IMPLEMENTATION OF THE WOS(TM) PROTOTYPES

HAUKE COLTZAU, DANIEL BERG, HERWIG UNGER

FB Informatik, Universität Rostock, 18051 Rostock, Germany
{django|dabe|hunger}@informatik.uni-rostock.de

Abstract
This article describes basic implementation details of the WOS(tm) prototypes. Special focus is set to the WOS(tm) Warehouses, the central object for the Resource-management. It figures out that warehouses give the possibility to handle a huge amount of inhomogeneous data and allow users to configure and maintain the WOS(tm) system individually.

Keywords: Distributed Systems, WOS, Resource Management

1 INTRODUCTION

The bigger part of today's existing distributed systems is constructed to interlink a small or medium amount of hosts [5] [3]. A quasistatic network architecture is often assumed, i.e. the hosts and their location must be known. The subsequent resource management is based on central catalogues [5] [7]. New hosts are added to these catalogues during the installation and configuration (e.g. [7]). As far as apparent from the publications dealing with the resource management of particular distributed systems, only a few systems, like CORBA [17] [6] and Globus [1] [2] can be seen as exceptions. The CORBA Object Request Broker (ORB) manages a small amount of resources and intercommunicates with other ORBs. Hence follows that CORBA resources are organized globally decentralized but locally central. So CORBA provides scalability and distributed catalogues, which are requirements for the implementation of large distributed systems. But due to the fact that the ORB is generally configured for more than one host, a single user is restricted in possibilities of configuring his own resources. Furthermore the user is constrained to implement the management and access restrictions for his resources by himself.

Concluding can be mentioned that there is no distributed system that manages a big amount of resources efficiently and allows individual and easy-to-use resource configuration.

The Web Operating System (WOS(tm)) [10]-[15]] was developed to overcome the described difficulties and to manage the huge amount of resources in the Web. This article presents the state of implementation.
2 DESCRIPTION OF THE IMPLEMENTATION

2.1 Basic Implementation Decisions

One of the first but significant decision was to chose the programming language, WOS(tm) shall be implemented in. WOS(tm) 0.1 was implemented in C but that caused a lot of portability problems and results in a lot of OS-dependent WOS(tm) versions. Therefore, for WOS(tm) 0.2 it was decided to use the Java programming language, which is system-independent. Hopefully, the promise write once, run everywhere will come true for the WOS(tm). Another big advantage of Java is the set of network functions, which are easy to handle and run stable in most cases. For the revised warehouse concept, which will be described later, the dynamic classloader opens a lot of possibilities regarding to warehouse abstraction and extensibility. The Java Abstract Window Toolkit (AWT) allows to create a user interface in a system-independent manner and is available for all noteworthy graphic oriented operating systems.

A significant disadvantage is to be found in Java’s processing-speed. This is substantiated in the byte code concept of that language and can only be changed by either using native C-Code for time critical operations or by using java to native compilers, which are available for some operating systems.

2.2 The WOS(tm) Prototype - from 0.1 to 0.2

The first prototype was implemented to find out, whether the idea of the WOS(tm) is realizable or not. The attention was turned to analyze the structur of a WOS(tm) node and its components. Questions regarding to security, portability and handling would be implemented in later versions. WOS(tm) 0.2 starts to deal with that questions, resulting in following changes:

**Programming Language:**

- WOS(tm) 0.1: was implemented in C and partially Tel/Tk, because of being close to the UNIX-system. This caused problems with portability.
- WOS(tm) 0.2: is implemented in Java.

**User Interface:**

- WOS(tm) 0.1: used CGI-Scripts started via netscape to offer a GUI and could be started from any Client.
- WOS(tm) 0.2: uses the Java Abstract Window Toolkit and binds the GUI to one host. See figures 2 and 4.

**Communication:**

- WOS(tm) 0.1: every communication was done using UNIX-native TCP sockets. There were no fault-detection mechanisms.
WOS(tm) 0.2: WOSRP/WOSP Communication layer and protocol enable system independent Java-socket based communication with fault-detection.

Global Service Detection:

- WOS(tm) 0.1: html resource warehouses were registered at AltaVista. A Tcl/Tk demon contacted AltaVista automatically and analyzed the warehouse of the remote host.
- WOS(tm) 0.2: WOSP-based search chains, started by the active warehouses.

Warehouses:

- WOS(tm) 0.1:
  - static databases, managed by RCU and RRCU
  - three different types with own access functions, which are described later.
- WOS(tm) 0.2:
  - active Warehouses, which are able to contact other warehouses
  - one common warehouse defining access methods and interfaces

2.3 The structure of a WOS(tm) 0.2 node

As shown in figure 1, a WOS(tm) 0.2 node consists of following components:

User Interface (UI): the User Interface is subdivided in two parts, the request menus and the warehouse editor. The request menus help the user to initiate a service request with special parameters. Possible parameters are for example: the preferred architecture and operating system, the execution time and -endurance, and the maximum charge that may be payed for the service. The User Interface analyzes the request data and launches the Resource Control Unit.

Resource Control Unit (RCU): The RCU accepts service requests from the User Interface and contacts several known warehouses to find a WOS(tm) node, where the requested service can be executed. First, the local service warehouse is contacted, then the wosnet warehouse. If no service was found, the search via AltaVista was initiated. If a server was found, the RCU asks for service execution and returns the results to the User Interface. After successful execution, the local warehouses are updated.

