Chapter 4 covers main architectural styles of the products and significant requirements for a reference architecture. Chapter 5 covers the architecture and is the main part of the thesis. Functionality and architecture of the products are abstracted and modeled. The case studies are described in detail and it is shown how specific tasks can be performed implementing the reference architecture. Chapter 6 provides an evaluation, a summary and conclusions of the thesis. It concludes which goals could be reached which problems arose and how to proceed further with the gained results.

2 State-of-the-Art

In this chapter I describe the state-of-the-art in Configuration Management and Software Distribution. CM covers version control, Software Distribution, online collaboration, support for the distribution of software, security related issues and further more.

Theoretical work has been done from the 1980s, when IEEE and ANSI published a Guide to CM [15] until now where these concepts are adapted to meet the current needs. Also most widely used definitions of software Configuration Management can be found in [16] and [17].

The documents mostly referenced to in the 1990s are papers from H. Feiler and S. Dart of the Carnegie Mellon University [1]. Their work will be introduced in 2.1. A lot of theoretical work for modern aspects of Software Distribution comes from Richard Hall. Hall covers agent-based Software Distribution in his thesis in detail [8] and will be introduced in 2.2. Further theoretical work from other authors is described in 2.3.

2.1 Configuration Management

The following concepts lay the foundations for all following related work described in the following sections. The functionality of Configuration Management systems has not changed very much, but the applications to get managed changed and made the process more complex.

2.1.1 Classic Configuration Management

I will give a short description of what Susan Dart concludes in [1]. She first states, that CM should cover the identification of configuration items, the control of an item's lifecycle, the status accounting for reporting change requests and the audits and reviews to validate a product's completeness. She further concludes that the construction of software products (manufacturing), fulfillment of organizational procedures (process management) and interaction of involved people (team work) should primarily be supported by a CM system.

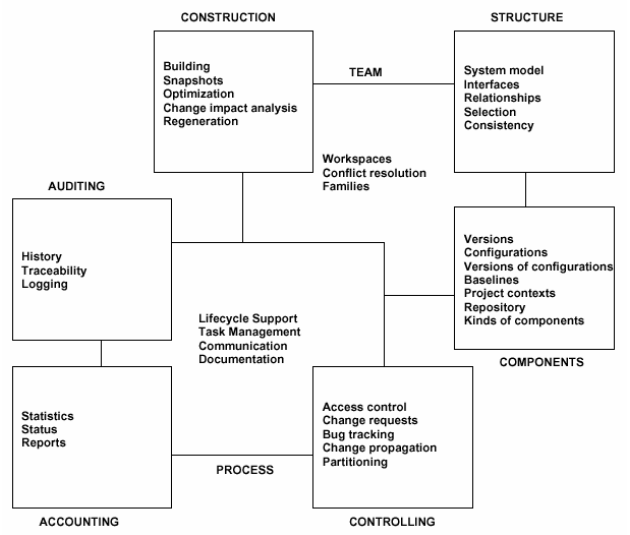

Figure 2: CM functional requirements (Dart)

The overview Figure 2 below provides a big picture of the different domains related to Configuration Management. The responsibility reaches from constructive issues of product development and version control to the management tasks of controlling and accounting. The importance of user roles is pointed out as technicians and managers both rely on the information of a CM system. Dart therefore describes the functionality in two related circles, one covering workspaces, version conflict resolution and product family management, while the other is related to task management, communication, and documentation. This view clearly integrates different technical and management tasks into one concept. A Software Distribution Environment has to support these roles and present reasonable use-cases for interactions.

These definitions should be part of the requirements of every CM related solution. Because the CM tasks are all producer related, there is an impact on my definition of a Software Distribution Environment. One major part is taken from the requirements presented here and another part is taken from the deployment process described in 2.2.

2.1.2 Concurrent Distributed Configuration Management

The first significant extension to the definition of Configuration Management comes from Stephen MacKay, who wrote a paper with theoretical background for distributed Configuration Management (DCM). DCM is simply a recognition of the state of software development in the 1990's [18].

Client/server concepts moved the development tools to the client and CM concepts have to be adapted to support the new environment. The increasing speed in WAN communications enable working simultaneously from different geographical locations. The development process ranges from a single repository configuration with WAN access, to real parallel development.

Simple techniques like branching, check-in/check-out and copy-modify-merge are substituted with concepts of Workspaces and Change-Sets. Workspaces allow all developers a virtual view of the software, where changes are made consistently. Change-Sets focus on groups of logical changes to a product, not on revisions of individual items.

Most of the functionality regarding workspaces and change-sets are implemented in Version Management related solutions. Continuus [19] and ClearCase [20] are examples that will be discussed in section 3.3. The conclusion for implementing a Software Distribution Environment is that storage that defines the interface to the Version Management system should be appropriately designed to support concurrent and distributed concepts.

2.2 Agent-based Software Distribution

Rick Hall provides an overview of traditional and modern concepts of Software Distribution in his thesis [7]. The thesis covers the design of an agent-based distribution framework and a platform and network independent implementation in Java (Software Dock).

A prototype has been implemented in 1999 and transferred to several organizations for use in technology evaluations (Lockheed Martin, Dassault Systems, Nortel Networks, Mitre). I describe software dock in the following section and in the comparison of CM and distribution products (see 3.3).

2.2.1 The Software Dock architecture

The software dock is an agent-based framework. It primarily consists of docks, agents and a software description format. Figure 3 provides an overview of the Software Dock architecture.

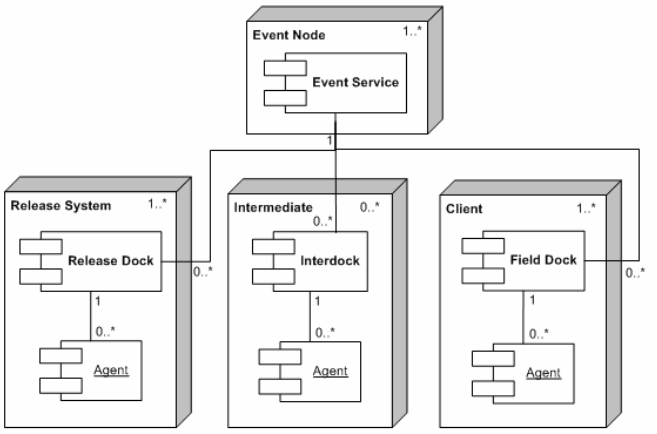

Figure 3: Software Dock architecture [7]

Docks are used to define interfaces that producers and consumers expose to the framework. The field dock is the consumer interface, it provides the only access to the consumer machine and to information about the current configuration and parameter. The release dock is an interface to a specific producer site. It serves as an access point to the release repository of the producer. It additionally provides an interface to browse product information.

Furthermore there is an Interdock that is used at the consumer site and serves as a store for global information not related to a specific client and as a cache for releases from the release dock. Therefore it reduces configuration work and saves bandwidth. All docks are implemented as server processes and are the access points for the agents used in the system.

Agents are created by the framework and perform the software deployment processes. Additional agents can be derived from existing ones to perform more complex deployment operations. As an example the installation agent docks on the local field dock to configure software. It checks the properties of the registry and determines which components (called artifacts) it has to gather from the release dock. In fact of dependencies an agent might have to call other agents to perform processes on the client before installation can proceed.

2.2.2 Event notification

The environment uses a local and global event notification infrastructure. Local events of a machine can be fired when changes to the clients' registry have occurred. In order to receive an event the agents have to subscribe to them. As soon as a change to an entry in the registry is performed, all subscribed agents are informed and can take actions like an update or adapt process. Global events are sent to a dock and not an agent. The docks can therefore receive events produced by their agents.

2.2.3 The software description format

Software Dock provides a semantic model for describing a software system at the producer's site and a target machine environment of the consumer's site. Hall defines the Deployable Software Description Format (DSD), an XML application to perform the deployment process [21]. The format defines a property structure. There are no predefined properties that have to be supported. The customer is free to define the properties that should be provided with the software. The consumer-site properties are stored in a platform independent registry on the client machine. Composition rules define logical operations upon properties that result in true or false. In addition to the rules there are assertion and dependency constraints defined. If an assertion constraint cannot be fulfilled then the installation is prohibited. The dependencies just define additional installations or upgrades that have to take place to successfully process the initial installation. Hall also evaluates other software description formats like the OSD Open Software Description [22] format an XML application and a W3 Consortium proposed standard from Marimba and Microsoft. This format is merely a software packaging description and it lacks from a detailed consumer-side model. Another

model, the MIF Management Information Format [23] is covered by Hall and is inappropriate to support the distribution process. The requirements for a description language are described in 3.1.3. An introduction of this language into a distribution architecture is described in 5.5.4.

2.3 Further Configuration Management concepts

Beside Dart's and Hall's work there are further concepts that influenced the CM products and also had an impact on my requirements definition.

2.3.1 Version Control using Concept Descriptions

A more recent work on version control is provided by Andreas Zeller in [24]. This thesis includes a lot of work done in the past five years related to Configuration Management. CM concepts often use serialized versioning, where the transition from one version to another is defined by the changes of the system. In [24] is concluded that because of the complexity of today's software systems and the more dimensional structure of changes, where related components work together and are separately revised, serialized versioning fails to deliver. The described solution uses Concept Descriptions to summarize components for different platforms or minor implementation differences in one version. Different versions are defined by additional, extended or missing features. This work is one of the first, which describes a reasonable abstraction of implementation details. Concept descriptions play a major role in Software Distribution systems. They can be used to describe a product's functionality in a very general way, regardless of the type of product described. Concept descriptions can be used to compare different products to decide whether updating a component is reasonable. They can also support a product or component catalogue, where search-term-based queries are used to find required functionality. Software distribution systems should use concept descriptions not only for versioning but also for delivering meta-information in the distribution process.

2.3.2 Versioning for WWW collaboration

The IETF WebDAV working group works on extensions to the HTTP protocol to support versioning within software projects. Extending the HTTP protocol has the strength to build on a widely used standard Internet protocol. Prior work on 'Versioning for WWW collaboration' has also been performed at the University of Bologna [25]. Prerequisites for Version Management over HTTP are described in [4].

Prior to the WebDAV protocol, authoring tools just supported read/write instead of check-in/check-out semantics. Existing versioning products have been accessed through proprietary HTTP extensions, which forced clients to support different interfaces for each implementation. The RFC also suggests that the interoperability of clients and servers with or without WebDAV support should be guaranteed and that the client implementation has to be simple. The extensions to the HTTP protocol are new methods (headers, mime types, document properties) and new behaviors. The support for versioning, parallel development and multi-resource locking results in a Web-based versioning systems. Additionally the usage of URIs and a new data format that allows for distribution makes the WebDAV protocol an interesting alternative for a lightweight end-user configuration client. However the distribution is described for packaged software and not yet useable for software under Configuration Management. The description of software components lacks detailed structured information compared to Hall's DSD (2.2.3). For now this approach is a solution for web versioning. Further progress in this area in the future will help Content Management systems and web-application builder tools implement Version Management and some Configuration Management tasks.

2.3.3 Content Change Management

Susan Dart has proceeded with her work to web content systems and has written an article [1] where she points out, why CM techniques should be applied to web Content Management systems.

She states that current development tools are just supporting version control but lack from support of Configuration Management tasks for a consistent install or update process. The current web crisis (like the software crisis) should be resolved with Configuration Management. Additionally new requirements like real-time updates to web sites are arising and have to be resolved by extending existing products or integrate CM tasks into web Content Management systems. Dart categorizes web pages from information systems to interactive distributed applications. She also states that most languages used are interpretive or use just-in-time compilers and that this leads to a style of changes on the fly. Dart also provides a complete list of current web Content Management tools in her article. She separates classic CM tools with web support from content tools with CM support. I will mention this in the product comparison in the following section.

I think that parts of the requirements described for web content can soon be applied to conventional software products. The use of interpretive languages also increases in conventional systems. Web Content Management cannot be discussed separated.

2.3.4 Integrating configuration and document management

Reagan Penner from the University of Saskatchewan wrote a preparation paper about his ideas of integrating configuration- and document management. In the paper a SCM language using XML is defined, to handle Configuration Management operations for documents. Penner states that code management is based on code fragments while documentation is based on information items. He defines an object based structure where the code base and the document repository are linked together and can be accessed with an XML parser. One part of the design is that the two systems are independent of each other as long as there is no integration. The integration itself links the information bases together. The XML definition of operations and parameters is a flexible way to define the functionality of a system. I use a similar notation for the operations in my framework, but there is no direct impact from the document management theme to my work [26], [27].

2.3.5 Change based Configuration Management

One part of concepts in CM are related to change management. Several articles can be found at CM Today [28]. It is stated that change management should operate on source code and not on executables, because the reproduction of an executable is not error free and accumulation of changes is difficult. For this thesis, where concepts are covered that have an impact on the functionality of a Software Distribution Environment, change management is not a different issue, because it operates with the same interface as a version control system.

3 Software Distribution Environment products

This chapter describes and evaluates Configuration Management and Software Distribution products. The following requirements are the base for a classification of the products.

3.1 Functional Requirements

First we define functional requirements for the products, where most is domain related functionality. These requirements will be used in the evaluation and the comparison chart. Functionality that cannot be categorized but influences the design of a reference architecture is described individually for each product.

3.1.1 Version management

Versioning describes the ability to store more than one version of a product with some automatism. Some applications provide a rule based version definition and local snapshots or workspaces for client-side caching of data. Rule based version definition provides automatic creation of product versions like “last version including component x and change y”. State management in versioning can be used for products, versions, or files. The support for local workspaces and snapshots enable better integration of remote users and provide an abstraction for more complex version branches. To reduce the branch complexity, version changes are combined to change set definitions. Further, release lines and similar abstractions provide a definition of release consistency, where it is possible to construct a running system anytime in the product’s life cycle. Keeping releases consistent depends on the programming model and the component description language and increases the product’s quality.

For parallel development it should be possible to support multiple users to perform cooperative work on the same or on different versions by some check in/out or similar semantics. Trigger and pre-conditions allow the integration of external products that provide additional build or validation steps throughout version changes. Through rollback, changes can be made undone on a file or a more complex change level. Milestones can be defined and stored as a revision history resulting in consistent software-configurations. Additionally build support provides functionality to integrate development tasks like preprocessing, compiling and linking. Change audits and defect tracking close the loop to development. Changes and all corresponding data, acting user, used workstation and timestamp information are tracked.

3.1.2 Configuration Management

First of all, CM has to support the storage of and operations on development artifacts ranging from source code and content to specifications of implemented features and further more. Feature definition provides a readable and understandable way to store, customize and patch software configurations. The product versions are defined by additional, extended or missing features (see also 2.3.1). Type, structure, and granularity of software components vary. Flexibility in supporting different definitions and descriptions of components is crucial for a successful CM system.

Managing products and releases is key point to Configuration Management. The support for multiple concurrently active releases is necessary for flexible product management. The granularity of configurable and distributable items enables to define the software in more detail. The granularity reaches from single product packages to object code or source code. To support customer or environment specific product customization, multiple configurations

of the same product should be supported and a policy for their usage should be defined. The definition of variants, extensions and optional components of the software enables automated component selection in the distribution process for specific environments and requirements of the installation. Product families might be supported to define relationships between product configurations.

A software description language is an abstraction that defines a software's content and structure and it's possible different configurations. It might describe parameters and options to provide enough information for installation. Additionally consistent code and content changes might be supported and are primarily addressed by Content Management systems. The software description can be based on package definitions or component description and an object store or repository. While most development related products use repositories, distribution systems work with package structures. Package definitions are more coarse grained than repository based definitions where components and related class files can be configured. Beside packages simple system snapshots can be used for distribution.

For distribution the used package format is important. One definition is the OSD format [22], additionally own proprietary or other formats like InstallShield packages are used [29]. Package transformation files allow customizing a products distribution, but maintaining a minimal set of packages. Important for a product's distribution are also dependencies to other products and a systems of systems definition [30] like the concept of merge modules used by InstallShield (3.3.5). Furthermore the installation on the target consists of system and user specific settings, separating them results in more flexible combinations of configurations.

3.1.3 Software description language

The use of a software description language allows formalizing the software definition together with the definition of the target environment where the software has to be installed, run and maintained. The target environment also covers installation requirements (see 3.1.3). Part of the target environment is the environment where the software is executed, the user, the license, the execution frequency and time. This definition differs because parts of the information gathered is produced after the product is installed. I will compare the languages used in the products during the evaluation and present requirements for a language that supports processes of the reference architectures. Software and customer abstractions are compared in sections 4.2 and 4.4. One language investigated further is the Deployable Software Description provided by Rick Hall [30]. An excerpt of the descriptions Document Type Definition is shown in Figure 4.

```
<!ELEMENT ImportedProperties (ImportedProperty)*>)>
<!ELEMENT ImportedProperty   (Name, Type, Description, Value, DefaultValue)>

<!ELEMENT Properties         (Property)*>
<!ELEMENT Property           (Name, Type, Description, Value, DefaultValue)>

<!ELEMENT Composition        (CompositionRule)*>
<!ELEMENT CompositionRule    (Condition, ControlProperty, Relation, Properties)>

<!ELEMENT Assertions         (Guard, (Assertions | Assertion)*)>
<!ELEMENT Assertion          (Guard, Condition, Description)>

<!ELEMENT Dependencies       (Guard, (Dependencies | Dependency)*)>
<!ELEMENT Dependency         (Guard, Condition, Description, Resolution, Constraints)>

<!ELEMENT Artifacts          (Guard, (Artifacts | Artifact)*)>
<!ELEMENT Artifact           (Guard, Signature, Type, SourceName, Source, DestinationName,
                              Destination, EntryPoint, Mutable, Permission)>

<!ELEMENT Activities         (Guard, (Activities | Activity)*)>
<!ELEMENT Activity           (Guard, Action, When, Description)>
```

Figure 4: Deployable Software Description Format - DTD excerpt

It uses a Property definition for software and consumer site (imported) descriptions and an artifact definition for software components. Conditions, rules, and dependencies constrain

which artifacts have to be handled and which activities have to be performed during distribution. The Condition attribute in the Assertion and Dependency element contain property values that are evaluated for installation customization. For example the property HTML viewer can be set to true or false. A feature called HTML help can be formulated as a dependent system. It is just selected for a specific configuration if the guard expression containing the viewer attribute evaluates to true. As a further example Java VM versions are common prerequisites. A profound description of the language and the assertion and dependency processing can is provided in [21].

3.1.4 Software Distribution

For the Software Distribution process I use the definition of Rick Hall [8] that has been introduced in 1.2.2 and conclude the following requirements.

First of all the distribution process has to be defined. Distribution plans formalize the participating hosts and users. Channel definitions are a specialization using a sender and one or more receiver nodes; user can subscribe to participate. Further the process is divided into activities. They may consist of packaging, deployment, and installation. Activity hierarchies can be used to cover nested installations. Additionally it should be possible to use different distribution mechanisms. For some scenarios, like implementation of an unattended setup, a push mechanism might be appropriate, for others, like reduced data transfer or on-use installation, pull semantics provide better service. Ideally these options can be combined using a distribution policy based upon the type and usage of software products. Some products implement the inter-server communication (e.g. scheduled push technique) different than the deployment to the client (e.g. User triggered push technique). A store-and-forward mechanism is used for mobile clients to support installation after offline periods.

Beside distribution methods several processes can be defined. The release process creates a distribution package from a release of the Configuration Management. The installation process installs a package on a client. The environment of the client and the software defined in the package determine which installation activity is executed. The updating process combines the installation with the update and removal of components. The activation and deactivation processes can be used during other tasks (e.g. it releases locks from application resources). Removal support should be based on target definitions. Software that is no longer used, needs to be updated or is out of license, can be removed from distribution targets. Beside these main processes license management can be performed to control the correct use of software in corporate environments. Licenses have to be stored and maintained throughout different versions of a product.

Transactional deployment allows a precise tracking of the process but requires support from all installation routines used. For character-based systems a console installation should be supported that simulates the graphical setup procedure. Silent installations are used to bypass user interaction, which can increase reliability. The deployment of the software can be executed automatically, based on a schedule or triggered by an external program. A specialized process is used for ASP services where components have to be installed for a period of time. Staged deployment additionally provides support for a staging area to test security holes and to test functionality under production conditions. Beside areas other pre-install checks and tests can be performed to increase the possibility of a successful installation.

The installation of software products can be based upon a user, a user group, a machine or IP address definition (see target abstraction 4.3). An extended target definition can be provided by an inventory management system. Through a query interface deployment schemes based on inventory information can be implemented. Additionally rules and channels are the most flexible way to configure Software Distribution. Rules combine information to determine distribution targets. Independent of the mechanism a dynamic membership functionality supports immediate changes to the target definition. An important criteria for Software Distribution are the supported content types. Conventional products consisting of executives, binary object files and shared libraries are well supported yet. Additionally server side

components for distributed systems, content for Web-based products and other types like documents or development artifacts might be distributed and registered or simply activated by an application server on the target machine.

3.1.5 Covered scenarios

The covered scenarios are a good real world requirement for a product. I will use the case studies presented in 1.3 and categorize the scenarios in more general. First the process coverage is a key factor of a scenario. Configuration Management might just cover the generation of distributable product versions or it additionally supports development teams and change management. A distribution environment might just cover product installation and system management and leave Content Management, licensing and billing optional. Additionally the environment structure is relevant. The definition of target groups generalizes the use of a product. It defines whether software download is supported for a large user base or also for specific customer sites using software with more configuration requirements. The definition of a company structure also generalizes the use. It defines whether one or more producers, distributors and customers can participate or whether individual end users or software companies can be supported as customers. The case studies have been chosen to reflect differences in these categories.

3.2 Non- functional requirements

The Non- functional requirements are key points for the acceptance of a Software Distribution Environment. Most are also common to other distributed systems.

3.2.1 Availability and performance

Availability and performance are key requirements in Configuration Management and Software Distribution, important features related to them are described. First, multiple distributed computing sites are beneficial. Depending on the implementation, fault tolerance, load balancing, proxying or caching can be provided. To profit from multiple distribution sites in a WAN environment, the receiver should be able to switch between location and resume after reconnecting to the source site. Another key factor is the architecture and implementation for supporting remote sites. Software distribution includes data intensive WAN operations that should be supported by routing mechanisms. For example the distribution of large product-packages can consume a lot of network bandwidth. This issue can be addressed with bandwidth throttling which can also be supported by the operating system. The network structure can further influence the distribution process. Depending on available bandwidth, specific deployment techniques are used. Even if bandwidth cannot be altered easily, more important content can be distributed using a priority queuing mechanism. Additionally any parts of a software product should optionally be transferred between production- and distribution locations through an offline source transfer. Further performance enhancements can be reached with delta transmissions, where only changes to content are deployed and with data compression where the efficiency depends on the type of content used. If data is corrupted upon transfer, which should be controlled by checksums or other mechanism, automatic retransmission can take place. The granularity defines the quality- and bandwidth savings of the recovery mechanism.

3.2.2 Security

First to secure the distribution environment, the access to specific functionality has to be restricted. The interaction can be modeled with user management or a complete definition of user roles and their system interaction. Additionally certificates should be used. Client certificates enable the identification of hosts gaining access to distribution services. User certificates enable the system to identify individual users. Server certificates enable users to ensure that only trusted authorities provide the services. Product certificates can be used to identify the producer of the software. All kinds can be useful in a Software Distribution Environment. If the environment makes extensive use of certificates an interface to a PKI infrastructure is beneficial. Further tempering of distributed content has to be avoided. File

integrity is enforced using hash computing and integrate hash results into the distribution package definition. Then the content should not be accessible to any unauthorized person. With content encryption the source or object code can be encrypted and therefore be unreadable and undecipherable for interceptors. Alternatively executables can be transferred without encryption if the activation of the product is separated through a licensing mechanism. If licensing occurs online, encryption is also important for the transferred data. The authentication and encryption can be supported using an external firewall and VPN infrastructure or to use integrated mechanism. Firewalls additionally try to protect against random denial of service attacks or virus intrusions. Independent of the security mechanisms used, user notification mechanisms are necessary to control and evaluate the level of protection. The system is enabled to inform user of certain security relevant operations. The reaction time and the possibility in causing resulting damage can be reduced.

3.2.3 Architectural styles

There are no specific requirements that have to be fulfilled by architectural styles, but the architecture states the modernity and flexibility of the product and an analysis is necessary for a reference architecture. A collection and discussion of common styles in software architectures can be found in [31]. The system model defines the components of the product. A client/server model defines client and server side components. A repository based system uses a central repository and a shared data design of system services. Agent-based systems use code or object migration to provide functionality on distribution servers or target clients.

Most client/server products use server components implementing the core logic and the distribution process, an administration client implementing a user interface to configure the process and configuration, and distribution activities. Depending on the implementation a target client is also used to install the products. Some products are implemented without server functionality or a target client. These variations provide more flexibility but can also restrict functionality. Client and server components can also be implemented Web-based. Web-based client components provide platform independence, are lightweight and even don't have to be installed before they can be used. Server side web components reduce the gap in distributing Web-based applications.

Further the product can be structured in layers, using different levels of abstraction to reuse base mechanisms and to make extensions and adaptations of the system more flexible (for example changes in the configuration and distribution process through Web-based applications). One step further is the creation of a whole distribution framework. The base functionality is implemented as system services in extension to the operating system. Distribution tasks are implemented on top of this base environment. Software interfaces, for example inventory management, can also benefit from the framework.

Another distinction can be made between process- and tool-related products. The process-related products implement a definition of the distribution process, where tool-related products are customized by companies to implement their externally defined processes. Therefore a company's process can be bought or supported. This leads to the assumption that the target group of the products should be separated in small, medium and large enterprises.

3.2.4 System interfaces

A distribution environment is by definition integrated into an existing environment and has to be connected with other systems (see 1.2.4). At least depth and width of integration possibilities should cover the functionality that is not implemented by the product. First an integration can be based on the development environment, where the software components are constructed and attributed. Development build operations and Version Management changes trigger operations of the distribution environment. For example a new revision of a file combined with an attributation can serve as a new customer configuration. The information stored in internal or external CM repositories can be used to automatically create package definitions or to update package-generating scripts. An inventory management system can be integrated to provide a target definition in the distribution process.

Additionally user or user groups are defined as targets for installation. Those definitions are best extracted from a directory service that stores organizational information or through an interface to an inventory database. A selection process should provide a flexible set of installation targets. Installer products and package formats can be integrated in the deployment process. The distribution environment may import externally created packages or use installation procedures provided by the operating system. Another type of installation procedure is the staging of static and dynamic web content. Content Management server based on Web-servers provide such features and can be integrated by defining the Web-server as the target machine with a customized installation procedure or by integrating content activities into a distribution plan. Licensing and payment systems provide additional relevant functionality and should be mentioned in the design of a distribution environment.

3.2.5 Interface techniques

Interfaces are used to integrate systems based on common communication techniques and language models. J2EE [32] and DCOM [11] are two RPC based communication mechanism for distributed systems. A tight integration between two applications should use one of them. A lightweight interface can also be implemented with the HTTP based SOAP protocol and Web-server integrated Java Beans [33]. Besides a protocol implementation it is also possible to integrate the products by customizing scripts or by scripting against a product's interface using languages like Java Script, VB Script, CGI, Phyton or others.

The standard mechanism for integrating application functionality consists of an API that is used by external programs or plug-in components running as a part of the integrated software. Another mechanism for synchronizing activities between two applications is the implementation of conditional execution and the usage of triggers. For example a product can be repackaged based on a trigger from the CM repository registering a new component's version. A further mechanism for integration covers a command line execution utility and other tools. It is also possible to provide an interface through an open software description like a standard package definition or wide used operation semantics like check in/check out.

All these different integration techniques should be chosen carefully. The more interfaces the more flexible is the integration, a good interface should also cover most of the functionality. It shouldn't be necessary to choose a specific technique for its functionality but to choose it, because it best fits for the integration needs.

3.2.6 Platform and protocols

Supported platforms and protocols extend the product's integration techniques. One factor are the supported operating systems. Beside the widely used Windows platform a distribution system has to support different Unix and Novell server platforms, Macintosh, and other systems. There are products that are just available for a homogenous platform infrastructure, which is very restrictive for a Software Distribution Environment. Other products provide a wide range of systems but are therefore more expensive to implement. Further the protocols supported are key points for integration. Some more specific interfaces are build upon their defined functionality. WebDAV extending the HTTP protocol can be used for basic Version Management over the internet. LDAP is used to access a query directory services. ODBC is used for database connectivity. XML is used as a general text based information exchange format. Domain specific XML applications implement functionality for Software Distribution like OSD. Windows Management Instrumentation [34] is a Windows based standard for inventory management that is built into the operating system. WBEM [35] are techniques for Web-based enterprise management. Again the right choice of platform and protocol support makes an integration suitable for a specific scenario.

3.3 Product Description

The evaluation is based upon the requirements defined above. Additionally individual functionality and relevance for the case studies (see 5.1) are described. Each product of the

comparison is described with the exception of Software Dock which has been introduced in section 2.2.

3.3.1 System Management Server 2.0

System Management Server (SMS) [36] from Microsoft is primarily a system management server. Therefore the support for software configuration and deployment is minimal. The architecture is based on a client/server model. The server functionality is provided with components that are implemented as system services (report generation, licensing, performance counting) or threads of the SMS executive system service (event manager for network logon, LAN and WAN data sender). The server uses a relational database for the storage of configuration data and gathered system data. Administrative clients are used to configure the system.

For installation tasks the SMS installer is used to create packages and package description files. The description is based on the OSD standard [22]. Additionally a SMS client is used on the target machine that consists of an implementation of the CIM protocol. The client can gather information about the computer to evaluate predefined requirements like disk or memory space. Further SMS supports patching of software. New versions of files can be installed automatically on computers with predefined older file versions. Uninstall and rollback support makes the deployment easier. Components and configurations can be removed or undone; a previous state of the client can be setup. Most of the configuration has to be done manually, less activities can be automated based on rules or conditions.

Dynamic distribution lists simplify administration of targets. User, Groups, and Machines can be created, grouped, moved, and deleted; the distribution of software is adapted dynamically to reflect the new structure. SMS supports routing of installation sources, in a way that data transfer occurs only once over WAN links. SMS also includes management of corporate- license- usage for each product. Operating system support is restricted to the Windows platform. Other features of the product are not related to software Configuration Management.

The usability for our case studies differs. The management of distributed software or Web-based software requires a complex distribution model. A definition of software component versions and their dependencies is not possible. The use for a company to distribute and update their products globally is minimal (see case study "Virus Scan"). SMS is designed to perform management tasks and the need for an installed client component is not acceptable for a large user base. For Case study "Workflow" SMS can be helpful for distributing and updating the software. The distribution is not fine grained and configurations cannot be handled on a component level, therefore configuring is subject to further manual interaction.

3.3.2 Perforce 4.0

Perforce implements a software Configuration Management software also based on a client/server model. It uses a central server for storing meta data and the product lines in a repository (called depot). The clients operate on a local workspace upon file copies.

