PodSystem: a management framework for customisable services

J. Walter

Coach: ir. R. Kersic (The industree B.V.)
Supervisor: prof.ir. M.P.J. Stevens
Date: February 2001
PodSystem: a management framework for customisable services

The industree B.V.

Master thesis of J. Walter
Author: J. Walter
Student ID: 361726
Eindhoven University of Technology
Department of Electrical Engineering
Research chair Information and Communication Systems (ICS)
Section Information processing Systems and Networks (EB)
Supervisor: Prof. ir. M.P.J. Stevens

Research conducted at
The industree B.V.
Daily supervisor: ir. R. Kersic
July-December 2000

Reviewers:
R. Kersic (industree), R. Pijpers (industree), M.P.J. Stevens (TUE)
PodSystem: a management framework for customisable services

Abstract
As part of the development of a new version of CablePilot, a framework for customisable services is being designed. This framework, the PodSystem, is based on the Java Management Extensions (JMX) [1] and consists of a collection of basic management services.

Although the PodSystem uses the Java Management Extensions (JMX) as its underlying technology, it has a rather different use for it than as intended by the JMX specification. Whereas the JMX specification places the focus on managing resources by means of MBeans, the PodSystem is focused on providing services to users. The most noticeable distinction of this approach is the user-based configuration and security management that is provided by the PodSystem, in contrast to JMX, which doesn’t provide any form of security management and only allows one configuration per MBean.

Consequently, the JMX specification and its reference implementation do not provide enough functionality for the PodSystem, though many useful concepts for managing services are introduced. Still, in order to facilitate in the implementation of the PodSystem, the JMX reference implementation has been chosen as the basis of the PodSystem, using those elements of JMX which are directly usable and replacing shortcoming parts of JMX with enhanced functionality.

By providing a set of standard management services, the PodSystem is able to relieve the developer of a Pod of a lot of responsibility and writing code needed for overhead functionality. This allows the developer to focus on building services, without having to worry about common tasks such as configuration management, security checking and notification handling.

By using the PodSystem as its basis, CablePilot not only is capable to be extended with new services for the management of CableFleet, but it also is able to interface with or manage resources other than those belonging to the CableFleet.
Table of contents

1 Introduction............................................................................................................ 10
2 Case studies.............................................................................................................. 12
  2.1 Installation and configuration of a cable modem ........................................... 12
  2.2 New management tasks .......... ................................................................ 13
3 Requirements of CablePilot ..................................................................................... 14
  3.1 Extensibility ............................................................................................... 14
  3.2 User based configuration and security ................................................................. 14
  3.3 Autonomous tasks ....................................................................................... 15
4 Service driven management..................................................................................... 16
  4.1 Service ........................................................................................................ 16
  4.2 Extensibility ................................................................................................ 17
  4.3 Manageability .............................................................................................. 18
  4.4 Maintenance ................................................................................................ 19
  4.5 Interoperability ........................................................................................... 19
5 PodSystem architecture ........................................................................................... 24
  5.1 PodServer ..................................................................................................... 24
6 The Pod ..................................................................................................................... 28
  6.1 Roles ............................................................................................................. 28
  6.2 Publication .................................................................................................... 29
  6.3 Configuration and security ........................................................................ 30
  6.4 JMX implementation ................................................................................... 31
7 Basic Pods ................................................................................................................ 34
  7.1 DeploymentPod .......................................................................................... 34
  7.2 DirectoryPod ............................................................................................... 35
  7.3 NotificationPod ............................................................................................ 36
  7.4 SchedulerPod .............................................................................................. 38
  7.5 MonitorPod .................................................................................................. 38
8 Extending the PodSystem ........................................................................................ 40
  8.1 Managing external Java components ............................................................ 40
  8.2 Distribution .................................................................................................. 40
  8.3 Stub interface ............................................................................................... 40
9 Conclusions .............................................................................................................. 44
  9.1 JMX-based implementation ...................................................................... 44
  9.2 CablePilot and PodSystem ........................................................................ 44
Appendix A: The industree B.V. and their products ................................................. 48
Appendix B: The Java Management Extensions ....................................................... 50
# Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CableDock</td>
<td>the head-end product from The industree B.V.</td>
</tr>
<tr>
<td>CableFleet</td>
<td>the range of products from The industree B.V.</td>
</tr>
<tr>
<td>CableJet</td>
<td>the cable modem product from The industree B.V.</td>
</tr>
<tr>
<td>CablePilot</td>
<td>the network management system from The industree B.V.</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise JavaBeans</td>
</tr>
<tr>
<td>ISP</td>
<td>Internet Service Provider</td>
</tr>
<tr>
<td>JAAS</td>
<td>Java Authentication and Authorisation Service</td>
</tr>
<tr>
<td>JSA</td>
<td>Java Security Architecture</td>
</tr>
<tr>
<td>JMX</td>
<td>Java Management Extensions</td>
</tr>
<tr>
<td>JNDI</td>
<td>Java Naming and Directory Interface</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>MBean</td>
<td>Managed Bean, used by JMX</td>
</tr>
<tr>
<td>NIC</td>
<td>Network Interface Card</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>SNMP</td>
<td>Simple Network Management Protocol</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>VoD</td>
<td>Video on Demand</td>
</tr>
<tr>
<td>WebNMS</td>
<td>AdventNet’s network management system</td>
</tr>
</tbody>
</table>
CablePilot
1 Introduction

As part of the CableFleet product line of The industree B.V. (cf. appendix A), the network management system CablePilot [10] provides the means to manage a collection of CableDocks and CableJets. It presents to a user an interface to the network devices via a web-based application using Java-Servlets. CablePilot uses Adventnet's WebNMS [9] as it underlying framework for managing the actual physical devices, i.e. the CableDocks and CableJets.

During the development of CablePilot, the need was expressed for a more capable version of CablePilot, that should be based on service driven management. Therefore, the assignment [11] was given to define the needed functionality of a service driven management system for CablePilot (chapter 4) and to explore how a possible implementation can be based on current Java technologies (chapters 5, 6 and 7). By means of an exploration of the possibilities of a future version of CablePilot (chapter 2), the limitations of the current version of CablePilot are identified (chapter 3).

This report presents the PodSystem (chapters 5, 6 and 7), a framework for customisable services, whose goal it is to implement the service driven management requirements of CablePilot (chapter 4). As the PodSystem is still in heavy development, not all aspects of the PodSystem are discussed. While only the basic functionality of the PodSystem is discussed, it provides an understanding on how the PodSystem may be used as the basis for a future version of CablePilot, without getting lost in implementation specific details. The PodSystem uses the Java Management Extensions (JMX) [1] as the underlying technology for its framework and discusses to what extent the functionality of JMX is applicable for the PodSystem. Consequently, this report assumes the reader is familiar with the Java Management Extensions (JMX) [1]. As a convenience, an excerpt from the Java Management Extension Whitepaper is provided in appendix B.
2 Case studies

To clarify the need for a more capable version of CablePilot, a few examples of new functionality of CablePilot are shown. Paragraph 2.1 shows how a subscriber of the cable provider uses systems, external to CablePilot, to access information offered by CablePilot. Paragraph 2.2 shows what new management capabilities are needed to allow for such functionality.

2.1 Installation and configuration of a cable modem

2.1.1 Goal
The easy installation of a CableJet cable modem by a subscriber and allowing a subscriber to choose the services he wishes to obtain from the cable provider.

2.1.2 Automatic CableJet Enabling: Install the cable modem
As an example, let there be a subscriber who wants access to the Internet through a cable modem and also wants to use the cable modem for telephony. The local cable provider sells or hires out these modems, CableJets, through their stores. The subscriber decides to buy a CableJet at one of these stores and in the store they record his name, address and account information. In return he is given the login information needed to access the AccountServer. Back at his home, he plugs in the CableJet in the TV-socket of the cable provider and connects the CableJet to his PC (via USB or NIC).

Choosing services
With the instructions he was given at the store, the subscriber connects to the AccountServer of the cable provider via an Internet browser. As no Internet service has yet been enabled for the cable modem, it automatically redirects every connection attempt to the AccountServer. After he logs in with his login code and password, the AccountServer shows him the services he can purchase or rent via the cable provider.

First, he chooses “Internet”. He then gets a choice of Internet service providers and he chooses the ISP and subscription he finds most suitable to his needs. After signing the agreement, he returns to the main menu. Here he is again presented with the services the cable provider offers. He chooses “Telephony”, and like the Internet service, he can choose between different telephony providers. However, for some reason he doesn’t want to choose a telephony provider right now, so he goes back to the main menu. Here he can make the choices that he made (ISP), permanent. As he does so, the modem is automatically configured by the cable provider for the services he purchased and available for direct use of those services. He now successfully has registered his modem with an ISP, his modem is fully functional, and he can start surfing the web.

