

***ObServer: An Architecture to
Monitor Networked Services with Java***

Roland Robe*, Robert Ott**

* Roland Robe, UBS AG
Postfach, CH-8098 Zurich, Switzerland
Phone: +41 1 238 26 81, Fax: +41 1 238 27 40
E-mail: roland.robe@ubs.com

** Robert Ott, Professional JNet Solutions AG (correspondence)
Postfach, CH-9542 Muenchwilen TG, Switzerland
Phone: +41 71 960 08 10, Fax: +41 71 960 08 14
E-mail: ott@jnet.ch

ObServer: An Architecture to Monitor Networked Services with Java

Roland Robe*, Robert Ott**

* Roland Robe, UBS AG
Postfach, CH-8098 Zurich, Switzerland
Phone: +41 1 238 26 81, Fax: +41 1 238 27 40
E-mail: roland.robe@ubs.com

** Robert Ott, Professional JNet Solutions AG (correspondence)
Postfach, CH-9542 Muenchwil TG, Switzerland
Phone: +41 71 960 08 10, Fax: +41 71 960 08 14
E-mail: ott@jnet.ch

Abstract

This paper describes an architecture called 'ObServer' that enables the observation of distributed services in a platform independent way using the Java platform.

Implementations of the ObServer architecture aim that a notification occurs when important services fail in their functionality and to initiate escalation procedures as necessary.

The architecture is divided into two major components that are ObserverManagers and attached Consumers. An ObserverManager is a container of ServiceObservers monitoring certain services. ServiceObservers report status information to its ObserverManager. ObserverManagers are able to publish such status information to various kind of Consumers. Implementations of Consumers can then display status information to end users or initiate escalation procedures when one or more services fail in their operation.

The paper also shows a sample application of the ObServer architecture. The example does observe various distributed services around the world. The services which are monitored in the example do include web servers (plain communication), SSL web servers (encryption using server certificates), web servers that requires a sign on (automatic sign on and content verification) and services that require non-http protocols.

Even though the described example shows mainly an observation of network based services, the ObServer architecture could easily monitor other services such as a security system of a building where electrical sensors (motion detectors, infra red barriers) need to be observed.

Introduction

Every system depends on its environment and therefore on the predictable and unpredictable conditions of that environment. This rule applies especially for networked computer systems that often fail due to the complexity of a network or due to some software bugs which only occur when the system is heavily used by a large number of end users.

There are basically two ways to increase the availability of networked computer systems. One solution could be to evaluate a system which is actually bullet prove. However, waiting for such systems does mostly end in a wait-forever-loop because there are almost always situation where a system could fail. Another solution to get systems productive is to find a way how a system can be adjusted to its environment. A third solution could even be to provide a system to end users which still contains some well known bugs and where these users are expected to use only the working functionality of that system. This third solution has often been applied when applications and operating systems get deployed to end users which are in reality quite buggy, but which provide user friendly graphical user interfaces.

The availability of systems such as application servers or security control software of buildings must be very high. In addition, escalation procedures must be defined how to react when such a system would fail and how it can be restarted to continue its normal operation.

The ObServer architecture described in this paper shows how various systems can be observed by an observer manager process. Such an observer manager process does also act as publisher of status information that can be used by Consumers. Consumers can use the status information to act as control agents to restart certain services or to notify a support crew with means such as via SMS (GSM mobile phone Short Message Service) or e-mail.

The ObServer Architecture

The architecture of the ObServer system can be divided into two major components. These components are the ObserverManager and a number of Consumers. In addition, an administrator interface is provided by the ObserverManager for configure purposes.



Figure 1: The ObServer architecture overview

Figure 1 shows the basic components of the ObServer architecture that will be described in detail in the next section. This section should give a basic understanding and definition of the terms ServiceObserver, ObserverManager and Consumer that will be used as references in the further sections of this paper.

The term ServiceObserver is used to identify an object that does know how to observe (monitor) a running service. An observed service can be anything that can reply to requests. The communication mechanism of requests and replies include all communication styles that are supported by the Java platform. Examples for such communication styles include, but are not limited to, TCP/IP, UDP/IP (core Java), serial (RS-232) and parallel port communication (Java extension javax.comm [Jcomm99]).

The term ObserverManager is used to identify a Java process (JVM) that runs an instance of an ObserverManager. An ObserverManager manages registered ServiceObserver instances, provides an administration interface (via TCP/IP socket communication) and publishes status information of its registered and running ServiceObserver instances.