The base of the functionality is the implementation of Perforce’s software lifecycle model [37], therefore it is development centered. Normal models define stages (development, QA, released) and promotion between stages. The management of version branches of files is hidden from the developer. Perforce uses their technique of Inter-File Branching where promotion of files is done explicitly by the developer [38]. Perforce defines use-policies that distinguish a main development line, maintenance lines and release lines. Every line gets file copies that are maintained separately from the main line. State is not maintained on a file but a line level of granularity. Multiple lines can be created on different levels. The main line can also just be labeled, tested and branched if tests are successful [39]. Retirement and reanimation of versions simply occurs when no more changes are made to the branch or new changes are made again [40].

Perforce performs checkouts by copying the lines to the developer's computer to gain local hard disk speed. Other operations are done via commands executed by the server, making the solution useable from 28.8k lines up [41]. For security concerns a host based access control is implemented, additionally firewalls and a secure shell access to the repository are supported. For web Content Management perforce supports different implementation models. First the Web-server can be given recent copies from the lines in the depot. Authors work on a specific line in the depot as usual. Second a test branch can be used for a test server and a production branch for the production server. Branching is then based upon the same mechanisms as for conventional software projects. As an alternative to external synchronization a tool is implemented that is attached to the Web-server and accesses the depot directly [42]. Integration is possible by using the functionality with a command line tool. Perforce can be used seamlessly instead of Microsoft Source Safe [43] and can also be used as a repository interface for an external project management environment. External tools can be integrated to support other merge mechanisms [44].

In terms of our case studies Perforce can be of use for both case studies as a versioning tool for the software development. The maintenance of several customer sites or the distribution of software for a large user base is not in the scope of the product. Therefore key functionality is left for additional deployment tools that are integrated with command line tools to access the repository data.

3.3.3 Not-so-bad-Distribution 1.4

Not-so-bad distribution is a project of Dave Dykstra at Lucent Technologies [45]. It is a Web-browser based tool that manages the synchronization of software packages. It can be integrated into environments for Configuration Management and Software Distribution.

It is possible to download software and register them for automated update, which forces security considerations. Certificates based on PGP [46] are used to assure a product's origin. Additionally protection is implemented to prevent damage through malicious or incompetent package providers. Just defined directories are accessible and functions available during installation can be specified. It is also suggested that different user accounts are used for distribution, installation and execution of software.

Maintainers make their package files available at an HTTP or FTP URL and create a Package Description file to specify local file location and installation parameters. They may include multiple binary executable types where users can select the types they want. Further a distribution file is used that contains the URL and secure checksums for the files in the package and is digitally signed. The file is made available though a Web-server. Packages can be updated by changing files and recreating the distribution file.

By using the URL in a browser the package is read and NSBD is started on the client. A user can schedule NSBD to periodically check for changes to the distribution file. New versions can be installed and files can be removed. Multiple changed files will be installed together. Users can permit maintainers to push updates by installing CGI scripts on their client that are invoked. The installation procedure is the same as above.

Currently NSBD is available as open source for Unix platforms. More information about integration- possibilities and the source code can be found at [47]. In terms of our case studies, the tool can be of great use to distribute software in a professional environment. Distribution for a user base external to the corporation might be a problem, because of the configuration necessary and the restriction to the UNIX platform.

3.3.4 CM Synergy 6.1

CM Synergy is a Configuration Management suite from Telelogic (former Continuus Software). It covers version, configuration, software component and change request management as well as distributed software compilation. The processes implemented can be customized. CM Synergy manages object-oriented data, controls every object type and supports user-defined object types, attributes, and methods. The relevant steps for parallel

development are implemented: access control, notification, comparison, merging [48]. Versioning can be used for projects, directories, files, and links. The versioning mechanism distinguishes between micro and macro revisions. Macro revisions are relevant for CM and potential releases. Configurations are defined by a rule system and also allow a creation of prior releases or for example “select version X and apply change set Y” [49], [50].

All components (including repositories) run on Windows and all major Unix distributions. Data in the repository is distributed manually, periodically or in real-time, depending on network bandwidth. Telelogic’s client/server architecture can be distributed across all supported platforms and uses commercially available RDBMS. With the Distributed Configuration Management (DCM) component location transparency enables a single view to configuration data. Another focal point is the support for external developer and content provider [51]. Of special interest is the Configuration Management for RAD development support. CM uses alternative build and test cycles, less formal change management, closer sharing between developers and rapid configuration changes. Less strict sharing of objects and a minimal set of maintained states eliminate overhead in the change process [19].

Additionally Change Synergy supports change management for user requests and defect tracking and reporting. This functionality fits for Web-based projects where software and content can be updated more frequently but have the same quality requirements. Web-based Version Management is implemented with different examination and approval processes. Beside web Content Management, the suite does not cover the process of Software Distribution. The additional product Web Synergy consists of a client that submits all kind of content and code from the original application to a Web-server. WebPages, applets, Java components and scripts can be versioned using check-in/check-out semantic.

For the case study "Workflow" the product might cover a lot of the development and configuration tasks and for the deployment of Web-based application, but it is not able to support server side components and on-site configuration of products explicitly [52]. One main benefit are the development changes on-site, that can be integrated in the change management process. For the case study "Virus-scan" solving the deployment based requirements are not the products focal points. CM Synergy is not designed for the distribution to a large anonymous user base.

3.3.5 InstallShield 7.0

InstallShield Developer is an installer software that is easing the process of software deployment [53]. The product supports different platforms (Windows, Unix, OS/400). My evaluation is based on the Windows product, but this should also fit for the other versions. Additionally AdminStudio is a product that covers distribution tasks beyond package creation and installer activities.

WindowsInstaller is a deployment software that allows to package software and other content (e.g. documentation, configuration files) [6]. The packages can be compressed and signed. InstallShield uses an XML file for the product definition that stores file properties, hashes for integrity and certificates for authenticity. The file format is praised as the interface for integration which is a bit inconvenient. The packager component creates packages for online distribution or splits them on disks or other media.

One part of the software is the builder component that creates setup procedures. Based on a cooperation with Sun and IBM specialties of different Unix platforms (e.g. on Solaris a product database registers information) have been implemented. Additionally a command line build capability provides creation of packages even over telnet sessions [54].

The setup process on the client executes with wizard-style panels. In the builder the panels are designed and conditions can be specified for visibility, data entry evaluation, panel sequence and further more. The conditions are based on logical expressions using product file properties. Many file types are recognized, their attributes accessible and file dependency scanning available. The installation can be run in wizard, console, or silent

mode. In console mode the wizard's interface is text based but provides the same functions as the graphical version. The silent mode works with an response file that provides the information for the panel definition. Multiple languages are supported.

To provide more flexibility the InstallScript language is introduced that is used also by Adobe, IBM, and Symantec. Scripts can manipulate the target consistent for different operating system platform. InstallShield integrates with source control systems such as: Microsoft Source Safe, Rational ClearCase, Merant PVCS and others through the Microsoft SourceSafe Source Code Control (SCC) interface [43]. New versions are recognized and existing package definitions can be used or updated.

AdminStudio extends the model with the components Repackager, Conflict Solver and Tuner. The Repackager creates packages that are used by other InstallShield products. It uses system snapshots to create packages by storing changes during installation of a reference workstation. The Repackager defines merge modules. Components like Java or VB Runtime are packaged once and then attached to any package definition. Additionally parts of the Windows Registry can be directly imported into packages. The Repackager creates INC Files for import into AdminStudio [55].

The Conflict Solver component checks conflicts and stores them in a database. Every combination of packages can be checked for component registration, file version, registry entries, shortcuts, INI Files, ODBC resources, NT service information, file extension registration or product code. Conflicts can be solved and a transformation file is created that changes the package content during installation to prevent conflicts. The package itself is not changed to reduce maintenance.

The Tuner component defines modifications on a package or transformation file. It defines pre- and post-validation and it can add custom files, registry keys, templates or shortcuts. Additional server or network based installation can be specified. The setup process can be customized, response files or predefined properties can be provided.

In terms of our case studies, InstallShield can be used to deploy software in a heterogeneous environment. Configuration management for more complex software or solutions for Web-based or distributed services is not addressed. The case study "Workflow" cannot rely only on this solution, it also needs a CM or at least an integrated Version Management solution. For the case study "Virus-scan" the deployment part of the CM process is covered well.

3.3.6 Tivoli Software Distribution 4.1

With the Tivoli Management Framework Tivoli provides a solution for managing heterogeneous networks. It supports AIX, HP/UX, Solaris and Windows. The framework works like a distributed operating system on top of the different platforms and provides base services for fault tolerance, logging, security, file transfer and further more. Based upon this framework modules for Software Distribution and for Inventorying are implemented [56].

The Software Distribution module consists of the following components: activity planner, change configuration manager and software package editor. The activity planer is used to define and schedule distribution activities. Operations of the framework can be used additionally. Activities can be grouped to form plans and can be executed conditionally. Distribution activities are controlled and monitored. The distribution process can be rolled-back, transactional distribution is also supported but lacks standardized installation procedures and custom scripts. The change configuration manager is a component that supports large networks of workstations [57].

The inventory module can be used to execute queries to define groups of target machines for the change configuration manager (CCM) [58]. CCM provides functions to define and group targets and different models to execute distribution plans. Push and pull installation and mixed methods are used. Direct access to a server, web download and email service are pull methods, data streaming and source redirection are push methods. Mobile clients are

supported by a store and forward installation. In a complex environment these methods are mixed and implemented on a server infrastructure that forms the different distribution models. Master server redirect sources to site server. WAN links have to be considered, compressed transfers and delta calculations are beneficial. Site servers use installation servers to perform load balancing or to split responsibility based on the distributed content (see also Figure 21: Tivoli transmission). Different installation methods can be used between the installation server and the client. Additionally the installation can be staged using a development, pilot and production environment, to predict failures during the distribution phase. The package editor is used to create software packages. The packages consist of the sources and information to install the software like registry values. The packager creates installation scripts automatically, those scripts can be further customized.

The software directly supports packaging with pre- and post snapshot and response file techniques. Snapshots are used to record system changes that are then performed on the target system. Response files are used as input for setup procedures and have to be supported explicitly. The installation can be undone, repaired, updated or removed. Locked files can also be updated and reboots can be integrated. External packager like Windows Installer or InstallShield can be used within the installation process [59], [60].

The solution seems to fit in every environment that can compensate the initial investments for the required base infrastructure and that benefits from the support of multiple platforms. For the case studies the product fits well to the requirements of a company supporting their customers in a tight relationship. Additional benefits are the further service of Inventorying and network management. The integration of a more complex Version Management strategy seems to be a lot of handwork and a separate product and an interface implementation would be necessary. For a company with a large user base the web and email distribution seems quite beneficial, even it should be used with the product and not because the product is the solution to these tasks.

3.3.7 Marimba Management Solution

Marimba provides a system management solution comparable to Microsoft SMS. Marimba supports Windows, Linux, Solaris, and other Unix operating systems. The solution is divided into modules. There is an infrastructure module that provides base functionality and additional modules for distribution, subscription, management and inventory [61].

The infrastructure module provides base services for distribution. Server use a transmitter and client use a tuner component to deploy content through channels. The module defines the processes Preview, Stage and Activation, Verify, Repair, and Rollback. The deployment can be ordered and scheduled. Dependencies and source filters can be defined. Any executable script can be used to pre- and post process or to execute during deployment. The deployment can also be used for operating system updates or security patches. It is efficient for server farms and their network infrastructure. Bandwidth control, compression and byte-level updates optimize transfer. A checkpoint restart of transmissions and transactional updates provide reliability. Security is implemented for user and content authentication and also admin session and deployment encryption. Central logging and display functionality is provided together with different log levels [62].

The distribution module is used for application packaging. Applications are prepared with a packager using OSD (see 2.2.3), an XML based software description language. Different packager are used depending on the software: windows installer, visual basic, java, a custom and a file packager. The definition contains a version comparison policy on a file or application level, application hardware and operating system requirements, script execution and reboots. The packages can be changed without recreation [63], [64].

The inventory module consists of an agent and a manager. The agent scans server in the background based on schedules. The agent reports inventory information via HTTP to the repository and it stores and forwards information for disconnected devices. Byte differences for subsequent scans can be calculated to reduce traffic. Information about hardware,

operating system, storage, i/o, peripheral and network parameter, installed applications and marimba defined installation requirements (for example file versions, user defined properties stored on the machine) are gathered. The repository uses a relational database viewed and queried with the inventory manager. Reporting generation is not integrated [65].

The management module defines transmitter, tuner and channels. Transmitters can be configured as repeaters to provide bad balancing. Transmitters can also be mirrors for higher availability and proxies for better performance. Target tuner can be defined using the client's locales, operating system or language. Targets can be grouped and groups can be nested and form virtual servers. Channels are defined based on source transmitter and target tuner and properties like security features. The tuner interface can be customized and the tuner itself is distributed. Additionally tuners for mobile user search for new content when connected to the network [66].

The software is administrated through a browser interface, additionally a command line tool exists. All modules can be integrated using an API with the SOAP protocol in an XML syntax. For example plug-ins for transmitter can process data or change channel behavior. The inventory repository database can be accessed and the scheme can be modified directly. Security mechanisms can use an external directory services and PKI infrastructure.

An additional product called Timbale covers the Content Management part. It consists of a Deployment Manager (Client) and a Deployment Service (Server). Code and content is staged in preview, staging and install phases. Dependency levels between operating system, application and data are defined and separate application and content building, integration and staging. Code and content can be updated automatic, scheduled and/or periodic. Deployment trigger allow nesting deployment sequences. DLL and registry management, macro and variable substitution allow application installation and customization. The distribution follows features of Marimba's core technology like byte level differences, compression, security, etc. An inventory management agent stores and forwards definable data via HTTP through default or customizable scan operations. Load-balancers and application servers can be integrated into Timbale [67].

The Marimba products fit very well for a company's internal administration. The distribution to customer sites can be accomplished for applications that don't need frequent updates. The installation of the tuner component for a large customer base is difficult. A detailed software component description, as it would be necessary for frequent updates, can not be accomplished by OSD.

3.3.8 NETDeploy 5.5

NETDeploy provides solution for software deployment. The infrastructure consists of an administrator application run on a Windows 2000 client. Deployment server und clients can be in a LAN, WAN or Internet environment. On the servers no additional software is necessary, therefore all operating systems are supported. To use features for large environments (priority queuing) a SmartPull Distributor component is used that is run on Windows 2000 server. The client platform is also restricted to Windows.

For deployment Windows Installer packages are used (OSD standard) where Windows Installation is available on the client. Additionally a proprietary installation function, SMS packages, zip files or self extracting exe files can be used. Applications, Windows service packs, drivers can be installed without customization of the setup process. Digital signatures are used for file integrity [68]. For the target definition NETDeploy uses the Windows 2000 Active Directory and its Group Policies [69]. A deployment policy (which user, group, or computer should get what) is defined and stored for deployment. The client evaluates whether the policy fits to its environment, it is not necessary to provide a complete Active Directory or Windows 2000 infrastructure.

The deployment differs between server-server and server-client communication. The server-server distribution is started by the administration client on a scheduled basis. Multiple

deployment server can be specified by package. Clients poll and automatically download new or updated deployment policies, schedules and software packages from the nearest server. Schedules can define a time, an allowed time span, user logon, PC startup, just-in-time delivery or priority queuing for installation. For all transmissions bandwidth throttling is implemented, a percentage of network capacity can be defined and only missing or updates files are transmitted.

A repair function is used upon corruption and can be specified in the deployment policy. A removal of application can be used and also scheduled for ASP rental models. The installation and deployment success is tracked and deployment policy satisfaction or exception can be reported. Notifications of major events through Email are implemented [70].

NETDeploy integrates WMI and the Microsoft Operations Framework [71]. It provides a COM interface and a process integration through an intranet portal. The deployment infrastructure is used for different scenarios: automatic installation of a thin client applications over the internet, in-house distribution of applications, PC complete installation for manufacturing and others. For our case studies the framework might be used for deployment. For individual customers ISDN or VPN channels can be used, for a large customer base the distribution over Web-server for deployment is possible, the necessary client component is disruptive. Work has to be done for Version Management and inventory integration.

3.3.9 Rational ContentStudio

ContentStudio is a Content Management solution for the UNIX and Windows platform. Through Rational's Configuration Management code and content have been integrated. Rational ClearCase is used as an artifact repository, that stores any versioned object or content. The versioning can occur through the original application like a word processor. A file and site history is stored and rollbacks are support. The automatically stored metadata in the repository is used to track creation, changes and expiration of artifacts.

Based on the versioning distributed development of code and content is possible. Mixed code and content changes are staged consistently. The incremental updates of content are more frequent and the deployment process is more complex than package creation of complete application. Application server like IBM Web sphere or Microsoft Component Services and therefore .NET and J2EE components are supported as a distributed environment. Different workflows for content change and software development are implemented that cover changing permissions, approving, testing and staging. Repository metadata stores which user can access which artifact during those activities. The multi-phased deployment uses staging areas with inner and outer firewalls and prevents links to secured resources. Test phases between staging steps are introduced. The launch of changes can be scheduled, expiration requirements can be specified.

The product is, in terms of the case-studies, more interesting for the small company integrating Web-services into their product and managing them using the strength of software development processes [72]. Further the Web-interface for the deployment of packaged products can benefit from ContentStudio, however it doesn't cover the distribution process itself.

3.3.10 Rational ClearCase v2001A

Rational ClearCase covers Configuration Management, version control, workspace and build management. It uses a client/server infrastructure and is implemented on Windows and Unix. There is a light, standard, and a multi-site version for distributed environments. While version control is distributed over servers and can be based on NFS, developers and their views are located on individual workstations.

The file versions and an annotated version history of all artifacts are stored in a repository. Dynamic and snapshot views of file versions are available for online and disconnected users.

Versions can be selected based on rules (e.g. "one used last time"). Automatic unlimited branching and binary sharing without copies enable flexible parallel development. Rational's own diff/merge technology allows to automatically merge files and solve conflicts. Rebuildability guarantees that a version that has been released can be recreated later. All common Version Management systems can be synchronized and major development environments can be integrated and viewed using ClearCase Explorer. For IDE´s, dependencies can be automatically detected and a report of bill can be constructed [73]. ClearCase defines projects, team members and associated activities. Projects are joined and activities are assigned, generating change sets of the product. Change Sets are collected to baselines of components and based on those application iterations are defined that can be released. Those iterations can be used for interfaces to a distribution environment. An event triggering mechanism allows customizing any ClearCase operation [20].

Rational products are integrated into the Unified Process, a process defined by Rational for the companies using their products [74]. In terms of the case-studies the unified process of ClearCase and ContentStudio is of good use for deployment to a large user base but overscaled for smaller companies or scenarios that primarily require distribution support.

3.3.11 Novadigm Radia

Novadigm's Radia Software Manager ships applications to clients in a LAN, WAN and Internet environment. A browser plug-in is downloaded from Microsoft or Netscape [75] Web-servers are used as the client-side component for the software deployment process. The plug-in is self-installing, self-repairing and self-updating. A web site presents available downloads for the clients and their currently installed software. Subscriber (clients) can autonomously detect the need of changes to their platform through the plug-in, no state has to be transferred between client and server. This technique called desired-state-model is a focal point of the product's architecture. Radia supports these pull techniques for installation, update, configuration and removal of products. A version roll forward and rollback mechanism keeps applications running. A verify option enables user to check the state of an application. Additionally global notifications can inform subscribers about available software versions. Updates can be installed without intervention or deferred on the client-side with an optional grace period [76], [77].

Providers implement an application wrapper for a service interface called E-wrap. E-wrap is a platform independent object-oriented capsule around an application. It provides the interface to the deployment process and the definition of dependencies and custom requirements. Simple conditional expressions allow modifications to software and content that is distributed and changes to the client's registry or operating system. Additionally the visual behavior during installation (setup wizard) and the application preferences can be defined. E-wrap also enables the access to policies stored in databases and directories like NDS and Active Directory. Policy associations can be established between organizational nodes of the directories. Additionally Windows Installer packages and the Windows Installer Service can be used [78]. C, C++, Java, Java Beans, Visual Basic, ActiveX components, other applications and other content are supported. Versions can be staged directly from the development environment. Radia can be integrated with Novadigm's own inventory management system, HP Openview, Tivoli or CA Unicenter. Interfaces to Version Management systems are more vague [79].

The light-weight client component allows this solution to be used for large-scale installation. The implementation of customized scenarios is also possible, but a more detailed concept for Version Management is necessary. The client- and server-side restriction to the Windows platform for the Software Manager makes implementation difficult.

3.3.12 Webstart

Webstart [80] is SUN's approach to software deployment for applications based on the Java 2 platform. Webstart is a reference implementation of the JNLP specification [81]. JNLP is intended for a Web-centric application model and consists of a deployment specification,

download protocols and a client execution environment. The supported applications are launched from resources that are accessible across the network through the HTTP protocol. Standard Web-server can be used, but some features need additional Servlet or CGI support. The technology is borrowed from the applet model. Applications should maintain most of their state on the server. Additionally a signing mechanism allows for trusted and untrusted applications. Versioning is supported based on pattern matching and a cache mechanism is specified providing a model that has initial first time activation costs but reduced network utilization.

The implemented process has no installation phase, but supports execution of external installers for platform dependent resources. A calling convention for invocation is described. Further an incremental download of resources for an application is supported, where JAR files, native files and support files are considered. Files can be downloaded before activation or on-demand by intercepting the Java class loader mechanism. A part attribute allows the partitioning of components that should be downloaded together. Also updates are incremental and supported by JAR diff files. The update process transparently uses the client cache for resources. Through the cache a limited offline-activation support is provided. For the JNLP-client / Web-server communication, the basic download, the version download and the extension protocol are used. The basic protocol can use timestamp information to distinguish versions. The version protocol uses unique URLs and version numbers to select components. The client cache relies on this information. A parallel use for different applications is therefore supported.

The JNLP description file defines how to download and launch an application. It is an XML document with a relatively simple structure. JNLP files can be nested and reference other files for installation of dependent products or site-specific installer. The file describes the needed JAR files, the required Java 2 platform version, packages it depends on, an application name and other display information, runtime parameters (version, heap size, resources platform), security information and system properties (operating system, architecture and locales). Additionally the document consists of application and applet descriptions and of component and installer descriptions (extensions). For applications and applets the main class and start parameters are defined. Extensions are downloaded with the extension download protocol and describe components that are shared between applications and installer applications that are executed once before the downloaded application itself. Wildcards can be used upon version numbers for selection of the right components.

The sequence for the client activation is: retrieve and parse JNLP file, determine runtime environment, download extension descriptors, run installer, download JAR and other files, verify signing and security, setup JNLP service and launch the application. The client execution environment consists of the runtime, HTTP proxy settings and a restricted sandbox that can be extended with required and optional APIs. JNLP supports a trusted and an untrusted environment and additionally a “j2ee specification conformant” and an “all-permitted” policy. Untrusted applications use the sandbox with restricted access to disk and network resources. For an application to be trusted, all JAR files are signed using the same certificate. Optionally the JNLP file can be signed. All certificates has to be trusted using root certificates.

For case study "Workflow" the product is not useable because the application is restricted to Java 2 platform products. For case study "Virus Scan" Webstart might be applicable depending on the distributed products. The creation of JNLP files and the component hosting on Web-servers should be supported and is beyond the capabilities of the Webstart reference implementation. A complete list of JNLP products can be found at [82].

3.4 Product classification

The product classification helps formalizing functionality of the introduced products. The classification has to deal with different range and scope of available product information. Therefore not all of the product’s functionality and not a complete feature support can be

discovered. Further a sound selection of classification criteria is difficult, because of the diverse focal points in the available literature. I use a plus symbol for supporting a functionality (+), a circle symbol for some partial implementation of a functionality (o), a (-) for no or just minimal support, and a blank for support is not in the scope of the product. The purpose of the comparison is not to find the product of choice but to further assimilate key functionality into a distribution framework.

Features/Products	System Management Server	Perforce	Not-so-bad-Distribution	CM Synergy	InstallShield	Tivoli Software Distribution	Marimba Management Solution	NETDeploy	Rational ContentStudio	Rational ClearCase	Novadigm Radia	Webstart
Versioning	**SMS**	**PER**	**NSB**	**CM S**	**INS**	**TIV**	**MM**	**ND**	**RCS**	**RCC**	**NO V**	**WS**
Versioning		+		+					+	+		
Rule based version definition		-		+					-	+		
State level product/version/file		-/+/+		+/+/+					+/+/+	+/+/+		
Local workspace/snapshot		-/-		-/-					+/-	+/+		
Release lines/Change Sets		+/-		-/+					-/-	+/+		
Release consistency		O		+					O	+		
Parallel dev. optimistic/pessimistic		+/-		+/+					+/-	+/+		
Trigger, Pre-Conditions		+		+					-	+		
Rollback /Revision history		-/+		+/+					+/+	+/+		
Build support		O		+[1)]					-	+		
Change audits / Defect tracking		+		+/+					+/+	+/+		
Configuration Management	**SMS**	**PER**	**NSB**	**CM S**	**INS**	**TIV**	**MM**	**ND**	**RCS**	**RCC**	**NO V**	**WS**
Mngmnt. of development artifacts	-	-	-	+	-	-	-	-	+	+	-	-
Feature definition and customization	-	-	-	-	-	-	-	-	-	-	+	-
Component def. and desc.	-	-	-	-	-	-	+	-	O	-	+	+
Product and release management	-	+	-	+	+	-	-	-	+	+	-	O
State level product/version/file	+/+/-	-/+/+	+/+/-	+/+/+	+/+/-	+/+/-	+/+/+	+/-/-	+/+/-	+/+/+	+/+/-	O
Multiple configurations	-	+	-	+	+	-	-	-	+	+	-	-
Options, Variants, Extensions	-	+	-	+	-	-	-	-	-	+	-	-
Product families	-	-	-	+	+	-	-	-	-	-	-	-
Software description language	+	-	-	-	+	+	+	+	-	-	-	-
Consistent code+content changes	-	-	-	+[2)]	-	-	-	-	+	-	+	-
Package / Repository-object based	+/-	-/+	+/-	-/+	+/-	+/-	+/+	+/+	-/+	-/+	+/-	-/+
Snapshots	+	-	-	-	+	+	+	+	-	+	-	-
OSD / Own / other package format	+/-/+	-/-/-	-/+/-	-/-/-	-/+/+	+/-/-	+/-/-	+/+/+	-/-/-	-/-/-	-/+/+	-/+/-
Package transformation files	-	-	-	-	+	-	-	-	-	-	-	+[7)]
Merge modules/systems of systems	-	-	-	-	+	-	-	-	-	-	-	+
Software/User specific package	-	-	-	-	-	+	-	-	-	-	+	-
Software Distribution	**SMS**	**PER**	**NSB**	**CM S**	**INS**	**TIV**	**MM**	**ND**	**RCS**	**RCC**	**NO V**	**WS**
Channels / distribution plan	-/+		-/-		-/-	-/+	+/-	-/-	-/-		-/+	-/-
Activities / A. hierarchy	+/-		+/-		+/-	+/+	+/-	+/-	+/-		+/-	-/-
Push / Pull / Mixed	+/-/-		+/+/+		+/-/-	+/+/+	+/+/+	+/+/-	+/-/-		+/+/+	-/+/-
Server-Server distribution	+		-		-	+	+	+	+		+	-
Mobile client support	-		-		-	+	+	+	-		+	O
Installation, Update, Removal	+		+		+	+	+	+	+		+	+

Repair, Patch, Rollback/forward	-		-		+	+	+	+	-		+	-
License management	+		O		-	-	-	+	-		-	-
Transactional deployment	-		-		+	+	+	-	-		-	-
silent/console installation	-/-		-/-		+/+	+/-	-/-	-/-	-/-		-/-	-/-
Automatic, triggered, scheduled	+/-/+		+/-/-		+/-/-	+/+/+	+/-/+	+/+/+	+/-/+		+/-/+	-/-/-
Activation schedule for ASP	+		-		-	-	-	+	+		-	-
Staged deployment	-		-		-	-	-	+	+		-	-
Tests / pre-install checks	-/-		-		+/+	-/+	-/-	-/-	+/+		+/-	-/-
Update specific file versions	-		-		-	-	+	-	+		+	+
User/user group target based	+/+		-/-		+/+	+/+	+/+	+/+	-/-		+/+	-/-
Machine/IP target based	+/+		-/-		+/-	+/+	+/-	+/-	-/-		+/-	-/-
Inventory management based	+		-		-	+	+	+	-		+	-
Rule/channel target based	+/-		-/-		-	+/-	+/+	+/-	-/-		+/+	-/-
Dynamic target membership	-/+		-		+	+	+	+	-		+	-
Content	**SMS**	**PER**	**NSB**	**CMS**	**INS**	**TIV**	**MM**	**ND**	**RCS**	**RCC**	**NOV**	**WS**
Server comp. (J2EE , NET)	-	+	-	+	+	-	+	-	+	+	-	-
Web (HTML, Java Script)	-	+	+	+	+	+	+	+	+	+	+	+
Objects (Java, VB, C++)	+	+	+	+	+	+	+	+	+	+	+	+
Executives (Byte code, EXE)	+	+	+	+	+	+	+	+	+	+	+	+
Shared libraries, DLLs	+	+	-	+	+	+	+	+	-	+	+	+
documents, charts	-	-	-	-	-	-	-	-	+	+	+	-
Availability and performance	**SMS**	**PER**	**NSB**	**CMS**	**INS**	**TIV**	**MM**	**ND**	**RCS**	**RCC**	**NOV**	**WS**
Load balancing	-	+[5)]	-	+[5)]	-	+	+	+		+[5)]	-	+[8)]
Proxying / data caching	-		-		-	+	+	+			-	+/+[8)]
Resume transfer / server switch	+/-		-/-		-	+/-	+/-	+/+			-/-	-/-
WAN Route definitions	+		-	+[5)]	-	+	+	+		+[5)]	-	-
Bandwidth throttling	-		-	+[5)]	-	+	+	+			-	-
Bandwidth specific deployment	+		-		-	+	-	-			-	-
Offline source transfer	+		-		-	-	-	-			-	-
Priority queuing	-		-		-	-	-	+			-	-
Delta transmissions	-		-	+[5)]	+	+	+	+			+	+
Data compression	+		-		+	+	+	+			-	+
Recover upon corruption	+		-		-	+	+	+			-	-
Security	**SMS**	**PER**	**NSB**	**CMS**	**INS**	**TIV**	**MM**	**ND**	**RCS**	**RCC**	**NOV**	**WS**
Access control	+	+	+	+	-	+	-	+	+	+	+	-
client certificates	-	-	-	-	-	-	-	-	-	-	-	-
Server certificates	-	-	-	-	-	-	-	-	-	-	-	+[8)]
user certificates	-	-	-	-	-	-	+	-	-	-	-	-
content certificates	-	-	+	-	+	-	+	+	-	-	-	+
Support for PKI infrastructure	-	-	-	-	-	-	+	-	-	-	-	-
File integrity, hashes	+	+	+	-	+	+	+	+	-	-	+	-
content encryption	-	-	-	+[6)]	-	-	+	+[6)]	+[6)]	-	-	+[8)]
Admin session encryption	-	-	-	+[6)]	-	-	+	+[6)]	+[6)]	-	-	+[8)]
Single / multiple Firewall support	-/-	-/-	-/-	+/+	-/-	-/-	-/-	+/+	+/+	+/+	-/-	+/+[8)]
User notification	-	-	-	-	-	-	-	-	-	-	-	-
Architecture and components	**SMS**	**PER**	**NSB**	**CMS**	**INS**	**TIV**	**MM**	**ND**	**RCS**	**RCC**	**NOV**	**WS**
Agent-based	-	-	-	-	-	-	-	-	-	-	-	-
Client/server	+	+	-	+	O	+	+	+	+	+	+	+