2.1.3 Automatic CableJet Account Management: Enabling Telephony
After the subscriber successfully registered his modem with an ISP, he starts using the Internet. He still wants to sign on for a telephony account, so he surfs to the web site of the cable provider. Here he can access the same AccountServer he used before. The AccountServer authenticates and authorises him (and his modem) and directs him to the main menu. Here he chooses “Telephony” and is presented with a list of telephony service providers. He chooses the one most suitable for him and agrees to the contract. After going back to the main menu and making the changes to his account permanent, he is now able to use a telephone with his cable modem to call and receive calls.
2.1.4 Introducing new services: Video on Demand

The subscriber is now capable of using all the features of his cable modem, Internet and telephone. His cable provider, however, has more services to offer and sends him an email, offering a new service he can use with his cable modem: Video on Demand (VoD). The only things he has to do is to register himself at the AccountServer to enable the VoD-service and to download a small program for viewing the movies on his PC (VoD-player) or TV (via the extra output connector on cable modem, that has always intrigued the subscriber). After he has registered, he can use the VoD-player (or even a telephone) to select a movie and watch it.

2.2 New management tasks

The previous example described a scenario in which the installation and configuration of a CableJet is very easy and can be done by the subscriber himself. It also described how new subscriber services, like Video-on-Demand, can easily be configured by a subscriber. All of this is done via an AccountServer, which is a separate server and not a part of CablePilot. Although not part of CablePilot, the AccountServer does need functionality provided by CablePilot, e.g. communication with CableJets and -Docks. Furthermore, although not explicitly mentioned, all operations in the AccountServer as well as watching a movie via VoD or making a telephone call may require some form of metering and billing. This too is not part of CablePilot, but needs interaction with its services, e.g. connection monitoring and metering.

Not only is the AccountServer able to access CablePilot to obtain information via an external interface of CablePilot, CablePilot itself can manage the AccountServer in order to configure the AccountServer for correct operation with CablePilot. To this end, CablePilot contains special management services, which are able to interact with the AccountServer and which allow external systems to interface with CablePilot.

Systems like the AccountServer can be made accessible, and manageable, by deploying new management services into CablePilot. Via the deployment of new services in CablePilot, new functionality can easily be offered to users of CablePilot.

While the deployed services in CablePilot manage different aspects of the system and might even provide management functions for managing external systems, they themselves have to be managed too. This implies a distinction between a normal user of CablePilot, who uses the services to manage (external) resources, and an administrator, who configures the services for those users and gives users the permissions to use those services. Therefore, not every user needs to have the same configuration of a service. For instance, two users A and B might both use the same service for managing a collection of CableDock headends. However, while user A is allowed to manage all of the CableDocks, user B is only permitted to manage one particular CableDock.
3 Requirements of CablePilot

The examples of the previous chapter describe functionality of CablePilot which is not yet offered, but which is desirable for a future version of CablePilot. The future version of CablePilot must not only provide access to the CableFleet objects via an external interface, it must also provide a framework in which new management services can be injected in CablePilot itself. These services not only are able to access the already available services in CablePilot, but they are also accessible to management applications themselves. Via this construction, CablePilot is extendable with new (management) functionality.

In short, CablePilot will use service driven management as its basis. This means that CablePilot will have to be a system, in which services can exist, which not only provide management tasks for managing the CableFleet, but also provide services that can interact with external (enterprise) systems. Via these services external systems have access to the CableFleet objects, and thus are able to provide services directly to subscribers.

The current version of CablePilot does not offer such functionality. Based on Adventnet’s WebNMS network management system, it provides a web-based management interface to users via several servlets and http-pages.

Although the current CablePilot does a good job in managing the objects of the CableFleet, it has a few shortcomings, which are directly translatable to requirements:

- **Extensibility**: Extending CablePilot with new functionality is a tedious job.
- **User based configuration and security**: CablePilot is not able to provide a customised interface to an administrator nor does it allow for fine-grained control over the services, that it offers, based on who is using the system.
- **Autonomous tasks**: CablePilot does not provide the means to add new custom management services other than those of its underlying system, WebNMS.

3.1 Extensibility

The current version of CablePilot does not provide a way for external services to interact directly with CableJets and -Docks nor does it have the means to add management services for systems other than CableFleet. In CablePilot, new functionality must be offered via servlets. In order to make this new functionality available, it needs to be incorporated in the existing navigational structure, which will require adjustments and recompilation of several servlets, sometimes followed by a complete restart of the system.

A future version of CablePilot needs to be open and extendable, as one can not predict what kind of services will be installed in the future. CablePilot must be capable of deploying new (management) services in the system. Once deployed, these services are able to interact with already deployed services in CablePilot and vice versa.

Giving CablePilot the capability to insert new functionality, services, in a dynamic manner without recompilation or restart, not only facilitates in the development of new functionality, but it also allows CablePilot to be more customisable according to the wishes of the cable provider.

3.2 User based configuration and security

CablePilot is a static system, in which the available functionality is partitioned into servlets included in html pages. Security control in CablePilot is limited by granting or denying users access to entire pages, based on the access class of the user. In order to provide a customised interface to a user, the degree of security control and configuration options must be increased.
3.3 Autonomous tasks

Although WebNMS provides the capability of automating some management tasks, such as data polling and event handling, CablePilot is not capable of defining management tasks other than those described. By providing CablePilot with the capability of running its own autonomous tasks, automated management tasks can be performed.
4 Service driven management

In a future version of CablePilot, service driven management will be used to provide users of CablePilot with new services that will aid in the management of the devices of the CableFleet and the management of other, external systems. Service driven management in CablePilot means, that management tasks are contained in services, which are made available for management themselves.

Service driven management implies a system that will serve as a container of management services and that will provide for a way to manage these services. These services provide functionality to management applications but could in the future also be used to provide services directly to subscribers of the cable provider.

The container and a set of standard management services together form the framework for service driven management. The goals of this framework are:

- to provide a unified interface to the services,
- to provide most of the functionality required to manage the services,
- to provide most of the functionality required for user-based configuration and security management.

4.1 Service

A service is an application with a well-defined interface and functionality and exports its features, such as methods and attributes, for use by users. The behaviour of a service is controlled by modifying the features of a service. This is done on a per user basis, so that each user has a customised version of a service.

The features may be configured by an administrator and therefore a feature must be accompanied with a human readable description. The collection of all these descriptions constitutes the publication of a service and provides a detailed description of the features and working of the service.

Every user has a customised version of a service, which means that each user will have its own configuration of a service. Not every user will have the same permissions on a service: for example administrators should be able to modify all attributes of a service in order to configure it for a particular user, while an ordinary user can only use certain methods of a service and may not alter any attributes. These settings are the security settings and they act as a filter for the interface of the service as they determine which features are accessible to the user, i.e. the perceived interface of the service.

A service consists of the following features:

**Attributes**: Attributes are the main means by which a service is configured. An attribute consists of a value and of the type the value is in; for example an attribute can be of the type Integer and have the value 5.

**Methods**: The functionality of a service is accessible via its exported methods. Each method has a list of parameters.

**Constraints**: A constraint is a condition, which must be satisfied by the configuration of a service. If a constraint is not satisfied for a particular user, a security violation occurs and the system is expected to act accordingly by not allowing further access to the service for that user until the constraint is satisfied for instance by an administrator or an automated management service.

Constraints can be used to set boundaries on attributes, but other uses of constraints are not ruled out. Constraints may also be used as a precondition for methods, i.e. if a constraint is not satisfied for a particular method, then the user cannot invoke that method.
Notifications: An occurrence in a service, e.g. a state change, may be propagated to the system and other services by means of broadcasting a notification. This notification is sent to all parties that have registered as a listener with the service. Reasons for sending notifications are, amongst others, attribute changes or problems encountered in services.

4.1.1 publication
The publication of a service describes the features of a service in a human readable fashion. The goal of the publication is to provide a description of a service, which can be used by a management application to present an understandable graphical interface to the user of the service.

When a service is deployed, its publication must be made accessible to users. This way, a management application can obtain the publication of a service and provide a management interface to an administrator, who then may configure the service for a particular user.

4.1.2 user based configuration and security
Whereas the publication of a service described the interface and features of a service, the configuration of a service contains the customised values of those features. Via the configuration, the behaviour of a service can be customised for each user. Each user has a customised configuration of a service, that contains the values of the attributes, the constraints on the attributes, that must be satisfied, and the security context of the user, which states the permissions of the user for that service.

The configuration of a service must be stored in permanent storage, so it can be used in case the system is restarted and all services are reloaded.

4.2 Extensibility

Extensibility in CablePilot means that CablePilot can be extended with new services in an easy manner, without the need for recompilation or restart of the system.

