Notification Server

Robert Hirschfeld
hirschfeld@windwardsolutions.com

Jeff Eastman
jeff@windwardsolutions.com

28. July 1998

Abstract

In enterprise information systems based on a two-tier distribution architecture, there are several clients working with shared resources. When designing the system you have to ensure that each client has a consistent view on the current state of the shared resource. If the resource in question is passive, i.e. the resource is not able to notify interested clients about changes of its (internal) state, attaching a *Notification Server* to that passive resource helps achieve a consistent view for each client.

Name

Notification Server

Aliases

Remote Dependents Collection

Context

You are developing an application for a distributed enterprise information system where desktop clients will access or modify shared resources (usually located within a passive[1] database server). You have decided to design your application based on a two-tier distribution architecture (figure 1).

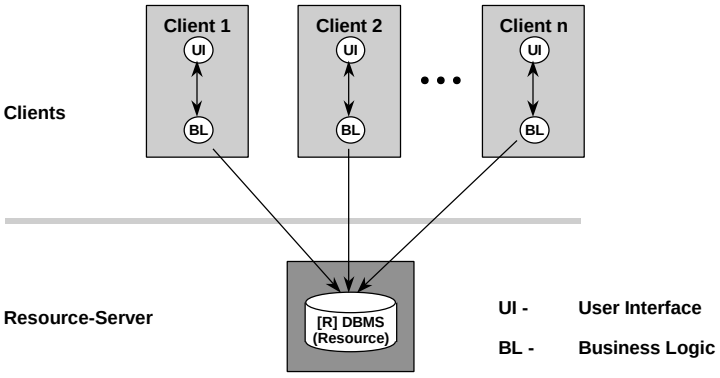

Figure 1

[1] To my mind the context can be widened to passive resources in general ...

Problem

How can you ensure that each client has a consistent view of the current state of the shared passive resource so that all users are in sync while working with the shared information?

Forces

Passivity. Passive resources (e.g., traditional databases) can not inform their clients about changes of the information they hold.

Consistency. Clients that are sharing a passive resource have to update their views to consider the current state of the resource while performing their responsibilities. This may require continuous checking (i.e. polling) of the server to detect changes, resulting in extra network load (figure 2). Depending on the polling time-frame/frequency clients become temporarily inconsistent with the database or may be affected by network traffic problems.

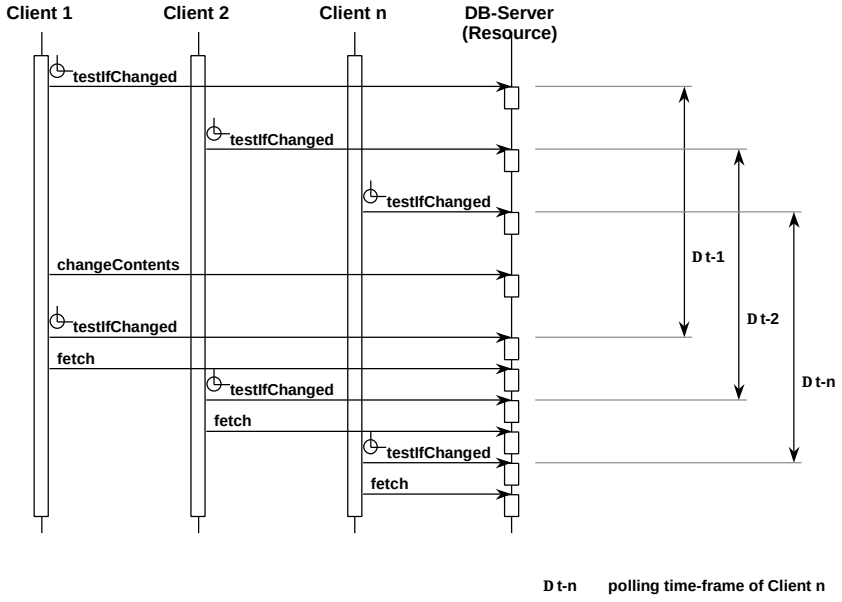

Figure 2

Generalization & Reusability. To be independent from a specific resource and its API, a possible solution should be lightweight by having no interaction with the resource itself.