Code or object migration	-	-	-	-	+	-	-	-	-	-	-	
Server side component	+	+	+	+	+	+	+	+[4]	+	+	+	-
Administrative client / web client	+/-	+/-	-/-	+/+	+/+	+/-	+/+	+/+	+/+	+/+	+/+	-/-
Target client / web client	+/-	+/-	-/+	-/-	+/+	+/-	+/-	+/+	+/+	+/+	+/+	+/+
Web-server based	-	-	+	+[2]	+	-	+	-	+	-	+	+
Layered structure	-	-	-	+	+	+	+	-	-	-	-	-
Framework	-	-	-	-	-	+	-	-	-	-	-	-
Toolset / Process related	-/+	+/+	+/-	-/+	+/-	-/+	-/+	-/+	-/+	+/+	-/+	+/-
Small/Medium/Large Enterprise	-/+/+	-/+/+	-/-/-	-/+/+	+/+/+	-/+/+	-/+/+	-/+/+	+/+/+	+/+/+	-/+/+	+/+/+
System Interfaces	**SMS**	**PER**	**NSB**	**CMS**	**INS**	**TIV**	**MM**	**ND**	**RCS**	**RCC**	**NOV**	**WS**
Software development (IDE)	-	O	-	+	+	-	-	-	+	+	+	-
Version management system	-	+	-	+	+	-	-	-	+	+	+	-
CM repositories	-	+	-	+	+	-	+	-	+	+	+	-
Inventory management	+	-	-	-	-	+	+	+	-	-	+	-
Directory Service	+	-	-	-	+	-	+	+	-	-	+	-
Installer products	+	-	-	-	+	-	+	+	-	-	+	+
Content Management (Web-server)	-	-	+	+[2]	-	-	-	-	+	-	O	-
Interface techniques	**SMS**	**PER**	**NSB**	**CMS**	**INS**	**TIV**	**MM**	**ND**	**RCS**	**RCC**	**NOV**	**WS**
J2EE / DCOM support	-/-	-/-	-/-	-/-	+/-	-/-	-/-	+/-	+/+	-/-	-/-	-/-
Java Beans / SOAP	-/-	-/-	-/-	-/-	+/-	-/-	-/+	-/+	-/-	-/-	-/-	-/-
Integrating scripts	+	+	+	+	+	+	+	-	+	+	-	-
Plug-in or API techniques	+	+	-	+[3]	+	+	+	+	+	+	+	+
Conditional execution / trigger	-	+	-	+[3]	+	+	+	-	+	+	-/-	-/-
Command line execution/other tools	-/-	+/-	+/-	+/+	+/+	+/+	+/+	-/-	-/-	+/+	-/-	-/-
Open software description	+	-	-	-	+	+	+	+	-	-	-	+
Platform and protocols	**SMS**	**PER**	**NSB**	**CMS**	**INS**	**TIV**	**MM**	**ND**	**RCS**	**RCC**	**NOV**	**WS**
Windows support for Client/Server	+/+	+/+	-/-	+/-	+/+	+/+	+/+	+/+	+/+	+/+	+/+	+/+[8]
Unix support for Client/Server	-/-	+/+	+/+	+/+	+/+	+/+	+/+	-/-	+/+	+/+	-/-	+/+[8]
VMS/OS390 support	-/-	+/+	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-
Mac support for Client/Server	-/-	+/+	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-	+/+[8]
NetWare/OS2 support	+/-	-/-	-/-	-/-	-/-	+/+	-/-	-/-	-/-	-/-	-/-	+/+[8]
HTTP/WebDAV support	+/-	-/-	+/-	+[2]/-	+/+	-/-	-/-	+/-	+/+	+/+	+/-	+/-[8]
LDAP support	-	-	-	-	-	-	-	+	-	-	-	-
ODBC support	+	+	+	+	+	+	+	+	+	+	+	-
XML support	-	-	-	-	+	-	+	+	+	-	-	+
WMI/WBEM support	+/+	-/-	-/-	-/-	-/-	-/-	-/-	+/+	-/-	-/-	+/-	-/-

1) with Continuus/OM, 2) with Web Synergy, 3) for merging, 4) optional, 5) for Version Management, 6) with SSL, 7) with JARDiff, 8) depends on Web-server

Figure 5: Product comparison table

4 Architectures of typical products

In this chapter the main architectural styles and models of the products are described. These styles and the requirements described in 3.1 and 3.2 lead to the composition of a reference architecture. Key concepts for an architecture are chosen and the implementation techniques of the products are pointed out. The UML notation [83] is used throughout the architecture related chapters 4 and 5. Recommendations in the use of the language can be found in Fowler [84].

4.1 Configuration and Distribution process

In this section I compare the range and technique of implemented Configuration and Distribution processes. The process is detailed for Tivoli, ContentStudio and Castanet, the other products are described related to them. First Tivoli is presented, a traditional distribution solution covering the core processes for automated software installation in a large environment. Figure 6 shows the main configuration and distribution processes covered by the Tivoli framework.

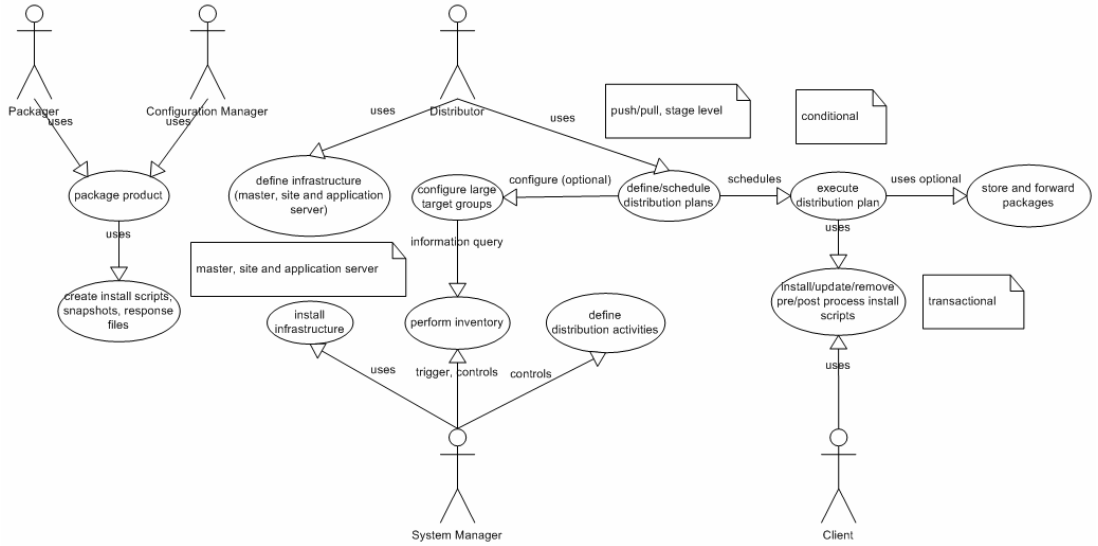

Figure 6: Tivoli processes

The packaging is done external and can use additional third party products. Additionally scripts, snapshots and response files are supported. The distribution is defined using plans with nested activities. The plans contain schedules that trigger the activities. Activities can perform several installation tasks (also uninstall, update, etc.) using different staging levels (test lab, production). Activities can be executed as transactions with rollback support. Inventorying is optional, for large environments inventory queries can be used to define target groups for a distribution plan. The second process description covers Marimba Castanet and is presented in Figure 7.

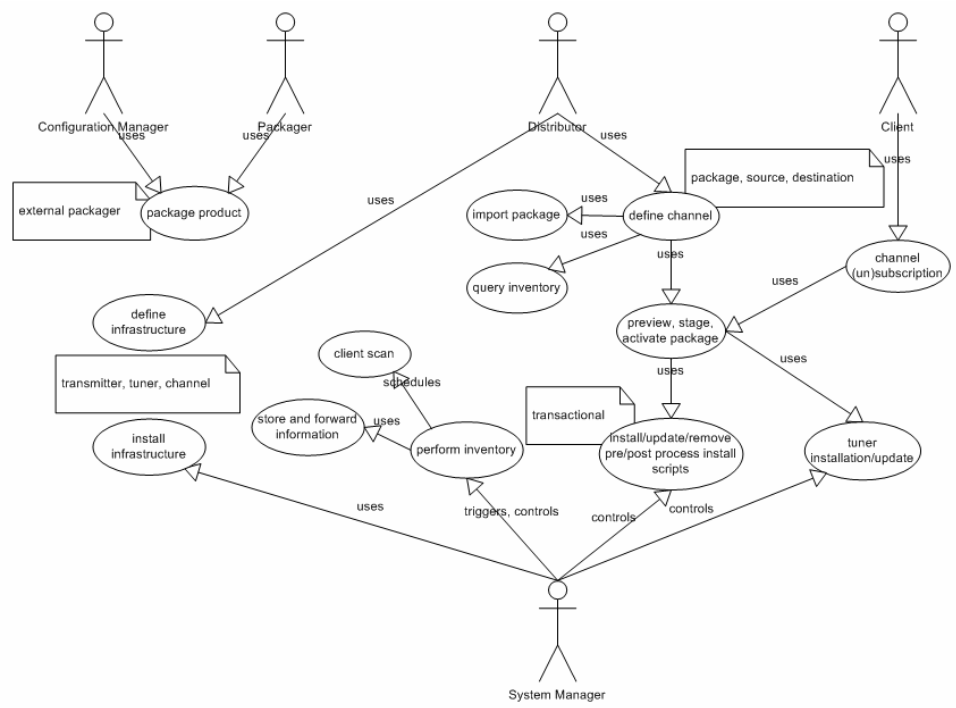

Figure 7: Castanet processes

Instead of plans and activities, channels are used to describe the Software Distribution. Instead of centrally defining plans to distribute software, client submit channels to products they want to have installed. A schedule triggers the transmission to the client. The installation process is transactional, the Inventorying optional and the client component self-installing. Especially for large environments, the infrastructure (transmitter, repeater, and tuner) has to be defined and installed. To provide a different process view, Rational ContentStudio introduces Use-cases related to Content Management (Figure 8).

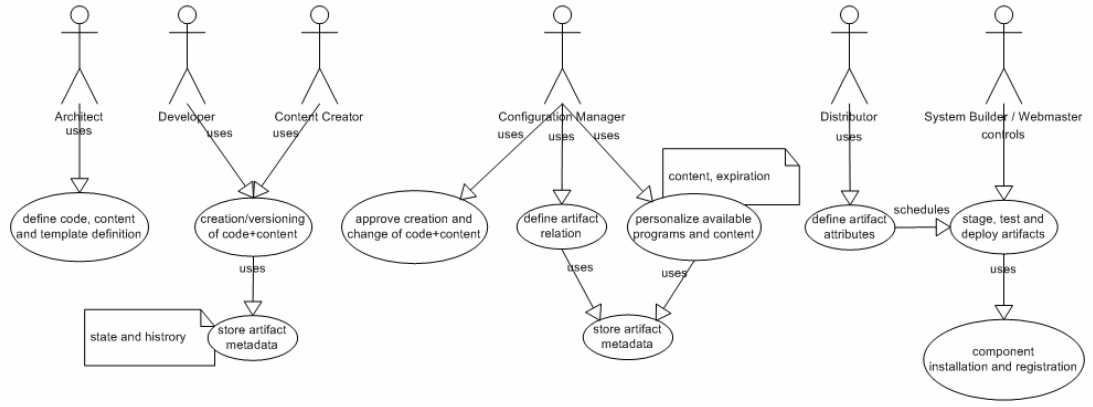

Figure 8: Rational processes

ContentStudio combines the code and content creation using templates provided with the product's design. All artifacts (code and related content) are stored together with metadata. The Configuration Manager approves changes and relates code and content. The Distributor personalizes the view to the artifacts and defines schedules for staging. A Web Master controls the actual staging process and component installation on web- and application servers. This process is web solution centric and doesn't deal with a client installation site, assuming a heterogeneous Web-browser environment. Compared to these models all other products can be related briefly. Microsoft SMS and CM Synergy are close to the Tivoli model. They use the same process for the definition and distribution of software and for Inventorying. Distributions are defined using plans and packages are created externally. Perforce uses a model similar to Tivoli but for a smaller environment. Products are not staged and no explicit test environment is used. Novadigm's Radia combines the staging, transactional installation and Inventorying of Tivoli with a Web-server based pull mechanism, where clients subscribe software packages. Also NETDeploy combines the push distribution between servers with a poll and download mechanism of clients, but it only support the packaging and installation processes. Radia and NETDeploy make extensive use of Directory Service information for package definition. InstallShield covers just the packaging and installation processes of Tivoli, but with additional installation options (silent, console, etc.). NSBD and Webstart are two simple Web-server based installation techniques that use an URL software description to install components from Web-server locations.

Summarized, the configuration and distribution processes can be defined as code and content creation, versioning, version selection and packaging, distribution definition, distribution, deployment (install, update, etc.), Inventorying and infrastructure planning and installation. The products support these processes with different techniques and focal points.

4.2 Software abstraction

In this section the software abstraction techniques of the products are described. The model of abstraction mainly influences the deployment technique. Client/server applications need different representations than Web-based applications, the chosen model restricts support on this basis. One model is the storage of product artifacts in a shared repository. For example the Rational Content Studio completely relies on the ClearCase Repository (see Figure 7).

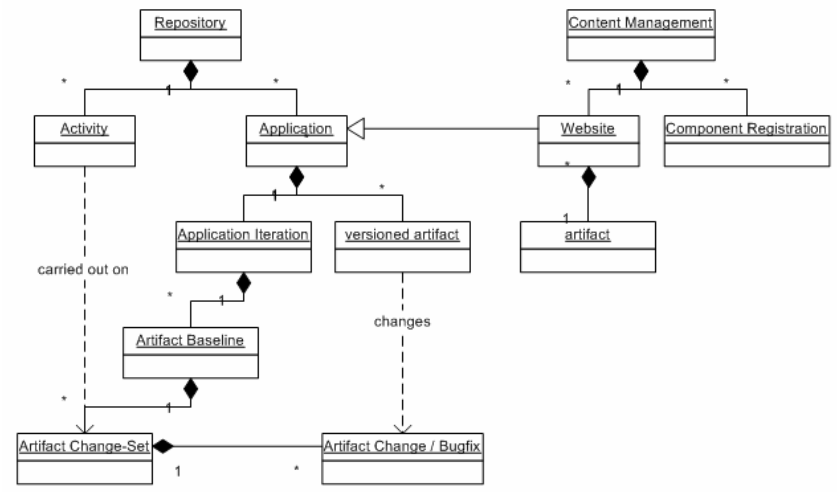

Figure 9: Rational software abstraction

This model uses the storage also as part of a version control system. The selection of artifacts for installation is done directly on the repository. The Content Management is reduced to a website definition and a component registration. The software deployment is reduced to component registration for web or application servers, a client-side installation is not

considered. No packaging mechanism is used. In contrast a packaging mechanism as it is used in InstallShield is shown in Figure 8.

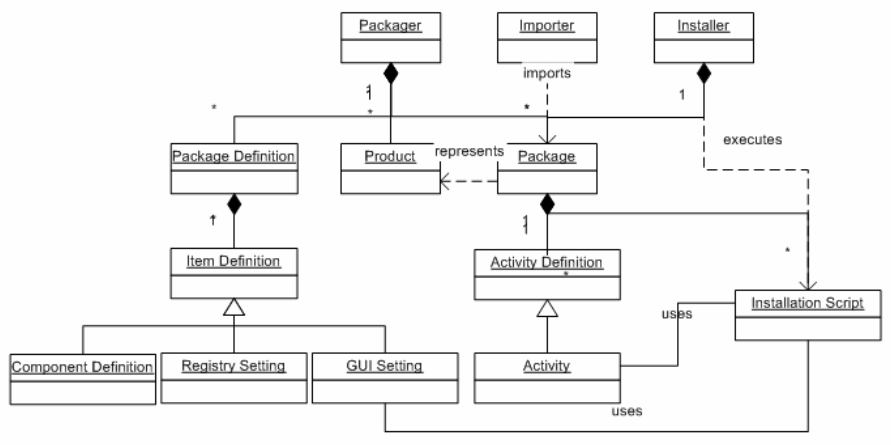

Figure 10: InstallShield Software abstraction

The product consists of an importer, packager, and installer tool. They all share the definition of a package, which consists of items and activities. Items are the product description necessary for installation and consist of program files, registry, and GUI settings. Activities are carried out upon them to install a product. Considering the software and the target system, different activities are required for installation. Additionally more or less activities and items are supported, depending on the package definition used. The packager instantiates package definitions for a specific client installation and creates an installation script that describes the activities. The script additionally defines installation options and application preferences like the visual behavior during installation. Items, activities and the installation script are contained within a package file. The importer covers externally created packages that are extended with an installation script. The installer component and the packages are loaded on the client and the installer executes the installation script defined by the package and performs the contained activities.

Perforce and CM Synergy use the Rational approach with a repository and a version control centric model. It is significant that Perforce also provides Content Management features and CM Synergy provides at least versioned content. The packaging paradigm is also used by System Management Server, Marimba, Tivoli and NETDeploy (for standard support see section 3.4). Unlike InstallShield the package definition can be automatically filled with Inventorying information from the client platform. The supported installation items and activities are quite the same. NSBD uses its own packaging format and supports only component items. Webstart doesn't use packages because all product files (Java classes) are loaded on-demand. The installation procedure uses a description file for all product components.

Radia uses an application wrapper (E-wrap) to provide a homogeneous platform independent description of the software. E-wrap is much like the packaging mechanism a software abstraction for the installation process. Instead of the possibility to import packages, the wrapper has to be implemented for all required applications. Conditional expressions replace the installation script capabilities of selecting specific installation options.

Beside this product support the OSD [22] and CDF [85] software definition formats are abstractions that are not very detailed but more generally agreed on. For distributed applications Corba [9] and DCOM [11] provide component models that are partially self-describing using registration services that are standardized only within their own model to distribute applications.

Summarized every product supports some kind of software abstraction. The abstraction used restricts the support of applications to the provided abstraction model. Web-based

applications and distributed components are tried to be covered by Content Management and conventional distribution systems. Two categories of models have therefore evolved.

4.3 Target abstraction

The target abstraction is necessary to provide customized software to the client. Target parameter influence which file or product version has to be installed. The more detailed the information is, the better are the resulting installation procedures. An inventory-based model provides such information. In environments with anonymous targets (e.g. Internet), where detailed Inventorying cannot take place, a pre-selection of versions can be done manually and the installation procedure can evaluate parameters and customize the installation on-site. An example inventory model for the System Management Server is shown in Figure 11.

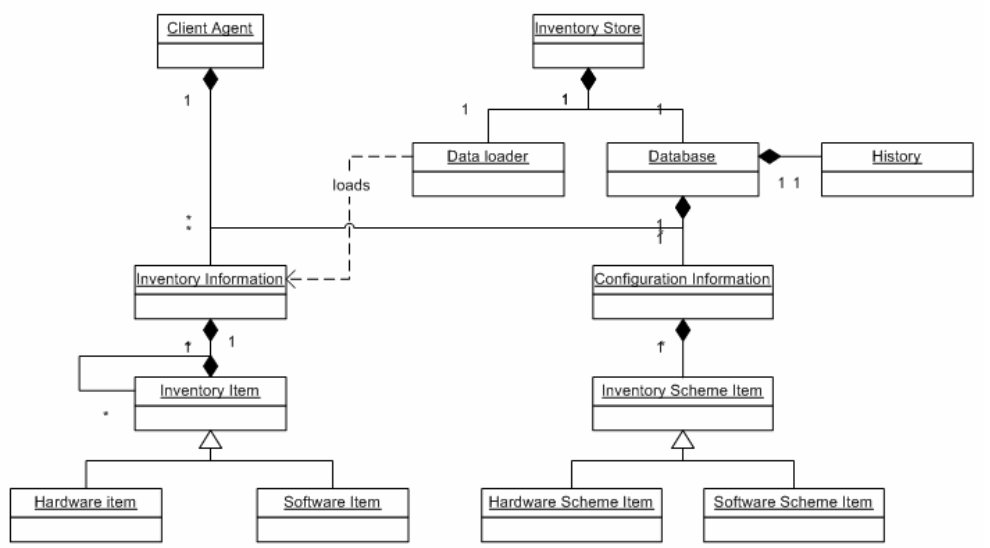

Figure 11: SMS Inventorying

The main component is the database, stored schemes and inventory items. Schemes abstract specific hardware platforms and software applications (e.g. which data should be gathered for a laptop). A configuration file stores this information with additional parameters. The client agent is activated on a scheduled basis and performs scans on the client platform. The created information items are stored in an inventory file. The file is transferred back to the inventory store over a multi layered server network that concentrates and forwards this information. The inventory store uses a separated data loader to fill the database. A history is additionally provided and allows to search for changes of the client's hard- or software [86].

In contrast to this concept the model of Webstart is described. Webstart loads applications on-demand and performs checks on installation invocation. Figure 12 shows a class view of the main components.

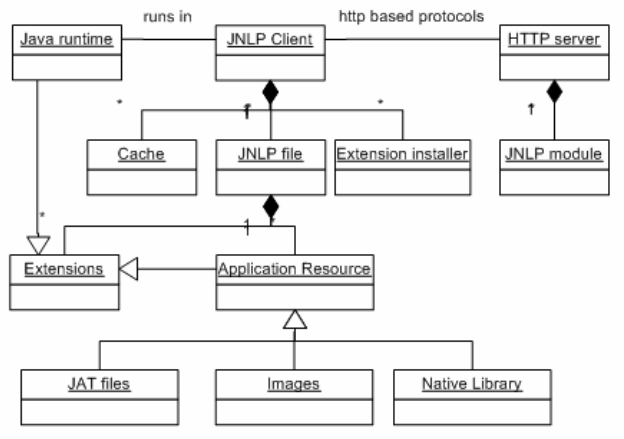

Figure 12: Webstart application invocation

Webstart implements an application environment without Inventorying requirements. A JNLP file describes all resources required by an application. Provided that the runtime can be executed, most hardware requirements can be ignored, because of Java's platform independence. Beside resources that don't require installation, extensions are defined and an extension installer is implemented that allow components to be installed during program invocation. The components are cached on the client side and can be versioned optionally. Also the runtime itself can be updated. The server has an optional JNLP module that is necessary if the versioned protocol is required. The request for a specific file version has to be satisfied by implementing a version scheme and a selection mechanism transparent to the client. Additionally the server provides application selection information that is presented to the user by the JNLP client for manual invocation and installation.

Tivoli, Marimba and Radia use an inventory-based approach. They all work with package-based distribution. Packages need the deployment information on generation and are therefore more dependent on an inventory. NETDeploy, InstallShield and NSBD use packages but lack inventory functionality. Manual creation of target information is necessary for successful installation. Perforce and Rational don't provide this functionality und also don't work with packages. Inventorying is not in the scope of the products' problem space. Beside specific product support several standards cover the target information. A common desktop interface layer is specified with DMI [87]. One effort is the Software MIF, a hierarchical data model. CIM is a newer object-oriented version [88]. Tivoli uses AMS, a MIF superset. Gnu Autoconf and the RedHat Package Manager are samples for the Unix operating system. The Registry is the target information repository on Windows systems. More detailed information can be found in [89].

Summarized, a target abstraction is a key issue for Software Distribution. The target can be described using Inventorying, content based or on-demand installation models. Goal of all models is making target information available to support the correct installation of applications.

4.4 Customer abstraction

The customer abstraction covers site and organization management, application customization, customer-based distribution models and licensing. In general it has to be distinguished between a technical term of a customer as a site of installation and a term that described the role and requirements and considers them in every use case of Version Management, Configuration Management and Software Distribution. Depending on the degree of detail the customer abstraction is not considered separately but as a part of the target abstraction.

First, the SMS site and organization management model is shown in Figure 13. It covers distribution, management, Inventorying and licensing information in a site hierarchy that reproduces the geographical and organizational structures of one or several customers. This approach has resulted from the assumption that one organization is in control of both sides of the deployment equation, i.e., the producer side and the consumer side [7].

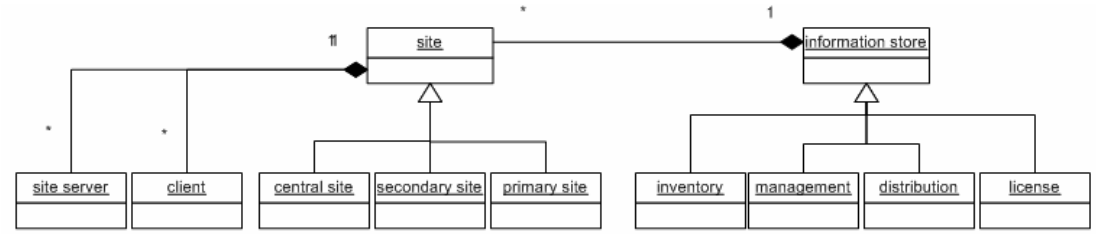

Figure 13: SMS site hierarchy

The hierarchy consists of one or several trees that define a parent-child relationship. A site can be primary (storing information) or secondary (delegating). The central site is the tree root. A site consists of a site server that stores (primary site) or forwards (secondary site) information upwards the hierarchy. One central information store is used throughout a site hierarchy. The client component produces information for inventory and management. The server uses additional centrally managed distribution and licensing information. A customer can be described as one or several sites. A large environment can use a single hierarchy per customer allowing to self-maintain the corresponding information store. In contrast small sites can be attached to one hierarchy with the software distributor acting as the root site.

In contrast the agent-based model of Software Dock doesn't centralize customer information and doesn't require information about the customer before installation takes place. This model is shown in Figure 14.

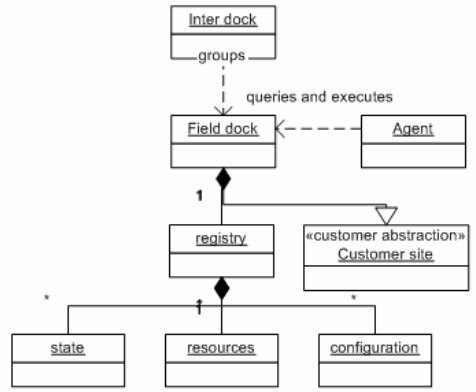

Figure 14: Software Dock customer abstraction

Software Dock doesn't separate the customer from the target, instead it provides an autonomous description. The Field Dock holds the information of the customer (consumer) site in a registry-like storage and provides access for agents through an interface. The agent is unaware of the customer environment before docking. Actions necessary for successful installations are derived from the software information carried by the agent and the state information carried by the dock.

To contrast the models of site management and agent-based customer interfaces shortly, a third content-based model is introduced (Figure 15).

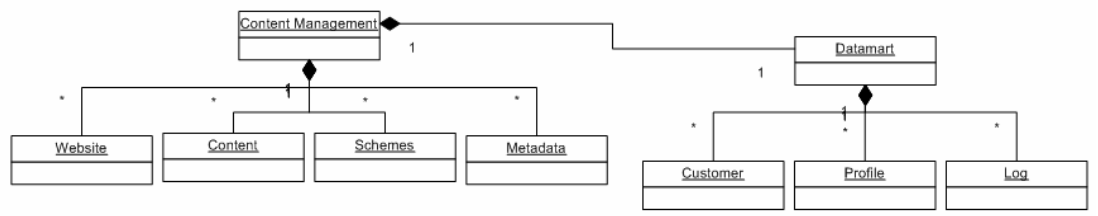

Figure 15: Vignette customer abstraction

Vignette implements an online tracking system for customers' interaction with websites [90]. Content is attributed based on scheme information. Attributes are stored in metadata. Based on this information the interactions create events that are stored in a datamart. The datamart can be extended with log files from the web or application server. Presentation of content and software changes are based on datamart analysis of the Content Management engine. Another model is used by NETDeploy which is based on the Windows 2000 Active Directory and its Group Policies. This model uses the hierarchical structure and grouping of resources in Directory Services as a target and customer description. Other products like SMS don't rely on this structure but execute queries to gather information. Further customer abstractions in Tivoli, Marimba and Novadigm are related to the SMS model. Most package-based installer environments rely on information of the target platform. Version Management systems lack such an abstraction.

Summarized, customer abstractions are partially considered in the products architecture. Organizational and individual configuration information and real-time interactions are part of the models. Different functionality requires this abstraction and the range of supported target groups and application is restricted by the abstraction. For the first and second model deployment information is gathered at the client, which has to execute a client-side component. In the third model the addressed applications are not dependent on deployment information. The model takes the customer information to the application and uses it for content and application profiles.

4.5 Components of a distribution environment

After the abstraction-based logical view of a distribution environment this section covers the components and building blocks as a physical view of the products' architecture. First the Tivoli model of the environment is shown. Tivoli as a distribution framework has a complex infrastructure that provides base services for distribution. Based on this infrastructure a distributed framework for Software Distribution is built.

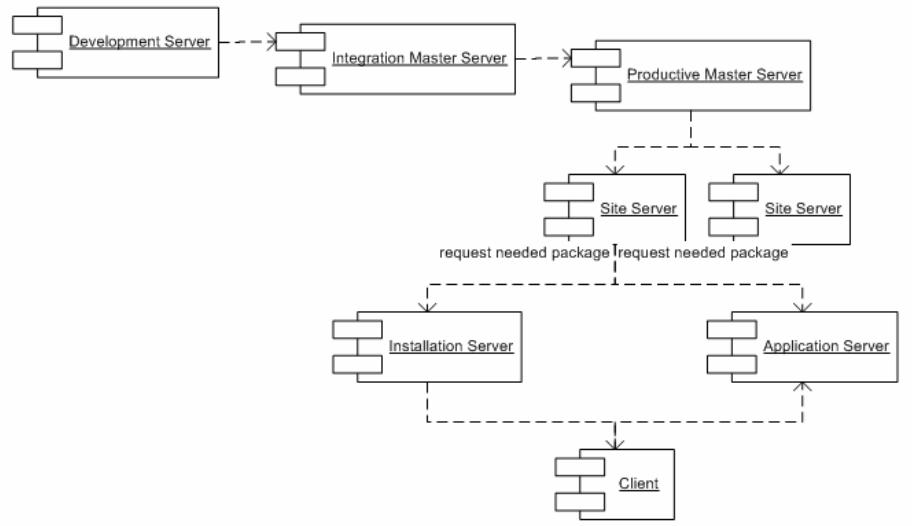

Figure 16: Tivoli distribution environment

All server and client components are part of the framework. The different stages of development, testing, and production can be physically separated sharing the same base infrastructure. The clients can be accessible through a hierarchy of site servers. The on-site infrastructure consists of Installation and Application servers providing server and client side software. The site-based infrastructure restricts the solution to departmental or office locations. The framework dependencies of clients require a direct physical connection to the corresponding site. On the other hand, extensive client information is available to the distribution process. A mixed push-pull model is used to distribute packages. It is based on replication and caching services of the infrastructure.