The term Consumer is used to identify processes that consume status information of one or more ObserverManager instances. A Consumer knows how to react on status information that is received from ObserverManagers. One implementation of a Consumer could send SMS messages to mobil phones when an observed service fails while another Consumer could be able to restart a failed service directly.

The ObserverManager Component

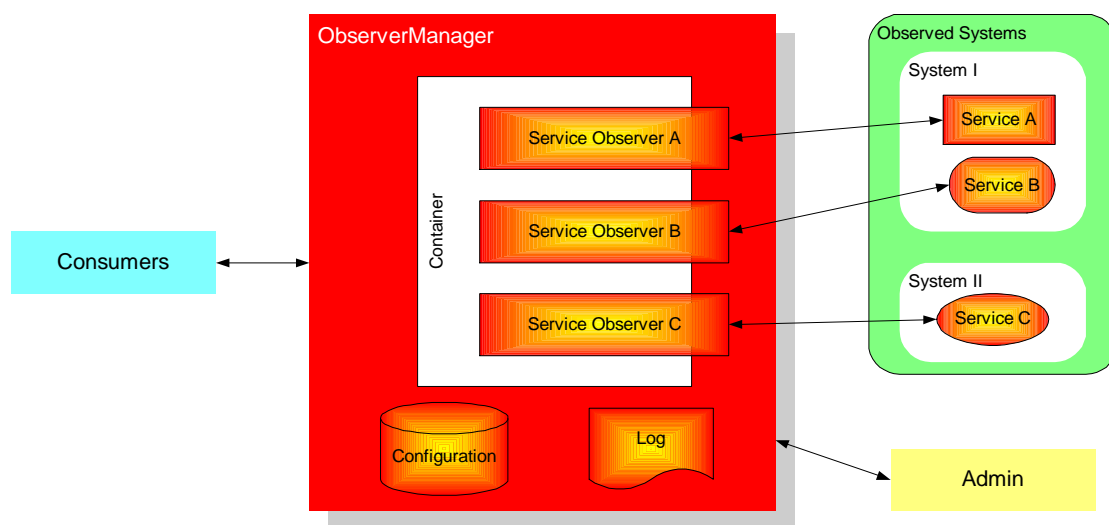


Figure 2: The ObserverManager component

Figure 2 shows an ObserverManager and its internal components in detail. An ObserverManager does hold all its ServiceObserver instances in a container object. In addition, an ObserverManager does have access to a persistence engine to hold configuration data of registered ServiceObserver's in a persistent way. Furthermore, a log engine is attached to the ObserverManager so that it can put logs status information of monitoring ServiceObservers for later report purposes. The figure also shows that the ObserverManager can publish status information of running ServiceObservers to attached Consumers.

Due that the ObserverManager facilitates the ServiceObserver instances with general facilities (engines and publishing interface), the ServiceObserver instances can concentrate on their only responsibility to observe services. ServiceObserver instances are also encapsulated from the underlying communication mechanisms to the ObserverManager's facilities.

The ObserverManager is also responsible for creating threads for the ServiceObserver instances and to start respectively stop these threads when necessary. As a result, the ServiceObserver's do not need to care about the complexity of multi-threading and can assume that every ServiceObserver instance is called with its own, single thread.

The ServiceObserver Component

```
public interface ServiceObserver {  
  
    public void init(ObserverManager manager);  
    public void destroy();  
  
    public ParameterList getObserveParameters(ObserverManager manager);  
    public PropertyList getStatusProperties(ObserverManager manager);  
  
    public void observeSystem(ObserverManager manager);  
  
}
```

Figure 3: The ServiceObserver Java interface

A ServiceObserver is an implementation of a common Java interface that must implement five predefined methods (see Figure 3). These methods are `init()`, `destroy()`, `getObserveParameters()`, `getStatusProperties()` and `observeSystem()`.

The `init()` and the `destroy()` methods can be used for initialization and cleanup purposes.

The method `getObserveParameters()` needs to return a list of parameter descriptors similar to a Java BeanInfo class. These descriptors are used to provide a flexible

administration interface that can be implemented independent of future implementations of ServiceObserver classes. For instance, an implementation of a HttpServiceObserver class would have parameters such as an 'URL' parameter (request URL), a 'Validation Content' parameter (whether or not the requested document contains the validation content) and a 'Max. Request Time' parameter (defines a time limit in which a page must arrive on the observer instance). Such an HttpServiceObserver class can also easily observe pages which are protected by SSL (Secure Socket Layer) using an implementation of the standard Java cryptography extension (e.g. the library provided by IAIK [Iaik99]).