4.2.1 deploy new services
When a new service is available for deployment, the system must instantiate the service and must perform initialisation operations to ensure that the service will be able to operate correctly in the system.
As part of the deployment process the system must publish the interface of the service, configure the service with its default configuration and set the default security settings of the service.
4.2.2 remove services
When a service is no longer needed, it may be removed from the system by an administrator. Removal of a service means removal of the program code of the service as well as its publication and configurations from the permanent storage.

4.3 Manageability

As CablePilot primarily is a management system, new services introduced in CablePilot must also be manageable. This means that a service has to provide a description of its features, including, but not limited to, its attributes and methods. This is done via the publication of the service.

4.3.1 central entry point
The system must be able to take control of the invocation process of services in order to perform management tasks, such as configuration management and authorisation. Therefore, a central entry point is needed.

This central entry point must be used by the services themselves as well as by external applications. Each access to the system, and thus each access to a service, must go through this entry point. Not only does this allow for automatic configuration and security management, but it also provides the system with the ability to deliver basic management services via a unified interface.

4.3.2 storage of the publication and configuration of services
The publication and configuration of the system and of all the deployed services must be stored in a permanent storage. This allows the system to restart or to recover from accidental crashes without losing the configurations of services.

4.3.3 modify configuration
Via the central entry point, only users with enough permissions are able to modify the configuration of a service for a particular user.

4.3.4 manage notifications
Via notifications, services are able to notify other services of changes in their state or other special occurrences. The system must be able to manage the notifications between services, automatically as well as manually. This means that the system must provide means for (automatic) notification routing between services and filtering of specified notifications for services. This relieves a service of the responsibility of
managing a list of listeners and filtering notifications for those listeners. Based on the default configuration of a service, notification routing may already be performed during the deployment phase of a service.

The notification capabilities of the system provide a subscriber/publisher relationship between services, as each service will receive the notifications of only those services that the service has been registered for as a listener.

4.4 Maintenance

As part of providing extensive management tasks, the system must be able to interact with services in an autonomous manner and perform operations to ensure the correct behaviour of services.

4.4.1 update services

Services may be updated with new code. This means that the program code of the service must be removed from the system and its new updated code must be loaded and deployed in the system. This must be done without restarting the whole system and without losing the functionality of the service for an excessive amount of time.

An updated service must be compatible in terms of its publication and configuration, as the service must immediately be operable once it is deployed.

4.4.2 monitor

An attribute or method may be bound by certain constraints. Failure to satisfy these constraints may result in erratic behaviour of a service. Therefore, the system must provide for monitoring services that are able to check the constraints and generate an alert when a constraint is not satisfied.

4.4.3 scheduler

In order to be able to automate management tasks, a scheduler must be available. This scheduler must be able to schedule the execution of tasks, such as a periodic self-diagnosis of a service.

4.5 Interoperability

Interoperability in CablePilot is twofold: internal and external. Internal interoperability means that deployed services are able to find and use each other, while external interoperability is achieved by two means.
External interoperability means making the system available to other systems via an own external interface, but it also means using other systems via their external interface. The latter can be achieved by implementing services that are able to use those external interfaces.

4.5.1 notifications
Services generate notifications when they want to notify other services of changes in their configuration or in other cases, for example when a problem is encountered. A service broadcasts its notifications to services that have registered themselves with the service.

A service not only generates notifications, but may also be the target of notifications generated by other services. If a service receives an notification, it will handle the notification in an appropriate way. For a service to receive notifications from another service, it has to register itself with that service. Once registered it has the status of listener.

4.5.2 lookup a service
If a service wants to obtain information about another service, like the names and descriptions of its attributes and methods, then the service must perform a look-up operation on the publication store. With this information, the service can then for example access methods exposed in the other service.

4.5.3 find services
If a service wants to use a service of a certain type, the service can look for services for which the configuration matches specified attributes. A search will result in a list of matching services. This allows a service to find and use services that match special conditions needed by the service.

4.5.4 use services
Once a service has obtained information about another service via a lookup call, it can use this information to access the attributes and invoke methods of the other service. It does this by using the central entry point, which provides methods for accessing attributes and invoking methods of other services.

4.5.5 external interface
The central entry point of the system also must provide an external interface, via which services are available for external (management) applications.
PodSystem
5 PodSystem architecture

In order to meet the requirements of CablePilot as described in chapters 3 and 4, a framework for managing services, the PodSystem, is being designed. The PodSystem is a framework consisting of a container for services, Pods, and a set of basic management services. The PodSystem uses the Java Management Extensions (JMX) [1] as the basis on which to build its management services. Via these management services the PodSystem not only relieves Pods of the responsibility of configuration and security management, but also provides means for Pods to actively react on events in their environment.

As noted in chapter 4, a service is an application with a well-defined interface and functionality. For the PodSystem this means that each Pod is accompanied by a publication, describing the interface of the Pod, and, for each user, configuration and security settings, which determines the functionality of the Pod for that particular user. However, as the JMX specification does not contain provisions for user-based security and configuration management, the PodSystem provides an advanced shell over JMX in the form of the PodServer. The PodServer replaces the MBeanServer of JMX as the central entry point of the system and provides automatic user-based security and configuration management.

As the PodSystem and its basic management Pods are still in heavy development, only the concepts of the PodSystem are explained in the following chapters. This will give an idea of the possibilities of the system without going too deep into implementation specific details.

First, the architecture of the PodSystem is explained, followed by the description of the Pod and the basic management Pods.

5.1 PodServer

The JMX framework uses an agent that acts as the liaison between an (external) management application and the MBeans contained by the agent. Every access to an MBean is controlled by the JMX MBeanServer, which thus provides a single point of entry for management applications as well as for MBeans.

![Figure 5.1 Diagram of a JMX Agent consisting of the MBeanServer and MBeans](image)
As the PodSystem is user-based, each access to a Pod must be preceded by security and configuration processing to ensure that a user has permission to use a Pod and, if so, that it is using the Pod with its own configuration.

Although Java includes security mechanisms in the form of JSA [2] and JAAS [3], the JMX specification doesn't define the use of those or any other security mechanism. Therefore, if access to an MBean should be controlled, this would have to be done by the MBean itself, which would result in a lot of overhead in each MBean and would basically be the same for each MBean. Not only the security is user based, also the configuration of a Pod is based on who is using it. Again, the JMX specification of an MBean does not take this into account, as it only allows one configuration of an MBean via its attributes. Therefore, the configuration of a Pod should not be handled by the MBean itself.

There are several solutions to this problem:
1. use a separate security and configuration manager object, which must be used by the MBean to enforce access control;
2. use an abstract MBean class that contains basic security and configuration functions, which handle the user based security and configuration management for the MBean;
3. make a custom implementation of the MBean server that provides the needed security and configuration management.

The first solution enables an MBean to directly use the JSA and JAAS security mechanisms, but it places the responsibility of handling security and configurations entirely with the developer. It also requires MBeans to be programmed according to a set of guidelines. As there are no means of verifying whether an MBean follows these guidelines without having the source code, this is an unacceptable solution.

The second solution provides a form of transparent security and configuration management, but at the cost of flexibility in the MBean, as it must extend the abstract class. Also, the abstract class should provide enough functionality to automatically provide security for new methods in the MBean, otherwise the same disadvantages of the first solution holds. This too reduces the flexibility of the MBean.

The third solution makes transparent security and configuration management possible, while maintaining the highest flexibility in an MBean. An MBean doesn't have to implement any security or configuration management functions, as this is all handled automatically by the MBeanServer. For each user the MBeanServer loads the appropriate security and configuration context and ensures that all access to the MBean takes place in this context.
The third solution seems like the logical choice: it provides transparent security and configuration management. However, in order to facilitate in the implementation of this solution a different approach to this solution is presented. Instead of subclassing the MBeanServer, a special MBean, the PodServer, has been designed, which takes over the role of the MBeanServer as the central entry point.

Via the PodServer the PodSystem and its Pods are accessible not only to internal objects such as other Pods, but also to external applications via a remote interface using Java’s Remote Method Invocation [7]. This allows the PodSystem to present a unified interface to all users, internal as well as external. After security and configuration management has taken place in the PodServer, the Pod is accessed by relaying the invocation to the MBeanServer.

Not only security and configuration management is provided by the PodServer, the PodServer also acts as a façade [8] for the Pods that provide basic management services. By placing the basic management services in Pods, the implementation of these services can be updated or replaced, without the need to update the code of the PodServer or other Pods.
6 The Pod

The Pod is the PodSystem implementation of the service as defined in chapter 4. A Pod is implemented as an MBean and its methods and attributes are made accessible via the PodServer to internal as well as external users.