The second model of NETDeploy is also based on a server infrastructure but also integrate existing file servers as part of the environment.

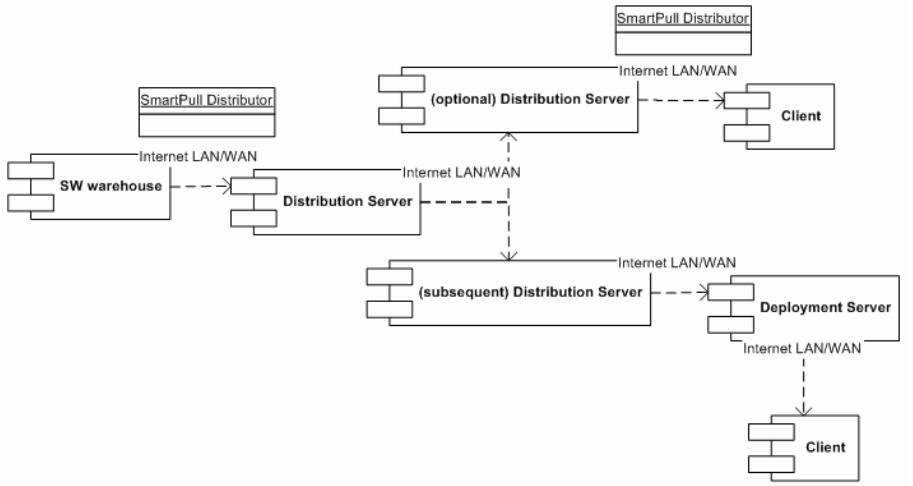

Figure 17: NETDeploy distribution environment

Hierarchically structured distribution servers distribute content over separated locations. The NETDeploy servers provide reliable, secure transfer of content using push and pull communication. The additional file servers act as passive components in the push/pull distribution and allow smaller client locations to be reached. The client is part of the environment and uses the NETDeploy client component. Tracking information is handed back by the client. In the smallest scenario an administrator uses a file server to distribute content to the NETDeploy clients. In this case limited functionality is available and no server infrastructure is necessary.

In contrast the third model introduces the agent-based components of the Software Dock system. The model has been introduced in 2.2.1. A further description of the components in a physical model is shows in Figure 18.

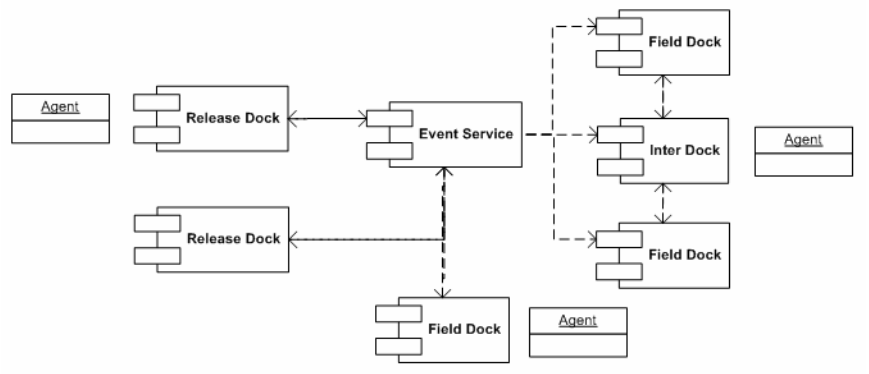

Figure 18: Software Dock components

The environment consists of Release, Inter- and Field Docks and Agents. The Docks correspond to the developer, distributor and customer locations. Release Docks are completely independent of each other. Field Docks can be grouped together by Interdocks. All Docks provide interfaces for Agents to receive data and execute distribution activities. A global event service that is distributed on a server infrastructure is used to communicate between Docks independent of Agents to provide state information about the distribution activities. While the server infrastructure and physical distribution can be implemented very flexible, the Field Dock again makes a client component necessary.

Beside these models several other approaches can be found. SMS uses the same model as Tivoli but doesn't provide a base infrastructure or staging semantics. Marimba and Radia use a channel, subscriber based model that is further introduced in the following section. Beside this difference the server and client infrastructure can be compared to SMS but has no hierarchical structure. Rational Content Studio uses a complete Web-server based model for the content delivery. Rational ClearCase uses a distributed Version Management model with a central repository server that builds the interface to Content Studio Web-servers. CM Synergy uses a repository-based client/server infrastructure similar to ClearCase for Configuration Management. Perforce uses a client/server infrastructure partitioned for related development groups. Webstart and NSBD use HTTP protocol extensions and a Web-server/Web-browser environment. InstallShield just uses tools that are executed on a client or distribution server and client installation routines.

Summarized, many different physical models for a distribution environment exist. Depending on the focal point of the software server infrastructure based, client/server based, repository-based, Web-server based and agent-based models are implemented. The products' models serve their purpose well, but require the implementation of a completely dependent infrastructure.

4.6 Code and Content transmission

Beside the described components the code and content transmission models are also quite different among the products studied in this thesis. Sequence diagrams show the interaction between the components, the used protocols, and the application mechanisms to achieve Software Distribution. The first model described is the Marimba Castanet channel-based approach. Figure 19 describes the communication between the Deployment manager, the transmitter and the tuner (client).

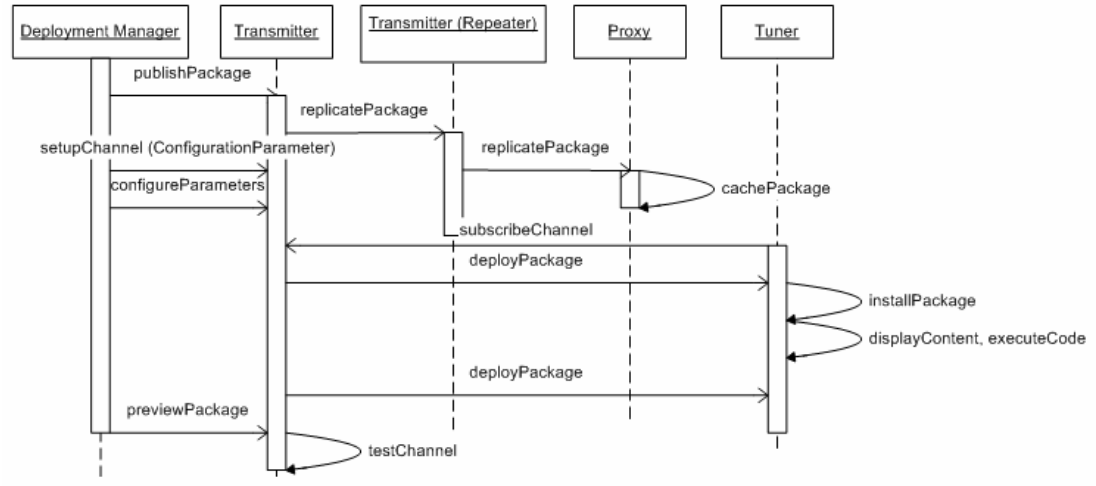

Figure 19: Marimba transmission

The Deployment Manager is used to administrate the package creation and the publishing on a transmitter. After package hosting one or more channels can be set up for a package distinguishing distribution parameters and target groups. The packages are replicated

between intermediate Transmitter. Proxies (special purpose transmitter) are used along the path to cache packages. The client uses the tuner component to subscribe channels. Depending on the deployment root of the channel, packages are deployed to the client from one or more locations. The client installs the package and executes or displays its content. Additional packages can be deployed without a client request through a push distribution. Additionally a channel can be setup and tested on a transmitter before activation.

The second model shown is the Web-server-based model of Webstart. Figure 20 shows the transmission of components using the JNLP based Java application description.

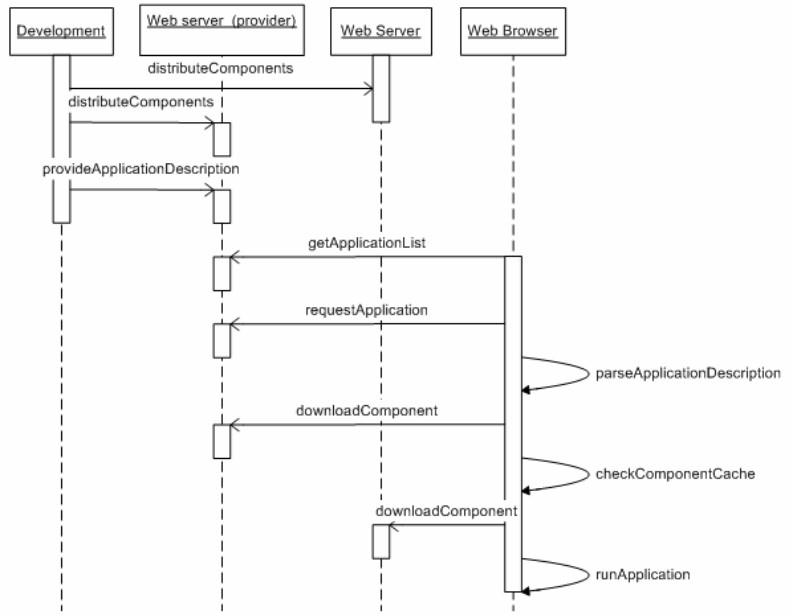

Figure 20: Webstart transmission

Components are distributed out of the development to several Web-servers. Additionally a JNLP application description is provided on one of the Web-servers. The Client enlists the applications available on the description-hosting Server. The user selects an application and the Webstart client downloads and parses the application description. Before executing the application the client recognizes cached components and downloads new or missing components from the Web-servers. The transmission of code on application activation restricts the environment to Java applications. These applications can be organized in several JAR archives that are extracted, classes that can be executed and cached and Java runtime classes. For components that rely on operating system services like COM or have a more complex registration and activation infrastructure like CORBA would also demand a more complex model for just-in-time installation.

The Tivoli site-based package distribution is shown as the third model. It consists of a framework server infrastructure, additional Web-servers for distribution information and installation clients as shown in Figure 21.

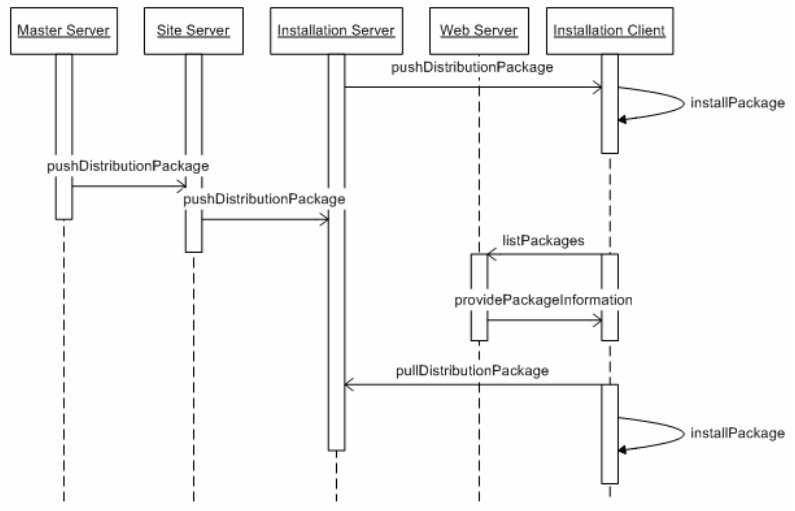

Figure 21: Tivoli transmission

The transmission model uses push communication along a distribution path. Packages are transferred within the framework between a master, several site and installation servers. The installation server is the client access point and provides push installation for clients. The additional Web-servers provide package and installation server location information. Clients can list packages, connect to the installation server and perform a pull distribution and installation. The pull installation is primarily used for partially disconnected mobile clients.

Based on this description the products can be related as follows: The SMS and NETDeploy transmission model can be compared to the Tivoli's push/pull transmission of packages. Novadigm Radia uses pull distribution and a publisher/subscriber model comparable to the Marimba approach. Additionally it allows intermediate publishing servers to adapt and personalize package content. NSBD uses a model similar to the Webstart model but it doesn't provide just-in-time installation and is not restricted to Java applications. InstallShield doesn't define the type of distribution, it just provides installable packages. Product extensions make these packages available on Web-servers. Content Studio itself hosts applications and provides content using Web-servers. ClearCase and CM Synergy as Version Management based systems use repositories and a distributed infrastructure to replicate products on a component version level. Perforce uses a client/server communication for development groups.

Throughout all models a multi-server staging environment is considered in communication whether consisting of passive Web-servers or active framework servers. The environment models of Tivoli and Marimba provide push and pull distribution with a channel-based and a site-based approach. Webstart provides just-in-time installation using pull distribution. Further all models share the definition of a client access point. Web-servers are commonly used for information retrieval. Active distribution servers implementing infrastructure services or passive Web-servers are used as access points for software installation.

4.7 IT infrastructure integration

The interfaces to other systems are described in this section as sequence diagrams. With this notation the high-level communication between the systems are shown and scenarios for a concrete interface implementation can be derived. The Tivoli integration Toolkit, the SMS Active Directory Service interface and the InstallShield Installer interfaces are covered. Further interfaces that are not considered here include Content Management, change management and e-commerce.

With the experience of many integration projects Tivoli's Distribution environment has been extended with the Integration Toolkit. This toolkit provides access to the functionality of the distribution classes [56]. Figure 22 shows the interaction with a fictitious distribution system.

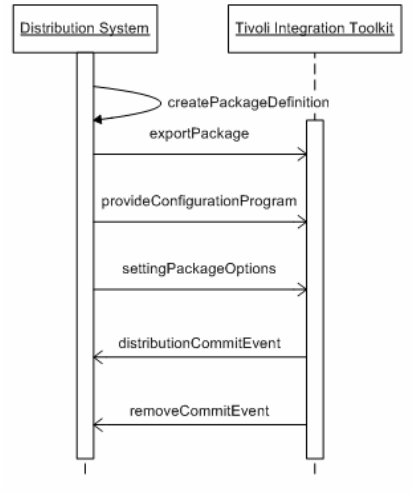

Figure 22: Tivoli integration toolkit

The distribution system creates a package definition that satisfies the Tivoli package specification (Figure 22), instead of using the internal package editor. The packages are then imported into Tivoli using the Toolkit. Package configuration information can be provided for the Tivoli installer program or a configuration program can be provided that is used to install the application. Package options are set and distribution activities are defined for the Tivoli activity planner. Tivoli's transactional distribution process provides all major transaction events through a toolkit interface. Events can be used to synchronize with the distribution process or to centralize state information. Rollbacks and distribution error events can be used to define new activities.

SMS has always relied on the Windows NT user and security services. With Windows 2000 an Active Directory-based management has been integration [91]. The communication as shown in Figure 23 is therefore a sample of a directory service integration specific to the Windows environment.

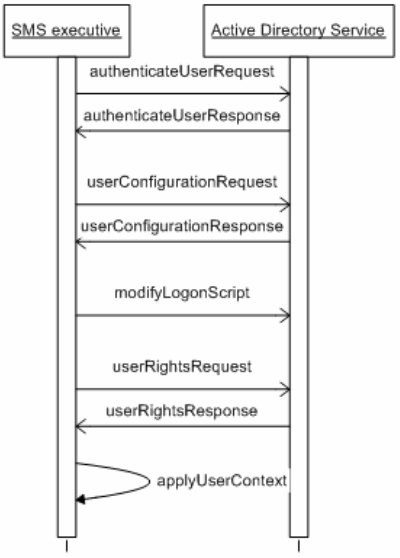

Figure 23: SMS Directory Service interface

The directory interface is used to authenticate Windows user accounts that are used on behalf of SMS for distribution and installation tasks. Additionally user accounts and user groups can be structured to represent organizational information. Logon scripts that initiate software installation are stored by the operating system but can be applied to users using the directory interface. User rights for installing products are evaluated and contexts of authenticated users can be applied to distribution processes.

Next the InstallShield Developer integration interface is described. InstallShield provides an Install Script interface to customize installation scripts and a COM interface to manage projects and create InstallShield packages. Figure 24 shows sample functionality to integrate with a Software Distribution system.

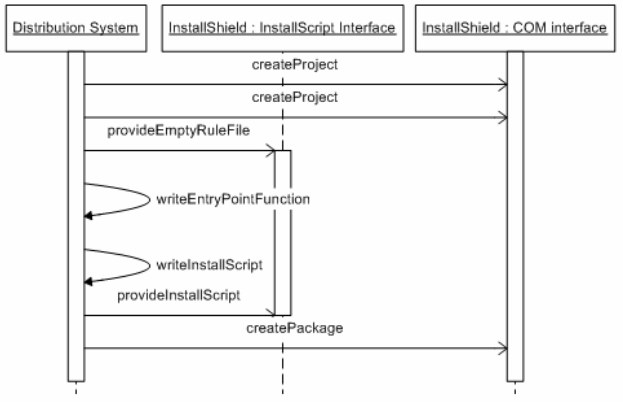

Figure 24: InstallShield interface

The COM interface can be used to access the component model of InstallShield Developer. A distribution system extends its installation functionality by automating the project management and package creation tasks. Projects can be versioned and files can be added and removed. For package creation different installers supported by InstallShield can be used. The scripting interface allows to customize the installation process. First a rule template is provided that can be extended in InstallShield. An entry point function for the installation process is implemented and specified in the script. The script can be extended to register components, specify version checks, and reboots. The script is then imported into InstallShield and executed with the client installer.

In addition to the described integrations all other products provide interfaces to customize their functionality. Some products like Marimba, SMS or NETDeploy rely on open software description standards. Others like Novadigm Radia, NSBD or Webstart provide a proprietary but open interface. InstallShield provides a common API and a scripting model. The Rational ClearCase and Content Studio integration is an example for an implemented version and distribution management integration that is not documented and therefore not customizable or reusable (see 4.2 for a detailed description). Perforce and Tivoli additionally provide all server functionality through a command line interface. Perforce also provides the client functionality this way and Tivoli additionally uses event notifications to synchronize state between systems. CM Synergy includes an interface to perform version and change management tasks on the central repository.

The architectural styles and models of the products show conceptual similarities and differences, functional focal points and integration techniques. Divergent models of the most important abstractions lead to completely different products, even if they share repository-based storages, push and pull distribution mechanisms and site- or agent-based component decompositions.

5 Reference architectures for Software Distribution Environments

In this section reference architectures are elaborated, based on the requirements and functionalities presented throughout chapter 0 and 0. The case studies introduced in 1.3 are discussed, conclusions for a reference architecture are presented leading to three different architectural models for Software Distribution.

5.1 Case Studies

Requirements and functionalities specific to the case studies are discussed in detail and the conclusions for an architecture are considered throughout the following sections. Sequence diagrams show the main processes that have to be implemented in the domain of Configuration Management and Software Distribution. It is further shown how the introduced products fit to the scenario and how they can be implemented.

5.1.1 Case study "Workflow"

case study "Workflow" covers the configuration and distribution of a workflow application that is implemented using the MQ series workflow client and server components. Key issues have been introduced in 1.3.1. The system provides a hierarchical model. On top, the domain concept separates multiple physical sites but shares a common workflow model. Synchronization is provided by the MQ series queue management. The next level consists of system groups that refer to a single location where one or more workflow systems are implemented. Systems grouped together use the same database and their communication is optimized. A single system consists of a one, two or three-tier architecture with a client and Web-server tier, a workflow server tier and a database server tier. Clients implement the workflow client API and can be Web-browser or Java-based applications using HTTP or RMI [92] over IIOP [93]. Their communication can be consolidated by client concentrators that forward their requests. Depending on the protocol a Web-server with servlets or the EJB-based WebSphere [94] application server is used. The workflow tier has several functionalities that can be distributed over multiple servers. The database can be run in a cluster to provide load-balancing [95].

The model described here covers the distribution of a MQ series system and takes into account multiple locations, multi-tier systems, a distributed server architecture, a large number of clients and the use of client concentrators. The Configuration Management tasks are mainly covered by the workflow administrator and can be substituted or extended with an own implementation, however the model fits both scenarios. One benefit is the homogenous structure of distributing a single system, on the other hand the heterogeneous operating systems infrastructure requires advanced setup techniques. The workflow system provides setups for all server and client components.

Below Figure 25 covers the key communications of the components necessary to distribute a MQ series workflow system. For distribution, information about a specific workflow system architecture has to be stored and maintained. A configuration database has to cover all installation-related domain, system group, system and node information. Clients in this case study are likely to be catalogued with an inventory system. Therefore an inventory database or a directory service is part of the configuration. A mechanism has to be provided that supports transfer of installation packages by implementing parts of the workflow API and querying the additional information. There exists an order dependency for installing workflow components. The server infrastructure is built before the client. The hierarchy has to be set up from a domain down to a system. An initial queue manager has to be installed for communication between components [96].

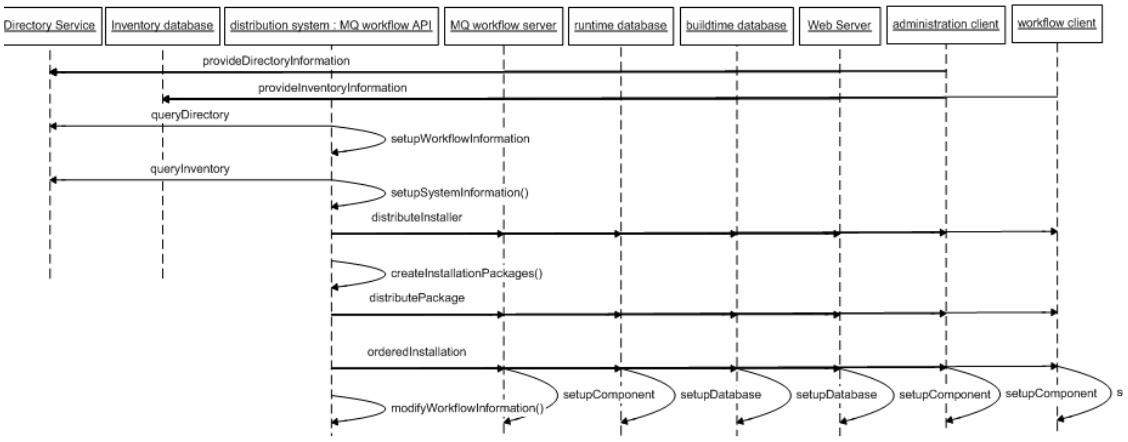

Figure 25: Case study "Workflow" processes

Next, the installation process, also shown in Figure 25, is described. All hosts except the workflow clients provide information to the directory service. Key information provided here covers system name, version and location, user and user group constellations and security policies. Additionally the installation process can be supported by storing which MQ series server components and database systems have to be installed. An inventory like the Tivoli inventory module (see 3.3.6) collects information about all hosts of a MQ series workflow system. An inventory client provides system information that can be used during package creation for version selection and installation customization (see 3.2.4). The information collection process is done independently of the installation process. The distribution system initially stores dependencies between workflow components to enable ordered installation activities. Next, it queries the directory service to setup workflow information. During this operation a plan of installation activities is created considering the component dependencies. Each activity is related to a target host and a MQ series workflow service to be installed. Furthermore, the inventory system is queried to select the product versions that fit the operating system and language requirements. Additionally the inventory provides information about already setup installer services and workflow components. Based on the product and system description the workflow information is setup, the activities are specified to whether install or upgrade installer services and workflow components and the packages that should be used are defined. From this point the distribution system is enabled to setup the installer components on the target systems. For a more complex distribution system like Tivoli additional distribution services have to be provided. Therefore a customized installation procedure and additional target systems have to be considered. This setup proceeds until all hosts are enabled to receive installer packages. During this process all required installation packages are created by importing or creating them with an external packager. After the distribution infrastructure is setup and all packages are ready for installation, the distribution activities from the installation plan are executed. First, except for deletions, the packages are distributed. An internal mechanism can be used or it can be delegated to a more or less reliable transport mechanism. Retransmission and resume operations have to be considered if an unreliable transport is used. The state of the distribution has to be maintained for all activities to start the installation activities themselves. Depending on the distribution system, the system or the user decides whether to install, upgrade or reinstall the package. For example the Tivoli integration toolkit can be used to distribute components and to maintain state of the installation process (see 4.7). A manual installation is not recommended for workflow components. The activities are executed in order based on the initial dependency information and the workflow system is finally distributed consistently.

From our products described in 3.3 the Tivoli distribution and Inventorying system meets a lot of requirements for the complex distribution tasks. SMS, Marimba and others shouldn't be taken into account as long as they are not implemented for other purposes. InstallShield

can be used as an external packager and all version control systems can be customized to provide the required information.

5.1.2 Case study "Virus Scan"

The case study "Virus-scan" covers the distribution of a virus scanner from the fictitious ImmuneWare company. Multiple platform and language versions are provided for a large client base through multiple Web-servers. A list feature should provide product information to clients for different versions which are active in parallel. The software's client components have to be registered on the target system and installation information is gathered. An additional billing and licensing mechanism is provided, where product's can be registered during installation. From the distribution standpoint the system has to provide pull operations, where the client decides when and how to install a product. The clients are anonymous and no inventory can be used. Therefore all relevant client information to select the right product version has to be provided through user interaction.

First a web and distribution server is required. The distribution functionality can be separated and the system can use clustered servers. The Web-clients on top of the operating system use the distribution system by executing operations on the Web-server. Additionally systems for versioning, licensing and payment are integrated.

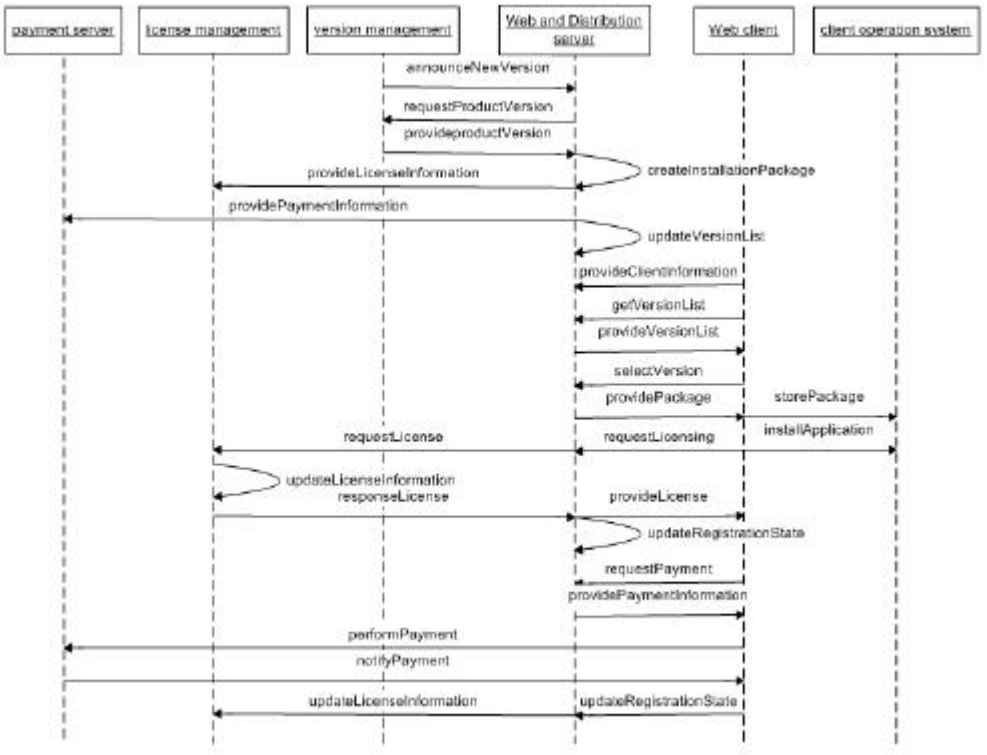

Figure 26: Case study "Virus Scan" processes

The Figure above shows the processes of this case study. A Version Management system announces new application versions to the distribution system that is hosted on a web and application server. The distribution system requests all necessary components of the product. Through these steps the Version Management strategy of selecting the right components is used. After all components have been requested and cached by the distribution system, it starts a packaging application through automatic or manual invocation. The packager requires platform and language information to customize installation procedures and parameters required during setup. License and payment information is added to the package, which format has to support such extensions. General payment and license registration information is also provided to the corresponding services. Then the list of versions provided through the Web-server is updated and available for clients immediately. This configuration

part of the process occurs independently and asynchronous to the installation phase. It is beneficial to separate the processes for these phases in the distribution executive.

In a further step end-users on the clients request a list of available product versions from the Web-server that consults the distribution system for a list of already packaged products. Therefore the distribution system has to store package state information throughout the lifecycle. The client selects a version for downloading and the server provides the package through FTP or HTTP. Afterwards the package is decompressed and the installation is started. The installer verifies the components and starts component registration, modifies system parameters (registry settings, environment variables), and performs optional system reboots. The download mechanism should be made reliable through retransmission and resume features and package integrity checks on the client. By decoupling the registration of the product from downloading, it is not necessary to store state information for the download process. This reduces load for a large number of client connections.

During application installation and depending on the installer implementation optional licensing and payment processes are performed. First the client requests a license and the distribution system contacts the license management. The license management creates a license item with customer information and a state of issued or pending, if further payment takes place. The license can also be an update of an existing item if the customer has already registered an earlier version of the product. The distribution system gets a response, provides the information to the client and updates its registration state. Further a payment mechanism can be invocated by the installer. The client requests payment information and the distribution system provides prices and the type of payment for the product. The information exchanged between the client and the payment service provider depends on the implementation, but is most likely credit card based (e.g. SUN iPlanet [97]). For simplicity reason, the payment system uses the same Web-server interface to provide and request additional information from the client. Finally the payment server provides payment information to the client, which updates the payment state information of the distribution system. The distribution system contacts the license management to update the license state. The license and payment model is limited. For example timeframe based use of applications or automatic update functionality cannot be easily incorporated. The server should not store any state information. The client installer can be uninstalled and the application might just use proprietary internet-based mechanisms.

From the described products Webstart for Java applications is an interesting alternative to download- and installer-based mechanisms. A Version Management system can be chosen that best fits into the infrastructure. The server can be based on WebSphere, .NET, or other component-based infrastructures.

5.1.3 Case study "Custom Web-service"

The case study "Custom Web-service" consists of Web-service-based applications that are part of an Internet portal. These services are used by server processes at the customers site and by Web-clients used by the customer and other portal visitors. The customer also uses a client/server application that is not considered here.

The Configuration Management tasks consist of the selection of the correct Web-service component version by incorporating a Version Management system. Further the customer database has to be instantiated and configured. A database version description file can be stored separately in the Version Management system or a specific product component generates the database and specifies the configuration options. The distribution tasks consist of the installation and registration of Web-service components. Additionally consistent code and content changes have to be managed for the Internet portal; a Content Management has to be integrated. Additionally the activation of the components and the changes to the customer configuration have to be synchronized and the customers' database has to be setup. A (de-)activation and uninstall of components and databases has to be supported. For these tasks a distribution system is integrated with a Web-server for the portal and an application server for the Web-based components. Further, the customer database and application

servers are part of the infrastructure and a Version Management system interface is implemented for the configuration related tasks.

