2002

Development of an Infrastructure for the Management of Smart Homes

Ramkumar Natarajan

Aditya P. Mathur
Purdue University, apm@cs.purdue.edu

Report Number:

02-022

DEVELOPMENT OF AN INFRASTRUCTURE FOR THE MANAGEMENT OF SMARTHOMES

Ramkumar Natarajan
Aditya P. Mathur
Baskar Sridharan

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #02-022
October 2002

Development of an Infrastructure for the Management of SmartHomes

Status Report and Research Plan

April 5, 2001

Ramkumar Natarajan, Graduate Student

Aditya P. Mathur, Professor

Baskar Sridharan, Graduate Student

Software Engineering Research Center

Department of Computer Sciences

Purdue University

West Lafayette, IN 47907, USA

Financial Support from:

British Telecom

Telcordia

Contents

1 Introduction

This paper is a summary of our research in the area of management of distributed and embedded applications. This section, in particular, highlights the research context and explains the problem of interest to us. A summary of our accomplishments so far and a research plan appears in Sections 4 and 5, respectively.

The SmartHome project grew out of our vision of a society that is getting increasingly "connected" amongst its living beings and the devices they use. The long term goal of this project is to develop and experiment with technologies with the potential to improve the lifestyle of humans across various sectors of the society by making use of current and futuristic computing and networking technologies. The powerful and low cost integrated circuits available now, the ultra thin and flexible circuits anticipated in a few years, and the more futuristic molecular devices, are all likely to assist humans in improving their lifestyle. We want to understand how these technologies could help humans in managing, and improving from its current state, their day-to-day life. We also want to develop an infrastructure that will allow efficient management of collections of devices that can be monitored and controlled via the Internet. Following are the key objectives of the SmartHome project:

1. Understand how the current and future technologies could assist in the "smartization" of devices and homes.

2. Develop an infrastructure for the management of smart devices and homes. Such an infrastructure must be useful to individual home owners as well as to the service-provider industry likely to mushroom around SmartDevices and SmartHomes.

3. Set up experimental SmartHomes for the evaluation of the infrastructure and obtain feedback from the potential users of this technology.

2 SmartHomes

We answer the following questions to expose the context of our research.

1. What are SmartHomes?

2. Why are SmartHomes needed?

3. What are the management needs for SmartHomes?

We then enumerate a set of requirements that we believe are crucial for the management of SmartHomes. Events and actions

2.1 What is a SmartHome?

A SmartHome is a physical domain comprised of devices (hardware and software) owned by one or more individuals that can be monitored and controlled via the Internet. SmartSpace is perhaps a more generic name for a SmartHome. A traditional home (space) becomes a SmartHome (SmartSpace) when it contains one or more SmartDevices. However, a ship ashore or a sailing boat also becomes a SmartHome when either contains one or more SmartDevices. Thus, our definition of a SmartHome is general enough to encompass a variety of "physically enclosed" and "privately owned" networks that connect a host of devices to the Internet or an Intranet.

We are liberal about the kinds of devices allowed inside a SmartHome. Of course well known computing devices and peripherals are considered as SmartDevices and can be made smarter with the addition of hardware and software. Also, traditional devices such as a refrigerator, VCR, TV, video camera, video game console, lights, garage door, microwave oven, dog food dispenser, etc. found in many homes, can be upgraded with hardware and software and connected to a network. Such an upgrade and connection transforms an otherwise non-smart device into a SmartDevice.

Devices such as parts of a boat, parts of an automobile, personal clothes, furniture, art ware, etc. are not prevented from being parts of a SmartHome. Existing and future IC technology will in all likelihood make it possible for such devices to be made smarter. For example, a SmartShirt could communicate with the washer and inform the person who put it inside the washer that the washer could not be set to the right cycle because this shirt has been mixed with the wrong set of clothes. A smart chair might adjust its contour to that of the person sitting on it simply by recognizing the person about to sit. A SmartSeat inside a car might adjust its position by recognizing the driver or the passenger. Obviously, there is no limit to creative thinking and smartization of devices.

Figure 1 shows several SmartDevices connected to a Local Network and, through a Gateway, to the Internet or an Intranet. All devices in this figure, shown inside a boundary, are assumed to belong to one or more owners. Devices such as a VCR, refrigerator, automobile, and the oven, that are found in most homes, might be connected via a gateway to the Internet. Devices such as a heart rate monitor, blood analyzer, and CATSCAN equipment, might be connected via a special gateway to an Intranet owned by a Hospital. We do not constrain the Local Network and the Gateway in any way. Thus, for example, devices in a mobile hospital might be connected amongst themselves via a wireless network and in turn via some sort of a Gateway to an Intranet that connects other mobile hospitals.

Figure 2 shows the relationship between SmartHomes and the three types of management entities enumerated earlier. Note that a Manufacturer and a Service Provider could enter a SmartHome via the Internet or the Intranet. The owner could enter the SmartHome owned either via the Internet/Intranet or directly from within. For example, while at home the owner might access the TV and program it using on-screen menu whereas from outside of the home

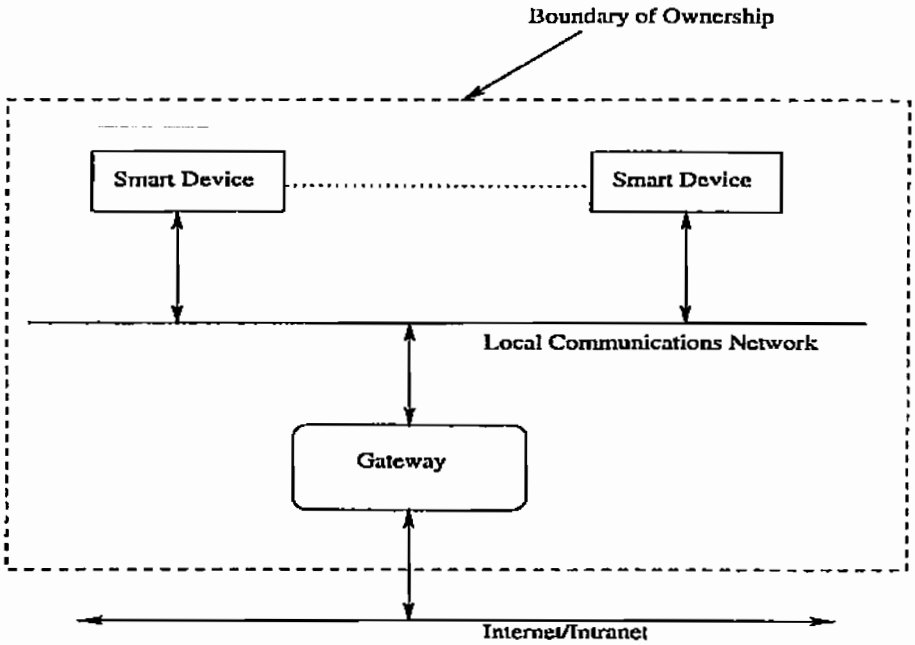

Figure 1: Connection of devices within a SmartHome to the outside world. The Gateway could be a special purpose device, such as a Residential Gateway from Telcordia Technologies, or a commodity Personal Computer.

she might program the same TV via the Internet.

2.2 Why SmartHomes?

The expansion of the Internet and the reduction in the cost and size of integrated circuits has given birth to a new breed of devices. These are devices that contain programmable logic which allows reconfiguration to suit individual tastes and requirements. The Internet allows such devices to be managed from almost anywhere in and outside the world. Programmability and connectivity of the devices entices people to use them in ways that could only be dreamed of in the past.

As an example, "fault tolerance" is introduced in the purchasing process when a soft drink can be purchased through a cell phone. Thus, if someone forgets to carry cash, or does not have the correct change, the cell phone provides an alternative way to meet one's need, that is to have a drink. Personal security, scheduling of entertainment, calling for medical assistance, providing medical assistance, are only a few of a variety of tasks that could be vastly improved with the widespread use of SmartHomes. In summary, it appears that SmartHomes will likely improve the overall quality of the individual lifestyle and hence their need.

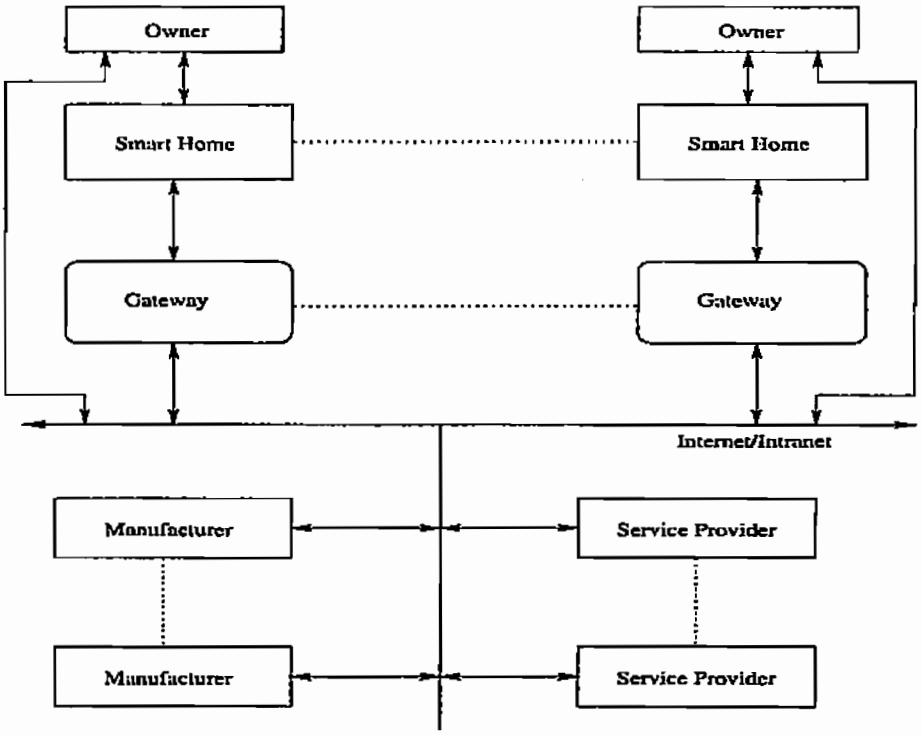

Figure 2: Accessibility of SmartHome to owners, Service Providers and Manufacturers.