The method `getStatusProperties()` needs to return a list of status property descriptors similar to the parameter described in the last paragraph. Status properties can be published and logged by the ObserverManager. All ServiceObservers have at least one property which defines the current status of the observer. This general property can have four major levels that are "OK", "WARNING", "ERROR", and "FATAL". In addition to this general property, a ServiceObserver class can implement its own additional properties.

The method `observeSystem()` does have the responsibility of observing its service. The method is not required to implement a loop, because it is called by the ObserverManager in certain time intervals defined using the administration interface. This method takes the observe parameters and fills the status properties after a system observation.

The Consumer Components

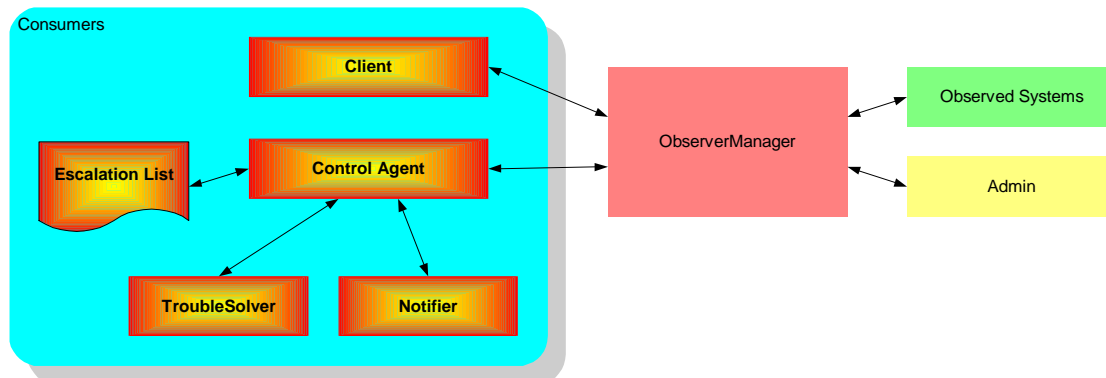


Figure 4: Consumer components

Figure 4 shows Consumers of an ObserverManager instance. The connection between the Consumers and the ObserverManager follows the basic publish/subscriber pattern. The publisher/subscriber pattern can be implemented by using communication API's such as iBus [iBus99], any of the JMS (Java Message Service) implementations [JMS98] or it can be implemented using direct TCP/IP socket communication by applying the java.net package. The implementation of the publish/subscriber pattern can therefore be a poll mechanism as well as a push mechanism. However, a poll mechanism is usually appropriate for most Consumer implementations.

A Consumer can be a client application that provides a graphical user interface showing status information of running ServiceObservers to the security staff of a company. Such a visualization client can gather the available status properties of the ServiceObservers through the ObserverManager and present these properties to end-users.

Other Consumers can be implemented as control agents. One instance of a control agent could try to restart a failed service automatically (a TroubleSolver) while other implementations of control agent could send SMS messages to mobile phones (Notifier). More complex implementations of control agents could look up predefined escalation procedures (Escalation List) and initiate workflow processes.

Sample Applications of the ObServer Architecture

Example of a Web ServiceObserver

Lets assume a company's department runs some web applications which are used around the world by their employees. Web servers do basically publish information and let employees use some HTML based applications. The applications provided by the department are business critical for employees as well as the management. Therefore, the web servers need to provide access at almost all time. The department decided to monitor all business critical http accesses of their web servers. Several monitor tools such as WebTrends [WebTrend99] have been evaluated but were limited in their functionality to run on different platforms and to analyse the http responses correctly.

Evaluated standard monitor tools have failed by reporting successful running web servers even though endusers were provided with error pages instead of the actual web applications. Furthermore, some web pages are protected by username and password authentication and are encrypted by SSL. Standard monitoring tools were also blocked out here. Reasons enough for the department to watch out for a solution for proper web server monitoring.