![Figure 6.1 Invocation of a Pod by an internal user (Pod B) as well as an external user (Management application in a different Java Virtual Machine)](image)

6.1 Roles

Looking at the intended use of the PodSystem, namely as the framework for CablePilot, the Pods that will be used in CablePilot can be divided into different roles, each with a specific kind of functionality. These roles do not restrict the possible functionality of a Pod, but they make it easier to understand the intended nature of Pods in CablePilot, namely Pods for management purposes.

A Pod is a combination of one or more of the following roles:

- **Application provider**: autonomous task, that performs background processing without user intervention
- **Service provider**: provides methods for use by users
- **Configuration manager**: manages the configuration changes of an (external) object and provides methods to propagate this information to the managed object and back to the management system.

6.1.1 Application provider

The main reason why JMX is chosen as the basis of the PodSystem is the independent manner in which MBeans exist in the system. MBeans are not limited in their capabilities in the same way as EJBs [4] are. EJBs live in a sandbox environment, which does not allow them to use certain features of the Java language such as using threads or listening on sockets, while MBeans don’t have these restrictions.
Because of this freedom, application provider Pods are able to perform operations in the PodSystem without user interaction, providing a degree of intelligence in the system. They are therefore also called autonomous Pods and are a valuable addition to the management system. There are two types of autonomous Pods:

1. Real autonomous Pods: they initiate their own tasks based on observations they actively make in the environment, e.g. actively monitoring by periodically polling a service.
2. Semi autonomous Pods: their tasks are initiated as a response to notifications sent by other Pods, e.g. a Pod that performs additional configuration tasks when another Pod is deployed, or an event logging Pod. Although not autonomous in the true sense of the word, they don't require direct instigation from a user to perform their tasks.

Typically, an application provider Pod starts its task, when it is deployed, and stops its task, when it is removed from the system. Like all Pods, it is configurable via its configuration. However, because the execution of the Pod is not initiated by a user, but by the PodSystem deployment process, the Pod needs to have its own, private configuration and security context, i.e. it must have its own user account.

### 6.1.2 Service provider
A service provider is a Pod that offers a service in the form of its methods to the users of the PodSystem. As every invocation of the service provider is initiated by a user, this allows the behaviour of the service to be configured for each user separately. This is done via the configuration of the user for that Pod.

### 6.1.3 Configuration manager
A configuration manager is a Pod, that handles configuration changes of an (external) managed object, such as an EJB, a servlet or even another Pod. It acts on behalf of the component when changes in the configuration of the component are made. A configuration manager may also be a factory of the managed object, for example it may be a factory of an EJB Home object or an EJB Remote object. As every Pod has a configuration, it should also have a configuration manager. Typically, this is the Pod itself.

### 6.2 Publication
In order for a management application to present an understandable management interface of a Pod to a user, the features of the Pod are described in the publication of the Pod. In JMX an MBean is described by its MBeanInfo object, which is either supplied by the MBean itself or by the MBeanServer via introspection.

Introspection by the MBeanServer is not a viable option to get a description of a Pod. Because not all of the features of a Pod can be described in the MBeanInfo object, introspection by the MBeanServer will not completely describe the Pod. For instance, constraints on attributes are not part of an MBean, and therefore will not show up via introspection. Furthermore, introspection by the MBeanServer only supplies a basic standard description of an attribute or method, because it only has information about the types of an attribute or method.

In order to supply a better description of an MBean, the JMX specification defines the DynamicMBean interface. Implementing this interface enables an MBean to supply its own description in the form of an MBeanInfo object to the MBeanServer. However, this places the responsibility of supplying a description with the MBean and does not allow the description to change without recompilation of the MBean.

In order to relieve the developer of a Pod of the responsibility of writing tedious code for filling the MBeanInfo object and in order to provide the system with the ability to dynamically change the description of a Pod, the publication of a Pod is not part of the Pod itself, but is stored in the external storage via the
DirectoryPod. The publication of a Pod contains descriptions of the features of the Pod, which are the attributes, the methods and the notifications.

6.2.1 Attributes
The attributes of a Pod are like the attributes of a normal MBean: they allow the behaviour of a managed resource, in this case the Pod, to be configured. For each attribute, the publication contains a description, which state the following:

- the name of the attribute
- the type of the attribute
- the purpose of the attribute
- the effect of altering the attribute
- the way the attribute may be constrained

6.2.2 Methods
Methods are the means of a Pod to offer its services to a user. The description of a method also contains the descriptions of the parameters of the method. The description of a method includes:

- the name of the methods
- the effect of the method, i.e. the service offered by this method
- the precondition of the method
- the postcondition of the method

and for each parameter of the method:

- the type of the parameter
- the purpose of the parameter
- the way the parameter may be constrained, as part of an imposed precondition

6.2.3 Notifications
The publication of a Pod contains a list of notification types that the Pod is able to listen to and a list of notifications that the Pod broadcasts. This allows other Pods, for instance the NotificationPod, to register themselves with the Pod as listener for its notifications, and it allows for the automatic routing of notifications to this Pod by the NotificationPod.

6.3 Configuration and security
Every user of a Pod is assigned its own configuration of a Pod. The configuration of a Pod contains the customised features of the Pod, i.e. the values of the attributes. The configuration also contains the security settings of the user, i.e. the permissions and the constraints the user has for that Pod. The configuration is managed by the DirectoryPod and made accessible via the PodServer.

6.3.1 Attributes
Whereas the attributes of an MBean are contained in the MBean itself, the attributes of a Pod are part of the user-based configuration of the Pod and are stored in the external storage of the PodSystem. Consequently, they are therefore only accessible via the PodServer.

6.3.2 Permissions
As part of the user-based configuration of a Pod, the security settings state the permissions of the user for the features of a Pod. The permissions determine whether a user is allowed to read or write an attribute and whether a user is allowed to invoke a method of a Pod. The PodServer maintains security in the PodSystem by constantly checking the permissions and the constraints of a user with each invocation the user makes.
Consequently, the PodServer provides transparent security and configuration management and relieves the developer of a Pod or an external application of writing code for security and configuration management.

6.3.3 Constraints
As part of the security settings of a user, the PodSystem allows attributes and methods to be constrained. A constraint specifies to what extent an attribute can be changed or under which conditions a method may be invoked. With constraints, the PodSystem has a fine-grained control over the amount of freedom the user has when accessing methods or modifying attributes.

For attributes this means, that a user cannot modify an attribute if the result would invalidate the constraint. For example, a constraint on an attribute of type integer could be, that the value of the attribute must not be larger than 100. The user then, can not assign a value larger than 100 to the attribute.

As far as methods are concerned, constraints provide a form of precondition checking. If a user tries to invoke a constrained method, the PodSystem checks whether the constraint is satisfied, before the method is invoked.

6.3.4 Descriptor file
Every Pod has a descriptor file. This file contains the publication of the Pod and the default configuration and security settings. During the deployment of the Pod, this file is parsed and the information contained is used to initialise the Pod and make it known in the PodSystem.

6.4 JMX implementation
As the Pod will be implemented as an MBean, it is important to know what restrictions are imposed on the Pod and its JMX environment.

The JMX specification states, that an MBean must either implements its own corresponding MBean interface or the DynamicMBean interface. If the MBean implements its own MBean interface, the MBeanServer will use introspection on the MBean in order to determine which attributes and methods are available for management applications. If the MBean implements the DynamicMBean interface, the MBeanServer will directly relay incoming requests to the MBean via the attribute and invocation methods of the DynamicMBean interface. The PodServer makes use of this functionality of JMX and relays invocations of Pods to the MBeanServer after configuration and security checking have taken place.

6.4.1 PodMBean interface
In order to facilitate the construction of a Pod for the developer, an interface will be made that encapsulates the basic functionality a Pod must implement.

```java
public interface PodMBean {
    void setPodServer(PodServer pserver);
}
```

The method setPodServer is called by the DeploymentPod when the service is deployed. This method is necessary, because only via the PodServer a Pod may access its configuration and use other services.

6.4.2 PodRegistration interface
The MBeanRegistration interface of JMX allows an MBean to exercise control of its registration and un-registration in the MBeanServer. A Pod, however, may never implement this interface, because the preReg-
ister method of the interface supplies the MBean with a reference to the MBeanServer. This is not allowed, because it allows an MBean to access other MBeans without security checking. Therefore, a Pod must only interact with the PodServer to initiate communications with other Pods and, thus, it must not implement the MBeanRegistration interface. During the deployment phase of a Pod, the DeploymentPod determines whether a Pod implements the MBeanRegistration interface and if it does, the Pod will not be deployed.