2.3 Why manage SmartHomes?

Management of SmartHomes is necessary to ensure that the needs of its users are met in a timely and efficient manner. In the context of SmartHomes the term "management" refers to the monitoring and control of individual, or a collection of heterogeneous, devices. Monitoring is the checking of one or more status indicators. Control is exercising of actions that could alter the state of the device. Remote checking of whether or not an ATM is out of cash or checking whether or not a water softener is low on salt are two examples of monitoring. Remotely commanding a blood analyzer is an example of control. Though the remoteness of the operations is not essential to their being management tasks, it does seem to make them more useful.

To understand management needs of SmartHomes, we identify three categories of people or organizations who will likely need to manage SmartHomes. These are (a) individuals who own the devices, (b) organizations that are required to service the devices, and (c) manufacturers of the SmartDevices. A home owner belongs to category (a), a department store, such as Sears, to category (b), and a manufacturer, such as Sony, to category (c). Whereas people in category (a) and organizations in category (b) are expected to be directly involved in the day-to-day management tasks, manufacturers in category (c) are likely to be concerned with the provision of suitable features in the embedded software for effective management. For example, the manufacturer of a SmartCar ought to be concerned with what features to provide

for the owner to remote monitoring and control and for the auto-dealer to check, and perhaps service it, remotely.

2.4 Requirements for the management of SmartHomes

Table 1 lists what we perceive to be the management needs, or requirements, of people and organizations mentioned earlier. It is reasonable to expect that individuals and organizations in the managerial roles will need to monitor and control the devices. However, their specific requirements might differ. For example, a Service Provider might not be interested, or authorized, to monitor the channel being currently viewed by the Owner. Also, the Owner might not be interested, or authorized, to check the status of the fuel injection system inside an automobile.

The Service Provider and the Manufacturer are likely to have similar requirements. However, one might or might not authorize the other to perform certain monitoring and control tasks. For example, the Manufacturer might provide a special feature in the embedded software that controls the automobile engine. Using this feature the Manufacturer would be able to download performance data. This data might be important enough to the business of the Manufacturer to not allow access to others including the Service Provider. The Service Provider might also restrict the Manufacturer from obtaining information about the service agreement that it has signed with the Owner.

2.5 Event notification and correlation

Event notification and co-relation mechanisms are common in distributed systems. Event based systems allow for loose coupling of components and enable building complex interactions among components of a distributed system.

The following are a few reasons why an Event Recognition, Notification and Co-Relation (ERNC) system would be beneficial as a component in the infrastructure for the management of SmartHomes.

- *Asynchronous interaction*: Using the ERNC, one have asynchronous interactions with components of a SmartHome. Thus, for example, instead of a user checking every 10 minutes whether a washing cycle is done, a smart washing machine combined with the ERNC can be programmed to notify the user (via e-mail or other means) upon completion of the washing cycle.

- *Automation of repetitive tasks*: A key benefit of managing SmartHomes is that it promises to improve the perceived quality of life of its users. This benefit can be further augmented by automating mundane and repetitive tasks. Thus, for example, with a SmartLock and the ERNC, the home security system can be enabled automatically each time the SmartHome is locked.

- *Allow complex interactions between devices*: The ERNC can be viewed as a mechanism for embedding intelligence into the HomeWabash system. For example, the ERNC can be used to dim the SmartLights soon after the SmartProjector is turned on. While the SmartProjector by itself

Table 1: Requirements for the management of SmartHomes.

User Category	Requirement	Explanation
Owner	Monitoring	Check the state of any device owned.
	Control	Send a control command to any device owned.
	Access specification	Specify access controls.
	Event specification	Attach time and state dependent actions to devices owned.
Service Provider	Monitoring	Check the state of any device under contract. The state space to be monitored will likely be different than the one that can be monitored by an owner.
	Control	Send a control command to any device under service contract. The control command set will likely be different than that for the owner.
	Access specification	Specify access controls to servicemen and owners.
	Event specification	Attach time and state dependent actions to devices.
	Service assignments	Assign devices to specific service personnel.
	Service schedule	Schedule on-site and remote maintenance services.
	Accounting	All finance related services.
Manufacturer	Monitoring and Control	Perhaps all Monitoring and Control needs of the Service Providers will also be the needs of the manufacturers. However, the manufacturer may need special access to the devices for the purpose of servicing and obtaining performance data. This could be provided by the Service Provider.
	Upgrade	Device manufacturer might want to upgrade embedded software on all devices of a kind. This could also be done by the Service Provider.

may be capable only of producing a notification of an "on" event, when coupled with the ERNC it can give the user the illusion of being a co-ordinated smart-device aware of other devices and their interactions.

- *Allow interactions between heterogeneous devices:* A key requirement of the HomeWabash system is the need to handle heterogeneity. We do not expect a single standard for smart-devices or for their management. Nor do we expect a single standard for communication mechanisms between and to SmartDevices. However, from a user's perspective, it is of great advantage if this heterogeneity is hidden and instead a consistent interface is presented for monitoring and control. The ERNC can facilitate the interaction between heterogeneous devices through the loose coupling presented by the event-response mechanism. The power of using the ERNC as a building block to a complex management system is that no component needs to be designed to work specifically with any other component. For example, a Bluetooth enabled PDA can send a song title to an IP enabled MP3 player which then downloads this song to be played later. While the PDA and the MP3 player may not have been designed to interact, or indeed even be aware of each other's existence, the ERNC system provides a flexible coupling mechanism that facilitates interactions between these devices.

- *Facilitate greater reach for* SmartDevices: We expect SmartDevices to have a limited scope in terms of their communication capabilities. Thus a device cannot be expected to make use of the diverse array of communication mechanisms to interact with the user. However, this restriction can be overcome when the device is coupled with an ERNC system that does not have the same limitations as the device. The complexity of managing the diversity in communication mechanisms to the user is now handled by the ERNC similar to how a Residential Gateway handles the complexity of communicating with an individual device. For example, a washing machine may only be capable of producing a simple notification to the local Residential Gateway. Using the ERNC, this notification can be forwarded based on the user's current location and environment, to her e-mail, mobile phone, pager, car dashboard, etc.

2.6 Embedded (EA) versus distributed applications (DA)

We are concerned with the management of embedded applications (EA) and distributed applications (DA). In the context of SmartHomes, embedded applications are those that reside inside a SmartDevice. The software that resides inside a VCR, a Microwave oven, an automobile, CATSCAN equipment, etc., constitutes embedded applications. A collection of components, perhaps distributed over many computers across the globe, to manage a group of SmartHomes, is an example of a distributed application. An embedded application could be distributed. For example, most automobiles contain several microprocessors controlled by software components that communicate amongst each other to realize an automobile function. Modern airplanes also have similar applications that are embedded as well as distributed. In addition to the embedded applications being distributed, the services for these devices may likely be realized using distributed applications. For example, a Home Management Service for managing SmartHomes may be a distributed, component-based application.

Considering that embedding and distribution are the likely characteristics of applications in the context of SmartHomes, we use the following terminology for the purpose of this work:

Embedded applications (EA) are intended to control a single SmartDevice and are hosted physically in

some form of memory (e.g. flash ROM, atomic disk, etc.) that resides within the device to be managed. An embedded application may or may not be distributed.

Distributed applications (DA) are collections of at least two or more components hosted on one or more computers and provide services using some variant of the client-server model. A component may be replicated to improve its accessibility.

It is likely that EAs are significantly more constrained than DAs in terms of the computing resources available to them. This will likely affect their architecture. For example, it is unlikely that an EA, even though distributed, uses a CORBA 3.0 or a DCOM implementation whereas a DA could. Certainly, progress in hardware technology could change this belief. However, when compared in relative terms, an EA is likely to be more constrained than a DA.

3 Research Issues

A careful examination of the requirements for the management of SmartHomes leads to the formulation of key research issues. One can imagine a multitude of approaches to fulfill the requirements listed in Table 1 for the management of SmartHomes. For our discussion we identify and examine two approaches. These approaches are: (1) individualized solutions and (2) standardized solutions.

3.1 Solutions to the SmartHome management problem

To understand the strengths and weaknesses of the two approaches mentioned above, consider a Service Provider who wants to enter into the business of servicing a kind of Internet appliance. The software infrastructure required to run such a business would necessarily include components to monitor and control the devices the appliances to be serviced. To maximize its profit potential, a Service Provider might want to manage a diverse portfolio of appliances rather than one-of-a-kind. Diversity in portfolio would imply different types of appliances from manufacturers. Under the *individualized solution* approach, each manufacturer develops its own monitoring and control interface thereby increasing the likelihood of incompatible monitoring and control interfaces for the appliances under service. Such incompatibility adds to the complexity of the software components for monitoring and control and hence to the cost of their development and maintenance. Expansion of business by acquiring one from another Service Provider will likely lead to the increase in software complexity. Other components of the software infrastructure, such as those that deal with accounting for pay-per-use appliances, or provide individualized services to the Owner, or assist in organizing a large domain of use into smaller subdomains for the ease of management, are also likely to face the problem of "added complexity".

Under the *standardized solution*, we foresee a software infrastructure consisting of components that provide features to meet the basic needs of the three categories of users and managers of appliances. These components are used by a number of manufacturers and Service Providers. The components are extendible to meet special needs. Note that standardized solution does not imply a unique infrastructure, but it does imply the existence of perhaps several infrastructures for management one of which is selected for use by a Service Provider depending upon its characteristics.