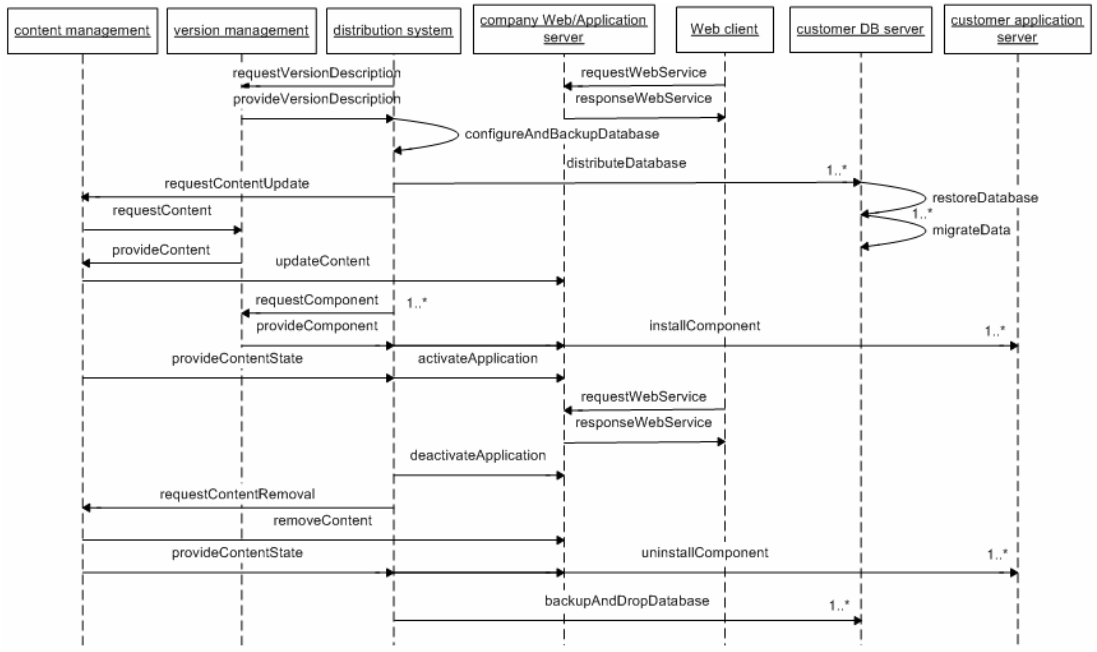

Figure 27: Case study "Custom Web-service" processes

The processes shown in Figure 27 can be described as follows: First the distribution system requests a version description for the customer database. The description covers the database scheme and the configurable application options. In conformance to this information the database is configured and customized. A database backup file is generated and distributed to the database server. The database is restored and optional configurations can be done on-site. By executing scripts against the database from the distribution system site, state for the distribution task can be maintained. Then a content update request is performed corresponding to the distributed database version. The Content Management system can use its own or incorporate the existing version control through an additional interface. The content is then updated on the Web-server and changes to the content state are provided to the distribution system. The distribution system contacts the Version Management for all corresponding components asynchronously. The Web-based components can be installed without any specific distribution system or external packager and installer application, reducing the expense of a distribution infrastructure in comparison to case study "Workflow". EJB or .NET components are registered by procedures of the application server that can be automated by the distribution system implementing a remote interface. The corresponding mechanisms are described in [10] and [12]. State information can be used for synchronization.

By activating the application new versions of services become available. Through the use of different entry pages provided by the Content Management clients can be redirected to the site implementing the new services. In a staged distribution model, a restricted area can be given access first to test for consistency. Next the site is made available to all clients. For Web-based services the registration information defined with the WSDL [98] protocol has to be updated. In a test environment a different registration has to be used temporarily. An activation and deactivation procedure has to implement an interface to registration functions provided for example with the Microsoft SOAP toolkit [99]. After activation in the production environment a period of time has to be considered to allow all clients to migrate to the new services. Afterwards a deactivation of the old application is performed and a content removal request is sent to the Content Management. The content is removed from the Web-server and content state information is provided for the distribution system. For the application server components an uninstall procedure is executed by implementing the described interface. Finally a database backup is performed and the database is dropped.

The customer related install und uninstall procedures are repeated for several sites depending on the participation in specific Web-services. Throughout the process clients asynchronously request services from the Web-server. As long as an application is activated its services can be requested by clients. The active periods have to overlap to provide optimal service availability.

The distribution part can be implemented with an infrastructure like Novadigm or NETDeploy. Also Tivoli, SMS or Marimba can be used but might not be cost effective without other synergies. Content Studio might be used for Content Management and ClearCase for Version Management.

5.1.4 Comparison

To compare the scenarios key distinguishing factors are taken from different sections of the product classification introduced in 3.4. Detailed information about the factors are given in 3.2. This section covers the functionality required by the Case Studies in terms of these factors. The differences in functionality are visualized using net diagrams. The stronger requirements for a factor are, the longer is the corresponding vector. The larger the created area, the higher are the overall requirements with respect to the chosen factors.

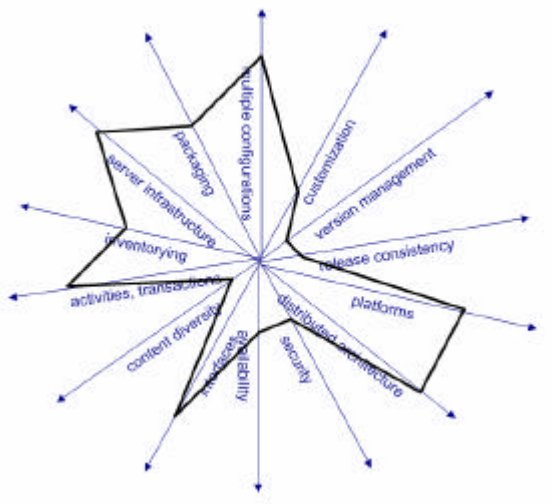

Figure 28: Case study "Workflow" key factors

The case study "Workflow" covers the distribution of a distributed system and therefore uses a complex software abstraction. This results in increased requirements for platform independence, a distributed architecture, a transaction based distribution process and a server infrastructure like Tivoli [3.3.6] for the distribution system itself. Additionally interfaces to directory and inventory systems are necessary, where the Inventorying itself provides key information for the distribution process. The software components consist of installation requiring applications, therefore packaging support is important. Because the case study covers MQ series workflow, multiple configurations for different workflow implementations have to be stored for the whole workflow lifecycle. Less relevant is customization, which is done within the workflow system once it is distributed. Version management and release consistency are not considered primarily important, because of relatively seldom changes to the base system. Security related issues are delegated to the underlying network infrastructure, which has to be already invested for a workflow system. The content diversity is minimal because the distribution system just has to deal with installation packages.

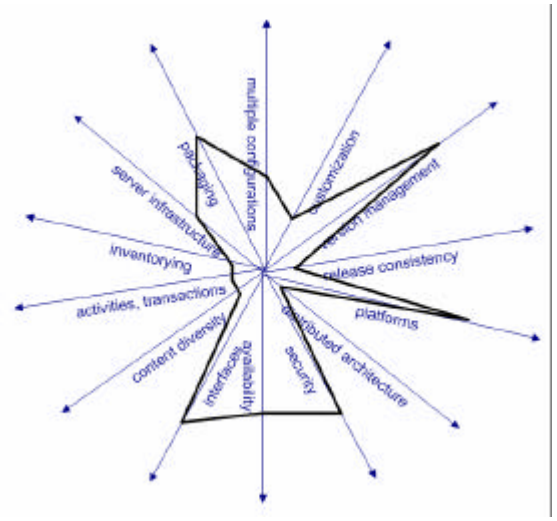

Figure 29: Case study "Virus-Scan" key factors

The case study "Virus-scan" has a very different requirements profile. Figure 29 shows the corresponding net diagram. Factors that are very important in the context of distributing standard software to a large user base are platform support, Version Management, packaging, security and availability. Platforms can be diverse for a anti-virus software. Version management is necessary because the product can be distributed in different versions in parallel and frequent changes are necessary due to newly detected viruses. Packages are necessary as the currently common distribution solution for standard software. Security is a major factor for the license-based activation and the transmission of customer related information. The client/Web-server communication has to be SSL secured for this purpose. Of course, the external payment mechanism has to be considered. Availability is a typical requirement for a Web-based Software Distribution to anonymous customers. The interface requirements are more specific to the scenario of license and payment integration. This is normally a minor issue in comparable systems. Further multiple configurations can be necessary depending on the target system and distinct functionality in the offered product editions. Less relevant for the case study are content diversity, which is restricted to packages, and transaction and activity-based distributions, which is overscaled for single package deliveries. The server infrastructure just consists of several Web-servers providing content and packages for clients. Customization is not relevant except for individual license schemes based on customer information. An Inventorying is not possible because of the customers' anonymity and a distributed architecture is neither necessary nor existing.

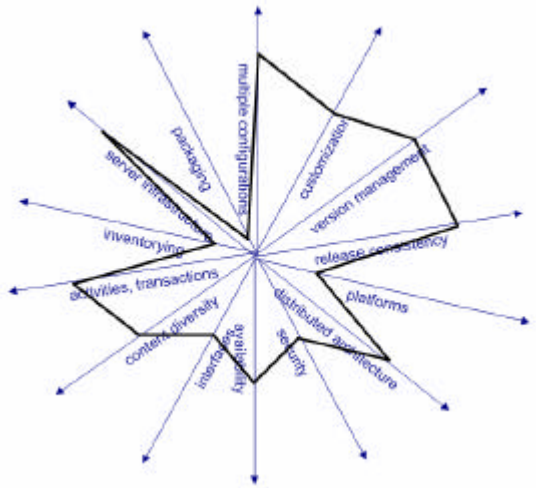

Figure 30: Case study "Custom Web-service" key factors

The case study "Custom Web-service" has again a very different requirements profile. The corresponding net diagram is shown in Figure 30. Compared to case study "Workflow" the

factors server infrastructure, distributed architecture, activity, and transaction based distribution as well as the support of multiple configurations are important for the same reason as above. The configurations are even more complex because the database configuration and setup is not separated from the distribution process. Because the company itself develops the application, parts of the configuration are highly customized. More frequent software changes due to a high responsibility for customer requirements require an integrated Version Management. Specific to the scenario is the Content Management interface. The database, the customized products, and the content integration extend the content diversity. Two aspects influence the availability requirement. First the installed products also work without the distribution system, but during staging of a product it is necessary to provide continuous availability for the clients of the Web-services. Security is already implemented for customer sites to enable remote administration of the products. This part is additionally sensitive because of the exchange of patients' medical data. The Web-services have to be secured to prevent misuse by non-authorized software. Platform support is less relevant because of Microsoft based operating systems, database systems, and Web-servers.

5.1.5 Categorization

After the comparison this section categorizes the functionality in the case studies and discusses similarities and differences based on the description starting with 5.1. Three major categories of functionalities are further differentiated. A category of necessary functionality that is used in all case studies even with minor differences. Another category of necessary functionality which is specific for the scenario is described. A third category consists of optional functionality that might not exist in scenarios that are comparable to the Case Studies. Additionally other commonalities based on major abstractions, used distribution components and implemented processes as described from 4.1 to 4.5 are discussed.

Common functionality:

First all three scenarios utilize a Version Management mechanism. The "Virus-scan" and "Custom Web-service" case studies require versioning on different levels. case study "Virus Scan" requests versions of products to create packages, while case study "Custom Web-service" requires a software on a component level and additionally requires the management of different database scheme versions. The case study "Custom Web-service" has no versioning requirements, nevertheless such functionality can be seen common for Configuration Management related tasks. A further common functionality is the integration of a packaging facility. The "Workflow" and "Virus-scan" case studies both require such integration on a product level for workflow setups or setups for the anti-virus software. The case study "Custom Web-service" directly distributes components and content without any packaging. Furthermore a distribution mechanism is necessary in all three scenarios. This mechanism is different in all scenarios. The case study "Workflow" requires a complete distribution infrastructure to setup the workflow system, the second doesn't require a specific infrastructure and uses the Web-servers HTTP or FTP as the distribution protocol, The third study uses a direct remote installation of components. The network infrastructure itself is extended with more reliable and fault tolerant connections. A further abstraction used in all Case Studies are the target abstraction, that consists of inventory management in the first case, an anonymous user base in the second and well-known server systems with a homogenous hardware and operating system base. Common to all scenarios is the use of distributed, web- and application server incorporation, infrastructure on which the different distribution techniques are built upon. Therefore common functionality related to site and server descriptions, user management, and system status tracking can be extracted.

Specific functionality:

First the distribution strategy differs. Server-side functionality is supported more effectively with push semantics (Study "Workflow" and "Web-service"), client-side functionality (Study "Virus-scan") can use push and pull mechanisms. Additionally the MQ series workflow application's client components are part of a client/server application, where push semantics

should be used for consistency. Further different distribution processes are implemented. case study "Workflow" and "Virus-Scan" use packager applications to distribute content, case study "Custom Web-service" directly installs components, distributes a database scheme or backup file and stages and synchronizes content. Therefore the software abstraction (see 4.2) differs and the software description language has to be extended. The "Workflow" and "Virus-scan" studies support languages with the used packager application, the "Web-service" study uses the internal descriptions of components, content, and database schemes. The components are self-describing through their interface definition that is used by the application servers. The Content Management uses meta-data to store artifact descriptions. The distribution system has to integrate these internal mechanisms to perform installations. case study "Workflow" and "Custom Web-service" have to order installation activities and execute them reliably. The "Workflow" scenario uses transaction-based distribution, the third implements distribution state management, staging and an activation/deactivation mechanism on an application level. Study "Web-service" could also implement transaction based distribution, but a Content Management interface has to be considered for synchronization for example based on the Tivoli integration toolkit (see 4.7). Both cases distribute a distributed system. Additionally the implemented transport mechanism differs depending on the type of distribution. A reliable transport with resume and retransmission features is used for transaction based systems. The case study "Custom Web-service" doesn't implement such semantics and has to ensure consistency through the staging process. case study "Virus-scan" requires high availability of the download servers and supports reliable transport only through HTTP retransmission [100] features.

Finally functionality related to integrated environments (see 4.7) is compared. Even if a target abstraction can be found in all scenarios the utilization of an inventory management system is specific to case study "Workflow" because of the well-known environment where all involved hosts can be integrated. Anonymous Web-based environments have to be satisfied with less reliable target information. Browser version and operating system can be gathered from the HTTP header, Java VM and other plug-in support have to be selected manually or evaluated with proprietary mechanisms. Another specific functionality is the integration of Content Management. Because of the increased costs and complexity of the infrastructure such systems are currently restricted to highly dynamic and integrated portals. The dependencies for a distribution system are minimal and synchronizing a staging process can be realized with relatively simple communication semantics. Use in less specialized environments will increase through further development of such systems.

Optional functionality:

Besides the core domains of version, configuration and distribution management, optional functionality covers related domains. The directory service provides environmental information that is more or less necessary for the described scenarios. For distributed scenarios (Study "Workflow" and "Web-service") such information can be integrated to reduce expense of changes to the environment, increase data consistency and reduce manual interaction. In homogenous environments services of the operating system can be used. In simple Web-based environments most required data is gathered manually during download or installation. Licensing and Payment can be implemented optionally. Licensing can be incorporated into all systems. The license models differ between the scenarios. Site licenses fit for server functionality and license pools for client/server applications. A single user license is the simplest mechanism needed in anonymous environments and therefore found in systems comparable to case study "Virus Scan". Payment functionality is not part of the distribution process, but beneficial for scenarios that support a large user base. In case study "Virus Scan" cost reduction and effective user support can be reached. The private environment of the first study won't benefit from such efforts because of the limited number of distribution activities. For case study "Custom Web-service" the customer relation can be compared to Study 1. For Web-services currently no model is implemented; a possible revival for micro-payment systems. Also change management and content Configuration Management have been introduced in 1.2.4. Further development of Web-service and

content-based information systems will change the model of case study "Custom Web-service" and represent distribution and Content Management in a single platform.

Other commonalities and differences:

Beside functional requirements the distribution components in these environments can be compared. A distribution system is necessary in all models. case study "Workflow" requires this system in different locations, case study "Virus Scan" and 3 can be restricted to a single optionally clustered system. Web-clients have to be supported. The Web-server is used as a gateway for the provided distribution functionality. In case study "Workflow" the Web-server is only used as a Gateway during runtime. Additionally an application server is necessary to implement Web-service-based distribution models like in case study "Virus Scan" or to run Web-service-based applications like in case study "Custom Web-service" or client/server based functionality like in case study "Workflow". The Version Management in the last two scenarios is a separated system with tight interactions during the distribution process. Even different artifacts have to be stored and related operations have to be supported. The optional functionality of directory services, Inventorying, payment and licensing is separated in distinct systems. The architecture can therefore be extended independently. Also Content Management is separated for very specific scenarios. In all scenarios installation and distribution activities depend on the operating system platform. Further administrative clients for customizing the distribution process implement an interface to the distribution system.

The main difference is based upon the implemented configuration and distribution process. In case study "Workflow" the distribution is ordered, transaction-based, and package-based. The distribution system has to maintain a configuration model of a workflow system. The Configuration Management of the software is mainly part of the workflow system itself. The CM tasks in case study "Virus Scan" cover package customization for operating system and language support. The distribution process consists of hosting self-installing packages on a Web-server infrastructure. case study "Custom Web-service" has an own configuration task for the customer database and also manages software on a component level during the distribution process. The content distribution is external and synchronized.

The code and content transmission techniques differ in push/pull semantics, content type, network environment and reliability and availability requirements. A single technique won't fit each purpose and result in additional expenses and costs. The IT infrastructure integration differs because of the optional functionalities implemented. An architecture should therefore include interfaces to the described systems independent of the provided functionality.

5.1.6 Conclusions

As the primary conclusion of the discussed scenarios it has to be stated that no single architecture can cover all the requirements. On the one hand the distribution models differ in functionality as discussed in 5.1.4. The type of products that can be distributed, the target group covered with the scenario and the infrastructure necessary to implement the model can hardly be abstracted with a single architecture. Further differences in the processes, used components and transmission techniques have been discussed in 5.1.5. On the other hand the major abstractions for software, target and customer, mechanisms used in the configuration and distribution process and supported features can be categorized as introduced from 4.1 throughout 4.4 and discussed in 5.1.5. In this context the different architectures remain comparable. Even if three case studies and three architectures are concluded, no one-to-one relationship can be established. A service-based-approach is used in contrast to agent-based systems or push-systems. Also channels as an abstraction for the distribution path for components, packages or other content are not considered further. A good introduction to push-based communication models is given in [101]. Additionally, Marimba provides detailed information about channel-based products [61].

5.2 Client/Server model architecture

All architectures in this thesis are described in UML using the "4+1 view model of software architecture" process suggested by Kruchten [102]. The process implementation has been supported by the language descriptions of Fowler [84] and the refinement recommendations of Larman [103]. As the first reference architecture a Client/Server model is introduced. This model is primarily intended to be used with scenarios comparable to case study "Workflow" but also has to be considered for models like in case study "Custom Web-service". The architecture also follows the changes to multi-tiered software as stated in 1.2.3. To point out differences, the architecture is set into relation to the other two models in the following sections. Further derivations for a generic model are considered.

5.2.1 Logical view

The logical view of the architectural model covers UML class diagrams and a description of class semantics, relationships and dependencies. Figure 31 shows the configuration related classes and the classes common to the whole process.

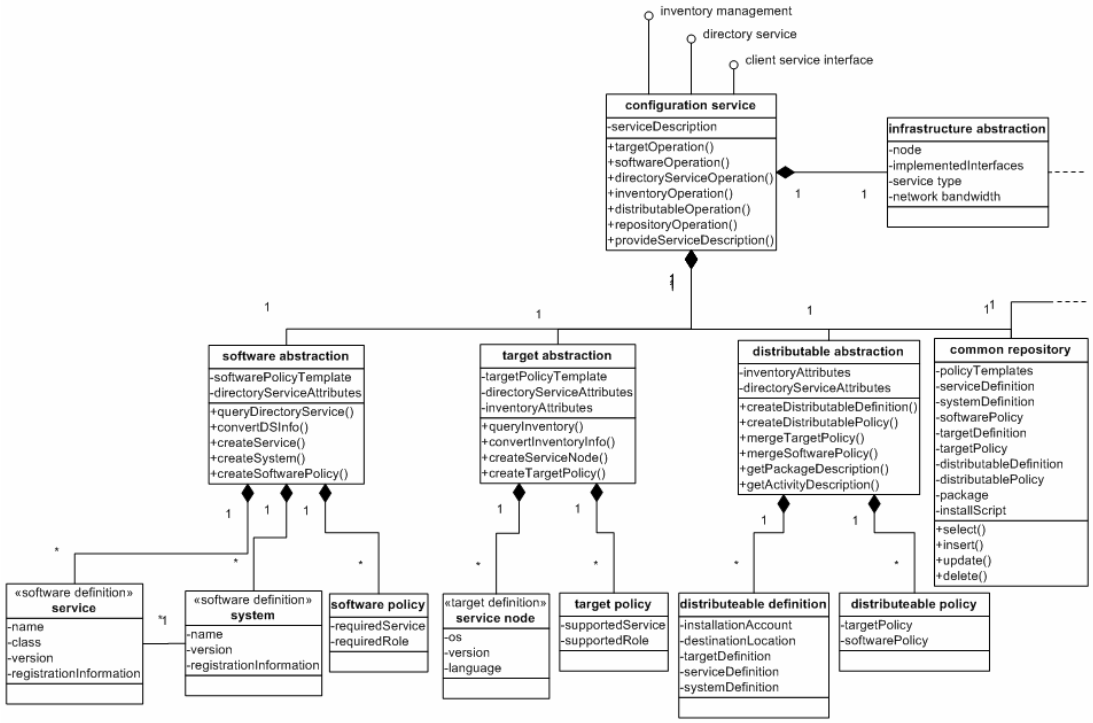

Figure 31: Client/Server model logical view - configuration

The model consists of a main class providing services to clients and external systems. It performs operations on its subclasses for the configuration process. The processes of configuration and distribution are kept separated but use a common repository to share and exchange data. This is reasonable because of the tight integration of distribution solutions for distributed systems like client/server products. Normally configuration and deployment of these products is done by the same company, possibly on-site. The repository could be separated to support physically separated processes, which is not necessary in this situation. In distinction the processes of configuration and distribution are kept separated because client/server products normally have very different configuration procedures but very similar distribution methods. The separation should therefore provide more flexibility in the implementation. Further software and target abstractions are used to model the real-world objects. They correspond to the requirements and definitions in 4.2 and 4.3 and also consider an integration of inventory and directory service interfaces. A customer abstraction as supposed in 4.4 is omitted because normally one system configuration has just one customer. Most requirements like type of service can be implemented in the software abstraction and others like licensing or payment do not apply. The software abstraction is modeled with a definition of a client/server system and constituting services. Both definitions get a name and

a version attribute and provide registration information for operating system integration. These attributes are examples of many more that might be useable like a service type that specifies different service functionality within a client/server system. The software and target policies contain descriptions of supported and required services and roles of the client/server system. These terms are example descriptions that can be further extended with interface or feature definitions. The require/support semantics explicitly describe behavior that can be compared during a negotiation of the software and target policy. The most important abstraction provided by the architecture is the distributable. It covers a single item of installation and a policy about when and how to install the item. The distributable definition is a compound of software and target attributes extended with attributes that exclusively define the installable item itself (for example installation destination and component registration activity). The policies and definitions of all abstractions are stored in the common repository. The infrastructure abstraction is used to provide information about the distribution system itself and covers physical nodes, implemented internal (clients), external interfaces (e.g. Inventorying), service type, and network parameters for system maintenance.

Figure 32 shows the logical view of classes related to the distribution process.

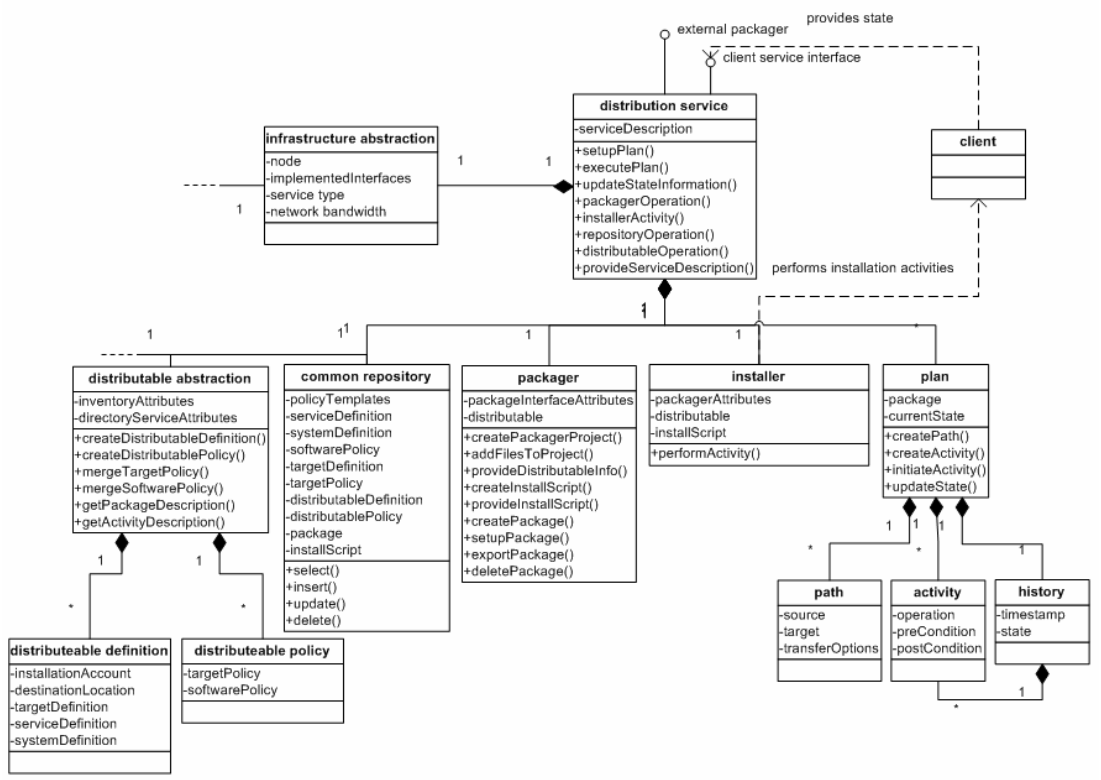

Figure 32: Client/Server model logical view - distribution

As with configuration classes, a main service class coordinates interactions and provides interfaces to other systems. The common repository is used to exchange the distributable policy and definition. The packager and the installer provide their well-known services based on a standardized interface that might have to be customized for specific products to fit. The packager creates packages from the distributable definition, the installer executes activities on the packages as defined in the distribution plan. The plan class provides the distribution process data, operations and an activity state engine. Its substructure consists of a path class that stores physical connections that can be used during distribution and relevant data transmission options. The activity class stores specific operations that are invoked by the installer during plan execution. In general, operations that should be considered cover the whole lifecycle of distributed software comparable to the description in 1.2.2. The installer activities can be customized with scripts or configuration files generated by an installer encapsulation class. Pre- and post conditions are used to decide whether a specific activity can be performed and has terminated accordingly. The history class is used to store state

information about performed activities. This state information is also used to update the path and activity class properties to customize related activities or change execution paths as shown in the following section. To benefit from history the packager interface has to support update, adapt and repair operations. The client component shown in the diagram is external to the class infrastructure but related to the installer and distribution service for activities and state management operations.

5.2.2 Process view

This view provides an insight to the dynamic behavior of the client/server model. Collaboration diagrams are given preference over sequence diagrams, because similarities in the class structure can be expressed more clearly. The focus of the process descriptions are operations that best describe the specific requirements in client/server products, more general parts are described in 5.5. First some configuration related class collaborations are shown.

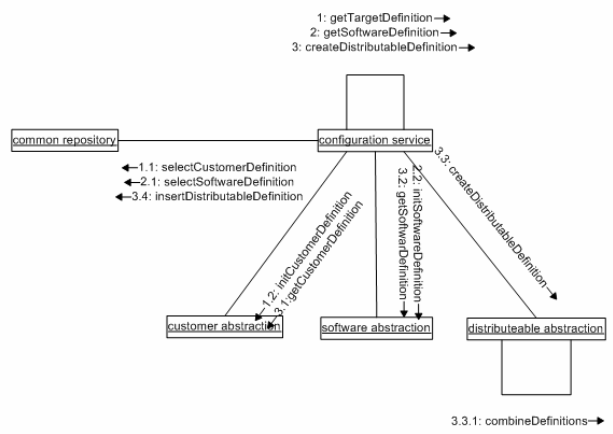

Figure 33: Client/Server model process view - configuration 1

Figure 33 shows the interaction between configuration classes to create a distributable definition from the target and software abstractions. The process has to initialize the definitions of the required target (1.1, 1.2) and software (2.1, 2.2) that are stored in the common repository. Then the target and software are selected (3.1, 3.2) and a definition is created for the distributable (3.3). The software and target attributes are incorporated into a compound attribute definition (3.3.1). Finally the distributable definition is also stored in the repository (3.4). The configuration service centrally invokes this operations but the process is guided by an administrator through the client service interface. A very similar yet more complex process is shown in Figure 34 for the policies that are contained in the abstractions.

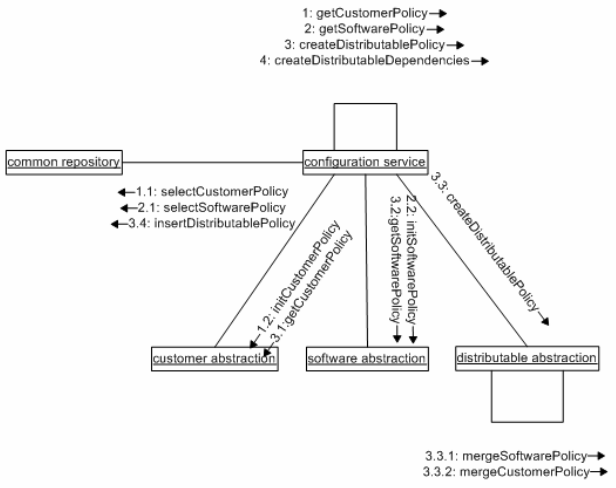

Figure 34: Client/Server model process view - configuration 2

First the policies of the target (1.1, 1.2) and the software (2.1, 2.2) abstractions are selected and initialized from the common repository. In the next step they are selected (3.1, 3.2) and a distributable policy is created (3.3). Then the merge operations for the policies are performed (3.3.1, 3.3.2). Because policies contain structured properties and rules the process is more complex and has to combine rules and properties. Additionally properties and rules of the distributable are added and new rules containing mixed properties can extend the distributable policy. A more detailed description about policy structures and content is provided in 5.5.4. At the end of the process the distributable policy is stored in the common repository (3.4). With this step the configuration process is finished and parts of the distribution process are described below.