Still, the concept of the MBeanRegistration interface is useful and therefore it is modified for use in the PodSystem. A new interface PodRegistration is constructed, which mimics the behaviour of MBeanRegistration, but is used by the DeploymentPod instead of the MBeanServer.

```java
public interface PodRegistration {
    PodName preRegister(PodServer pserver, PodName name)
        throws java.lang.Exception;
    void postRegister(java.lang.Boolean registrationDone);
    void preDeregister()
        throws java.lang.Exception;
    void postDeregister();
}
```

**preRegister**
The DeploymentPod will invoke this call-back method before registering the Pod. The Pod will not be registered if any exception is raised by this method. This method may be used to:
- Allow a Pod to keep a reference on the PodServer, so that it can look up its own configuration and use other Pods.
- Perform any initialisation that needs to be done before the Pod is exposed to management operations and clients.
- Perform semantic checking on the Pod’s name, and possibly provide a name if the Pod was created without a name.
- Get information about the environment, for instance, check on the existence of services the Pod depends on. When such required services are not available, the Pod might either try to instantiate them, or raise an exception.

**postRegister**
Allows the Pod to perform any operations needed after having been registered in the PodSystem and the MBean server or after the registration has failed.

**preDeregister**
This method is called by the DeploymentPod before the Pod is removed from the PodSystem. It allows the Pod to perform clean up operations such as freeing resources and stopping threads. A Pod may prevent undeployment by throwing an exception in this method, however, the DeploymentPod can overrule this if it’s necessary.

**postDeregister**
This method is called by the DeploymentPod after the Pod has been de-registered from the MBean server and unpublished from the DirectoryPod. It allows the Pod to perform any operations needed after having been undeployed in the PodSystem.
6.4.3 Concurrency

Unlike the concept of Enterprise JavaBeans [4], where every user is assigned its own instance of an EJB, JMX does not provide such kind of component management. Instead, every Pod will be instantiated only once, meaning that every user uses the same instance of a Pod. This requires that Pods are designed with concurrency in mind and, thus, that they must provide their own synchronisation mechanism to shield critical parts of their code.

As an alternative to this, the PodSystem could provide a form of access control to the Pods, allowing only one user to use a Pod at a time or, more complex, to allow multiple instances of a Pod. This, however, is beyond the scope of this phase of the definition of the PodSystem.
7 Basic Pods
As stated in chapter 5, the PodSystem provides a set of basic management Pods in order to manage the PodSystem and its Pods. In order to provide a unified interface to the PodSystem, these basic Pods are accessible via the PodServer only. The basic Pods not only provide functions that allow services to find and communicate with each other, but also provide extended services, like a scheduler and a monitoring service.

The basic Pods that constitute the PodSystem are the following:
- DeploymentPod: controls the life-cycle of a Pod
- DirectoryPod: provides access to the stored information of the PodSystem. Also provides configuration and security management
- NotificationPod: provides routing and filtering of notifications
- SchedulerPod: provides task scheduling
- MonitorPod: provides functions for observing the state of Pods

7.1 DeploymentPod
Although the JMX specification already contains a deployment mechanism for MBeans via methods of the MBeanServer, it does not provide for security checking and other features, such as updating already deployed MBeans with new code and automatic configuration of the features of the MBeans. Therefore a special Pod, the DeploymentPod, is being developed.

The purpose of the DeploymentPod is to control the lifecycle of a Pod, which means that it is responsible for deploying, updating and removing Pods.

7.1.1 Deployment
Deployment of a new Pod consists of the following important tasks:
1. instantiation of the Pod
2. publication of the Pod
3. configuration of the Pod with a default configuration and default security settings

Instantiation
The DeploymentPod enhances the deployment mechanism of the MBeanServer by performing advanced class loading and instantiation of the Pod before it is registered with the MBeanServer. Although the DeploymentPod could delegate the actual class loading and instantiation of the Pod object to the MBeanServer, it still provides its own class-loading scheme. This is done, because replacing the code of a class with new code requires a special class-loading and object destruction scheme, which is not provided by the MBeanServer.

Publication
The DeploymentPod is responsible for reading the publication from the descriptor file of the Pod. Once the publication is read from the file and checked for consistency, it is passed to the DirectoryPod, which will store the publication in the permanent storage.

Configuration
In order to make the Pod ready for use, the DeploymentPod uses the configuration stored in the descriptor file to create a default configuration and security settings. If the user is not assigned a customised configuration and security for this Pod by an administrator, it will receive a copy of the default configuration and security settings. The DeploymentPod passes the routing and filter information for the Pod to the NotificationPod.
7.1.2 Removal
When a Pod is no longer needed in the PodSystem, it may be removed by the DeploymentPod. As removal is the reverse of deployment, this not only means the unloading of the Pod classes, but also the removal of the publication and configurations of the Pod from the permanent storage.

The DeploymentPod takes care of the unregistering of the Pod from the MBeanServer and the actual unloading of the classes of the Pod. As with the deployment process, the DeploymentPod delegates the removal of notification routes, publication, configurations and security settings to the corresponding Pods.

7.1.3 Update
When new code for a Pod is available, the DeploymentPod is able to update classes of the Pod with the new classes. The DeploymentPod ensures that the Pod is deactivated and that the Pod classes are replaced by their newer versions. As Java and JMX don't have a standard mechanism of removing obsolete code from the Java Virtual Machine, the DeploymentPod uses a special class-loading and object destruction scheme to ensure, that a class is completely removed from the JVM and that its new code can be loaded.

7.2 DirectoryPod
As the PodSystem does not use JMX functionality to manage the information of Pods, i.e. their publications and configurations, a new Pod, the DirectoryPod, is being designed to fulfil the need of persistent information management. The main function of the DirectoryPod is to be the manager of the stored information of the PodSystem. As the DirectoryPod is the only Pod that has direct access to the stored information, all other Pods use the DirectoryPod to retrieve and store information from the storage. Therefore, the DirectoryPod also is the security manager.

7.2.1 Information storage
Whereas the PodServer is the central entry point for users to access the functionality of the PodSystem, the DirectoryPod is the central point of accessing the information of the PodSystem. The DirectoryPod contains for each Pod its publication, and for each user its profile, i.e. the collection of configurations of the Pods for that user. When a Pod wants to access its attributes, it uses, via the PodServer, the DirectoryPod to retrieve the value of the attribute in the configuration of the user of the Pod. Likewise, when a user, or management application, wants to obtain information about a Pod, it retrieves the publication of the Pod from the DirectoryPod.

The DirectoryPod contains methods for managing the publication and configurations, which includes for each Pod methods to:
- retrieve, modify, publish and remove a publication,
and for each user methods to:
- store and remove an entire configuration of a Pod,
- copy a configuration from another user, for instance the default configuration,
- retrieve and modify attribute values per configuration.

7.2.2 Query the information
For Pods and users to use other Pods, they first have to know of the existence of the Pod and its interface. The DirectoryPod therefore exports methods to query the information of the PodSystem. These methods allow a user not only to retrieve the publication of a Pod, but also to find Pods, which are in accordance with specified features or values of features.
Although the extent in which the information of the PodSystem can be queried is not yet determined, it is clear that, in order to be as flexible as possible, queries must be based on a structured query definition. Examples of these are the JMX query mechanism and the SQL used in many relational databases.

7.2.3 Security manager
An important aspect of the DirectoryPod is its role as the security manager of the PodSystem. Instead of providing a separate object for Pods to use in order to maintain the security in the PodSystem, the security manager is integrated with the DirectoryPod. As such, the DirectoryPod is able to provide automatic permission checking and constraint validation. Every access to the PodSystem by a user has to be authorised, therefore the PodServer first calls the DirectoryPod to check whether the user has enough permissions to perform the requested access.

Not only the permission on a feature of a Pod is checked, the DirectoryPod is also responsible for validating the constraints that may be imposed on certain features for the user concerned. By placing constraints on a feature of a Pod, a more fine-grained control of permission-based access is provided. When a constraint is not satisfied, the user is denied access to the feature. This construct allows a permission on a feature to be conditional, and even to be dynamic, as the content of the constraint could be under control of another process.

In order to manage the configuration of a user, the DirectoryPod exports methods to perform the following actions:

- retrieve, modify, add and remove security settings,
- retrieve, modify, add and remove constraints,
- retrieve, modify, add and remove user information.

7.2.4 External storage
The information of the PodSystem is stored by the DirectoryPod into an external storage by using JNDI [6]. Using JNDI has the advantage that the underlying technology for the actual storage is independent of the interface needed to access the storage. This provides the PodSystem with the ability to adopt a different storage medium when needed.

7.2.5 Notifications
In its capacity of security manager, the DirectoryPod will broadcast security notifications to notify interested parties of denied access attempts or failed authentication processes, for instance.

The DirectoryPod may also broadcast many other notifications, for example it broadcasts a notification for each attribute change. In order to facilitate in the handling of these notifications, it depends on the NotificationPod for automatic routing and filtering of its notifications.