3.2 Questions of interest

Discussion in the previous subsection leads to the important question that is driving our research: *"What should be the software infrastructure for the management of* SmartHomes *?"* We split this generic question into the following research questions:

1. What should be the architecture of the infrastructure?
2. How should an infrastructure be evaluated? We expect to identify and, where possible, quantify the key describing characteristics of infrastructures so that competing solutions could be compared.

We do not expect a unique answer to the first question above. Perhaps there are different architectures suitable under different management scenarios. By the term "architecture of the infrastructure" we mean a listing of the key components of the infrastructure, the relationships amongst these components, features offered by each component, and any special characteristic of each component [9].

Evaluation of the infrastructure is essential to our research. We have identified the following characteristics against which we plan to evaluate management infrastructure developed in our research.

1. Intrusiveness
2. Scalability
3. Support for heterogeneity
4. Support for secure management
5. Support for remote maintenance
6. Support for the generation of management applications

In the following subsections we explain each of the above characteristics.

3.2.1 Intrusiveness

Intrusiveness is a measure of the extent to which the infrastructure (a) requires a change to the application code, (b) degrades the performance of the entity being monitored, and (c) alters the state of the entity under observation. In a SmartHome, for example, it may be desirable that the current operation of a blood analyzer be not affected in any way when its state is monitored. Also, when the monitoring and control needs change, one should not have to modify the application code in order to accommodate the new needs.

3.2.2 Scalability

Scalability is a measure of how the infrastructure performs when the number of devices being monitored and controlled increases. Performance of the infrastructure could be measured as, for example, (i) the time to monitor, (ii) the time to send control signals to a device upon the occurrence of an event, (iii) the ease of distribution and reallocation of management personnel, and (iv) the time to upgrade embedded applications.

3.2.3 Support for heterogeneity

Support for heterogeneity refers to that aspect of the design of the infrastructure which allows its users to easily adapt new `SmartDevices` into a management domain. It is difficult to quantitatively measure this aspect of the infrastructure as it is perhaps always possible to accommodate any `SmartDevice` by appropriate modification of the management application. Hence, it is the ease with which new devices could be inducted into a management domain that becomes the distinguishing characteristic of a "good" infrastructure. In DAs heterogeneity arises due to the use of different technologies. For example, a management application might employ a mix of components that use CORBA [12], Java RMI [1], and JINI [2]. Monitoring and control of such applications raises interesting questions.

3.2.4 Support for secure management

Secure management and support for it refers to the use of techniques for managing `SmartHomes` in a secure way. A fully secure infrastructure implies that, when used correctly, the management application built around this infrastructure will make it impossible for intrusion and misuse of monitoring and control operations by unauthorized individuals.

3.2.5 Support for remote maintenance

Support for remote maintenance refers to those features in the infrastructure that allow Service Providers and Manufacturers to remotely enter a `SmartDevice`, identify the cause of any problem already reported or likely to occur, and, if possible, make repairs. In a sense this is akin to debugging remotely but within the environment of its use.

3.2.6 Support for the generation of management applications

Support for the generation of management applications refers to those components in the infrastructure that are intended for assistance with the generation of management applications given a specification of the management tasks. We believe that all categories of users would need to develop new management applications to induct new `SmartDevices` into their management domain and to satisfy previously unsatisfied needs. Our goal is to provide components in the infrastructure that will assist with the development of such applications.

4 Summary of past research

We describe the architecture of two infrastructures for the management of EA and DA, summarize the important characteristics of their implementation, and present results obtained from experiments designed to evaluate them.

4.1 Architecture of the infrastructure for the management of DA

The infrastructure for the management of DAs was designed and implemented to manage applications that use CORBA. These applications consist of several components that are often replicated and distributed across multiple computers. The components, both clients and servers, are usually unaware of the locations of those

from which they need to obtain services and communicate via a Object Request Broker popularly known as ORB. An owner of CORBA components is often interested in their monitoring and control. The architecture described in this section was designed and implemented to allow scalable and non-intrusive management of CORBA components.

In this architecture, named Wabash, scalability is achieved primarily through the use of *zone-based partitioning* and non-intrusiveness through a clever positioning of components called "Local Listeners". Several key features of Wabash also distinguish it from other similar systems. These features include the ability to perform dynamic load testing of servers, testing of servers via interfaces, and the ability to allow local and distributed management. The novel elements of the architecture and its implementation are summarized in the remainder of this section.

4.1.1 Components of the architecture

Figure 3 shows the components and their interactions as found in Wabash. For our discussion, let CM denote a component to be managed in a DA. Various components of Wabash are described below. The advantage offered by each component is explained in Section 4.3.4.

`Local Listener (LL)`: Each CM is encapsulated by an LL. All requests sent to or from CM must go through the LL. The LL is the supplier of information regarding a CM. For example, the LL is the source for all events that occur at CM. The request emanating from and the responses received at a CM are viewed as events. The LL maintains and supplies component specific information such as the component's name, number of exported interfaces and certain performance statistics. In addition, the LL stores manager specified event-action correlation pairs and recognizes events that occur at CM. The LL also executes the corresponding actions associated with an event. LL uses information from CM's interface to provide the functionality.

`Monitor (MR)`: The MR provides services for monitoring one or more CM(s) and for parsing and delegating event-action correlations. For example, for an event that involves more than one CM, the MR component parses the event description and delegates the recognition of the local events to the appropriate LL.

`Controller (CR)`: The CR provides services for controlling one or more CM through the LL. For example, CR provides a *stop()* that can be used to dynamically stop the CM from servicing requests. The *stop()* service also provides for a fine-grained control over the requests. For example, it is possible to specify that CM stop servicing certain requests *only* if they arrive from clients that reside in a certain network. The MR and CR provide features for managing the CMs in one zone and hence constitute the *zonal manager*.

`Database Manager (DB)`: The information and state of each CM is incrementally updated and stored in a database. DB exports services for storing in and retrieving data from the database.

`Zonal Services Gateway (ZSG)`: The ZSG serves as the gateway to access the services of the various components within a zone. For example, the LL uses the ZSG to access the services of the MR and CR. Similarly, the services of the LL can be accessed through the ZSG.

In addition to the above mentioned components, the architecture also includes a User Interface (UI) component for managing the DA.

4.1.2 Zone-based partitioning

In is likely that components of the DA are distributed over a wide geographical area with differing ownerships and communication speeds among the components. It was therefore decided to allow a manager to establish a one-level hierarchy of components by partitioning the space of components into logical *zones*. Each zone consists of a collection of components managed independently of components in other zones. In the description below we assume that the management of components is under the control of a human being who is referred to as "manager".

The partitioning of the components into zones may be based on several criteria such as ownership and communication bandwidth. We consider the case where components are owned by one or more managers each of whom may have different levels of access. Even though owned by different individuals, the components may collaborate to present a single service to their clients. Each set of components owned by an individual may receive management requests from the other. Partitioning the deployment space into distinct, though communicating, zones facilitates the application of access restrictions for managing the components and also simplifies the management of these components. For example, for an application deployed in two different zones, it is possible to assign one administrator for managing each zone. The administrator for a given zone may be granted access-permissions only for components deployed in the zone of responsibility. It is also possible to assign to a single administrator the responsibility for the management of two or more zones.

Type of the communication links, that connect the various components, could be another criteria for partitioning. For example, components that communicate using a single LAN can be grouped under a single zone. Typically, though not necessarily, components deployed over a single LAN are owned and managed by the same organization. Hence, the management needs of the administrator for these components are more for the locally deployed components than for those that may be deployed across a WAN. Hence, it would be beneficial to assign a zonal manager for managing each local group.

4.1.3 Using Wabash to manage a DA

The DA can be managed *manually*, *automatically*, or using a combination of the two. For manual management, a manager selects the appropriate component using the UI and a suitable command is sent to the selected component. The UI sends the command to the component's ZSG. The ZSG then forwards the request to the component's LL. The DA can be automatically managed using event-action correlations. For example, a sequence of management commands, referred to as *actions*, can be executed upon the occurrence of an event at a component. The event-action pair is sent to the ZSG through the UI. The ZSG sends the information to the MR for parsing and delegation. MR sends the local events to the CR and also to the appropriate LL. Upon the occurrence of the event at a component, the corresponding LL directs this event to the MR through ZSG and also executes the action(s) corresponding to this event.

Managing a DA manually or automatically requires that the requests and responses go to the component through LL. Upon the arrival of a request at a LL, the following actions are taken before the request is forwarded to the component:

- The request is time-stamped with its local time of arrival.