Solution

Introduce a *Notification Server* into your system architecture. This *Notification Server* is a system infrastructure component that is separate from and independent of each client and the shared resource, too. It is known to all clients, so every client may access the

Notification Server in order to register and unregister for change notifications, to notify other clients about changes the client caused in the database or to receive and react on change notifications any other client caused.

Participants

Shared Passive Resource. A shared resource is a part of the system that might be used by several potential clients. This resource is called passive if it is unable to notify its environment (the potential clients) about changes of its state.

Clients. Clients have to have access to the shared resource to fulfill their responsibilities. They may read as well as modify the resource's state.

Notification Server. A Notification Server is an entity attached to a shared resource. It allows clients to register to receive notifications when the state of the shared resource changes.

Structure

Figure 3 shows the basic structure of a system based on the Notification Server.

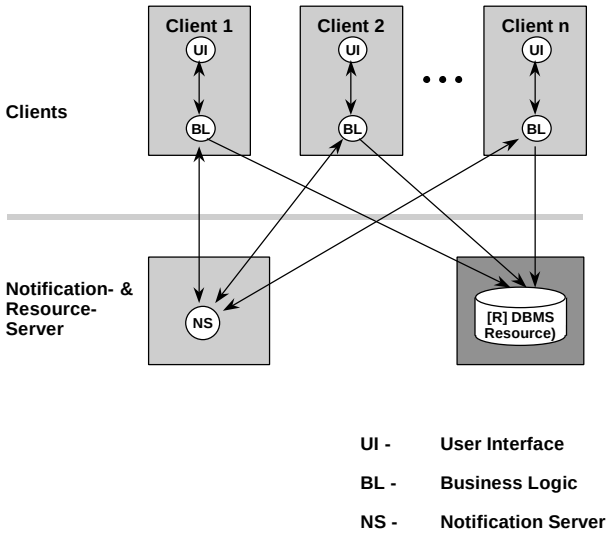

Figure 3

Dynamics

Every client can join a group of clients sharing certain resources (that is a database or a part of a database) by registering itself on the *Notification Server* responsible for these resources. After a client has changed a database resource, it informs the *Notification Server* about that change and the *Notification Server* in turn informs all the other registered clients (figure 4). A client that was notified about some changes may react on this notice by refetching information from the database. If a client isn't interested in

change notifications related to some resources anymore, it leaves the group by unregistering itself from the responsible *Notification Server*.

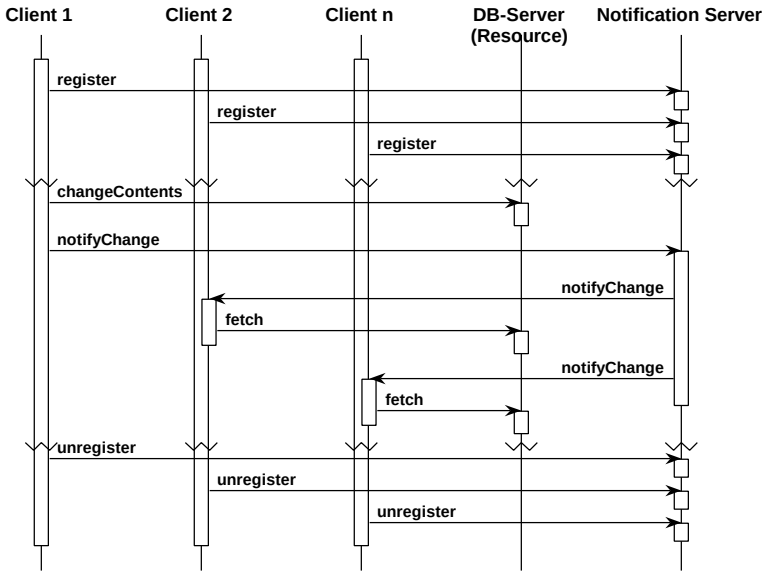

Figure 4

Consequences

Network Traffic. Because all registered clients are notified after a change, there may be high network traffic while these clients access information that was probably affected by the change.

Clients' Responsibility. It is difficult to require all possible clients of a shared resource to use its associated *Notification Server(s)*. Therefore all system parts sharing a resource have to agree to use its *Notification Server(s)* to coordinate notification of possible resource modifications.