The example shown in Figure 5 gives an overview of a possible implementation of the ObServer architecture. The system called 'ObServer Client' enables to check the overall performance between the system in Zurich and NewYork and visa versa. Lets assume a user in Bogota does have access to the network in Zurich but the user has difficulties to see the web pages on the web server located in Zurich. The system administrator advises the user to download a ServiceObserver as a signed applet provided by the 'Dedicated ObServer' system in Zurich. As soon as the applet is running inside the web browser of the end user, the system administrator can see the view from the user in Bogota. The administrator is

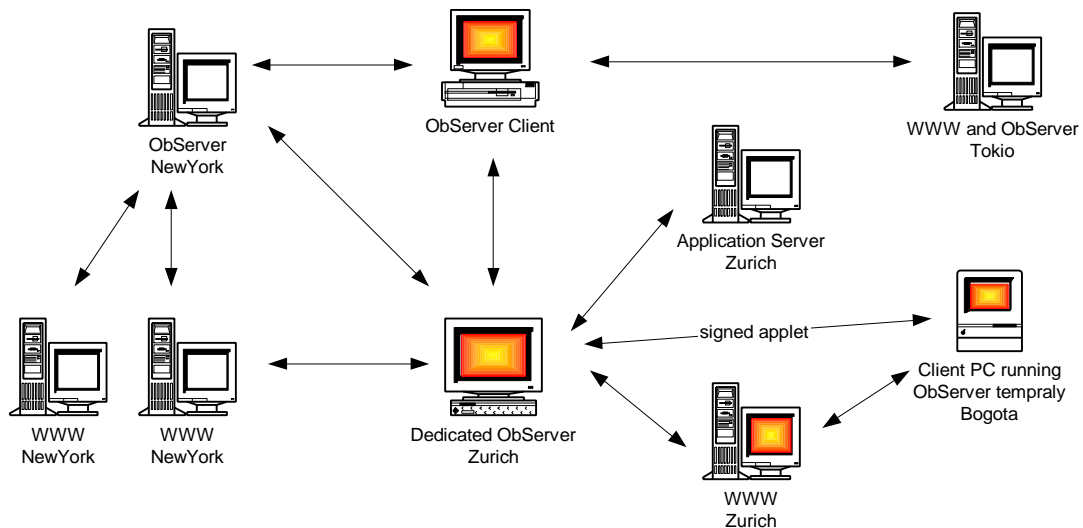


Figure 5: Observing distributed Web servers and applications services

now able to figure out whether the problem occurs from a miss configuration of the end user's system or whether the problem comes more from the network itself.

Due to the platform independent implementation of the ObServer architecture, it is easily possible to monitor systems from almost every system in a network. Standard monitor tools run either only on one node or they are mostly limited to run on a single operating system. Web pages do look different from different places because there are almost always firewalls, routers and proxies between these places. As a result, systems need to be observed from different locations within the network and the results should be report the operators of these systems. Implementations of the ObServer architecture using the Java platform do usually have a very small footprint and can easily be deployed on almost every platform. As mentioned, Service observer can even run within applets of end users.

Figure 6 shows how such a ServiceObserver can run inside a web browser's Java applet. The user simply clicks on a web link and receives an applet that contains a ServiceObserver instance. The downloaded ServiceObserver instance does also have a stub-communication piece that is connected to a skeleton-communication piece on the container of an ObserverManager. The downloaded ServiceObserver is now able to observe a system

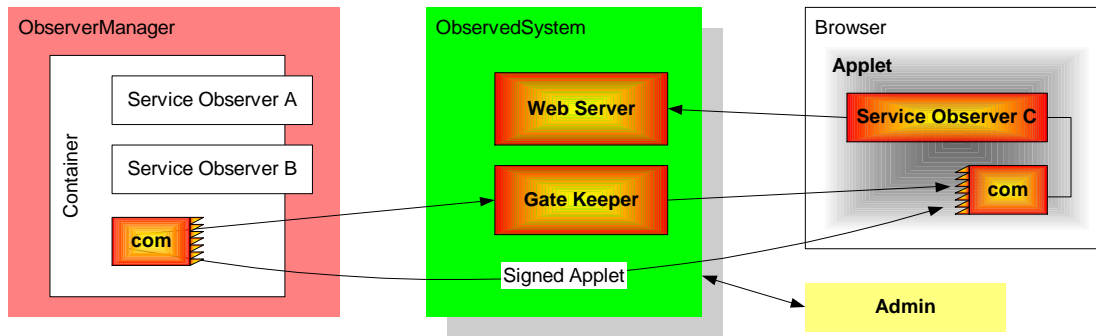


Figure 6: Observing services from a client's perspective

on one side and to communicate with the ObserverManager on the other side. A system administrator can now change observe-parameters and view observe properties of the ServiceObserver within the applet through the ObserverManager (as a Consumer).