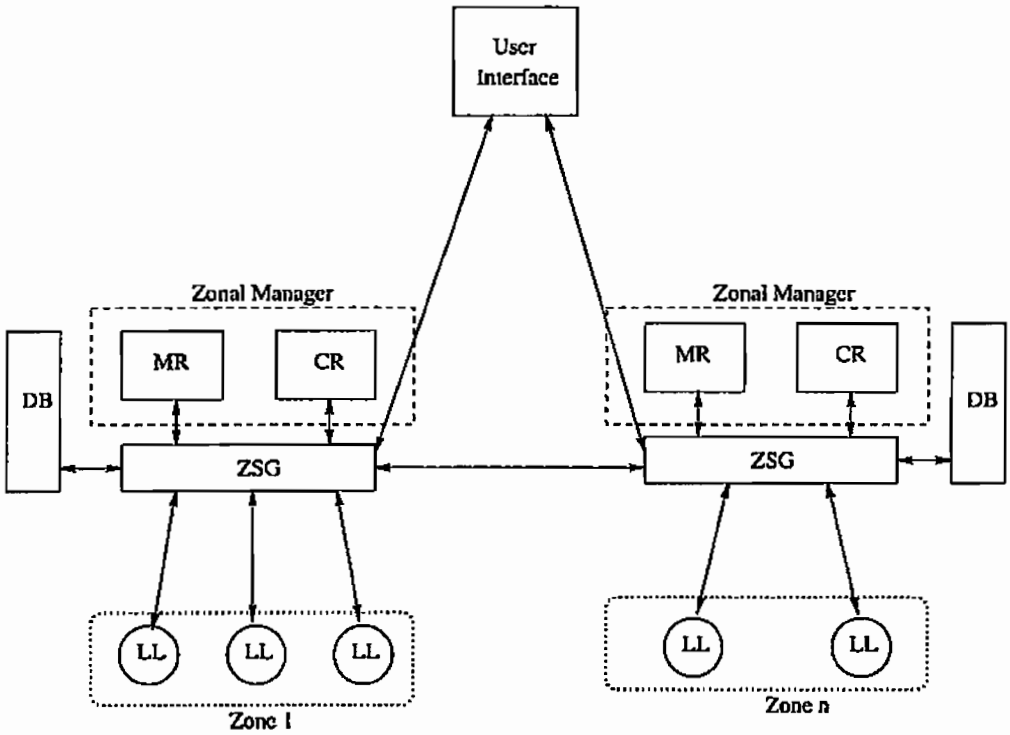

Figure 3: Components of the infrastructure for the management of DA.

- The performance statistics for CM are updated.
- Attributes of the request are sent to MR.
- An event-action list is consulted and appropriate action is taken if specified in the action part of the event-action pair.

4.1.4 Implementation of Wabash

The proposed architecture has been implemented using JDK1.2 and Visibroker 3.4 [11]. All components of Wabash 2.3 are implemented as CORBA components and hence the communication between them is via the ORB. The LL is implemented as a CORBA interceptor. Though Visibroker 3.4 does not support portable interceptors, the subsequent versions of Visibroker do and hence the LL module can be extended to work with various implementations of CORBA. A flat-file representation of a database has been implemented.

4.2 Features of Wabash

Wabash provides the following features:

1. *Performance* of the individual objects and the application.
2. Measurement of *method and interface coverage*

Table 2: Performance measures at each level.

Property	Appliction Level	Object Level Level	Interface Level	Method Level
Name	√	√	√	√
Host Machine Name	√	√		
# of unnamed objects	√			
# of named objects	√			
Started at	√	√	√	√
Up Time	√	√	√	√
# of hits	√	√	√	√
# of interfaces		√		
# of methods			√	
Maximum Latency				√
Minimum Latency				√
Average Latency				√
Memory usage				√

3. Mechanism for *fault injection.*

4. *State* of the components at each level of hierarchy.

5. Mechanism for *event-based monitoring and control.*

Each application is viewed at different levels of granularity, namely, application level, object level, interface level, and method level. The above features are implemented at each level of the hierarchy of an application as described below.

4.2.1 Specifying project information for testing and management

Before Wabash can be used to test or manage an application, some static information regarding the application is obtained. The association between machines and zones is specified. The information regarding the servers and the machines on which they are hosted is also specified. This static information is stored in a database and is referred to as a *project.*

4.2.2 Performance measures

Wabash collects performance statistics at each level of the hierarchy. The performance statistics can be viewed at run-time. They are also logged on to a stable storage for post-processing and other analyses. Table 2 shows the statistics collected at each level.

4.2.3 Coverage measurements

Traditional testing of sequential programs involves maximizing the code coverage of the various components of the application. Current research in the area of testing of component-based applications focuses on interface-based coverage criteria. Wabash aids in testing component-based applications by providing coverage information for the various interfaces exported by the application. It provides a graphical view of the ratio of *hits/total-hits*, and the percentage of the interfaces covered at each object, as a pie-chart. The coverage information helps the tester visualize an increase in interface and method coverage while running various tests.

4.2.4 Interface mutation and fault injection

The current architecture supports mutation of the elements of an object's interface. The tester can use this feature to create, activate and deactivate mutants of the methods in an interface and thereby evaluate the adequacy of a test suite. The mechanism that creates mutants is also used for the injection of faults at component interfaces. These faults include server crash, delayed response, and invalid response. The following mutants are generated:

- Swap any two parameters.
- Increment/Decrement an integer by one.
- Substitute a value of a parameter by null.

4.2.5 Event-based monitoring and control

This feature allows a user to specify an event of interest and to attach an action. The action is executed upon the occurrence of the associated event.

4.2.6 State information of a component

The tool also collects and displays state information at each level. A state of a component at a level is defined to be one of the following:

- INACTIVE - when a component has been statically defined to be a part of the application but has not been started.
- DENY - when a component has been started but all requests to the component have been denied service.
- ALLOW - when a component has been started and all requests are allowed service.

ALLOW is the default state of a component. The state of a component may be affected when a *control_action* is applied to it. In addition to the *control_actions* specified earlier, two special control actions are assumed to be pre-defined for any application. These are the *start* and *stop* actions applicable to any component. *start* is assumed to start a stopped component and *stop* stops a running component. Upon the start of a component, all components contained inside it are also started. For example, when a server is *started*, all the objects in the server are also started. Similarly, when an object is started, all the interfaces exported by the object are started.

For components at each level, Wabash displays the current state of all the components contained inside. The state of a component can be changed unconditionally by choosing any component and applying a control action on an *Any* event.

4.3 Evaluation of the infrastructure for the management of DA

Section 3.2 lists the criteria we propose for the evaluation of a management infrastructure. The criteria are described with a focus on the embedded nature of the applications. Though the basic criteria remain unchanged in the context of distributed applications, the semantics and relevance of the criteria do change. For example, in the context of EA, heterogeneity refers to the ability to support different types of device technologies whereas in the context of DA, it refers to the ability of the same management architecture to support different types of distributed systems technology. Similarly, in the context of EA, intrusiveness with respect to the service latency is not as relevant as in the context of DA. In the remainder of this section, we describe the semantics of the criteria enumerated in Section 3.2 and describe the evaluation of our architecture based on these criteria.

4.3.1 Intrusiveness with respect to the source code

As mentioned in Section 3.2.1, minimal intrusion is a desirable characteristic of any infrastructure for management. The components of the architecture interact with the CM through the services exported by its LL which is the only component that interacts directly with the CM. Hence, it is important that the LL be minimally intrusive. In our implementation, LL has been implemented as a CORBA interceptor. A CORBA interceptor can be loaded at run-time into the same address space as the component itself with no change or only a minor change to the source code of CM. The change to the source code depends on the language in which CM has been implemented. If the CM is implemented in Java, then some CORBA implementations can load LL at run-time into the component's address space. Also, this can be done without *any* modification to the source code. A CM implemented in C/C++ may require a small modification to instruct the CORBA implementation to load LL into the component's address space.

4.3.2 Intrusiveness with respect to service latency

It is important that the management architecture have very little, if any, effect on the latency of the services exported by the CM. We evaluated a prototype implementation of the architecture to determine the overhead of the architecture on the latencies of the exported services. We conducted experiments on a large, four-tiered, CORBA-based telecommunication application written in Java. The application was under development at Telcordia Technologies. At the time of experimentation, this application consisted of over 100,000 lines of code. Hence, non-intrusiveness with respect to the source code was an important criteria. For our experiments, we selected the services of a high-volume component of the application. The objective of the experiment was to determine the effect of the management architecture on the latencies of the services. We found the overhead to be between 4.65%-6.2%. More details of the experiment are found in [16].

4.3.3 Heterogeneity

In the context of DA, heterogeneity is the ability of the architecture to manage DAs built using different distributed-systems technology such as CORBA, DCOM, Java RMI etc. The architecture we have built is used to manage CM using LL. The various components of its architecture communicate with LL. Though the current implementation of the architecture uses CORBA to realize the components, it does not restrict the architecture's heterogeneity. We can manage a DA, if we can implement a LL, and also provide access to the CORBA services exported by MR, CR, and DB. This implies that ZSG may have to act as a gateway, i.e. communicate with the zonal manager using one technology and with the LL using another. For example, to manage Java RMI based DA, the services of LL needs to be exported using Java RMI. Also, the ZSG needs to communicate with LL using Java RMI and use CORBA to communicate with the MR, CR, and DB. Though we can manage DA by implementing LL appropriately, it may not be possible to load LL into CM's address space with only a minimal modification to the CM's source code.

We are yet to evaluate the architecture for scalability, security and support for automatic generation of management applications. This task is proposed to be completed in the future.

4.3.4 Related work

In this section, we describe briefly the work related to the management of distributed systems and compare this with ours.

MOTEL: MOTEL [13] provides for run-time monitoring and testing of pre-defined properties of object-oriented distributed applications. MOTEL defines a set of twenty events for modeling and expressing the behavior of such applications using linear temporal logic. MOTEL monitors the behavior of the system by monitoring the occurrence of the corresponding events. It uses a single observer to monitor the occurrence of the events. These events are then used to test whether or not the system violates the specified behavioral constraints. It achieves the twin tasks of run-time monitoring for occurrence of the events and testing for violation of the behavioral constraints by instrumenting the source code of the application. The process of instrumenting the source code is automated. MOTEL has been implemented for CORBA-based distributed applications.

HiFi: HiFi [7] uses an event-based abstraction for modeling and monitoring the behavior of distributed applications. It provides for the specification of the events to be observed at run-time. Each event can be attached to an action which can be used for steering the application. HiFi uses a hierarchical system of observers to monitor the occurrence of events. This minimizes the intrusion of the monitoring system on the performance of the application. HiFi, like MOTEL, instruments the source code of the application to achieve its task.