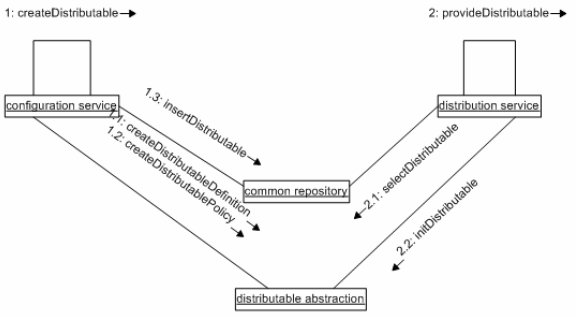

Figure 35: Client/Server model process view - shared data

Figure 35 shows the interaction of the configuration and distribution service during exchange of the distributable. The definition and policy handling (1.1, 1.2) has been described above. Instead of providing the distributable directly to the distribution process, the configuration service stores the definition in the common repository (1.3). This allows the two processes to perform their operations independent of each other. The distribution service is able to define plans for distributables as soon as they become available. There is no need for a complex application level communication protocol. The distribution service selects the distributable from the common repository (2.1) and the distributable abstraction is initialized (2.2). After the abstraction is set up the process can continue to define the distribution plan.

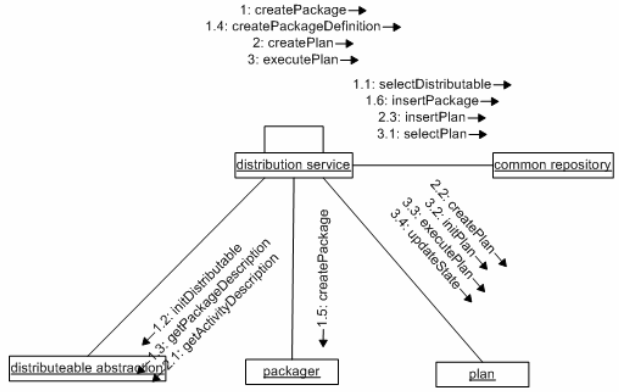

Figure 36: Client/Server model process view - distribution 1

Figure 36 shows the creation of packages and plans out of a distributable. First the distributable data stored in the repository is selected (1.1) and the distributable is initialized (1.2). In a further step, a package description is extracted (1.3). The distribution service is responsible for converting the description to a format readable for the packager (1.4) and creates a package by invoking methods of the packager interface (1.5). Finally the package is stored in the repository (1.6). In the second step of plan creation (2) the activities that can be performed on the distributable are extracted from the description (2.1) and provided during the creation step for a distribution plan (2.2). The setup of the plan also consists of ordering and selecting the required installation activities. After the plan is set up the execution step (3) can proceed by selecting (3.1), initializing (3.2) and executing the plan (3.3). During

execution state updates are provided that are used to update the current state of the execution plan (3.4).

Figure 37 provides a more detailed description of the distribution process.

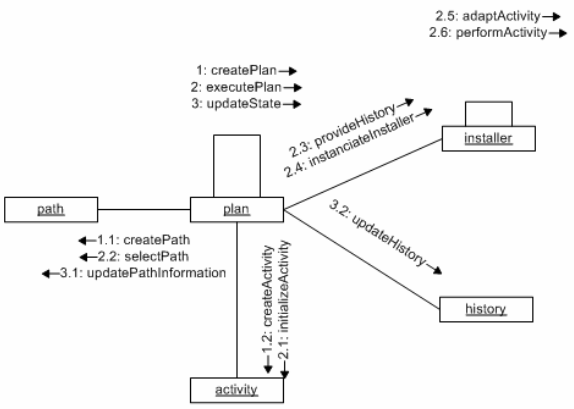

Figure 37: Client/Server model process view - distribution 2

The first phase of plan creation (1) is detailed in a path creation for destination information (1.1) and followed by an installation activity definition (1.2). The second phase of plan execution (2) consists of the initialization of a defined activity (2.1), a path selection for distributing the installer and the packages to be installed (2.2). Then the state history of the distribution plan is provided to the installer (2.3) and the installer in invoked (2.4). The installer adapts the activities based on the state information (2.5) and performs the activity on the target system (2.6). As a result of the second phase the state information of the performed activities is provided to the distribution service in the third phase (3) and influences the path information (3.1) and updates the plans activity history (3.2).

Most operations provided in the process view are customized to client/server environments. The following architectures show very different behavior in one place while other mechanisms work quite similar. A common approach for the processes is described in the generic architecture (5.5).

5.2.3 Deployment

The deployment view of an architectural model covers the physical system nodes and provides a description of how they are distributed and how they interact.

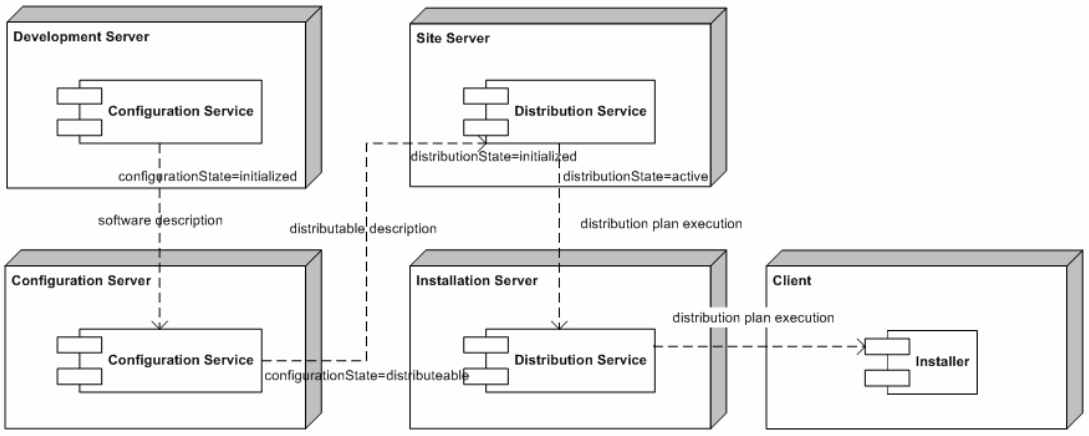

Figure 38: Client/Server model deployment view

A deployment diagram of this architecture is provided in Figure 38. The model is a simplification of the deployment view provided for the Tivoli deployment system (see 4.5), which fits very well for client/server systems. A development server is the starting point for

the software description exchanged during initialization of the configuration process. The configuration server hosts the configuration service and the common repository. The configuration is executed in this node. The common repository has to use a replication mechanism to provide the shared data to site servers where distribution takes place. The site server node normally corresponds to a customer's physical location. In large installations additional installation server are introduced to provide load balancing and fault tolerance of the process. A site server uses installation servers to provide distributables based on the path information in the distribution plan. The client as the last node is the target of the installer that performs the installation activities.

5.3 Standard software model architecture

The Standard software architecture covers models that can be compared to case study "Virus Scan" and are currently widely used. The architecture is driven by the current changes in the distribution of mass-applications using the Internet infrastructure. The term "Standard software" is used to describe the type of applications distributed that run on a client platform usually without any dependency to other systems.

5.3.1 Logical view

The logical view provides a model for the configuration and distribution classes of the architecture. Similar to the "Client/Server" architecture a service-based design is used and also other classes and their functionality are comparable. A process to build a more generic architecture out of the main building blocks is described in 5.5.

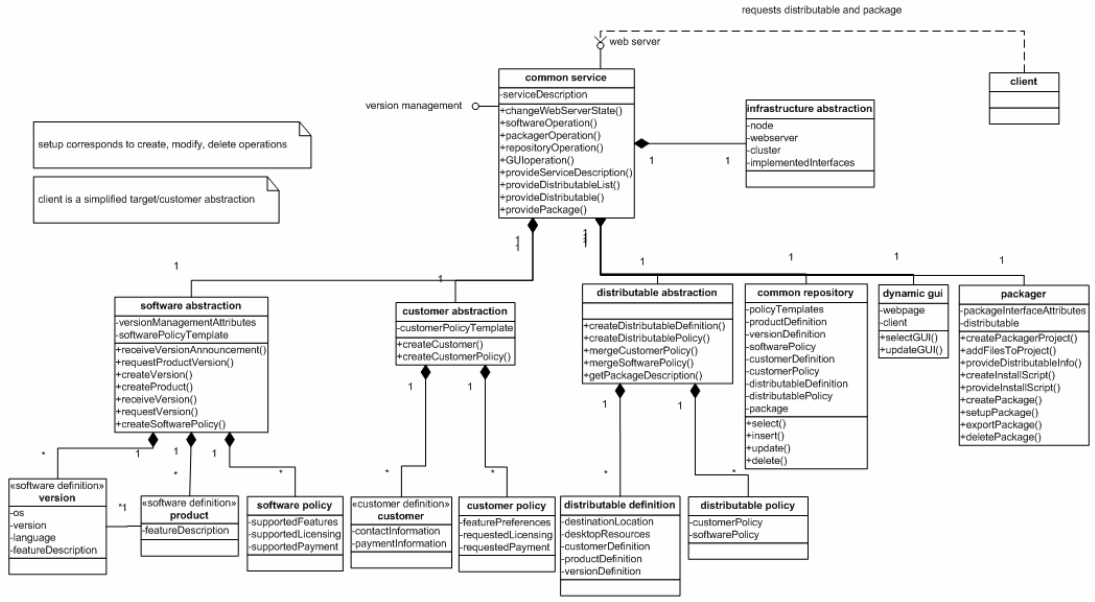

Figure 39: Standard software model logical view

Figure 39 provides a complete overview about all modeled classes. A common service, as the main class, implements all major tasks. This architecture is similar to the "Client/Server" architecture by providing interfaces to clients and external systems. In this architecture a version control system and a packager application are used to provide additional functionality. It differs in the respect that it serves a user interface to clients and can be accessed directly with a Web-browser. Therefore a GUI component presents Web-pages that are dynamically filled with distribution process information. The interface provides product lists, feature descriptions, download, licensing and other information. The GUI component should be customizable and extendable. The information provided by the version control is used in the software abstraction. The class contains a software definition that is structured into a product and version class. The product provides feature descriptions that can be used during the selection by the customer. Additionally the feature implementation is described

for each version of a product and it reached from missing to supports and extends. The Version Management related processes are described in the following chapter. This approach can be compared to the feature description definition of Andreas Zeller [24] that has been introduced in 2.3.1. Further a software policy is used that constraints the use of the product, like support for operating systems and other requirements. A customer abstraction is used to cover the interactions with the large number of clients connecting to the common service. For this model the customer definition contains contact and payment information that is partially used for the distribution process to provide distributable lists that correspond to already released products. Further a customer policy contains feature preferences that influence the product selection, licensing requirements that can restrict and prevent consummation of products and payment requirements that can customize an external process that should be integrated into the architecture. The payment related issues are not covered in depth. The customer information is exchanged manually through the client interface. Compared to the Client/Server architecture a target abstraction is omitted for several reasons. First, because less information is available about the anonymous client environment. Second, because the installer used in deployment is provided with the product and customization of installation activities are minimal. Third, a customization based on information that could be provided is possibly not in the interest of the client if products are requested that are installed on different machines than on those where downloaded. Therefore the client is provided with information about available products and decides which one best fits its needs. Based on the combination of the software and customer abstractions a distributable is defined that also consists of a definition and a policy. The definition is derived from the other abstractions. It additionally contains information relevant for customizing the package creation process, like destination paths, desktop resources and other attributes. The process of the policy merge operation is described in the following section. The distributable class is independent of the other abstractions and therefore used to provide the product description to the client and the packager.

The packager class contains the same information and has the same functionality as described for the "Client/Server" architecture. It also has to perform operations that transform the software into packages and it uses package descriptions provided by the common service that are based on the distributable definition and policy. Comparable to the Client/Server architecture is the common repository that is used to store all definitions, policies and packages created during the process. Further solutions using the "Standard software" type of distribution can be hosted widely distributed but demand and contain very similar repository information. A replication mechanism, as described in 5.2.3, can be used to support this model. The infrastructure class contains all relevant information about the common service and is comparable to the "Client/Server" architecture, even if the information stored is different according to the implemented service functionality. Compared to the "Client/Server" this model doesn't contain an installer class, which is part of the client distribution environment and has to be provided as a package as described above. Further, the installation process is not covered at all omitting a distribution plan and all related functionality.

5.3.2 Process view

In this section the configuration and distribution processes of the Standard software model are described. First, the configuration of products is shown with special considerations of the integrated Version Management.

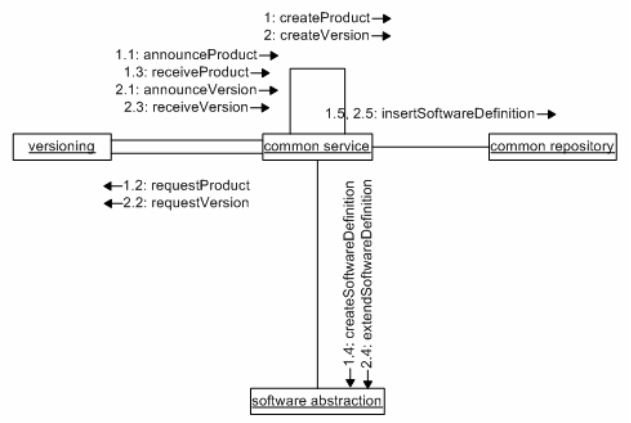

Figure 40: Standard software model process view - Version Management

Figure 40 shows the operations between the common service, the software abstraction and the Version Management. Normally, the phase of product creation (1) is initiated by the Version Management by announcing new software products (1.1). Then the common service requests (1.2) and receives the product (1.3). In the next step, the software definition is created (1.4) by handing over the product description. The definition is stored in the common repository (1.5). The phase of version creation (2) is analogous but operates on a version level. New versions are announced, requested, and received (2.1-2.3). The software definition of the corresponding product is extended by the version description (2.4) and the software definition is updated in the repository (2.5). The version interface can also be used for other software definitions as described in the following architecture (section 5.4) and can also be generalized (section 5.5).

Figure 41 covers the interactions of the service, repository and the different abstraction classes. The process of policy related operations can be shown as follows. First the customer policy is selected from the database (1.1) and the abstraction is initialized with the data stored in the repository (1.2). The same operations are performed for the software abstraction (2.1, 2.2). In the phase of policy creation (3) the target and software abstractions are selected (3.1, 3.2) and a distributable is created (3.3). Then the merge operations are performed (3.3.1-3.3.2) by the abstraction class. In contrast to the "Client/Server" architecture the operations contain negotiations of the product and version features as well as customer licensing and payment requirements. The basic merge mechanism is the same and can be generalized as shown in 5.5.3.

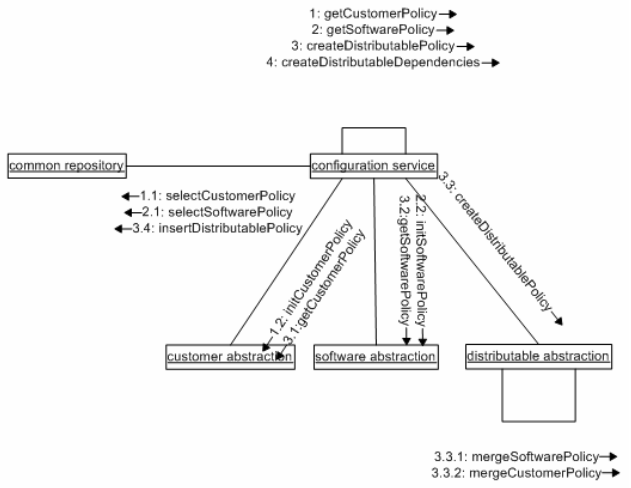

Figure 41: Standard software model process view - configuration

The attribute combination process for the distributable definition class is comparable to the "Client/Server" architecture with respect to the participating abstractions as shown for the policy related operations. The same phases are executed during definition creation.

After this configuration-related process the distribution process of this architecture is shown in Figure 42.

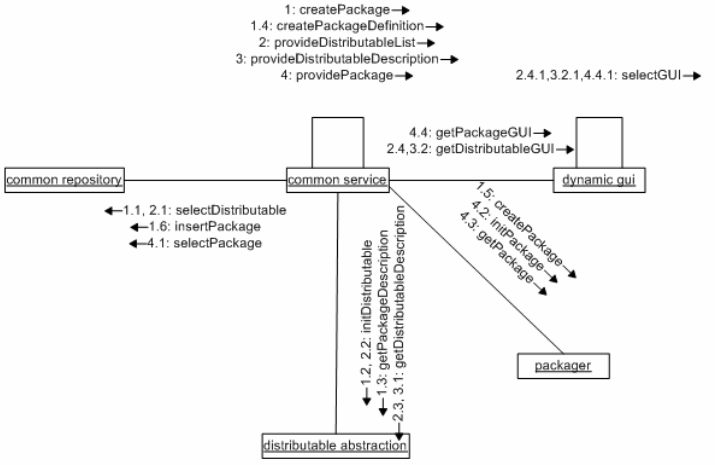

Figure 42: Standard software model process view - distribution

First and independent of the distribution process a package creation phase is initiated (1). The corresponding distributable is selected from the database and initialized (1.1, 1.2). Then a package description is generated by the distributable (1.3) that is used by the service class to generate a package definition that has to be readable for the packager class (1.4). The definition data is used during the package creation operation (1.5). The package is then stored in the common repository (1.6). In the second phase a GUI for a distributable list is provided to client (2). First the distributable is selected from the repository (2.1) and initialized (2.2). A distributable description readable by the client (e.g. XML) is generated (2.3) and the corresponding GUI is requested from the dynamic GUI class (2.4) that selects the Web-pages from an internal cache mechanism, to receive better performance (2.4.1). The resulting dynamic interface is provided to clients in an HTTP response package. The client selects a distributable that it is interested in (not visible in the Figure) and starts the third phase by receiving a more detailed distributable description (3). The initialized distributable provides a more complete and comprehensive description (3.1) and together with the dynamic GUI (3.2, 3.2.1) the description is presented to the client. In the last phase of the distribution process, the client requests the package of a specific distributable (4). The package is selected from the repository (4.1) and initialized (4.2). The service class receives the package and its content in a distributable format (4.3) and sends it back to the client.

The distributable and package process can be compared to the client/server architecture, the direct GUI creation and the installer-less process mark the differences between these models. Further operations required by the client are less complex, primarily install and uninstall activities are of interest, updates and repair mechanism are nowadays also provided by several platform installers like the Windows Installer [104].

5.3.3 Deployment

This section provides a deployment view of the Standard software architectural model. Figure 43 shows the corresponding UML notation.

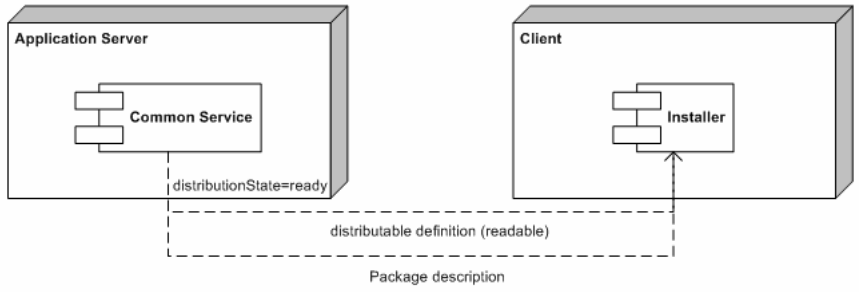

Figure 43: Standard software model deployment view

This view is simpler than for the "Client/Server" architecture, because the number of involved nodes is dramatically reduced. The model consists of an application server component that hosts the common service, the repository, and the client web-interface. Additionally the repository can be hosted on a separated database server without requiring changes to the model. The server sends distributable definition, package descriptions and package files to the client. The client component only consists of the installer executive that is transferred from the application server and invoked on the target machine. The distribution state that controls the availability of products can be defined and managed within the distributable description and evaluated by the common service during the distributable list generation for the client. The communication is HTTP based and also uses HTTP based data transfer. Simple FTP-based transfer is sometimes problematic due to firewall configurations and large environments.

5.4 Web-service model architecture

This architecture covers Web-service-based models that can best be compared to case study "Custom Web-service". In distinction to the distributed architecture Web-service-based applications with additional database support are considered to be distributed. No client/server applications are supported and no client software components other than Web-browsers are considered. Therefore a clear distinction between the models is made and a selection for a specific scenario should be possible.

5.4.1 Logical view

Again the logical view for the model is provided first and the classes related to the configuration process are shown in Figure 44.

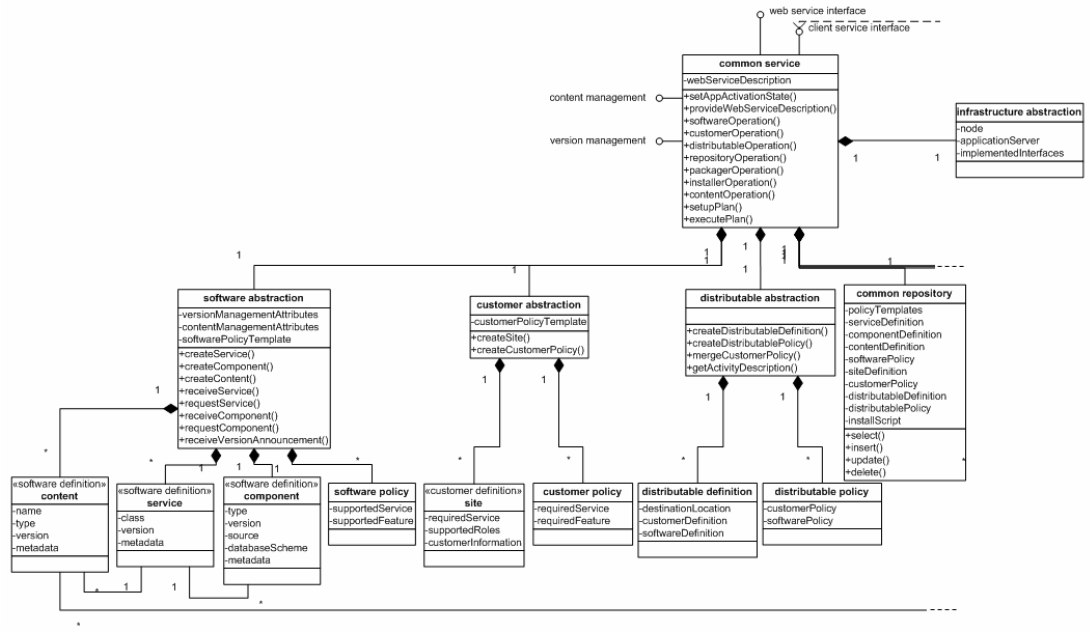

Figure 44: Web-service model logical view - configuration

The main classes and the base structure of the model is comparable to the first two models, yet there are some very important differences. The model uses a software- and a customer abstraction and omits the target abstraction. The software abstraction uses a definition of a Web-service and their components and a content definition. All classes contain a name and version attribute. The content and the component classes additionally provide a type attribute that is used to identify the operating system or application server registration method for components and the way of displaying content for Web-browser. The service additionally contains a class attribute that is used for application negotiation (e.g. print service, file service). All definitions contain a metadata attribute that is a more complex data type and consists of all other attributes used to describe the structure of the service, component, and content classes. Normally this information is used to install and register files, to know about deadlines of content actuality and to avoid the necessity for a target abstraction, an Inventorying mechanism or other expensive infrastructure with similar functionality. Further, the software abstraction has a corresponding policy that implements required/support semantics for specific services or features of services. The metadata has to contain information about these restrictions otherwise rules regarding this data cannot be resolved. The software definition can be used to generalize the metadata definition found in different service and component models like COM, CORBA or EJB. The customer abstraction is therefore primarily used to implement and resolve the requirements for services and features. In contrast to the abstraction in the Standard software architecture, implementers of this model have implicit knowledge about the customers and can configure and customize the customer definition manually or with support from a directory service. Payment, Licensing

and other information is irrelevant, showing that the same terms in these architectures have a quite different structure. The customer abstraction could also be implemented as a target abstraction, but its policies depend more on organizational than on infrastructural requirements, which should be used to distinguish between those practices. The distributable, repository and infrastructure class shown in the Figure have the same responsibilities as shown in the last two architectures and their operations just slightly differ corresponding to the Web-service centric model. The common service class can be compared to the "Standard software" architecture, even if the distribution process is quite different as shown in the following section. The common repository/common service approach has also been used in the Standard software architecture.

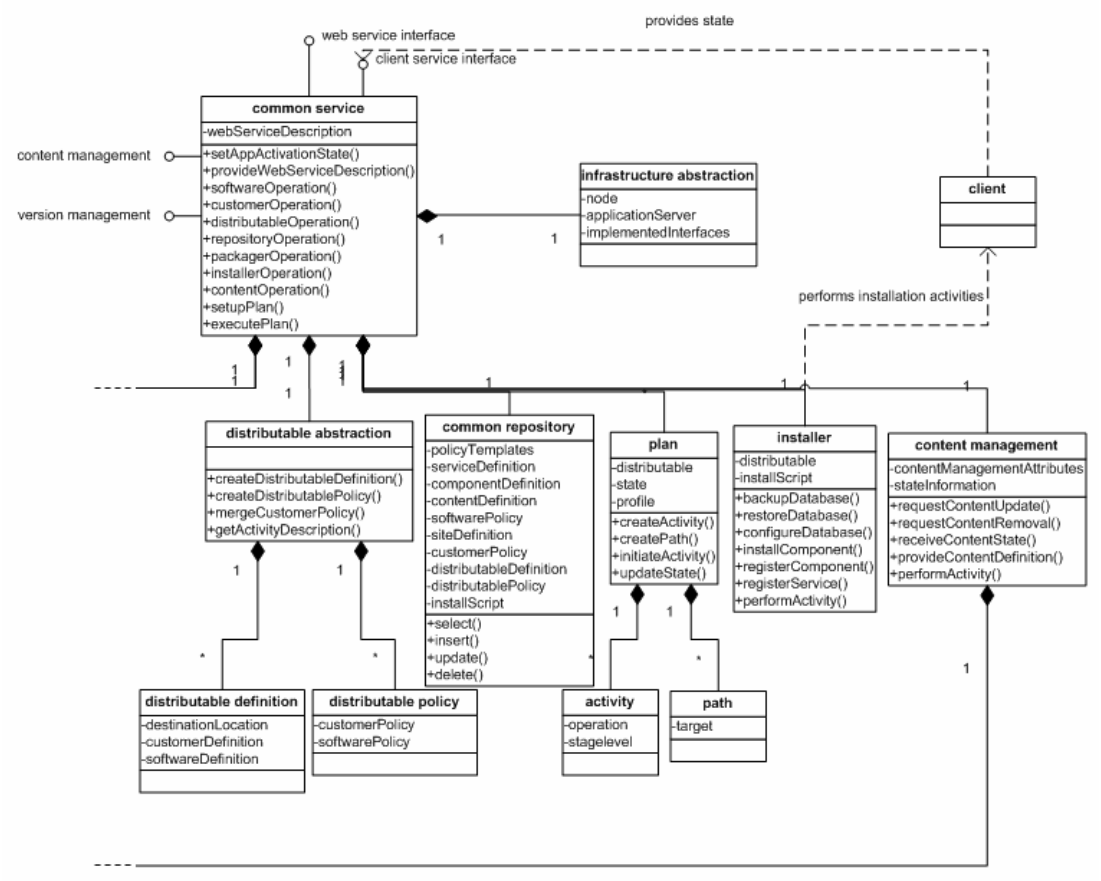

Figure 45: Web-service model logical view - distribution

Figure 45 shows the classes related to the distribution processes. The distribution process works different than in the "Workflow" and "Standard software" models and can be best described as "staging-like". First, most components distributed are self-describing, perform self-registration, are distributed, fine-grained, and wide-spread. Therefore the model omits a packager class. Second, a Content Management system is integrated that provides an interface to the distribution system for staging content. Content related activities have to be integrated into the process that basically executes registration functions and performs installer-based activities for components that are not able to be registered without an installation function like database-related activities where backup and script files have to be executed on the target machine. Further a history class is omitted because the distribution process doesn't adapt installation activities based on this information. The plan, activity and path classes contain a structure comparable to the "Client/Server" architecture, even the attributes and operations differ because of changes to the distribution process. The plan class additionally provides profiles that allow different distribution activities related to the same set of distributables and the same path definitions. This functionality allows a mixture of service and component releases and subsequent content updates to be modeled and executed using a similar infrastructure. The activity class additionally contains database and content

related functionality and operates on distributable definitions (single or several files) rather than on packages. The client shown in the Figure represents the target platform and is normally not a client in terms of the other architectures. The target is more likely a database server or an application server. Nevertheless state information is provided that is used to update the state of the staging process.

5.4.2 Process view

In this chapter the process view of the Web-service architecture is shown. First, the distributable definition creation is shown in Figure 46.

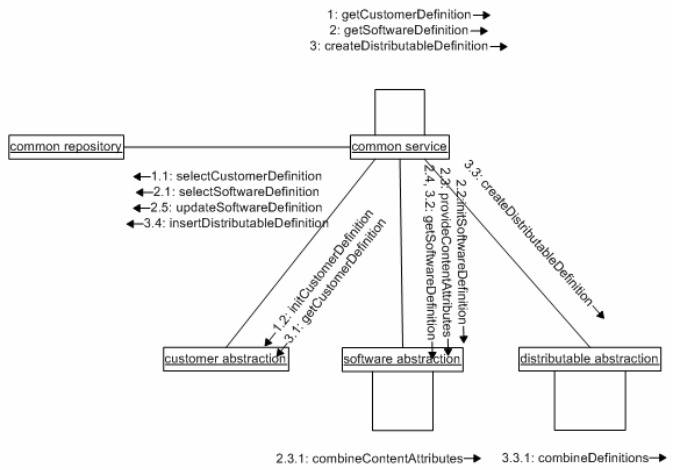

Figure 46: Web-service model process view - configuration

The process follows the descriptions of the first two architectures but additionally supports an external definition of a content definition. The customer definition is read from the database and initialized (1). The software definition is read, initialized, and in the third step the content metadata including other attributes are provided through the Content Management interface (2.3) and combined with the existing attributes of the content definition of the software abstraction (2.3.1). Then the software definition is provided (2.4) and the data in the common repository is updated (2.5). The third phase for distributable creation (3) can be compared to the other architectures with respect to the different attributes stored. Also the creation process and merge operations for the customer, software and distributable policies are processed in the same way as described in 5.2.2 and 5.3.2. Even if no Version Management is incorporated into the model, the interface provided in the "Standard software" architecture can be adapted to support service- and component-based operations and therefore also be compared to the interface introduced in Figure 40.

Figure 47 depicts the process of plan creation and execution as required by a Web-service-based model. The distribution process of the Web-service model can be shown. The Content Management system is integrated and a staging-like approach is chosen to provide the required flexibility in a distributed environment.