7.3 NotificationPod
Notification handling is a well-defined mechanism in the JMX specification and provides a way for MBeans and management applications to respond to events in MBeans. An MBean that wishes to broadcast notifications has to implement the NotificationBroadcaster interface, which allows the addition of listeners and sending of notifications to these listeners. An MBean that wants to receive notifications has to implement the NotificationListener interface. The MBean then adds itself as a listener to the broadcasters it wishes to receive notifications from. If a listener wants to filter certain notifications, it can specify a filter during registration with the broadcaster. All notifications that do not match the filter, do not arrive at the listener.
As notifications are meant to be explanatory messages to inform an object, e.g. management application, of certain occurrences in a Pod, they should be easy to generate and require as little overhead as possible from the Pod. The current JMX notification model, however, has several disadvantages that make it an unacceptable model in the PodSystem:

1. The MBean itself is responsible for sending a notification to all of its listeners. This means the MBean has to manage a list of listeners and has to invoke the notification handler of each listener. Not only does this place a lot of overhead at the MBean, but, more importantly, it also blocks the MBean while the listener handles the notification. The latter is totally unacceptable, as it disturbs the normal flow of execution of a service and can even be the cause of a malfunction of the service, when the notification handler of the listener takes too long to return or, more disastrously, raises an uncaught exception.

2. The broadcasting MBean is also responsible for filtering the notifications that are sent to a listener. The listener, therefore, has no other choice than to trust the MBean to perform this filtering. This dependency is not acceptable in the PodSystem, as neither the listener nor the management system have any control over the messages it may receive.

3. The listener cannot specify which method should be called for which type of received notification. Only one method is available as notification handler, making it once again the responsibility of the MBean to filter incoming notifications.

To overcome these disadvantages, while still making use of the notification model, a special Pod is being designed, the NotificationPod. Its purpose is to provide “fire and forget” notification handling for Pods that broadcast notifications and acts as a router and filter for notifications.

### 7.3.1 Router

As the main disadvantage of the JMX notification model is the dependence of a broadcaster on the fast and correct execution of the notification handlers of its listeners, the NotificationPod solves this problem by acting as a message queue between the broadcaster and its listeners. All notifications sent to the NotificationPod are placed in a queue and processed in a separate thread. This ensures that the notification handler of the NotificationPod does not block the broadcasting Pod for a longer time than necessary.

![Figure 7.1 Separation of threads in the NotificationPod](image-url)
system. When a broadcasting Pod is registered in the PodSystem, the NotificationPod will automatically add itself as a listener to the Pod, and when a listening Pod is registered, the NotificationPod will automatically add the listening Pod to its list of listeners.

### 7.3.2 Filters

Notification filtering with the NotificationPod is done by specifying the type and the origin of notifications which must be filtered (deny filter) or let through (grant filter) for a Pod. Also, the method which should be called as the notification handler can be specified. The NotificationPod then filters incoming notifications and lets through only those notifications that match the filters. For example, a Pod A that wishes to receive attribute change notifications of Pod B would register a grant filter with the NotificationPod, containing the type `attribute change`, the origin `Pod B` and the method `handleAttributeChange` as the notification handler.

This not only relieves the broadcasting Pods of the responsibility of filtering the notifications, but it also allows management of the filters, as filters can now dynamically be added to and removed from a Pod by a management application.

### 7.4 SchedulerPod

JMX provides a timer MBean that sends out an alarm at a specified time that wakes up all the listeners registered to receive timer notifications. Notifications can be added to the timer MBean and will be sent to all listeners at a specified time. The timer MBean is nothing more than a scheduler for notifications and its usability is limited to objects which implement the NotificationListener interface. This provides an adequate form of scheduling in a management environment as intended by JMX, where each MBean manages an independent resource and doesn’t interact with other MBeans. An MBean is expected only to need a timer to periodically trigger an event in order to facilitate in the management of the resource associated with the MBean.

Although this form of scheduling could be used by the PodSystem to time the execution of tasks, the JMX reference implementation is not usable, because it does not use the security mechanism of the PodSystem and would result in tasks being executed in an undefined security context. In order to overcome this problem the PodSystem uses its own scheduler, the SchedulerPod. Instead of managing a list of notifications, as does the timer MBean, the SchedulerPod manages a list of tasks.

Tasks are executable objects which implement the Runnable interface and which will be assigned their own thread of execution and security context. The SchedulerPod is responsible for the timely execution of a task and manages the threads in which the tasks are executed.

Because tasks are independent objects and therefore not necessarily part of a Pod, they may exist in the system without a management module. It would, however, be wise to have a Pod manage the configuration of a task. This Pod then acts as a configuration handler and possibly as a factory of the task, while the task uses the configuration of the Pod for its own purposes. By using tasks, Pods are able to appear autonomous without having to worry about thread management; a Pod can even be a task itself by implementing the Runnable interface and adding itself to the SchedulerPod, making this Pod an autonomous Pod.

### 7.5 MonitorPod

As part of the JMX specification, a model for observing attributes of MBeans is defined in the form of MonitorMBeans. MonitorMBeans provide a useful management service for inspecting normal MBeans by periodically polling the value of the attribute of MBeans. They are, however, not usable for monitoring the configuration of a Pod. Pods do not use the attribute mechanism as specified by JMX, but, instead, use the configuration manager, i.e. the DirectoryPod, of the PodSystem to access their configurations.
Another drawback of the monitor model of JMX is that a MonitorMBean can only monitor a single attribute of an MBean. In case of many monitored items, this would require many MBeans in the system and, more importantly, it would present an administrator with an unstructured collection of management interfaces of the monitors. In order to facilitate in the management of monitoring, the PodSystem provides its own monitor management via the MonitorPod. The MonitorPod manages a collection of monitor objects and provides a uniform interface for managing these monitors. Unlike the JMX monitor MBeans, the monitors used by the PodSystem are not limited to monitoring the attributes of a Pod on a periodic basis.

The MonitorPod manages two kinds of monitors, active and passive monitors. Active monitors are like the monitors of JMX, in that they observe the state of a feature of a Pod in a periodic manner. Active monitors are useful when a method of a Pod must be invoked periodically to perform a self-diagnosis. Passive monitors are notification listeners and only react to specified notifications. Passive monitors are useful for monitoring the value of an attribute, if a change in the attribute results in a attribute change notification.

When a monitored item does not satisfy its constraints, the MonitorPod sends a notification to the NotificationPod. The MonitorPod provides methods with which monitors may be created, modified and removed. The MonitorPod takes care of creating the actual monitor objects, performs scheduling of active monitors with the SchedulerPod and registers the passive monitors with the NotificationPod.
8 Extending the PodSystem

The PodSystem as presented in the previous chapters provides a framework on which an extensive management application can be build. This chapter describes how the PodSystem can be extended with extra services that provide management of external resources and with services that will make the PodSystem a distributed management server.

8.1 Managing external Java components

One of the roles of a Pod is that of configuration manager for components which are not part of the PodSystem themselves. Special Pods can facilitate in managing these components by providing methods to control the lifecycle of a component, e.g. deployment, and methods that act on behalf of the component when changes in the attributes of the component are being made or when notifications are sent. Examples of such components are Java Servlets [5] and Enterprise JavaBeans [4].

A characteristic property of an Enterprise JavaBean (EJB) is its inability to exercise control over its configuration parameters. During the deployment phase of an EJB the application developer determines its parameters. While this in itself may seem a good idea, it has the extra restriction that the parameters cannot change once the EJB is deployed.

Although most EJB containers will have means of parameter management for an EJB, this is not part of the EJB specification and therefore will differ for each vendor. In order to provide a (uniform) way of managing the parameters of an EJB, the normal parameter handling of an EJB must be circumvented. Therefore, instead of accessing the parameters via its standard JNDI context, an EJB could also opt for retrieving its parameters via the PodSystem by means of its own configuration manager Pod.

8.2 Distribution

One of the main advantages of using a container with a central entry point is that a user doesn’t need to know on which server a component resides. A good example of this is the Enterprise JavaBeans server [4], where an external client retrieves and uses an EJB-object via JNDI [6] (the entry point) without having to know on which server the EJB-object is deployed. The transparency the user experiences can be used by the PodSystem to provide distributed management tasks to the user and to the system itself.

Multiple PodSystems could be linked via a special connector Pod and made accessible through one central PodServer. This central PodServer then would relay management tasks to the appropriate PodSystems.

This construction also allows for advanced functionality such as load balancing with redundancy and fail-over. For example, the central PodServer may decide whether or not to deny access to heavy-loaded servers, making the Pods in those servers unavailable for users, until the workload of those servers has been decreased.