MOSS: MOSS [8] is a system for monitoring and steering of parallel and distributed applications. It provides features for monitoring and steering of distributed applications by creating objects that mirror the state and methods of the application. In addition to mirroring the state and methods of the original application, the objects also include additional state and methods for the purpose of monitoring and steering. These "mirror" objects are analogues of the original application as they share the state via monitoring and implement the methods via remote method invocation. The act of steering is performed through the application object's original methods via remote object invocation. MOSS has been implemented for CORBA-based distributed applications. For the

purpose of monitoring and steering, MOSS requires a modified IDL compiler for producing an instrumented stub/skeleton. The IDL compiler instruments the `get` and `set` methods for the attributes to perform monitoring. The object methods are also instrumented for the purpose of steering.

In the above, we see three approaches for incorporating the monitoring and control code into the distributed application. All three use some form of instrumentation either at the level of the source code or the interface code. MOTEL and HiFi instrument the source code while MOSS instruments the interface by modifying the stub and skeleton. The main strength of Wabash, in comparison to the above approaches, is the lack of instrumentation at any level. Wabash uses LL for monitoring and control. Since LL has been implemented as a CORBA interceptor, for CORBA-based applications written in Java, LL can be incorporated into the distributed application at run-time by simply specifying the location of the code base for LL as a command line argument to the Object Request Broker. This criteria is important for heterogeneity - the lesser the amount of instrumentation required, the more is the support for heterogeneity and more easier to support distributed applications written in languages other than the one for which the management system is designed.

Unlike MOTEL and MOSS, Wabash uses a zone-based partitioning approach for improved scalability with each zonal manager handling those monitoring and control functionality that pertain only to the components in the assigned zone. This approach is similar to the hierarchical system used by HiFi. Whereas HiFi uses a multi-level hierarchy, Wabash only employs a single level. For applications where the frequency of the events is low and the number of components in the system is small, the single level monitoring architecture performs better than a multi-level hierarchy [7].

Both Wabash and HiFi provide the ability for dynamic specification, notification, and action correlation of events. Wabash also provides for dynamic, fine-grained control of the components. This feature is missing from the three approaches presented earlier even though MOSS provides for some limited form of control through program steering.

As mentioned earlier, the LL component of Wabash is a CORBA interceptor that uses only the information obtained from the interface of the components. This provides for the ability to dynamically generate clients for the components. This feature, not present in the three systems reviewed above, has been used to provide dynamic load testing functionality in Wabash.

4.4 Architecture of the infrastructure for the management of EA

4.4.1 Components of the architecture

Figure 4 shows the architecture of the infrastructure for managing an embedded application (EA). Components of this infrastructure are described below.

Gateway: The Gateway provides uniform access to the services exported by the EAs (labeled D1, D2, D3, and D4 in Figure 4). It helps connect the EAs to an Intranet or an Internet. The Gateway component is made up of (1) Event Recognition, Notification and Correlation Component (ERNC) and (2) Device Communication Component (DC). ERNC, in the Gateway, is responsible for recognition and notification of events that occur at the EA that are attached to the Gateway. ERNC maintains the list of interested events and the corresponding actions. On occurrence of an event, it consults the list and takes the appropriate action. It also notifies the Proxy

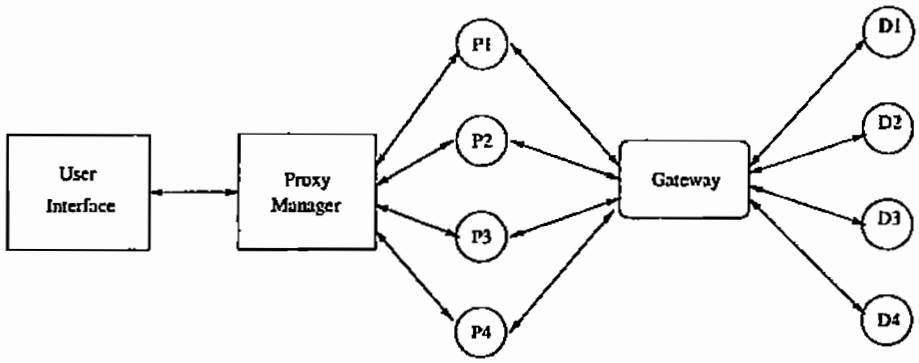

Figure 4: Components of the infrastructure for the management of EA.

Manager. DC acts a bridge between the devices and the outside world. The complexity of communicating with heterogeneous devices is the responsibility of DC.

`Proxy` and `Proxy Manager`: The Proxy provides the interfaces for remote access and management of an EA. Each proxy is mapped to an EA. The Proxy Manager (PM) provides a uniform interface for the creation and management of the proxies through the Proxy Communication (PC) component. PC exports the interface of the proxies for access using different communication protocols such as CORBA, Java RMI and HTTP. The Proxy Manager also contains an ERNC component that detects events that occur at the proxies. For example, when a user wishes to access to the functionality of a device from a remote location, the request is sent to the device through the corresponding proxy. Such events, related to the invocation of certain device functionality, are detected and notified by ERNC.

In addition to the components mentioned above, the infrastructure also contains a user interface component through which the services of PM can be accessed. UI can be used by (1) a home owner to manage the EA or (2) a service provider to manage the gateway, proxies and the proxy manager.

4.4.2 Implementation of `HomeWabash`

Figure 5 shows the components and their interconnections of `HomeWabash`, an implementation of the infrastructure for managing EA. The proxy and the management components have been implemented using JMX [3] and Java Beans, respectively. The proxies are implemented as JMX MBeans and are dynamically configurable. The interfaces exported by the proxies are exposed through CORBA, Java RMI, and HTTP. The proxy manager has been implemented as a JMX MBean Server. `HomeWabash` has been implemented using JDMK which is Sun Microsystems' implementation of the JMX specification. The functionality of each device is mapped by the corresponding proxy. The functionality of the proxy is dynamically queried and exported as a WML interface to the home owner by a Java servlet. The Java servlet is hosted inside a firewall. The implementation lets a user to remotely monitor and control the devices using a WAP-enabled device.

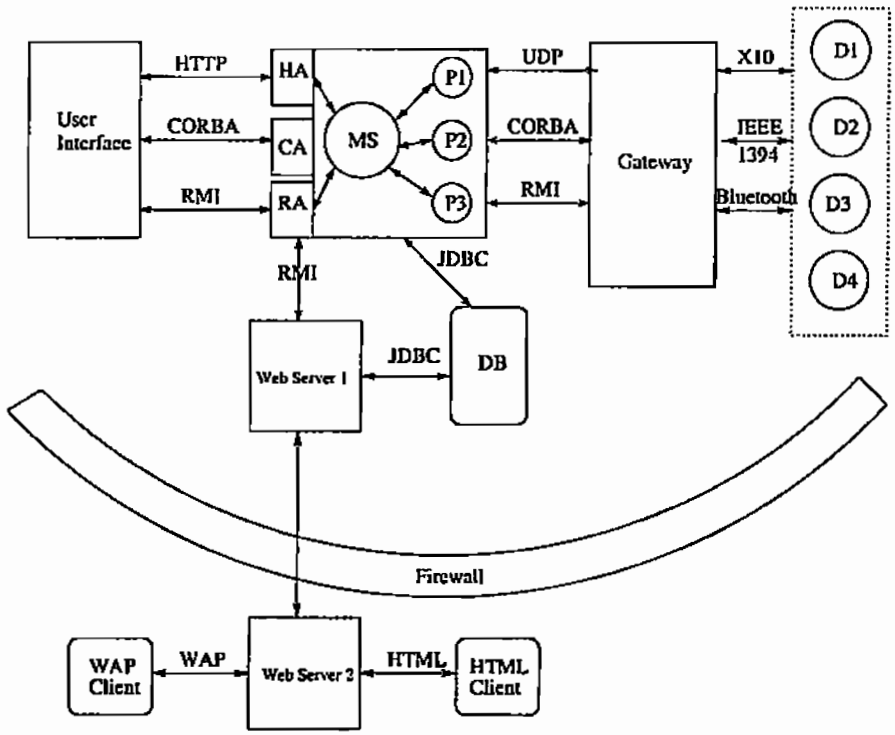

Figure 5: An implementation of HomeWabash for the management of EA.

4.5 Features of HomeWabash

HomeWabash is expected to be used by three categories of users: Home Owner, Service Provider, and Manufacturer. The features of HomeWabash provide for the management tasks listed in Table 1. Each of these tasks can be performed using any WAP-enabled device such as a cell-phone, a Handheld device, and a Web browser. For example, the Home Owner can control devices owned using a cell-phone while a Service Provider can instantiate, register and deploy proxies for devices incorporated in the SmartHome. A Manufacturer can use a Web browser to detect the malfunctioning of a device and can up-load new code into a SmartDevice. HomeWabash also provides for the persistant storage and retrieval of user profile, usage and accounting information, and proxy state. Persistant storage of the proxy state helps in providing better fault-tolerance against failures of HomeWabash.

4.6 Evaluation of the infrastructure for the management of EA

4.6.1 Intrusiveness with respect to the source code

The EAs are managed through a corresponding proxy component by mirroring the services of EA in the proxy. This approach does not require any change to EA's source code and hence the proposed architecture is non-intrusive with respect to the source code of EA. However, this implies that the proxies are coupled,though loosely, to the EA. Hence different types of proxies are required to manage different types of EA and a change in the services exported by EA may require a change in the proxy as well.

4.6.2 Heterogeneity

Heterogeneity, in the context of EA, is the ability to manage a mix of device communication technologies using a single architecture. Heterogeneity is handled by the Gateway component which provides an uniform mechanism to access the services of EA. For example, to manage a mix of devices that are based on the technologies such as Bluetooth, IEEE 1394, X10, etc., the Gateway must be capable of communicating with the device using the corresponding technology. This would ensure the support for managing heterogeneity though at the cost of considerable complexity in the Gateway component.

The architecture of HomeWabash has not yet been evaluated for its scalability, security, support for remote maintenance and support for automatic generation of management applications.