Remote Resource Control Unit (RRCU): The RRCU receives service requests from other WOS(tm) nodes and examines, whether execution is allowed or not. Therefore the resource warehouse is asked. The RRCU transmits the answer to the Client - RCU. The service execution itself is also managed by the RRCU, which contacts the resource warehouse a second time to verify the access rights. After that, the service is executed. In WOS(tm) 0.2, services are forced to save their results in a HTML file, which is sent to the Client RCU.

193
Communication Layer (WOSRP/WOSP): The Communication Layer provides system independent methods and interfaces. A WOS(tm) specific communication protocol (WOSP) can hold and transfer any serializable data. Comparable to UNIX, a demon (WOSRP) manages the sending and receiving of messages.

3 WAREHOUSES IN THE LOWER WOS(TM) - LEVELS

3.1 Description of Warehouses

The main task of a warehouse is to store, manage and maintain WOS(tm) specific data. It must be considered that the stored data differs in meaning and structur, depending on the special task of a warehouse. Therefore, there is a need of a common warehouse implementation, which meets following requirements:

- store any kind and type of data
- offer proper access methods, which can access any data in any warehouse
- define task-specific interfaces for the interpretation and management
Figure 2: The Service Request Window
3.2 Realisation of Warehouses in the WOS(tm)

To meet the requirements, WOS(tm) warehouses are implemented in a binary tree structure, as shown in figure 3. The value of a node can be any serializable Java-Object. Every node also has a left and down reference, so the structure can be described as level-oriented and multi-nested. The first three levels of this structure are called Entry, Criterion and Value.

![Diagram of warehouse structure](image)

**Figure 3: the structure of a warehouse**

An Entry concludes all information of one Warehouse-dataset. A name is not requested, but could be used as a sort-qualifier to appreciably decrease the access-times for huge warehouses. A Criterion allies all data of one special type, which are the Values. It has a significant role regarding to common interpretation interfaces, as described in the next section.

3.3 The Dynamic Link Classes (DLC)

Java provides the possibility of loading classes dynamically into the virtual machine at runtime. Using interfaces makes it possible to define the functionality for any warehouse type by defining methods, that are expected from the loaded class. Now, those methods can be accessed, whereas the Values of one Criterion are parameters. An example shall clarify this:
One task of the resource warehouse is to compare incoming service requests to its stored data and to decide, whether the execution of the requested service will be allowed or not. For that purpose, an interface StdResourceWarehouse must be defined, which contains among others the following prototype:

```java
public boolean isIn (String req, Tree val)
```

The next step is to create a class, which implements exactly that method for one Criterion. For example, a class IPList would expect `req` to be the IP of the requesting client and `val` to be the list of all IPs that are allowed to execute the requested service. If `val` includes `req`, `isIn` returns `true`, else `false`. In case of `req` is empty, `isIn` returns `false`.

At runtime, any service request is handled following the same scheme:

```
Repeat
    load Entry
    Repeat
        read Criterion
        load DLC
        result = DLC.isIn
        While (more Criteria AND (result = true))
        While (more Entries AND (result = false))
    If (result = true)
    Then allow execution
    Else do not allow execution
```

If a new `Criterion` shall be added or an old one shall be changed, the method `isIn` must be implemented in a new class respectively changed in the existing one. For some cases, WOS(tm) provides already implemented standard-classes, like `IPList`, `TimeList` and `Services`.

### 3.4 Warehouses in WOS(tm) 0.2

In WOS(tm) 0.1 and 0.2, there are three warehouses:

- **Resource-Warehouses** are the central objects in the WOS(tm) resource management. They include all information about services a WOS(tm) node offers. Generally, entries answer the question: *Who may execute which service within which period of time and are there any other restrictions?*

- **Local-Warehouses** contain information about those services on other WOS(tm) -nodes, that have already been successfully executed previously. They simplify the search for requested services and as a result decrease needless and expensive network-traffic.

- **Wosnet-Warehouses** give a general survey of known WOS(tm) -nodes and clusters.

To expand and improve the service-search, the Resource-Warehouse gets more functions. Now, this warehouse is able to forward requests to other known warehouses und return their reply to the requesting client. So now it is possible to realize search chains. The static and passive warehouse becomes an active Warehouse.
Figure 4: The Warehouse Editor
3.5 Future Warehouses

If a warehouse grows, it gets more difficult to maintain and update the mass of data, that is stored. Since the Wosnet is based on the Internet, hosts can be added or removed. The availability of services underlies the same dynamics. From this it follows the requirement of self-organized warehouses, which have to

- search services frequently to allow the fastest reply to its users requests
- update known services to keep the essential data as small as possible and to decrease needless network-accesses during the service-search.

Those adaptive warehouses are in a position not only to decide where they can get information from (active warehouses), but also which information is needed to fulfill the users demands best. This decision can be reached by means of previously defined profiles and determined statistics.

4 CONCLUSION

This paper shows the state of the implementation of the WOS(tm) prototype. We have released version 0.1, that offers the opportunity to include further parts of the WOS(tm) system to all members of the WOS(tm) team and builds a demonstration tool for the achievements that can be reached using the WOS(tm) philosophy.

Version 0.2 continues this works, especially the resource management has been improved. The implementation of version 0.4 is still going on. We intend to use an execution manager, improve the fault tolerance as well as the security mechanisms and build in the WOS(tm) file system. We expect version 0.4 to be ready in october 1999.
REFERENCES


200