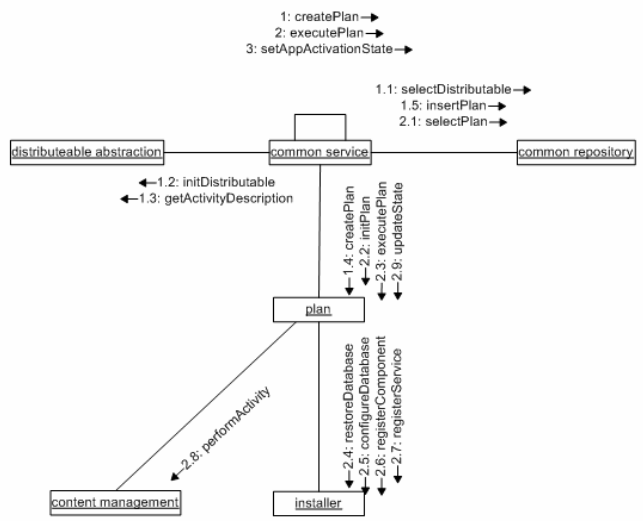

Figure 47: Web-service model process view - distribution 2

The distribution process contains three phases. In phase one a distribution plan is created (1), which is related to all required distributables as stated in the corresponding policy and contains all activities for component, database, and content installation. First, the distributables are selected from the repository and initialized (1.1, 1.2). Second, a description containing Web-service related activities is provided (1.3) and used in the operation of plan creation (1.4). This step can be further detailed for the path and activity subclasses. These operations can be compared to Figure 32 of the Client/Server architecture logical view. Finally the plan is stored in the repository (1.5). In the second phase the created plan is executed (2). The plan is selected from the repository and initialized (2.1, 2.2). The plan is then executed and based on its activity definition one or more of the described service, component, database and content operations are performed (2.4-2.8). Activities that should be supported cover the restore and configuration of a database, the registration of services and components and the staging of content. Further activities for deregistration or database backups should be considered. An important issue to support staging regards content updates that are independent of the deployed components. The plan class therefore supports different profiles that cover the base distribution and subsequent content updates as described in 3.3.9.

5.4.3 Deployment

This section describes the deployment view of the Web-service architecture. Figure 48 shows the system nodes interacting in this model.

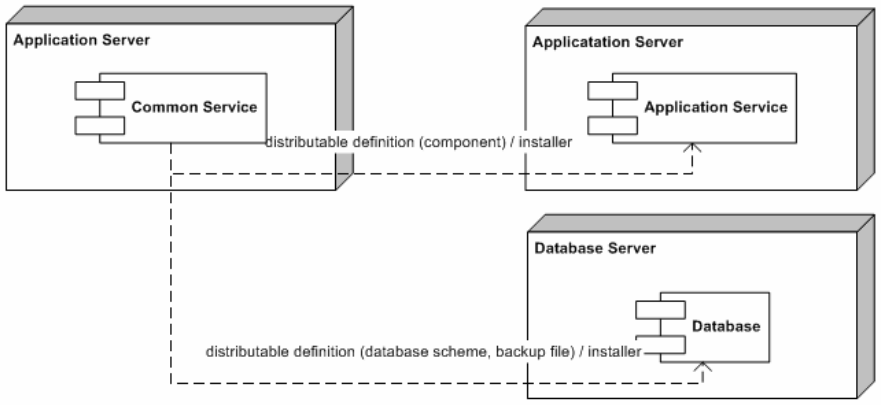

Figure 48: Web-service model deployment view

An application server hosting the common service is required. It is able to host a Web based application service and has integrated HTTP support, object pooling, transaction support and

many other advantages. The Web-service-based system consists of several application and database servers hosting content, components and databases. The distribution process provides distributables and installer applications to the destination nodes for installation activities on the target site. The common service might also be distributed across several sites for geographically distributed application servers. In this case a repository replication is most appropriate to allow on-site installation of components and a distribution consistency of the Web-service functionality. Additionally all nodes can be clustered to provide load balance and fault tolerance.

5.5 Towards a generalized model architecture

This section elaborates on a generalized model architecture that shows the mechanism and abstractions used in all described architecture models. General assumptions are made about the configuration and distribution processes with respect to the requirements of the product studies of chapter 0 and the case studies of section 5.1. The similarities of classes and processes within the described architectures have to be found out and should lead to a generic model. This generalization process is described in the following section, before the architecture is discussed in more detail.

5.5.1 Architecture generalization

A process of architecture generalization has been used to derive a generic architecture from the three architectural models provided in the last sections. As stated in section 3.2.3 the products in the field of software configuration and distribution use very different architectural styles depending on implemented processes and purpose. To provide a more consistent view the assumption has been made that a service-based infrastructure and a central repository are building blocks that fit for most scenarios. Further the architectures discussed aim to integrate different styles depending on their advantages for common scenarios. Therefore the widespread use of packaging and installer applications as separate tools lead to models that correspond to this situation, while the process implementing services can be based on current technologies of distributed systems. Bass et. al. [105] suggest an approach to relate and derive architectures. Bass defines unit operations and refers to standardized modification operations observed in architectures developed over the past years in the field of large scale information systems, internet-standards, and Human Computer Interaction (HCI) devices. Unit operations like generalization and specialization allow to construct further developed models from existing architectures. Additional operations should be introduced that are used to construct architectures as commonalities and variants of each other. This view of relations in architecture models has been described by Coplien et. al. [106]. For the generic architecture optional components and components with a set of common and distinct attributes and operations can be specified and allow to define a relationship between the architectures.

A main difference between distribution systems lies in the type of supported applications and distribution artifacts. It is necessary to provide abstractions for data and operations that can be customized for specific scenarios. The used software, customer and target abstractions are very general definitions of real-world entities that can be used in all three models. Further their definition and policy subclasses are very general equivalents for the structure, content and behavior of these entities.

The more complex task is the abstraction of the different processes implemented in the distribution models. An easy way to achieve a generic model is again the definition of optional process phases. The Client/Server and Standard software architectures for example, provide packaging functionality, while the Client/Server and the Web-service architectures both provide installer-based deployment activities and define distribution plans, both not necessary for the Standard software model. In a generalization all operations should be covered that at least exist in one of the models. To avoid redundancy in the resulting architecture similar operations should not occur twice and a form of orthogonality in operations should be defined.

A more sophisticated problem are differences in the same process phase. This situation can occur in different forms. The functionality can be implemented (a) using different classes or (b) using the same classes and different operations or (c) using different implementations within the same operations of the same classes. The last approach is the most desirable one, because it has the smallest change effect on the architecture. In most cases this simple categorization doesn't fit for architecture modeling. Some operations are similar on one level of granularity while they differ in detail. As an example the Client/Server and the Web-service architecture both model the execution of a distribution plan. While the executePlan operation has the same functionality in a coarse-grained view, the more detailed view shows that the Client/Server model utilizes a history class for state management, while the Web-service-based model omits this functionality.

Furthermore, the supported interfaces of a software system are a key aspect of every architecture. For example the support for external interfaces differ in all three models. As a generalization an interface definition that allows to integrate such systems, without specifying which systems to use and which functionality to expect is reasonable. Several examples exist in the three models like Inventorying and Content Management. Their impact reaches from optional functionality to process influencing and tight integrated operations. Finally a generic architecture can establish additional functionality and constraints that can hardly be discovered by designing a single architecture. For example a software description language concept to describe policies and the definition of policy templates are introduced during establishment of the generic architecture.

5.5.2 Logical View

A generic model has to cover classes implementing the configuration and distribution processes that can be derived and further refined for different scenarios. The following two Figures cover the logical view of the separated configuration (Figure 49) and distribution (Figure 50) services.

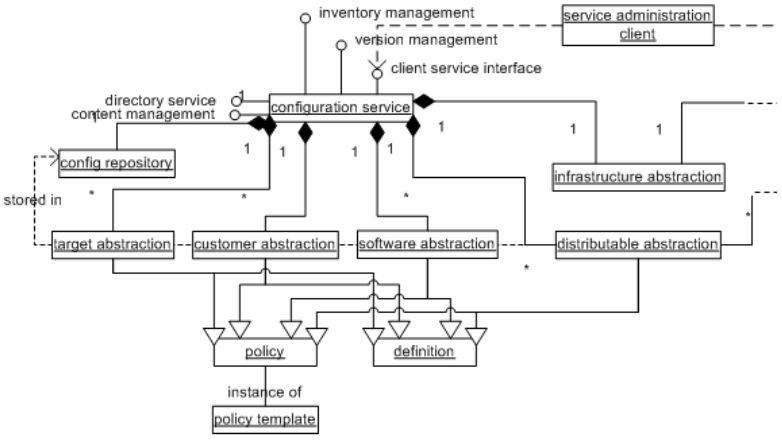

Figure 49: generic architecture logical view - configuration

The configuration classes consist of a main class that represents a configuration service. The responsibility of this class is the communication and instantiation of subclasses, the access to a common repository, the communication to administration and end-user client processes through a service interface and the implementation of the interfaces to external systems. One category of subclasses of the configuration system are the abstractions used to encapsulate models of the real world configuration use-cases. The main abstractions are the software, target and customer abstractions. They correspond to the requirements and definitions of sections 4.2 to 4.4. All abstractions consist of two subclass categories, a definition that defines variable content and a policy that defines the flexible behavior of the abstraction. The structure of the policy and definition classes is generalized for the abstractions to allow polymorphic operations and to reduce complexity. Additionally templates for policies are

provided that restrict the descriptions of behavior to properties that are relevant for the configuration and distribution of software (for a further description see 5.5.4). To reduce the generality and to make the implementations more meaningful, attributes of the specific architectural models in the last chapters can be inserted.

The software abstraction covers the software to be configured and distributed. Its main purpose is to describe the software content and attributes together with the features and the relationships and dependencies between software components as the software policy. This abstraction can be found in any distribution environment and is therefore used throughout all specific models. The target abstraction covers the target environment of a software, the site, the node where the software has to be installed and in detail capabilities of the operating systems and environmental parameters of the network. Again properties that are relevant to the processes covered here are related in a policy that defines for example which target is available for specific installation activities or is able to host parts of the distributed software. A third abstraction covers the customer as the organizational part of the distribution environment. Often required in this context are definitions of nested user groups from a directory services that define who gets software installed and also inventory information that user and not machine related. Additionally software products and operating systems allow host and user specific customizations. Further more the abstraction can be extended with features required by customers or with licensing and payment conditions that have to be met before installations can take place. Most definitions and extensions of the later two abstractions are scenario dependent and therefore not applicable in all specific models.

The most important abstraction provided by the architecture is the distributable. To provide a single view for package based and component based systems, the distributable covers a single item of installation and a policy about when and how to install the item. The distributable definition is a compound of software, target and customer attributes (see 5.2.1). Figure 50 shows that relationship between the abstraction classes in detail.

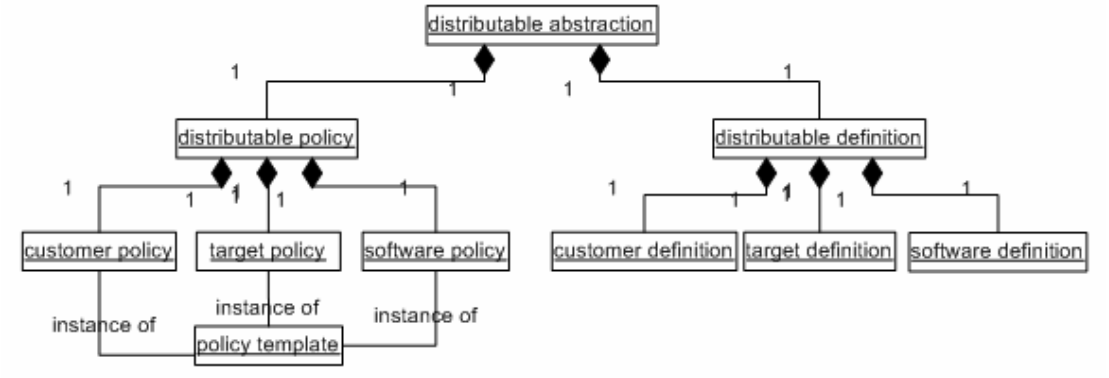

Figure 50: generic architecture logical view - abstraction details

The definitions are combined in the distributable object by selecting specific attributes that are required for proper installation operations, in more detail, that are used for display product information or referenced in the distributable policy. The policy is generated in a more complex merge operation that is described in detail in the process view of the generic architecture (see 5.5.3). The distributable is also part of the distribution classes and counts to the shared class category. Also part of this category is the infrastructure abstraction and the service administration client. The infrastructure class covers the description of the configuration and distribution system itself and describes its system components and distributed architecture. The client implements the configuration and distribution service interface and allows to manage and adapt the system. A more detailed description is given in the interface section 5.5.7. The last subclass covered by the logical model consists of a configuration repository that stores the definitions and policies described above. The repository can be relational or object-oriented and therefore be able to instantiate objects on behalf of the configuration service class. This abstraction is used throughout all models

however the content that is stored slightly differs. The distribution classes are described as shown in Figure 51.

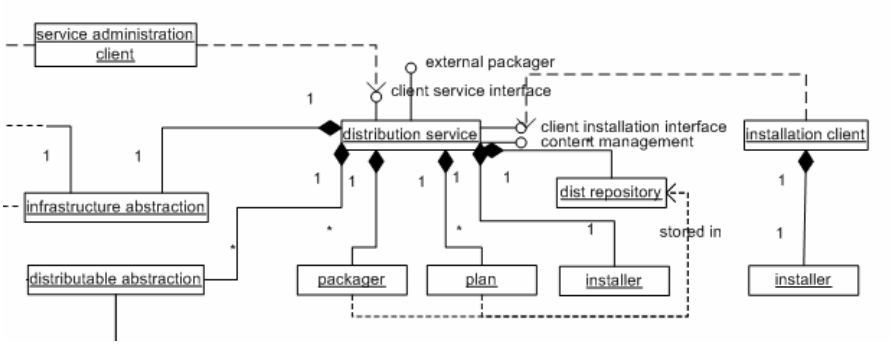

Figure 51: generic architecture logical view - distribution

The classes shared with the configuration part have been described above. The distribution part consists of a service class that has the usual class responsibilities within the model. It instantiates subclasses and implements interfaces to clients and external systems. The model further consists of classes required to distribute software. All classes operate on the distributable abstraction when performing their operations. The first subclass is a packager component that is used to combine distributables that have to be deployed together, compress them and provide some installation procedures that might be operation system or language specific. For software that is not language neutral the selection mechanism has to be evaluated before packaging. For language neutral software the language requirements can be evaluated during setup. Therefore, the packaging class needs access to the distributable policy that describes such dependencies. Packaging is optional and some software models need no specific installation procedure because they are directly registered by the operating system or an application server. In such cases the packaging is omitted. Another class required is the distribution plan. This class covers the "how-and-when" of software installation. It consists of subclasses that define a schedule for activities, a path for distribution and state and state history management. An installation plan is optional. In simple client-side installation scenarios (case study "Virus Scan") no distribution planning occurs and the class can be omitted from the architecture. The next subclass of distribution is the installer. This class covers the process of providing an install script within the distribution system that includes all performed activities and implements operating system independent and specific operations carried out at the client. The installer is therefore part of the distribution service and the client operation environment. The last subclass described covers a repository that stores all data from the distributable, plan, packager, and installer classes. The repository is separated from the configuration repository to model constellations where the two main processes are implemented in different systems. Exchange of distributable definitions and policies maintain a consistent model as described in the process view (see 5.5.3). The interaction between the subclasses described above is depicted in Figure 52.

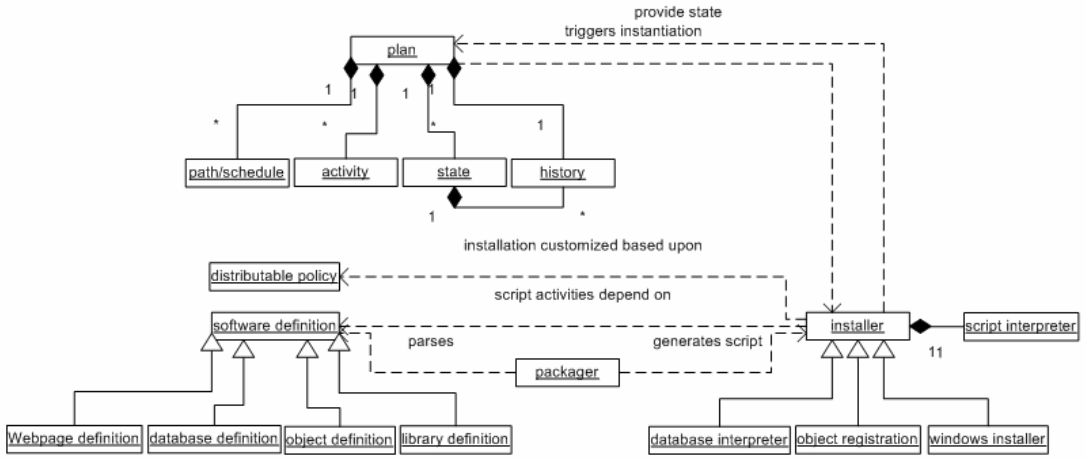

Figure 52: generic architecture logical view - distribution details

Figure 52 shows the subclasses of the plan class that define the path, schedule, activity, state and history of the distribution. Activities correspond to install, uninstall, repair, update, and other installation operations performed by an installer or the operating system. It is therefore the main class that controls the distribution process. The defined activities can be implemented by an installer and be based upon package operations or can be directly executed on distributable definitions and operating system mechanisms. The dependencies between the distributable and the packager are defined as parsing the software definition and generating an installation script. The installer is related to the distributable policy that defines the activities performed during plan execution. The activities performed by the installer are related to the software definitions within the distributable. As examples, Web-page, database, object, and library installation procedures have to be handled differently and explicitly by the distribution environment. The history class is related to one plan and contains all performed activities. The plan class has to provide the history information during the instantiation of the installer class. The installer customizes its operations based on performed activities as shown in the following section. Without history information the performed operations are independent of the current distribution state (see 5.4).

5.5.3 Process view

To gain a better understanding of the interactions between the classes described in the logical view, the process relates them in common operations used in the configuration and distribution processes. The generic architecture covers them with semantics that can be derived for specific scenarios throughout the models in sections 5.2 to 5.4. Collaboration diagrams are used again because the syntax is more related to the class diagrams above, sequence diagrams can be derived directly.

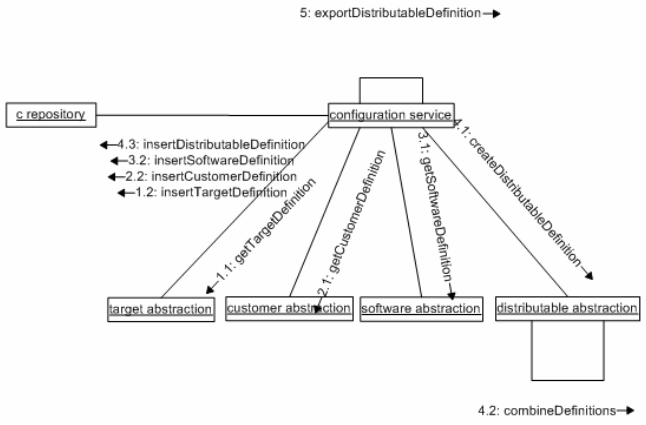

Figure 53: generic architecture process view - configuration 1

Figure 53 describes the process of creating a distributable definition out of the definition of the three main abstractions. The definitions of software, target and customer are first created by the corresponding abstractions triggered by the configuration service class. Get-operations correspond to the instantiation of an object. The operations 1, 2 and 3 are performed asynchronously, the sub-operations synchronously. The operations don't have to occur in the described order. For example the customer definition can be created before the target definition. This is of additional importance when external interfaces like Inventorying or directory services provide this information independently. However all three operations have to be finished to start defining a distributable definition. The combine operation refers to the process of creating a compound attribute definition based on the abstractions and extended with attributes from the distributable. The distributable then can be exported for the distribution parts of the process.

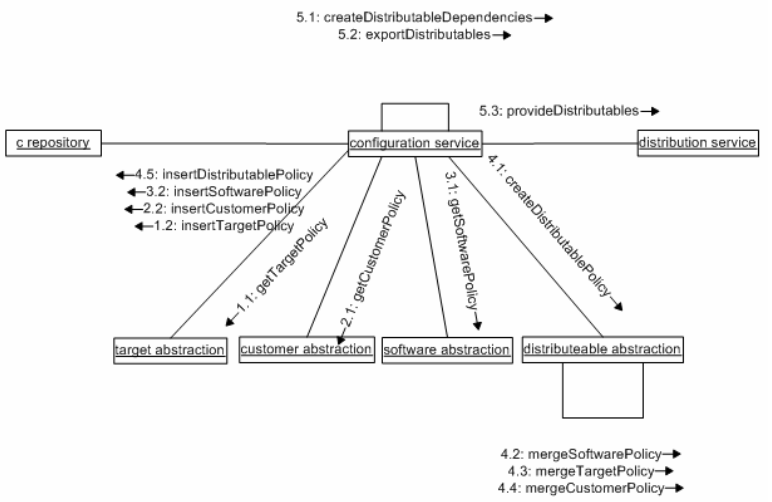

Figure 54: generic architecture process view - configuration 2

Figure 54 shows the second part of the operations that correspond to the creation of a distributable policy. The operations can be nested with the operations shown in the figure. Definitions and policies can be provided together in a single operation or separated where the definition is based on inventory information while the policy is provided by customization through a system administrator. Most operation semantics can be derived from the last Figure. Different from there the merge operation is more complex than the combination of attributes. In detail it is a two-step process of combining and extending. A more detailed description of the operation is given in 5.5.4. After the merge operation the distributable is stored in the repository and can be exported in an exchangeable, readable format (preferable XML) and provided to the distribution service.

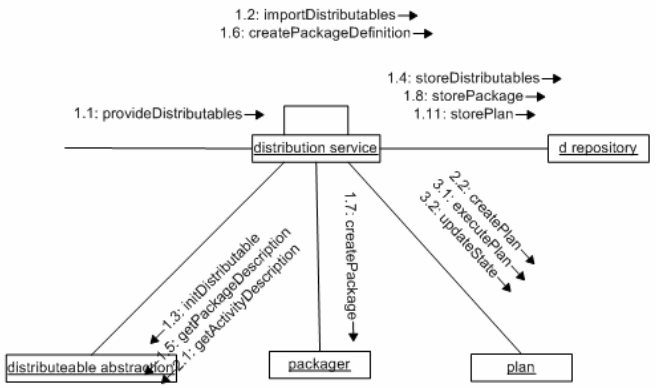

Figure 55: generic architecture process view - distribution 1

Figure 55 shows the class interaction during package creation and plan definition and execution. The distribution operations of the model are described here. The operations are performed within a serial process. No other order than the one described above should be used, because most steps are based on the outputs of the previous. First, in the preparation phase (1), a distributable is imported corresponding to the export function of the configuration service and stored in a separated repository. Then the distributable abstraction is initialized and a package description is created based on an internal processing of the distributable policy. The distribution service that implements the packager interface has to convert the description to a packager specific format during the "createPackageDefinition" operation. Further the packager is invoked to provide a package. During this operation the packager might additionally access the repository for installation sources. Afterwards the package is stored in the repository and the planning phase (2) can be started. All packager operations have to be considered optional for models that provide direct installation methods without packaging the components. First the activities that have to be performed are extracted from the distributable policy and used in the "createPlan" operation to setup activities, paths and schedules. The plan creation step is an administrative task that can be restricted by policy rules. For example specific services have to be set up on different nodes using different paths. Further database setup or class registration operations are defined for the installer. Finally the third phase of installation can be started. The plan is executed and operations of the installer result in an updated state of the plan. To describe the phase 2 and 3 in more detail Figure 56 also shows some subclasses of the plan class and the corresponding sequence of operations.

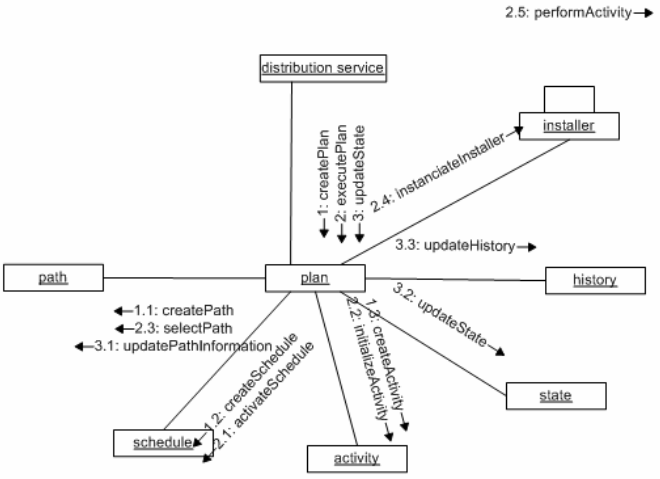

Figure 56: generic architecture process view - distribution 2

During plan creation a path is setup with information about the destination site where the software has to be installed and attributes about the network infrastructure (e.g. fall-back servers). Then a schedule is created where the time and frequency of operations are defined. For example a content update might be implemented with a frequent recurring schedule. Subsequently an activity related to the schedule is created (these steps can also be performed in reverse order) where the installer, that has to be invoked, and its operation options can be specified. Models without an installer use activities for operations that are performed by the operating system. The "executePlan" operation is also a nested operation where the schedule is activated. Asynchronously through schedule activation the plan is executed and a series of activities is initialized. Based on the activity definition a path is selected and the destination information is used to optionally migrate the installer to the corresponding site. Second the history information is provided to the installer to customize installation. The installer is invoked and the installation activities are adapted and performed. The activities defined and the activities performed by the installer can be divergent. Not all operations performed by the installer might be configurable during runtime. The installer might also provide state information about finished activities during installation. Self-registering components normally don't rely on installer operations. For some models the installer operations are

exchanged with registration functions of an application server. The plan class uses it to update the current state, the path information, the state of related activities and the activity history.

5.5.4 Policy descriptions

In general the software policy should specify capability information that can be required, restricted, forbidden or otherwise constrained by target and customer policies. A policy can be described as a software description provided as a hierarchically structured document. This approach is a derivation of the Deployable Software Description Format definition provided by Rick Hall [30] as introduced in 2.2.3 and 3.1.3. The original model works with a distinction of properties (software attributes) and importedProperties (attributes of the installation environment). This model should be extended to provide different policy definitions as shown in Figure 57.

```
<!ELEMENT DistributableProperties (DistributableProperty)*>)>
<!ELEMENT DistributableProperty   (Name, Type, Description, Value, DefaultValue)>

<!ELEMENT SoftwareProperties (SoftwareProperty)*>)>
<!ELEMENT SoftwareProperty   (Name, Type, Description, Value, DefaultValue)>

<!ELEMENT TargetProperties   (TargetProperty)*>
<!ELEMENT TargetProperty     (Name, Type, Description, Value, DefaultValue)>

<!ELEMENT CustomerProperties (CustomerProperty)*>
<!ELEMENT CustomerProperty   (Name, Type, Description, Value, DefaultValue)>

<!ELEMENT Composition        (CompositionRule)*>
<!ELEMENT CompositionRule    (Condition, ControlProperty, Relation, Properties)>

<!ELEMENT Assertions         (Guard, (Assertions | Assertion)*)>
<!ELEMENT Assertion          (Guard, Condition, Description)>

<!ELEMENT Dependencies       (Guard, (Dependencies | Dependency)*)>
<!ELEMENT Dependency         (Guard, Condition, Description, Resolution, Constraints)>
```

Figure 57: Distributable Policy DTD

The software, customer and target abstractions each use a policy with their own property collections. The Condition attribute in the Assertion and Dependency element contains property values that are evaluated for customization. The distributable policy can be described as a compound that contains all three property collections and an own collection of properties provided during configuration. In a policy merge process the attribute definitions have to be combined first, next compositions and assertions can be combined. As the last step the conditions that contain property values can be extended by introducing properties from different policies or by combining existing conditions. The transformation process of policies has to be investigated further; a more detailed view is out of scope of this thesis.

5.5.5 Commonality in abstractions

Commonalities found in the abstractions for software, customer, target, distributable, and infrastructure are described in this section. The software abstraction can be seen as the most divergent one, because it is mainly dependable of the system to be distributed. By separating the definition from the policy, requirements common to all systems can be modeled the same way, while the differences in the software structure can be made explicitly. The software definition and policy are hold in generalized classes, classes for specific architectural models are derived from it and build up on the functionality generally introduced. The generic architecture should be further developed to support a generic model for the internal structure of a software, useable in most cases and customizable for specific scenarios. For example a nested system and system components model could provide enough flexibility. Though integration of Version Management and directory services make this approach more complex.

The target abstraction in the models is, where available, also comparable. A target system mainly consists of the same attributes and policies. Differences exist between modeling client or server environments and on different operating environments like PC servers and

mainframes. The internal structure of the abstraction is mainly the same. One problem that arises is the optional use of an Inventorying system, which strongly influences the target information gathering process. The inventory interface has to make this process transparent and use the same target operations as the internal process. Further the abstraction can be optional which has been addressed by the class structure and the configuration process that can be easily reused without a target abstraction.

The customer abstraction is used in very different ways and its definition is more diverse than it is in the abstraction of a computer environment. Different characteristics when modeling customers should be considered. The relationship level influences the kind of information stored. For example information about a large anonymous group of customers has to be handled explicitly within the distribution system, while a small group of well-known customers can be handled outside the system and is implicitly modeled within the system as different software configurations. The customer abstraction itself in such cases can be used to further select configurations based on a customer policy. Nevertheless, the abstraction has also very similar characteristics to the target abstraction, external interfaces and omitted support for such an abstraction can be handled as described above. The distributable abstraction as a compound of the three abstractions is more an exchangeable format and a scenario independent representation of the system to be distributed. The infrastructure abstraction can be used for any complex distributed environment that has to be managed. Such systems also integrate management functions from the operating system and integrate infrastructure state information into the executable processes.

5.5.6 Commonality in processes

Separation of the configuration and distribution processes is important. The first phase is more environment and data-driven, while the second is more process driven. The focal points can be very different, while standard-products have minimal configuration requirements, they demand a complete internet-based distribution infrastructure. On the other hand, complex Client/Server or Web-service-based products have much more complex configuration requirements and a different distribution process. The generic architecture tries to present a model that can be customized to fit for most scenarios, therefore parts of the implemented processes have to be adapted or omitted to build a feasible solution. Further, the distributable abstraction used in the model enables the configuration and distribution processes to be separated. Therefore the different key aspects can be considered without a spread influence on the architecture. As stated above, interfaces can extend and even change the process implementation, but should not effect the architectural process models. Their special position is detailed in the following section.