The standard Java security model of applets does restrict applets to only communicate to that web server from where the applet has been loaded. Either an applet must be signed to get more access to other network services or it must communicate through a gatekeeper process running on the web server from where the applet has been loaded.

One of the world's leading financial services groups, the UBS AG, did implement the ObServer architecture. Various kind of ObserverManager's and ServiceObservers are running at UBS AG. These ServiceObservers do check the availability of free disk space, the performance and availability of systems, the number of open unix file handles, web applications, central login procedures and much more. Distributed ServiceObservers monitor the web from different locations and different operating systems. Some ServiceObservers do even observe other ObserverManagers within the network! Some ObserverManagers run on dedicated ObServer systems while others run on the same systems as web servers are running.

The simple ServiceObserver Java interface allows to implement new ServiceObservers within hours!

Other Examples of ServiceObservers

The web is only one example to apply of the ObServer architecture. The architecture allows to observe anything that is be able to reply on requests. Lets think about houses and buildings. An ObserverManager could inform you about open doors or windows when leaving the house. An implementation of a ServiceObserver could use an implementation of the EIBAgent architecture [EibOtt99] to access the status of EIB [Eib98] motion detectors or infra red barriers components of a building. Consumers could then send SMS messages when a system fails or an unexpected situation occurs.

Possible examples of ServiceObservers could also include the observation of free lines of a private branch exchange (PBX) of a phone center, the monitoring of the temperature of a server room or even to monitor the state of a traffic light system. Further examples are probably only limited by our fantasy.

Conclusions and Summary

Sure, huge systems like IBM-Tivoli [Tivoli99] or CA Unicenter [CaUni99] are very powerfull and provide a wide spectrum of functionality to increase the availability of systems. However, these full featured systems are in many cases to huge or to expencive for just monitoring one or more systems. In many cases, an implementaion of the ObServer architecture is more than sufficient to increase the availability of a system.

The ObServer architecture described in this paper can be used to increase the availability of services in a platform independent way. The Java platform provides the necessary portability across different operating systems and environments. Due that Java components can run inside web browsers, systems can even be observed from a users perspective without any installation on the users computer.

One of the world's leading financial services groups did implement the ObServer architecture to increase the availability of internal web services that are provided to about 30'000 users. SMS messages get sent to mobile phones and pagers of support staff as soon as certain services decrease their availability. As a result, failed systems can mostly be restarted or tuned before end users even realize that some services failed in one or another way.

However, the best ObServer implementation can not undo software bugs or systems that are not tuned. It is still extremely important to carefully evaluate and test systems before these systems are provided to end-users. Especially networked services that are mostly provided to a large number of users must run on operating systems that provide the required availability.

An implementation of the ObServer architecture, however, does increase the availability of services and should therefore be applied to increase the overall satisfaction of end users.

Scott McNealy said once "The Network Is The Computer™" [Nealy85] which became reality in the last decade. Therefore, we need to guarantee the availability of networked services by applying architectures such as the ObServer architecture in our environment!

References

- [CaUni99] Information on Computer Associates Unicenter is available at <http://www.cai.com/products/uctr.htm>
- [Eiba98] EIB Handbook, Rev. 2.21, EIB Association, Brussels, Belgium, 1998
- [EibOtt99] Connecting EIB Components to Distributed Java Applications, presented at ETFA'99, October 1999, Barcelona, Spain
- [Iaik99] Institute for Applied Information Processing and Communications, Graz University of Technology, Austria, information available at http://jcewww.iaik.tu-graz.ac.at/IAIK_JCE/jce.htm
- [iBus99] Information on iBus is available from Softwired Inc. at <http://www.softwired-inc.com/products/ibus>
- [Jcomm99] Java Communication extension specified by JavaSoft, available at <http://java.sun.com/products/javacomm>
- [JMS98] Java Message Service specification version 1.0.1, available at http://jcewww.iaik.tu-graz.ac.at/IAIK_JCE/jce.htm
- [Nealy85] "The Network Is The Computer™" is a slogan by Scot McNealy from 1985 and a trademark of Sun Microsystems Inc.
- [Tivoli99] Information on Tivoli is available at <http://www.tivoli.com>