4.6.3 Related work

In this section, we briefly describe related research in the area of management of SmartHomes and compare it with ours.

AutoHan: The AutoHan [14] system is a reference implementation of a SmartHome. AutoHan provides for remote management of SmartHomes. The devices in the AutoHan implementation export their services using XML [4]. The services are published as events and registered with a device registry named DHan. The devices use the Universal Plug and Play (UPnP) [5] Generic Event Notification Architecture (GENA) to send and receive events. The devices are monitored by subscribing to the events and controlled by modifying the device attributes in the DHan registry. The events themselves are transported to and from the device using HTTP. The IHAN residential gateway connects the devices to the Internet and provides for remote access.

OSGi: The Open Services Gateway Initiative (OSGi) [10] focuses on the residential gateway that connects the devices to the Internet. OSGi defines a Java-centric set of APIs to allow devices to bind to the residential gateway and thereby export their services for remote access. It also defines APIs that allow certain services such as a Digital Music Library Service or Security Service to bind to the gateway and hence push these services to the devices.

VESA: The VESA Home Networking Committee [6] proposes a home network architecture based on XML. In this model, each device holds an XML page that describes the attributes and services exported by the device. The device can be monitored by reading and parsing the XML page and it can be controlled by modifying the XML page. This requires the devices to parse XML data.

HomeWabash, like the other approaches described above, requires the use of a residential gateway for remote access to the services offered by the devices. All the three approaches described above require changes to the device, and hence to the EA, for management. For example, AutoHan requires an IP stack running on these devices, VESA requires the devices to be capable of handling XML data, and OSGi requires the devices to run a Java Virtual Machine(JVM). It may be possible for each of these requirements to be incorporated in the future devices but we believe that there will be no single specification/standard that all devices will meet. Hence it is important to handle this heterogeneity without any assumption about the kind of support in the device. OSGi, to some extent, is based on this assumption, i.e. as long as the device supports a JVM, the OSGi architecture can be used for its remote management.

Unlike AutoHan and VESA, Wabash does not require the devices to adhere to any one management standard/specification. Instead it is based on the assumption that there will be multiple standards used by the devices. Hence, HomeWabash, like OSGi, moves the complexity from the device to the residential gateway. In the HomeWabash architecture, the Gateway component is assumed to be able to communicate with heterogeneous device types. The moving of complexity to the residential gateway has an advantage that it can even support devices that do not run a EA. The current implementation of HomeWabash supports the control of X10 and IEEE 1394 devices which do not host an EA. Such devices are controlled through additional hardware in the residential gateway. This hardware can, for example, attach itself to the IEEE 1394 bus and sense the activity on the bus. This additional complexity in the residential gateway has the following advantages over the other approaches: (i) it does not require the device to run any specialized application and (ii) it does not require the device to implement any specialized protocol.

None of the other approaches, discussed above, use the proxy-based management as employed in HomeWabash. The proxy-based approach de-links the management architecture from the device network and hence leads to increased flexibility. For example, HomeWabash implementation uses the Java-based JMX management architecture. However, it is also possible to implement HomeWabash using CORBA-based Wabash architecture where the proxies are the CORBA-components to be managed. The proxy-based approach has its limitations. For example, it is possible that the state of the proxy and the actual device may not be synchronized at all times. This could happen when the device's state has been changed from within the SmartHome physically and the proxy's state has not yet been updated.

4.7 A XML based Policy-Driven Management Information Service

Due to the rapid growth in the number and flexibility of services in current networks and integration across organizational boundaries, present day and evolving distributed systems tend to be highly heterogeneous and dynamic. A management solution to such systems must possess the following attributes:

- Ability to delegate authority to the lowest possible level to handle the sheer scale of Internet based distributed systems.
- A vendor and implementation and operating system neutral information model for resource management to facilitate the interchange of management information. Such an information model should be capable adequately representing any entity of the system and is important given the heterogeneity and dynamic nature of emerging distributed systems.
- Ability to be configured at run-time to meet emerging requirements, either by providing additional functionality or by adapting to environmental changes. This ability is required to handle the dynamic nature of evolving network systems.
- Ability to interact and cooperate with other management solutions, in order to deal with inter-organizational integration and interaction.

Next we describe a prototype implementation of a management information server that can be used as a building block in creating a management system that exhibits the above properties.

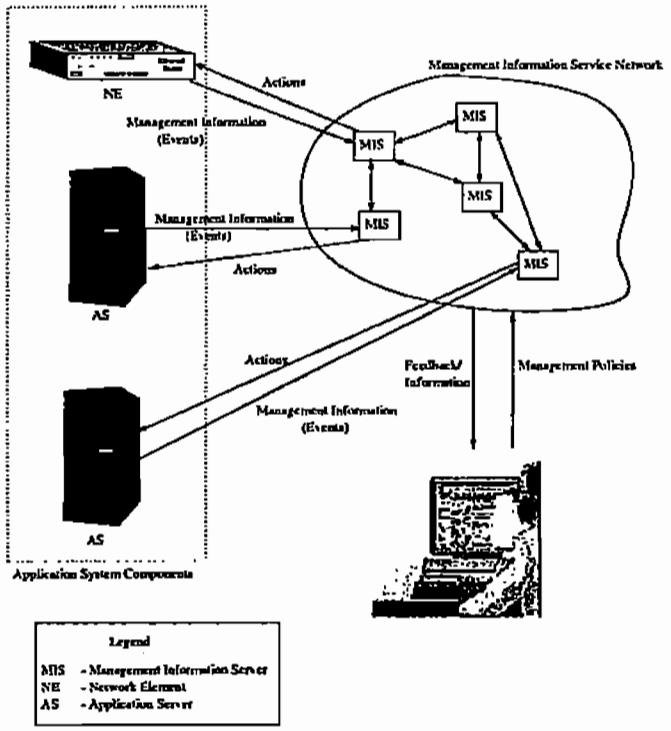

Figure 6: Architecture of a Management System built around a Management Information Server.

4.8 Architecture of a Management Information Server

In the context of the management of SmartHomes, management information consists of events and management policies. Events are generated by application components such as software objects, servers, smart appliances, and network elements. They typically provide operational data about the application. Policies are generated by management systems and are an expression of directives from the management that drive the application towards meeting goals specified by the management. At the information server level, policies would constitute a set of event-action pairs that express the management goal. These lower-level policies would be derived hierarchically from a set of higher-level policies that describe the management goal in increasingly abstract terms.

Our Management Information Server (MIS) is a management system component that receives events from an application components and acts on them based on specified policies. Thus, the management information server can be viewed as a generic extension of the basic unit of a publish and subscribe service. The type of actions that a MIS can perform on the information it receives can range from simple subscription based forwarding to sophisticated aggregation, filtering and logging of information. In addition, custom extensions can be provided to initiate specific actions based on recognized information. Figure 6 shows the architecture of our management system and where the proposed MIS fits in this architecture.

One sample usage scenario of the MIS is in managing a diverse set of web servers capable of externalizing events in some form. In this case, we can convert the externalized events into XML fragments which will form management information input to our MIS. The manager can now specify policies that take different actions,

either on the application system's components or otherwise, based on events. As an example, a policy to divert traffic or alert administrators in the presence of high loads can be enforced on a web server that is capable of providing traffic events per unit time.

The policies and management information in our system are specified in XML. We use a generic template for all incoming communications to the server. This consists of a standard template for a message that is divided into a header and body. The header serves as a wrapper that contains essential information such as message type, origin, destination, authorization information (if needed), timestamp etc.

Using the header, the server identifies the message and passes it on to appropriate subsystems for further processing. The body of a message also has a generic template that is determined by the contents of the message header. For instance, an event message would have a body that contains event type, event priority, timestamp, event text and other such event related elements.

Similarly a policy message would have a body that contains policy user (the intended subsystem that should use this policy), policy data (the actual policy contents), activation timestamp, active interval and other policy related elements. Each of these elements themselves can be XML fragments that can be further interpreted. For instance, the policy data can be a XML fragment that specifies the event template and the actions to be taken when events matching the templates are detected. Due to the self-descriptive nature of XML documents, we believe that this model will lead to easy extension of the generic templates to match any particular representation of events and policies chosen by the user.

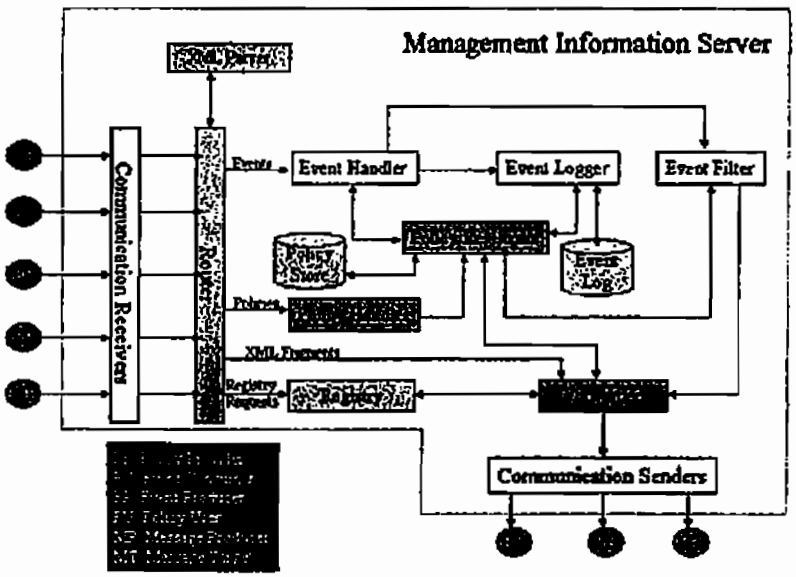

Figure 7: Architecture of the Management Information Server.