5.5.7 Interfaces

The interfaces to external systems have been designed to be stable in different implementations of the architecture. The interfaces are implemented in the configuration and distribution service to avoid impact on the abstractions used. The interfaces have to be separated into data-providers like inventory and directory services, and process-influencing ones like version and Content Management. Data-providers are used in operations like queryInventoryDatabase and combineInventoryAttributes that can easily extend the representation of the software or customer definitions. Requirements of such interfaces have been stated in 3.2.4 while different techniques to integrate such systems have been pointed out in 3.2.5. Integration of the process related interfaces is more challenging because they cannot use one similar mechanism. To be most effective, the provided functionality has to be fully supported by the architecture while the overhead for systems that don't have corresponding support should be hold minimal. Derived from the three models the architecture should define supported operations in a general way without restricting the use to a specific product. As an example the "performActivity" operation of the Content Management interface can be used to integrate any distribution operation that is implemented. An encapsulation class should be used that overrides this method and implements the functionality of a specific Content Management system.

6 Evaluation, Summary, and Conclusions

This chapter summarizes the architecture generalization, suggests extensions and provides an overview about what can be concluded from the thesis.

6.1 Generalization process

In this section a short description of the generalization process is given. The first part that has been worked out for this thesis has been the product comparison. Throughout the product evaluation and comparison different models within the products could have been pointed out. Those models were different in terminology, distribution mechanisms and functional focal points. The next step has been to categorize the different scenarios that had been considered in the design of the products and to create case studies that best represent the different structure and behavior of configuration and distribution processes. This lead to the formulization of three Case Studies that represent the different categories. The cases have been further extended with support for different interfaces to be considered in the architectures. The decision which interfaces to require from which case study could have been made different. The Version Management, Inventorying and directory service interfaces are also relevant for the "Web-service" case study while others like Content Management represent case study-specific functionality.

By describing all key aspects of the Case Studies most relevant building blocks for a logical view and functionalities for the process view have been initiated. Therefore the next step was to create a first architectural model for all three studies. A service and repository based infrastructure has been constructed and a separation of the configuration and distribution tasks has been made. These first assumptions have been lead by experiences gained in other conceptual work on architectures. In a further step the more detailed requirements should have been introduced which were much more divergent. Through discussing the requirements of groups of products together with my contributing advisor, we decided to create three architectures that represent the different behavior but share a common terminology. Then the three architectures that corresponded to the Case Studies have been designed in detail in a second step. Those models had for example very different definitions of a software system. Another iteration was necessary to make the different models more homogenous in terminology and to describe the processes in terms of variants and extensions of base configuration and distribution operations (e.g. creation of policies or distribution activities). Abstractions for software, target and customer were introduced, which were able to describe real-world entities in general. A distributable abstraction has been created to provide a consistent model for artifacts in the distribution process. Packager and installer utilities have been encapsulated and directly integrated where useable. Further a more common mechanism for the interfaces to clients and external systems has been modeled. The architectures were then homogenous enough to redesign the first generic model in a third step and to build a generic architecture that shares terminology and abstract logical and process similarities. The class hierarchy used a service infrastructure, the different abstractions, the Software Distribution utilities and client and external interfaces. The processes for configuration and distribution used general methods that could be easily extended and adapted. This iteration uncovered some points of weakness within the structure of the case study architectures that have been further eliminated. The deployment views have been left unchanged which requires that a generic architecture has to be adaptable for different deployment models. The service infrastructure comparable in all architectures and a repository replication mechanism should support this approach.

6.2 Further extensions

The following extensions can be thought of for further research. With a combination of the Client/Server and the Web-service model a transaction based, package less distribution environment can be build. This model is useable for all situations where more than one Web-service host has to be integrated. A problem designing such systems are the very fine grained activities that have be modeled with transactional semantics. One way to solve this issue is to create activity hierarchies that accumulate changes of related service classes based on composition rules or dependency constraints (see 2.2.3). As a second extension a combination of the Standard software and the Web-service architecture creates an environment for ASP based Web-service solutions. Such systems adopt the Web-service model for business to business solutions with licensing und payment integration and the customer abstraction of the Standard software model that handles a large anonymous user base. With the generic architecture as a base model for distribution environments such combinations can be designed without the risk of an architectural drift as described in [105]. On the contrary such new models force to extend and partially redesign the generic architecture that itself is developed further through this process while maintaining its stability.

6.3 Not covered in the architectures

In this section I provide a short description of topics not covered in the architectural models. One import issue in all distributed systems is security. Some related issues have been addressed in 3.2.2 but the architectures lack a corresponding mechanism. Furthermore the physical and process view of an architecture cannot describe performance and availability issues sufficiently. The deployment view introduces replication, caching and clustering of services and repositories but the requirements stated in 3.2.1 should be further incorporated into the architecture. Especially solutions for the requirements above should be modeled in separate classes, because most issues are also addressed by operating system functionality and a decision whether to implement them or rely on an existing infrastructure shouldn't be restricted through the architecture. From the product evaluation an distribution approach related to the Marimba Channel design or the Webstart application invocation is not considered. The Channel model is just a different view of a distribution plan where the producer and consumer of data are dynamically related. The path class can also be used to provide such channel semantics. The Webstart model is currently more applicable for Standard software products, has a strong language dependency and is only supported for Java.

6.4 Conclusions

Based on the existing infrastructure and the software to be distributed, all requirements for a distribution environment are defined. The different architecture models therefore have been built bottom up, by providing the relevant background and investigating and comparing real world scenarios and searching for common abstractions. The distribution solution chosen in a specific situation has to follow architectural styles that correspond to scenarios that cover products similar to the one that have to be distributed. With higher similarity cost effectiveness and better process support can be reached. A complete match with one architecture cannot be expected. Changes to software structure and installation mechanisms will proceed and a consolidation of current environments can be expected. More long-term the architecture models will utilize some domains that are currently covered with interfaces while the distribution models will be simplified and the responsibility for reliability and other features will be shifted to the operating system.

7 Appendix

The following diagrams show a complete picture of three architectures' logical views.

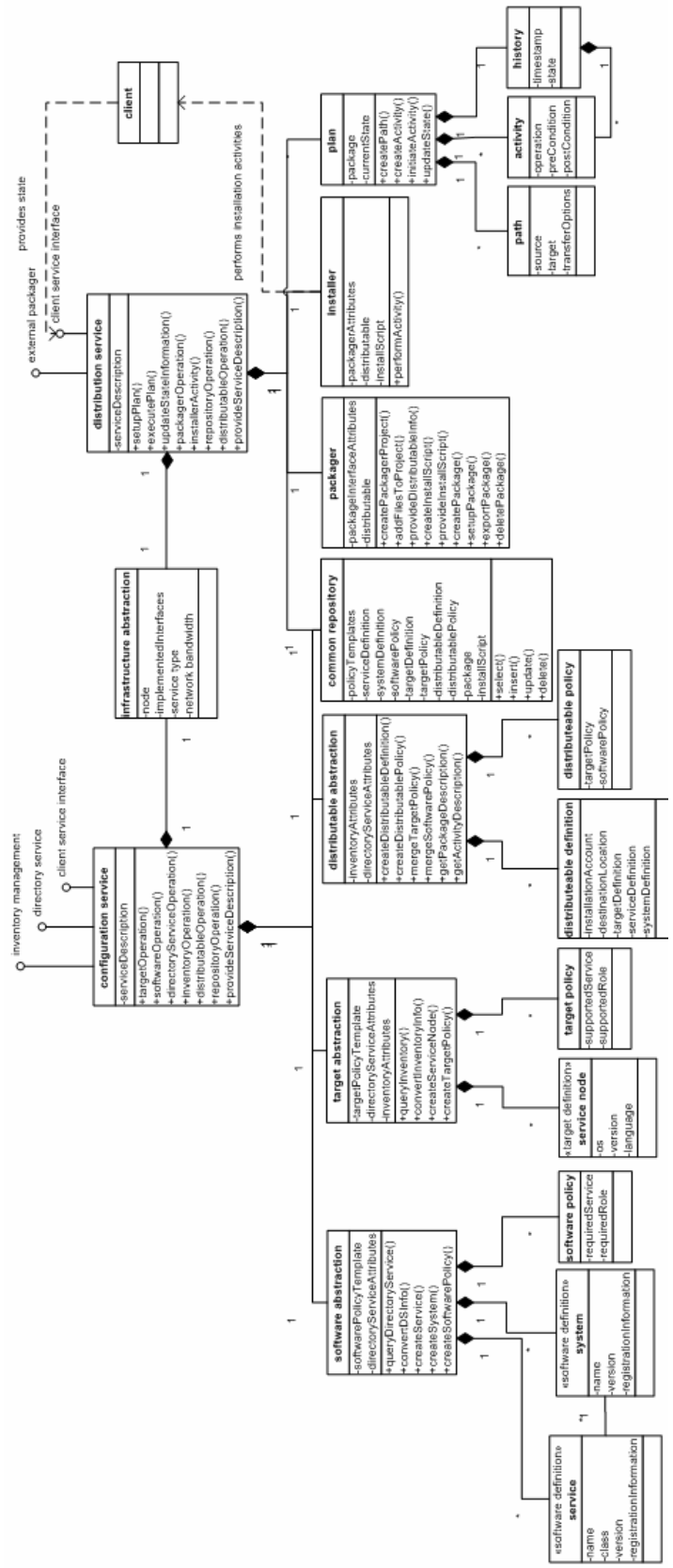

Figure 58: Client/Server model logical view

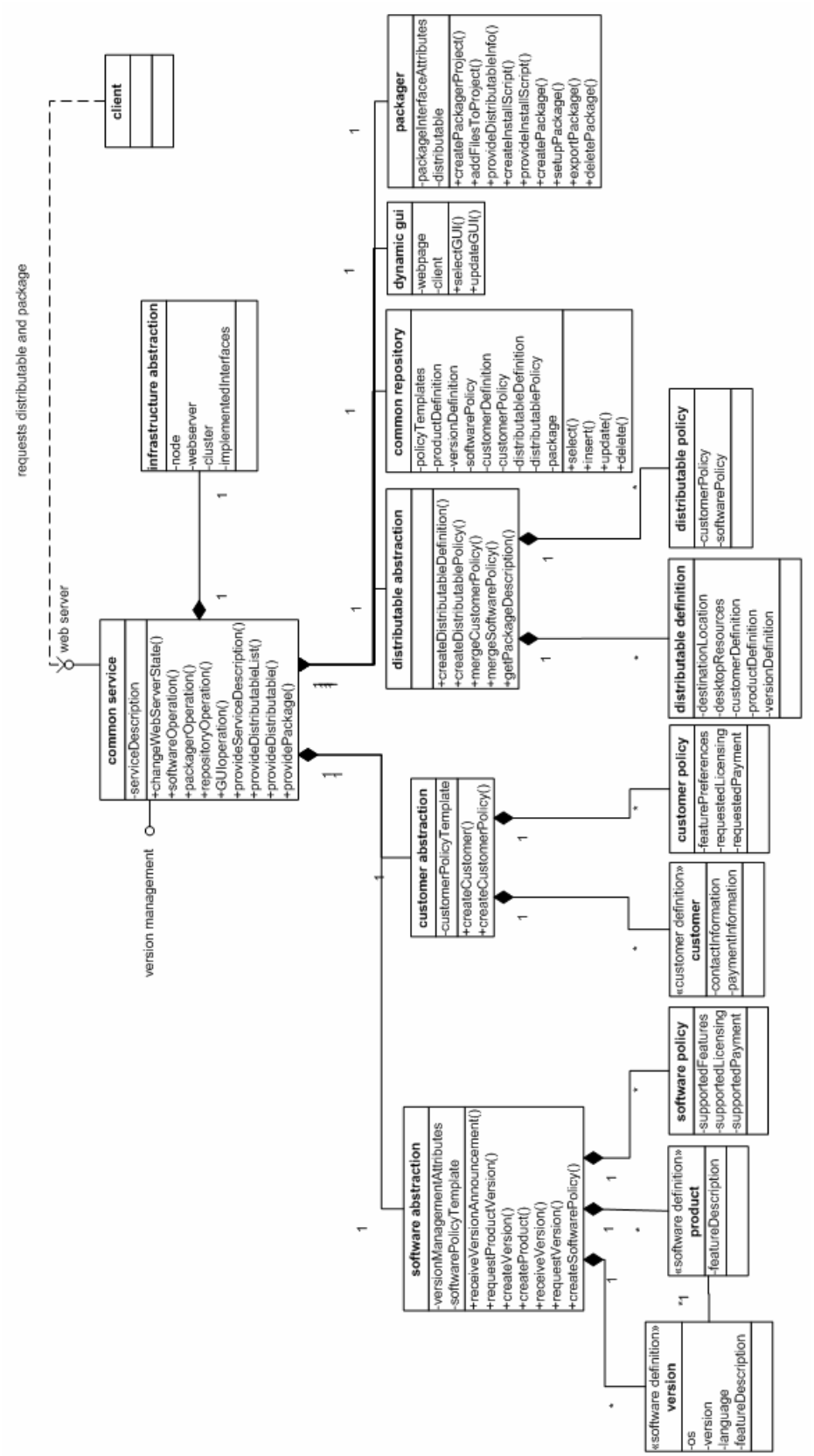

Figure 59: Standalone Software model logical view

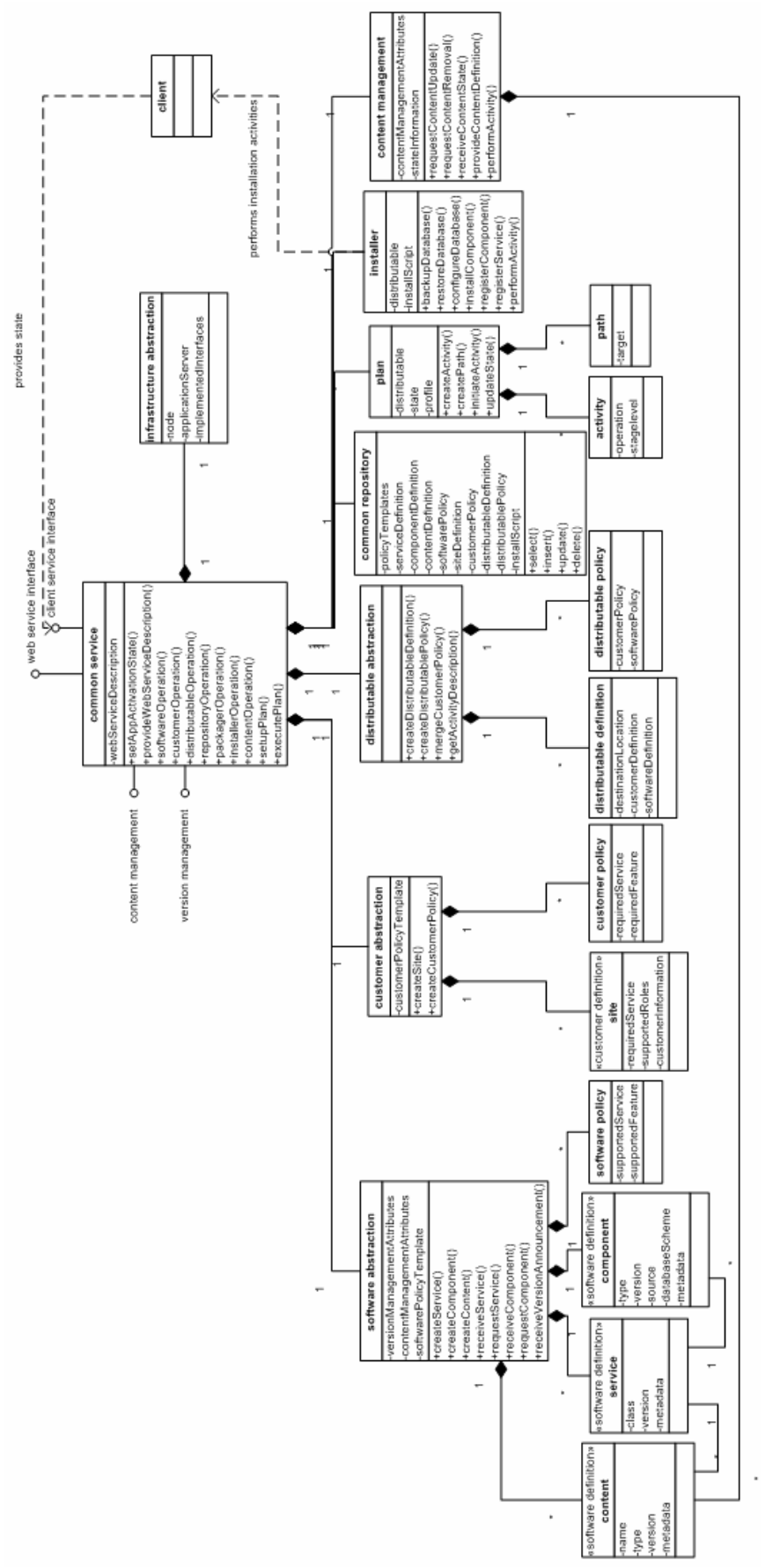

Figure 60: Web-service model logical view

8 Table of Figures

9 References

[1] **Dart,** *Concepts in Configuration Management Systems*, Carnegie-Mellon University, US 1992

[2] **Carnegie Mellon University,** *Configuration Management*, http://www.sei.cmu.edu/legacy/scm/, US 2001

[3] **Dart,** *Content Change Management: Problems for Web Systems*, http://www.stsc.hill.af.mil/CrossTalk/2000/jan/dart.asp, US 1999

[4] **IETF Network Working Group,** *RFC2518, HTTP Extensions for Distributed Authoring - WEBDAV*, http://www.ietf.org/rfc/rfc2291.txt, US 1998

[5] **Microsoft,** *System Management Server Reviewer's Guide*, http://www.microsoft.com/smsmgmt/default.asp, US 1998

[6] **InstallShield,** *InstallShield for Windows Installer Whitepaper*, http://www.installshield.com/isd/resources/default.asp, US 2001

[7] **Rick Hall,** *Software Dock*, http://www.cs.colorado.edu/serl/oldindex.html, US 1999

[8] **Rick Hall,** *A characterization framework for software deployment technologies*, http://www.cs.colorado.edu/serl/oldindex.html, US 1999

[9] **Object Management Group,** *Corba*, http://www.corba.org/, US 2001

[10] **Sun Microsystems,** *Enterprise Java Beans*, http://java.sun.com/products/ejb/, US 2001

[11] **Microsoft Corporation,** *DCOM*, http://www.microsoft.com/com/, US 2001

[12] **Microsoft,** *.NET Framework* http://www.microsoft.com /net, US 2002

[13] **SUN Microsystems,** *JAVA*, http://java.sun.com, US 2001

[14] **World Wide Web Consortium,** *SOAP (Simple Object Access Protocol)*, http://www.w3.org/TR/2001/WD-soap12-part1-20011002/, US 2001

[15] **IEEE,** *IEEE Guide to Software Configuration Management Standard 1042-1987*, http://standards.ieee.org/reading/ieee/std/se/1042-1987.pdf, US 1987

[16] **IEEE,** *IEEE Standard glossary of software engineering terminology*, IEEE Press New York, US 1990

[17] **IEEE,** *IEEE Standard for software Configuration Management plans*, IEEE Press New York, US 1990

[18] **MacKay,** *State of the Art in Concurrent, Distributed Configuration Management*, http://seg.iit.nrc.ca/English/index.html, Canada 1995

[19] **Telelogic,** *CM Synergy*, http://www.telelogic.com/products/synergy/cmsynergy.cfm, US 2001

[20] **Rational,** *Rational ClearCase product information*, http://www.rational.com/media/products/clearcase/, US 2001

[21] **Rick Hall,** *Specifying the Deployable Software Description Format in XML*, http://www.cs.colorado.edu/serl/oldindex.html, US 1999

[22] **Microsoft, Marimba,** *OSD*, http://www.w3.org/TR/NOTE-OSD.html, US 1997

[23] **IETF Distributed Management Task Force,**
Enabling your product for manageability with MIF files,
http://www.ietf.org, US 1994

[24] **Zeller,** *Configuration Management with feature logics,*
Computer Science Report 94-01, Braunschweig, Germany 1994

[25] **Vitali,** Using versioning to support collaboration on the WWW,
http://www.cs.unibo.it/~fabio/bio/papers/1995/WWW95/version.html, Italy 1994

[26] **Penner,** *Structured SCM environment using XML*,
http://www.cs.usask.ca/grads/rtp130/papers/predesign.doc, Canada 1998

[27] **Penner,** *Collection of resources for document management*,
http://www.cs.usask.ca/homepages/grads/rtp130/bookmark.htm, Canada 1998

[28] **University of Colorado,** *CM Today*,
http://www.cs.colorado.edu/users/andre/configuration_management.html, US 2000

[29] **InstallShield,** *InstallShield*,
http://www.installshield.com/products, US 2001

[30] **Rick Hall,** *Evaluating Software Deployment Languages and Schemes*,
http://www.cs.colorado.edu/serl/oldindex.html, US 1999

[31] **Shaw,** *Software Architecture: Perspectives on an emerging discipline*,
Prentice Hall, US 1996

[32] **SUN Microsystems,** *Java 2 Enterprise Edition*,
http://java.sun.com/j2ee, US 2001

[33] **SUN Microsystems,** *Java Beans*,
http://java.sun.com/products/javabeans, US 2001

[34] **WMI,** *Microsoft Windows Management Instrumentation*,
http://msdn.microsoft.com/library/default.asp?URL=/library/psdk/wmisdk/wmistart_5kth.htm, Microsoft, US 2001

[35] **Distributed Management Task Force,** *WBEM Web-based enterprise management*
http://www.dmtf.org/standards/standard_wbem.php, US 2001

[36] **Microsoft,** *System Management Server 2.0 Reviewer's Guide*,
http://www.microsoft.com/smserver/evaluation/overview/revguide.asp, US 1999

[37] **Perforce,** *Software Lifecycle Modeling*,
http://search.perforce.com/perforce/life.html, US 2001

[38] **Perforce,** *Inter-File Branching in Perforce*,
http://search.perforce.com/perforce/branch.html, US 2001

[39] **Perforce,** *High-level Best Practices in Software Configuration Management*,
http://search.perforce.com/perforce/bestpractices.html, US 2001

[40] **Perforce,** *Perforce 2001.1 User's Guide*,
http://search.perforce.com/perforce/doc.011/manuals/p4guide/index.html, US 2001

[41] **Perforce,** *Networked Software Development*,
http://search.perforce.com/perforce/wan.html, US 2001

[42] **Perforce,** *Web Content Management with Perforce*,
http://search.perforce.com/perforce/wcm.html, US 2001

[43] **Microsoft,** *Microsoft Source Safe Integration*,
http://msdn.microsoft.com/ssafe/, US 2001

[44] **Perforce,** *Perforce 2001.1 Administrator's Guide*,
http://search.perforce.com/perforce/doc.011/manuals/p4sag/index.html, US 2001

[45] **Lucent Technologies,** *NSBD Manual Page*,
http://www1.bell-labs.com/project/nsbd/nsbdman.html, US 1998

[46] **PGP,** *PGP 7.0 User's Guide*,
http://www.pgpi.org/doc/guide/7.0/en/, US 2001

[47] **Lucent Technologies,** *Up to Date Software with Not-So-Bad-Distribution Program*, http://www.bell-labs.com/project/nsbd/, US 1998

[48] **Telelogic,** *Untangling Configuration Management*, http://www.continuus.com/developers/developersACEC.html, US 1999

[49] **Telelogic,** *Telelogic SCM*, http://www.continuus.com/developers/developersACEB.html, US 2001

[50] **Telelogic,** *Change Sets Versus Change Packages*, http://www.continuus.com/developers/developersACEA.html, US 2001

[51] **Telelogic,** *Distributed Change Management*, http://www.telelogic.com/products/synergy/cmsynergydcm.cfm, US 2001

[52] **Telelogic,** *Task-based Configuration*, http://www.continuus.com/developers/developersACEA.html, US 2001

[53] **InstallShield,** *Developer Getting Started Guide*, http://www.installshield.com/isd/, US 2001

[54] **InstallShield,** *InstallShield Multiplatform Whitepaper*, http://www.installshield.com/isd/resources/default.asp, US 2001

[55] **InstallShield,** *AdminStudio Getting Started Guide*, http://www.installshield.com/isas/, US 2001

[56] **Tivoli,** *Tivoli User's Guide*, http://www.tivoli.com/support/public/Prodman/public_manuals/td/sw_dist/gc32-0651-01/en_US/HTML/sd40ug02.htm#ToC, US 2001

[57] **Tivoli,** *Tivoli Software Distribution Factsheet*, http://www.tivoli.com/news/press/pressreleases/en/2000/supplement/software_dist_factsheet.html, US 2001

[58] **Tivoli,** *Tivoli inventory*, http://www.tivoli.com/support/public/Prodman/public_manuals/td/inventory/GC31-8381-04/en_US/HTML/inv02.htm#ToC, US 2001

[59] **Tivoli,** *Tivoli Reference Manual*, http://www.tivoli.com/support/public/Prodman/public_manuals/td/sw_dist/GC32-0716-00/en_US/HTML/sdref02.htm#ToC, US 2001

[60] **Tivoli,** *Introduction to the Software Distribution process*, http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/, US 2000

[61] **Marimba,** *Marimba Management Solution*, http://www.marimba.com/products/intro.htm, US 1999

[62] **Marimba,** *Server Infrastructure Module*, http://www.marimba.com/products/change_management/server/infrastructure.html, US 2001

[63] **Marimba,** *Desktop Software Distribution Module*, http://www.marimba.com/products/change_management/desktop/software-distribution.shtml, US 2001

[64] **Marimba,** *Server Software Distribution Module*, http://www.marimba.com/products/change_management/server/software-distribution.html, US 2001

[65] **Marimba,** *Server Inventory Module*, http://www.marimba.com/products/change_management/server/inventory.html, US 2001

[66] **Marimba,** *Mobile Management*, http://www.marimba.com/products/change_management/desktop/desktop-infrastructure.shtml, US 2001

[67] **Marimba,** *Timbale White Paper*, http://www.marimba.com/products/whitepapers/Timbale_White_Paper.pdf, US 2000

[68] **ManageSoft,** *NETDeploy Technical Specification*,

http://www.managesoft.com/products/technical/index.xml, US 2001

[69] **Microsoft,** *Windows 2000*,
http://www.microsoft.com, US 2000

[70] **ManageSoft,** *Managing software for mobile and remote users*,
http://www.managesoft.com/products/mobile.xml, US 2001

[71] **Microsoft,** *Microsoft Operations Framework*,
http://www.microsoft.com/mof/, US 2001

[72] **Rational,** *An overview of Rational Suite ContentStudio*,
http://www.rational.com/products/cstudio/whitepapers.jsp, US 2001

[73] **Rational,** *SCM Specification Evaluation Guide*,
http://www.rational.com/products/whitepapers/scmseg.jsp, US 2001

[74] **Rational,** *Rational Suite*,
http://www.rational.com/products/entstudio/whitepapers.jsp, US 2001

[75] **Netscape,** *Netscape Directory Server 4.0*
http://home.netscape.com/directory/v4.0/index.html, US 1999

[76] **Novadigm,** *Radio Application Manager Factsheet*,
http://www.novadigm.com/products/radia/index.html, US 2001

[77] **Novadigm,** *Radio Software Manager Factsheet*,
http://www.novadigm.com/products/radia/index.html, US 2001

[78] **Novadigm,** *E-Wrap technology whitepaper*,
http://www.novadigm.com/products/radia/index.html, US 1999

[79] **Novadigm,** *Radio Inventory Manager Factsheet*,
http://www.novadigm.com/products/radia/index.html, US 2001

[80] **SUN Microsystems,** *Webstart*,
http://java.sun.com/products/javawebstart/, US 2001

[81] **SUN Microsystems,** *JNLP specification 1.0.1*,
http://java.sun.com/products/javawebstart/download-spec.html, US 2001

[82] **SUN Microsystems,** *JNLP products*,
http://java.sun.com/products/javawebstart/partners.html, US 2001

[83] **OMG,** *UML*,
http://www.uml.org, US 2001

[84] **Fowler,** *UML Distilled (Second Edition)*,
Addison Wesley, US 2000

[85] **World Wide Web Consortium,** *Channel definition format*,
http://www.w3.org/TR/NOTE-CDFsubmit.html, US 1997

[86] **Microsoft,** *System Management Server User's Guide*,
http://www.microsoft.com/smsmgmt/default.asp, US 1998

[87] **Distributed Management Task Force,** DMI Desktop Management Instrumentation,
http://www.dmtf.org/standards/standard_dmi.php, US 2001

[88] **Distributed Management Task Force,** *CIM Common Information Model*,
http://www.dmtf.org/standards/cim_spec_v22/, US 1999

[89] **Rick Hall,** *Agent-based Software Configuration and Deployment*,
http://www.cs.colorado.edu/serl/oldindex.html, US 1999

[90] **Vignette,** *Vignette Content Suite V6*,
http://a416.g.akamai.net/7/416/812/3d6380b3e5467e/www.vignette.com/Downloads/WP_VCS_v2.pdf, US 2001

[91] **Microsoft,** *Using Systems Management Server 2.0 with Windows 2000*,
http://www.microsoft.com/smsmgmt/SMS_Win2k.asp, US 2000

[92] **Sun Microsystems,** *Java RMI (Remote Method Invocation)*,
http://java.sun.com/products/jdk/rmi/index.html, US 2001

[93] **OMG,** *IIOP Internet Inter-ORB Protocol*,
http://www.omg.org/cgi-bin/doc?formal/01-12-53, US 1999

[94] **IBM,** *WebSphere*,
http://www-4.ibm.com/software/webservers/appserv, US 2002

[95] **IBM,** *MQ Series Workflow Concepts and Architecture*,
http://www-4.ibm.com/software/ts/mqseries/workflow/, US 2001

[96] **IBM,** *MQ Series Workflow Installation Guide*,
http://www-4.ibm.com/software/ts/mqseries/workflow/, US 2001

[97] **SUN,** *iPlanet*,
http://www.sun.com/software/iplanet, US 2001

[98] **World Wide Web Consortium,** *WSDL Web-service Description Language*,
http://www.w3.org/TR/wsdl, US 2001

[99] **Microsoft,** *The SOAP Toolkit*,
http://msdn.microsoft.com/library/default.asp?URL=/library/sdkdoc/soap/kit_intro_19bj.htm,
US 2002

[100] **World Wide Web Consortium,** *RFC 2616 Hypertext Transfer Protocol*,
ftp://ftp.isi.edu/in-notes/rfc2616.txt, US 1999

[101] **Hauswirth, Jazayeri,** *A Component and Communication Model for Push Systems*,
http://www.infosys.tuwien.ac.at/Staff/pooh/papers/PushIssues/, Austria 1999

[102] **Kruchten,** *The 4+1 View Model of Software Architecture*,
IEEE Software, US November 1995

[103] **Larman,** *Applying UML and Patterns*,
Addison Wesley, US 2001

[104] **Microsoft,** *Windows Installer*,
http://www.microsoft.com/msdownload/platformsdk/sdkupdate/, US 2002

[105] **Bass, Clemens, Kazman,** *Software Architecture in practice*,
Addison.-Wesley, US 2000

[106] **Coplien, Hoffmann, Weiss,** *Commonality and variability in Software Engineering*,
IEEE Software, US 1998