8.3 Stub interface

Developers that use the PodSystem need to invoke methods on the PodServer object to access a Pod or a configuration. While this is consistent with the desire to have one entry point, it is not a convenient way of accessing objects. It would be better if the developer could directly use a client-side stub object of a Pod to access the Pod and its configuration. This stub then translates all calls to the Pod and its configuration to calls to the PodServer. This way, the developer uses objects in the normal way and the PodSystem would still have one entry point.
Conclusions
9 Conclusions
The previous chapters described the PodSystem, a management framework for customisable services. By providing a set of standard management services, the PodSystem is able to relieve the developer of a lot of responsibility and overhead. This allows the developer to focus on building services, without having to worry about common tasks such as configuration management, security checking and notification routing.

By using the PodSystem as its basis, CablePilot not only is capable to be extended with new services for the management of CableFleet, but it also is able to interface with or manage resources other than those belonging to the CableFleet.

9.1 JMX-based implementation
Although the PodSystem uses the Java Management Extensions (JMX) as its underlying technology, it has a rather different use for it than intended by the JMX specification. Whereas the JMX specification places the focus on managing resources, by means of MBeans, the PodSystem is focused on providing services to users. The most noticeable distinction of this approach is the user-based configuration and security management that is provided by the PodSystem, in contrast to JMX, which doesn't provide any form of security management and only allows one configuration per MBean.

Consequently, the JMX specification and its reference implementation do not provide enough functionality for the PodSystem. Nevertheless, many useful concepts for managing services are introduced by JMX. Therefore, in order to facilitate in the implementation of the PodSystem, JMX has been chosen as the basis of the PodSystem, using those elements of JMX which are directly usable and replacing other parts of JMX with enhanced functionality in the form of basic management Pods.

The PodSystem uses the JMX concept of a protocol adapter to provide a central entry point to the PodSystem. However, whereas JMX uses an adapter to create an entry point to the system for external applications only, the PodServer is used by external as well as internal resources, e.g., Pods.

Because the JMX specification does not provide any form of security management - not even the standard Java security architecture is supported - , the PodServer takes over the role of the MBeanServer for controlling access to Pods. By doing so, the PodServer is able to provide transparent security and configuration management based on the connected user.

9.2 CablePilot and PodSystem
The requirements of CablePilot are met by the PodSystem in the following ways:
Extensibility: Extensibility in the PodSystem is accomplished by the DeploymentPod. The DeploymentPod makes it possible to insert new Pods in a running system and to update a Pod with new code.

User based security and configuration: The DirectoryPod provides user-based configuration and security management. With every action a user performs on the PodSystem (e.g., invoking methods or changing attributes) the DirectoryPod checks the permission of the user and associates the action with the configuration of the user.

Manageability: The Pods are made accessible via the central entry point of the PodSystem, the PodServer. The PodServer also serves as the interface to the basic management Pods, such as the DirectoryPod and the NotificationPod. Via DirectoryPod, the features of a Pod are configurable for each user separately, while the
NotificationPod makes it possible to manage the communication between Pods by providing the PodSystem with the ability to automatically route and filter notifications.

**Maintenance:** PodSystem provides several means for allowing maintenance on Pods. Maintenance on a Pod is provided by the DeploymentPod, which contains methods to update Pods with new code. Automatic maintenance tasks are made possible by the MonitorPod and SchedulerPod. The MonitorPod manages monitors, which take actions based on observations made in the environment, while the SchedulerPod schedules tasks in a timely and periodic manner.

**Interoperability:** Interoperability is achieved by two means in the PodSystem. Firstly, internal interoperability or, more precisely, co-operability, is provided by the PodServer. Via the PodServer, Pods can invoke methods on each other and use the NotificationPod for sending notifications to each other in an asynchronous manner. Secondly, external interoperability is achieved by the PodSystem itself in the form of the external RMI interface of the PodServer. The PodSystem can be made more interoperable by providing new adapter Pods that present an external interface of the PodServer via other connection technologies like SNMP and CORBA. While this will make the PodSystem available for other systems, other systems can be made available for management by providing a Pod that communicates with such a system via some protocol. This allows CablePilot not only to manage the objects of the CableFleet, but also to manage other external systems.

**9.2.1 User Interface**

The PodSystem does not specify how a management application like CablePilot should present a graphical user interface of a Pod to a user. It is up to CablePilot to define a graphical user interface using information from the publication and configuration of a Pod.
Literature

[1] Sun Microsystems, Inc.
Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303, U.S.A. 650-960-1300
http://java.sun.com/products/JavaManagement/

Java Security Architecture
Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303, U.S.A
http://java.sun.com/j2se/1.3/

Java Authentication and Authorisation Services
Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303, U.S.A
http://java.sun.com/products/jaas/

Enterprise JavaBeans Architecture
Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303, U.S.A
http://java.sun.com/products/ejb/

Java Servlet Technology
Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303, U.S.A
http://java.sun.com/products/servlet/

Java Naming and Directory Interface Application Programming Interface (JNDI API)
Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303, U.S.A
http://java.sun.com/products/jndi/

Java Remote Method Invocation
Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303, U.S.A
http://java.sun.com/products/jdk/rmi/

[8] Gamma, E.; R. Helm; R. Johnson; J. Vlissides
Design Patterns
1995 Addison-Wesley Publishing Company
PodSystem: a management framework for customisable services

[9]
Adventnet WebNMS
http://www.adventnet.com

[10]
R. Kersic
CablePilot 200 v1.1 Commercial Requirements Specification v0.7 review
Doc. Id: RKE.003.02
2000 The industree B.V.

[11]
R. Kersic
CablePilot 200 Graduation Assignment Description v0.1 accepted
RK.006.02
2000 The industree B.V.
Appendix A: The industree B.V. and their products

The industree B.V. designs, develops and markets broadband data communications systems over CATV and wireless networks. All products are based on open industry standards that enable QoS implementations and the integration of data, video, and voice services. Their CableFleet® system is designed to save valuable installation time and enables cable operators to realise their cable modem roll-outs at considerable lower costs. A break-through that will enable massive roll-outs for cable operators in the shortest time frame possible.

Figure 0.1 Diagram of The industree CableFleet

The CableFleet® cable modem system consists of three elements:
- **CablePilot® 200**: the cable modem network management software that offers cable operators a unique provisioning solution. Designed to save valuable installation time and to enable cable operators to realise their cable modem roll-outs at considerable lower costs. A break-through that will enable massive roll-outs for cable operators in the shortest time frame possible.
- **CableDock® 200**: the new innovative head-end solution, an Interactive Network Adapter that provides expandable capacity and scalable processing power to synchronise investment levels with capacity needs. The CableDock® 200 consumes only half the valuable rack space typically required by current generation INAs.
With its capacity to integrate In-Band and Out-of-Band services as well as wireless applications, the competitively priced CableDock® 200 is a true future-proof investment for cable operators. **CableJet® 900/910:** the high-speed Euromodem compliant cable modem, prepared for future service enhancements via robust software download mechanism. The CableJet® 910 enables the home user to plug in this EuroModem using USB technology and avoid costly service calls and scheduling delays.

The CableFleet® cable modem system focusses on **PROVISIONING** and **REVENUES.**

**Provisioning:** The combination of products offers cable operators a unique provisioning solution that will speed up cable modem roll outs by 50%. We started the design and development of this provisioning solution after thorough analysis of the issues that cable operators are facing today, as they are to start large scale commercial roll-outs of new services via their cable infrastructure. The biggest hurdle for cable operators is to get the potential customer base on line quickly. Getting cable modems installed and provisioned, and getting all the user-specific information into the relevant databases is a very time consuming, labour intensive, and error-prone process. With the solutions currently available, a cable operator can typically add about four new users per installation engineer each day. Clearly, with today's shortage of technically skilled staff, and their hourly rates, cable operators are urgently looking for improvements. We are very excited to introduce the solution, based on our next generation INA and management software and cable modem, combined in the CableFleet® cable modem system.

**Revenues:** The CableFleet® system enables operators to offer integrated data, video and voice-services to their homes via one cable modem system. Cable operators will be able to maximize their revenue stream by offering these integrated services.
Appendix B: The Java Management Extensions

The following excerpt from the Java Management Extensions White Paper is intended to provide a basic overview of the Java Management Extensions, on which the Pod System, discussed in the report, is based.