Figure 7 shows the internal architecture of a management information server. The operation of each functional block (sub-system) in the architecture can be independently controlled through management policies. The server prototype has been built in Java, and components of a single information server are capable of being

distributed over a network or can be run on one machine.

The router component provides the flexibility and extensibility of the system by identifying the generic templates and classifying messages based on this, passing them on to the relevant subsystem. The policy handler and policy registrar handle and store policies, respectively. The event handler uses policy registrar services to determine actions to be taken on receiving events from the application system components. The event logger is used to log events into a database for later analysis and audits. The event filter is the means by which intelligence and authority is delegated to the MIS. It aids the management decision-making process by performing sophisticated prioritization, filtering, aggregation, averaging, threshold detection and other operations on incoming events. The forwarder and registry service combine to form an efficient addressing unit to perform conventional subscription based forwarding of events.

5 Research Plan

Our goal is to design, build, and evaluate an infrastructure for the management of SmartHomes. Towards this end we have identified EA and DA to be the key types of applications in SmartHomes and have identified intrusiveness, heterogeneity, scalability, support for security, support for remote maintenance, and support for automatic generation of management applications as criteria for evaluating and comparing different architectures of the infrastructure. We have developed architectures, Wabash and HomeWabash, for the management of DA and EA respectively. Wabash and HomeWabash have been evaluated with respect to the intrusiveness and heterogeneity criteria and compared against similar architectures reported in the literature. In this section, we outline our plan for research in the near future.

5.1 Evaluation of Wabash

Section 4.3 describes the evaluation of the Wabash architecture with respect to the intrusiveness and heterogeneity criteria. In the following we outline how we propose to conduct further evaluation.

5.1.1 Scalability

We intend to evaluate the scalability of the architecture by obtaining quantitative measurements. The number of components to be managed and the bandwidth of communication links between the zones are most likely to be the key factors in determining the scalability of the architecture. We intend to obtain quantitative measurements of scalability by varying the two factors and then measuring the overhead of the architecture on (i) the service latency and (ii) the response time of management commands.

5.1.2 Support for automatic generation of management applications

Our future work in the area of building management infrastructure will focus on the generation of management applications. For this purpose, we intend to perform qualitative evaluation of the architecture for its support in generating management applications. The likely characteristic of a "good" support for such a cause would be the ability to "plug" in a new component into the architecture without requiring any modification. For example, the architecture currently supports the monitoring and control functionality. Suppose that it is required

to generate a management application to dynamically display the interactions between the components. Ideally, it must be possible to simply "plug" in the new component, that incorporates the new feature, into the existing architecture.

5.2 Evaluation of HomeWabash

Section 4.6 describes the evaluation of the HomeWabash architecture with respect to intrusiveness and heterogeneity. We are yet to evaluate the architecture with respect to the remaining criteria. Though support for secure management is important in the management of SmartHomes, it is not focus item in our research. In the following subsections, we describe our plan to complete the evaluation of the HomeWabash architecture with respect to heterogeneity, scalability, support for remote maintenance and support for automatic generation of management applications.

5.2.1 Heterogeneity

We believe that the support for heterogeneity is a key criteria for evaluating any architecture for the management of EA. There are at least two reasons in support of this belief. First, many device communication technologies such as X10, IEEE 1394, Bluetooth, etc. are expected to share the market of SmartHomes. Hence it is important for the architecture not be biased towards one or the other technology. Second, there are several competing communication standards for devices, each with their own advantages and disadvantages. These standards, at least those that survive, will likely have a share in the market of SmartHomes and hence the architecture must not be biased in favor or against one standard. We intend to perform qualitative evaluation for the support for heterogeneity by experimenting with various devices and device communication protocols and studying their impact on the components of the architecture.

5.2.2 Scalability

The HomeWabash architecture can be used by a Service Provider to manage one or more SmartHomes or a Home Owner to manage his/her SmartHome. Scalability of the architecture is important for both. In both cases, (a) the number of devices and (b) the bandwidth of communication links that connect the residential gateway to the Internet or an Intranet, are most likely to be the key factors in determining the scalability of the architecture. We intend to evaluate the architecture by varying (a) and (b) and studying the impact on factors (i)-(iv) described in Section 3.2.2.

5.2.3 Support for remote maintenance

Remote maintenance of the device is akin to remote debugging. During debugging of software, the state of the application is often required. It may also be required to invoke certain methods exported by the application. The HomeWabash architecture supports remote access to the device through the Proxy and the Gateway components. The state of the device can be obtained by extracting the state of the proxy. However, the state of the proxy and the device may not be synchronized. Invoking methods on the proxy does not have the same problem as the corresponding method, exported by the device, is invoked synchronously. The HomeWabash architecture does not fully support the task of remote maintenance. We intend to identify techniques, suitable in the context

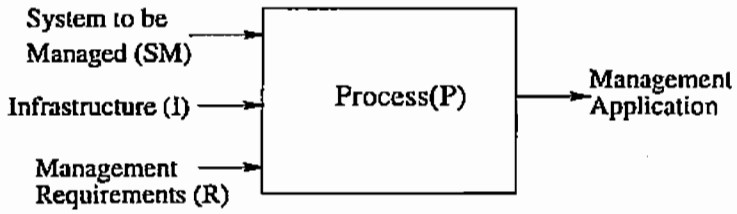

Figure 8: Process of generating management applications.

of SmartHomes, for tightly coupling the proxy's state with that of the device. We then intend to evaluate the support for this task qualitatively and also quantitatively.

5.2.4 Support for automatic generation of management applications

As our future work will focus on the automatic generation of applications for managing SmartHomes, it is important to evaluate the HomeWabash architecture with respect to this criteria. A key characteristic of the architecture would be its ability to let new components be plugged into the HomeWabash architecture seamlessly and without requiring any modifications to the architecture. For example, using the current implementation of the architecture, it is possible to capture the occurrence of events such as a *play* or *fastforward* request to the VCR. For example, if one would like to automatically generate a new management application that is also capable of *metering* the usage of requests, then it must be possible to dynamically plug in the new component into the HomeWabash architecture. We intend to qualitatively evaluate the architecture with respect to this criteria.

5.3 Automatic generation of customized,"pluggable" management applications

We have described two architectures for the infrastructure needed for the management of SmartHomes. Our future work in this area will focus on creating a process that will use the infrastructure for the automatic generation of management applications for SmartHomes. The generated application must be "pluggable" i.e. it must allow a new component to be dynamically loaded into the management architecture at run-time and without requiring any modification to the architecture. The management application must also be capable of handling customized specifications of the topology and interconnection of the devices in the SmartHome. Figure 8 shows the process (P) for generating management applications. To generate the management application, the process would use the following inputs:

- System (SM) to be managed
- Infrastructure (I) for the management
- Requirements (R) for the management

We plan to focus on identifying the needs and issues in creating such a process and building and evaluating a tool that automatically generates customized management applications.

5.4 Architecture of the ERNC system

An Event is an asynchronous occurrence containing parameterized details of an activity that has occurred within a distributed component. [15] For the purpose of the ERNC, we identify the following components that can generate events:

- Devices
- Residential Gateway
- Device Proxies
- Proxy Manager

As listed in Table 1, Home Owners, Service Providers and Device Manufacturers might all require the services of the ERNC system. However, their requirements from the ERNC system could be different. In addition, the level of access that they have to the ERNC system would also be different. While their requirements might be different, the essential task that each of them expects the ERNC to perform would remain the same. The following would be the essential task that would be accomplished by the ERNC system:

On the occurrence of specified event(s) take the following action(s).

The difference is thus modeled not through the functioning of the ERNC but through the specification of events and actions by the various categories of users.

The following model of the SmartHome system is used in constructing the ERNC system:
The physical devices are assumed to be capable of generating notifications corresponding to events that can occur within. Further, we assume that these notifications can be received by the Residential Gateway. Note that under some circumstances, the notification may not be directly generated by the device, but the Residential Gateway can itself detect the occurrence of the event and generate an appropriate notification. However, from the ERNC's viewpoint, such notifications are considered as directly originating from the corresponding device. Devices may or may not support a query and extraction of all possible notifications that it can generate. While we do not make this a requirement, the support of such query mechanisms enable easier interfaces. Given this model, our primary focus is not on how devices generate notifications, or how these notifications can be received. Instead, we focus on how the ERNC can interpret, match and take appropriate actions based on these notifications and on providing a flexible and powerful specification mechanism for event notifications and event-action pairs.
With the above model of SmartHomes, we can examine all communications going into the ERNC system as messages. Thus, both notifications of events and the specification of event-action pairs enter the ERNC system as messages. We have chosen to represent these messages as well formed XML documents. XML has the following benefits that we believe make it suitable for this purpose:

- Expressiveness
- Simplicity

- Human readable
- Easily parsed using commercial tools
- Easily transported across various communication mechanisms
- Vendor, Operating System and implementation neutrality

Appendix A lists the initial Document Type Definition (DTD) we have defined for specifying notifications and event-action pairs. As shown, every message consists of a fixed header and a variable body. The header identifies the type of the message used in the interpretation of the body. This allows the definition of an extensible set of messages where new message types are identified using the header and handled accordingly.

While the ERNC system is based on event notifications modeled as well-formed XML documents, we do not expect devices to be capable of generating notifications consistent with the format we have specified. Instead, we envision the use of translation mechanisms at the Residential Gateway that will translate the native device notifications into the corresponding template that the ERNC can recognize and act upon. This is in keeping with our architecture where the heterogeneity at the device level is handled by the Residential Gateway.

The device proxies can be designed to produce notifications in the desired format. However, if this is not possible (for instance, the device proxy was written by a different vendor and there is no access to the source code), a similar approach like the one used for devices can be adopted, where filters are used to translate device proxy notifications into the desired format.