The Java Management Extensions

JMX defines a management architecture, APIs, and management services all under a single umbrella specification. The specification for JMX has been developed by Sun Microsystems, Inc. along with management industry leaders, following the Java Community Process. JMX provides developers of Java technology-based applications across all industries with the means to instrument Java platform code, create smart agents and managers in the Java programming language, implement distributed management middle-ware, and smoothly integrate these solutions into existing management systems. The JMX architecture is divided into three levels:
- Instrumentation level
- Agent level
- Manager level

In addition, JMX provides a number of Java APIs for existing standard management protocols. These APIs are independent of the three-level model, yet they are essential because they enable JMX applications in the Java programming language to link with existing management technologies. This part of the JMX specification is referred to as the:
- Additional Management Protocol APIs

JMX Architecture

The JMX architecture is built according to a three-level model. This gives flexibility by allowing subsets of the specification to be used individually by different developer communities utilizing Java technology.
- **Instrumentation level** - gives instant manageability to any Java technology-based object. This level is aimed at the entire developer community utilizing Java technology. This level provides management of Java technology which is standard across all industries.
- **Agent level** - provides management agents. JMX agents are containers that provide core management services which can be dynamically extended by adding JMX resources. This level is aimed at the management solutions development community and provides management through Java technology.
- **Manager level** - provides management components that can operate as a manager or agent for distribution and consolidation of management services. This level is aimed at the management solutions development community and completes the management through Java technology provided by the Agent level.

In order to build upon existing management technologies, the JMX specification also provides interfaces to the most widespread protocols in use today:
- Additional management protocol APIs provide a means of interacting with other management environments. The Additional management protocol APIs are aimed at the management systems development community and provide integration with existing management solutions.

JMX Components

The following section provides a brief introduction to JMX terminology. The JMX specification provides compatibility test suites which confirm that specific components comply fully with the appropriate section of the JMX specification.
JMX Manageable Resource

A JMX manageable resource is a resource that has been instrumented in accordance with the JMX Instrumentation Level Specification and tested against the Instrumentation Level Compatibility Test Suite. A resource can be a business application, a device, or the software implementation of a service or policy. In order to be instrumented, a resource can be fully written in the Java programming language or just offer a Java technology-based wrapper. Anything that needs to be managed, now or in the future can be instrumented and considered as a potential resource.

A managed bean, or MBean for short, is a Java object that represents a JMX manageable resource. By design, MBeans also follow the JavaBeans™ components model, thus providing a direct mapping between JavaBeans components and manageability. Because MBeans provide instrumentation of managed resources in a standardized way, they can be plugged into any JMX agent.

JMX Agent

A JMX agent is a management entity implemented in accordance with the JMX Agent Specification and tested against the Agent Level Compatibility Test Suite. A JMX Agent is composed of an MBean server, a set of MBeans representing managed resources, and at least one protocol adaptor or connector. A JMX Agent may also contain management services, also represented as MBeans. The MBean server is a registry for MBeans in the agent. The MBean server is the component which provides the services allowing the manipulation of MBeans. All management operations performed on the MBeans are done through Java technology-based interfaces on the MBean server. Protocol adaptors and connectors let management applications access a JMX agent and manipulate the MBeans it contains. Protocol adaptors give a representation of the MBeans directly in another protocol, such as HTML or SNMP. Connectors include a remote component that provides end-to-end communications with the agent over a variety of protocols (for example HTTP, HTTPS, IIOP). Since all connectors have the same Java technology-based interface, management applications use the connector most suited to their networking environment and even change connectors transparently as needs evolve.

JMX Manager

A JMX manager is a management entity implemented in accordance with the JMX Manager Specification and tested against the Manager Level Compatibility Test Suite. A JMX manager provides an interface for management applications to interact with the agent, distribute or consolidate management information, and provide security. JMX managers can control any number of agents, thereby simplifying highly distributed and complex management structures.
Both JMX agents and JMX managers integrate services that give them autonomy and intelligence. These services enable agents to handle their resources and let managers forward information back and forth between agents and management applications. Agents are more autonomous because they can incorporate certain management tasks, such as polling. The intelligence is embodied in simple logic that can keep managers from escalating unimportant alarms. Both of these measures can reduce network traffic and make management applications more resistant to outages.

In the JMX architecture, services are also MBeans that can be added and removed as needs evolve. This gives scalability to agents and managers, which is critical when these are deployed on thin clients.

The JMX specification currently defines the interface for such basic services as a registry for MBeans, queries of this registry, operations on resources and the forwarding of events back to managers, dynamic loading of new MBeans, creation of relationships and dependencies between MBeans, timer functions and attribute monitoring. Other management services that will be integrated into the specification include bootstrapping and persistence, network policy management, discovery of agents and managers, and security.

**Additional Management Protocol APIs**

The goal of these APIs is to provide a standard way for Java management applications to interact with existing management technologies. Typically, an application will use one of these APIs to access a legacy system and expose its attributes as a JMX manageable resource. This resource will then allow any JMX-compliant management application to manage the legacy system through a JMX agent. These APIs therefore create a bridge between existing and future technologies.

Based on the experience and feedback of the community utilizing Java technology, the JMX specification will include an SNMP Manager API, WBEM Manager and Provider APIs, and TMN Manager, Alarm and Topology APIs.
A New Management Paradigm

Why Java Technology is a Must for the Next Generation of Management Systems

The service driven network is taking over from the client-server model as it becomes incapable of satisfying the demands of today's customers. That is, the world is moving into the service age. Only a service-centered, service-driven network can satisfy these demands. Such a network can only exist if it is manageable, and the management system must be as flexible as the services it is managing.

JMX provides the flexibility, inter-operability, and dynamic management capabilities that are required for the service-driven network.

Management of Java Technology

The JMX instrumentation level delivers a light-weight instrumentation technology that surpasses any existing management technology in terms of flexibility. JMX provides instrumentation and hence manageability to any Java technology-based object, in all areas of the industry. JMX instrumentation allows any Java technology-based resource to be spontaneously managed by any JMX compliant agent.

Using JMX leaves application developers free to put the effort into their core business, rather than putting excess effort into adding manageability. Often, adding a few lines of Java technology-based code is enough to have instant manageability. This provides Java application developers with a standard way of instrumenting Java technology-based code. Developers writing code not based on the Java programming language can also add manageability by adding a Java technology-based wrapper which includes the instrumentation for their application. Code instrumented in this manner gains manageability that is not tied to any protocol or information model.

JMX MBean technology gives:

- Standard manageability for any Java application, sometimes in just three to five additional lines of code
- The ability to embed all necessary management information in a standard way in the resource to be managed
- The ability to provide a wrapper for instrumented resources not based on Java technology (even proprietary or custom solutions) with Java technology-based management systems

A JMX compliant agent is automatically capable of managing JMX resources. A non-JMX agent may also support JMX resources. Finally, the JMX instrumentation layer does not introduce any dependencies on external classes, a resource is entirely self-contained.

JMX instrumentation is aimed at the entire developer community that utilizes Java technology. JMX meets the demands for easy, rapid instrumentation of Java code allowing any resource to be spontaneously and dynamically managed.

Management through Java technology

The JMX agent and manager levels provide a flexible, distributed, and dynamic management infrastructure to be implemented in the Java programming language. By extending the Java programming language, JMX enhances the capabilities of existing solutions and enables the rapid creation and deployment of new types of management solutions. These solutions can be extended dynamically to incorporate new equipment and services in a plug and play manner.

JMX agents and managers developed using Java technology offer the following benefits:

- Platform independence
- Protocol independence
- Information model independence
- Management application independence
The JMX agent and manager levels are tailored for use by the following segments of the management solutions developer community:

- Management application suppliers
- Management platform vendors
- Equipment and device manufacturers
- System integrators

Each of these groups should consider the unique benefits that Java technology brings to management solutions. If they already use Java technology, JMX is the specification they should rely on to achieve standard Java technology-based management.

This approach uses the best of Java technology to create a new generation of management systems. JMX gives developers the freedom to integrate with existing and future Java technology in any field. For example, Jini™ technology-enabled devices are already manageable using JMX technology.

Integration with Existing Management Solutions

The JMX additional management protocol APIs bring the breakthrough technology of JMX into the reach of any existing management solution. The JMX technology though, does not attempt to replace existing legacy systems. This is simply not an option for end-users who must consider the value of the investment they have made. Introducing new technology beside an existing management system is also not a viable option, as this would create two separate management entities unable to cooperate or communicate effectively.

Instead, JMX offers a vertical integration, either providing manager or agent services through the technologies already in place. JMX provides a means for the seamless introduction of the latest Java technologies into existing management systems.

JMX includes an open interface that any management system vendor can leverage. Using this interface, a JMX agent and its resources can present management information consistent with various management models, such as:

- SNMP
- CIM/WBEM
- CORBA
- TMN
- LDAP

The JMX specification includes the definition of several management protocol APIs. These allow JMX managers to access agents in a legacy system and communicate with them through an existing protocol. For example, the SNMP manager API provides the services needed to write applications that manage SNMP agents or act as SNMP proxies. The definition of a WBEM client API allows you to write Java applications that access a CIM Object Manager.

Advantages of JMX

JMX is a unique management specification because it encompasses more than the specification of an interface. JMX also provides developers with the following:

- The shared expertise gained through the Java Community Process
- A complete set of tools for developers
- A proven technology base