In the current implementation, we have modeled the ERNC system as a service in the HomeWabash system. This service is provided by an Event Engine MBean that is registered with the JMX MBean server. With this representation, the ERNC system from a Home Owner's perspective, acts like any other device proxy. The ERNC system depends on the underlying communication technology used in the implementation of device proxies, for the distribution and delivery of notifications. The system leverages the JMX notification mechanism for the distribution and delivery of notifications. Identification of a common means for distribution and delivery of notifications in the presence of different communication mechanisms would be a focus area for future work.

An important part of our work will be identifying the requirements for a ERNC system in this setting and developing the ERNC message specification to enable the handling of a diverse set of event-action pairs and notifications. The requirements for the ERNC system needs to be thought out not only from the point of view of the Home Owner, but also from that of the Service Provider and Device Manufacturer.

The impact of the ERNC system on the evaluation criteria specified for HomeWabash needs further analysis. A well-designed ERNC system can facilitate the goal of automatic generation of customized "pluggable" management applications. However, the impact of such a flexible ERNC system on the scalability, latency and intrusiveness of the solution needs to be analyzed.

By implementing management policy at all levels, our proposed MIS-based architecture differs from conventional publish-subscribe semantics based systems. With the MIS we have developed a building block for an extensible, flexible management solution capable of handling heterogeneity. Future work involves experimental evaluation of the system in handling heterogeneity apart from efficiency, overhead and scalability of the system. The benefits and limitations of XML in expressing policies and events in such a scenario also need to be further analyzed.

5.5 Testing EA and DA

Testing of various components of an embedded and distributed applications often poses tough challenges. Traditionals techniques for testing software can be applied if the components are tested in-house. For example, when the owner of a camcorder reports a malfunction, the manufacturer could reproduce this malfunction in-house, test the application embedded within the camcorder, debug it, remove the error, and download the upgraded application via the internet. However, it might not be always easy or desirable to attempt to reproduce behavior in a laboratory setting. Instead, one might want to do so in-situ, i.e. while the presumably malfunctioning device or an software component is in its normal domain of operation.

We wish to study how monitoring and control might aid in remore testing and debugging of both EA and DA. This study is expected to lead to monitoring and control features specially suited for testing and debugging. Note that typical software debuggers can be abstracted as monitoring and control applications. We are interested in studying how the management infrastructure can itself support remote testing and debugging tasks.

Acknowledgements

Our sincere thanks to Balakrishnan Dasarthy, James L. Dixon, Frederick D. Porter, David Waring of Telcordia and David Griffiths, Richard Dennis, and Paul McKee of British Telecom for their support and the time they spared to discuss with us various critical issues in the SmartHome project.

References

[1] http://java.sun.com.

[2] http://java.sun.com/jini.

[3] http://java.sun.com/products/JavaManagement/.

[4] http://www.w3c.org/XML.

[5] http://www.upnp.org.

[6] http://www.vesa.org.

[7] E. Al-Shaer. *"Hierarchical Filtering-based Monitoring Architecture for Large-scale Distributed Systems"*. PhD thesis, Old Dominion University, July 1998.

[8] G. Eisenhauer and K. Schwan. "An Object-based Infrastructure for Program Monitoring and Steering". In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 10--20, Welches, OR, USA, August 1998.

[9] David Garlan and Mary Shaw. *"An Introduction to Software Architecture - Advances in Software Engineering*, volume 1. World Scientific Publishing Company, River Edge, NJ, 1993.

[10] L. Gong. "A Software Architecture for Open Service Gateways". *IEEE Internet Computing*, 5(1):64--70, February 2001.

[11] Visibroker for Java Programmers Guide. Version 3.2. Inprise Corporation, San Mateo, CA, USA.

[12] Object Management Group Inc. *The Common Object Request Broker: Architecture and Specification (CORBA)*. Revision 2. John Wiley, 1995.

[13] X. Logean. "Monitoring and Testing Tool for Distributed Applications". Technical report, Swiss Federal Institute of Technology, 1998.

[14] U. Saif, D. Gordon, and D. J. Greaves. "Internet Access to a Home Area Network". *IEEE Internet Computing*, 5(1):54--63, February 2001.

[15] Spiteri, M.D. and Bates, J. *An architecture to support storage and retrieval of events*, Proceedings of MIDDLEWARE 1998, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Lancaster, UK. Sept. 1998.

[16] B. Sridharan, B. Dasarathy, and A. P. Mathur. "On Building Non-Intrusive Performance Instrumentation Blocks for CORBA-based Distributed Systems". In *Proceedings of the 4th IEEE International Computer Performance and Dependability Symposium*, pages 139--143, Schaumburg, IL, USA, March 2000.

Appendix A: Document Type Definition for Event Notifications

```
<!DOCTYPE message [
<!ELEMENT message (head, body)>
<!ELEMENT head (sender, dest, authinfo, messtype, time)>
<!ELEMENT sender (type, desc, opt?)>
<!ELEMENT dest (type, desc, opt?)>
<!ELEMENT authinfo EMPTY>
<!ELEMENT messtype (#PCDATA)>
<!ELEMENT time (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT desc (simpledescr|devicedescr)>
<!ELEMENT simpledescr (#PCDATA)>
<!ELEMENT devicedescr (hostname, mgrname, domainname, devicename)>
<!ELEMENT hostname (#PCDATA)>
<!ELEMENT mgrname (#PCDATA)>
<!ELEMENT domainname (#PCDATA)>
<!ELEMENT devicename (#PCDATA)>
<!ELEMENT opt  (#PCDATA)>
<!ELEMENT body (info, text)+>
<!ELEMENT info (name, oldvalue, newvalue?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT oldvalue (#PCDATA)>
<!ELEMENT newvalue (#PCDATA)>
<!ELEMENT text (#PCDATA)>
]>
```

Document Type Definition for Event-Action Pairs

```
<!DOCTYPE message [
<!ELEMENT message (head, body)>
<!ELEMENT head (sender, dest, authinfo, messtype, time)>
<!ELEMENT sender (type, desc, opt?)>
<!ELEMENT dest (type, desc, opt?)>
<!ELEMENT authinfo EMPTY>
<!ELEMENT messtype (#PCDATA)>
<!ELEMENT time (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT desc (simpledescr|devicedescr)>
<!ELEMENT simpledescr (#PCDATA)>
<!ELEMENT devicedescr (hostname,mgrname,domainname,devicename)>
```

```
<!ELEMENT opt  (#PCDATA)>
<!ELEMENT body (eventactionpair)+>
<!ELEMENT eventactionpair (eventtemplate, actionlist, activateat?, validinterval?)>
<!ELEMENT eventtemplate (headertemplate, bodytemplate?)>
<!ELEMENT actionlist (action)+>
<!ELEMENT activateat (#PCDATA)>
<!ELEMENT validinterval (#PCDATA)>
<!ELEMENT headertemplate (sendertemplate?, desttemplate?, authinfotemplate?,
          messtypetemplate?, timetemplate?)>
<!ELEMENT bodytemplate (nametemplate, oldvaluetemplate?, newvaluetemplate?, texttemplate?)>
<!ELEMENT sendertemplate (typetemplate?, desctemplate?, opttemplate?)>
<!ELEMENT desttemplate (typetemplate?, desctemplate?, opttemplate?)>
<!ELEMENT typetemplate (#PCDATA)>
<!ELEMENT desctemplate (simpledescrtemplate|devicedescrtemplate)>
<!ELEMENT simpledescrtemplate (#PCDATA)>
<!ELEMENT devicedescrtemplate (hostname?, mgrname?, domainname?, devicename?)>
<!ELEMENT opttemplate (#PCDATA)>
<!ELEMENT authinfotemplate EMPTY>
<!ELEMENT messtypetemplate (#PCDATA)>
<!ELEMENT timetemplate (#PCDATA)>
<!ATTLIST timetemplate
 cycleperiod CDATA #REQUIRED
 fromtime    CDATA #REQUIRED
>
<!ELEMENT nametemplate (#PCDATA)>
<!ELEMENT oldvaluetemplate (#PCDATA)>
<!ELEMENT newvaluetemplate (#PCDATA)>
<!ELEMENT texttemplate (#PCDATA)>
<!ELEMENT action (forward | log | invokeop | generatenotification)>
<!ELEMENT forward (url)+>
<!ELEMENT url (mailurl | dburl | rmiurl | rawurl)>
<!ELEMENT mailurl (#PCDATA)>
<!ELEMENT dburl (hostname, portnum, instance, username, password)>
<!ELEMENT rmiurl (hostname, portnum?, objectname)>
<!ELEMENT rawurl (hostname, portnum, conntype)>
<!ELEMENT hostname (#PCDATA)>
<!ELEMENT portnum (#PCDATA)>
<!ELEMENT instance (#PCDATA)>
<!ELEMENT username (#PCDATA)>
<!ELEMENT password (#PCDATA)>
```

```
<!ELEMENT objectname (#PCDATA)>
<!ELEMENT conntype (#PCDATA)>
<!ELEMENT log (database | file)>
<!ELEMENT database (url, querystring, queryparamlist)>
<!ELEMENT querystring (#PCDATA)>
<!ELEMENT queryparamlist (queryparam)+>
<!ELEMENT queryparam (#PCDATA)>
<!ATTLIST queryparam queryparamtype (attribute|operation|oldvalue|newvalue|
          text|none) #REQUIRED>
<!ELEMENT file (#PCDATA)>
<!ELEMENT invokeop (optarget, opname, param*)>
<!ELEMENT optarget (hostname, domainname, managername, devicename)>
<!ELEMENT domainname (#PCDATA)>
<!ELEMENT managername (#PCDATA)>
<!ELEMENT devicename (#PCDATA)>
<!ELEMENT opname (#PCDATA)>
<!ELEMENT param (paramtype, paramvalue)>
<!ELEMENT paramtype (#PCDATA)>
<!ELEMENT paramvalue (#PCDATA)>
<!ELEMENT generatenotification EMPTY>
```