Implementation

Granularity. The application of a *Notification Server* is not limited for a database as a whole. Instead you could divide your database conceptually into smaller parts that can be handled even as passive resources with their own *Notifications Servers*. You should try to divide every resource into disjoint parts to avoid additional and sometimes useless notifications. If you are not able to obtain a disjoint division, you have to ensure that clients working with those resources are registered by all *Notification Servers* of the joint set of resources.

Communication. If your applications are intended to run within a distributed heterogeneous environment, you should decide to apply standards like CORBA and CORBAservices from the Object Management Group (OMG) as much as possible ([OMG94, OMG95, OMG96]).

Notification. It will take some time to inform all registered clients about recent changes if there are many of them registered with the Notification Server. Do not use a synchronous communication model to avoid the blocking of the notifying client or the Notification Server - not only from the client to the Notification Server but also from the Notification Server to the other clients. The Event Notification Service will work best for an asynchronous decoupled communication ([OMG96]).

Callbacks. When you implement the callback mechanism that notifies a client about notifications multicasted by the *Notification Server*, apply the concepts of the Command[2] pattern to construct a simple and flexible client interface ([GHJV94]).

Initial Connection. If a desktop client wants to establish an initial connection to the Notification Server (e.g. for registration purposes), it has to look for it. Here the Naming Service may help by supporting you to access the Notification Server by a symbolic/meaningful name ([OMG96]).

Sharing & Concurrency. A Notification Server is usually shared among many clients. So, if you don't use the Event Notification Service ([OMG96]) you are responsible for coordinating concurrent access to the shared Notification Server to avoid race conditions. Here you can apply the OMG Concurrency Control Service to serialize access ([OMG96]).

Location. A Notification Server should be located on a reliable system because it must be available as much as the resource server. One solution could be to co-locate Notification Server with its associated resource.

Availability. If a Notification Server goes down for any reason, you should handle the situation. One way to do this is the application of a Strategy ([GHJV94]), where one strategy deals with the normal case - the available Notification Server - and another Strategy addresses that case where no Notification Server is alive, e.g. by polling the database as if there was no Notification Server.

Related Patterns

Observer[3] ([GHJV94]): In the context of the Observer pattern the *Notification Server* would play the role of the Subject and all registered clients are Observers. In opposition to the Observer pattern the subject observed by all clients is not the *Notification Server* but the shared resource - the database. So there is a separation between the "real" Subject observed and the notification mechanism used to propagate changes. Moreover in the Observer pattern the Subject notifies all Observers about the changes including the Observer that has initiated this notification. This additional notification is not desired in the context of a *Notification Server*. The *Notification Server* is different from the Observer pattern both in its structure and behavior.

---

[2] *Command(233):* Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

[3] *Observer(293):* Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically ([GHJV94]).

Publisher-Subscriber[4] ([BMRSS96]): Like in the Publisher-Subscriber scenario the *Notification Server* (the Publisher) publishes events about changes of a shared resource it relates to. All clients (the Subscribers) that are interested in those information registers themselves with the *Notification Server*. If applied to the *Notification Server* the Publisher would send unnecessary or even undesired notifications to the Subscribers. The example makes use of the Event Channel as a variation of the Publisher-Subscriber.

Asynchronous Completion Token [5]([PHS98]): T.B.D.

Example

The following example was implemented in a CORBA-based environment. It makes use of CORBAservices like Naming and Event Notification ([OMG96]). Here all clients that are working on a shared database notify each other about the changes they made within the database by using the same Notification Server (figure 5). There are synchronous and asynchronous communications between the clients and the Notification Server in both directions. Synchronous communication is used by the client to register with and unregister from the Notification Server and, in certain circumstances, by the Notification Server to instruct a client to unregister. Asynchronous communication is used to propagate change notifications. Notifications from the clients to the Notification Server are concentrated via an event channel, here an n-to-one connection, and will be forwarded to each client by using an event channel as a one-to-one connection. The Notification Server registers itself in the Naming Service, so it can be found and accessed easily by new clients for registration issues.

[4] *Publisher-Subscriber(339):* The Publisher-Subscriber design pattern helps to keep the state of cooperating components synchronized. To achieve this it enables one-way propagation of changes: one publisher notifies any number of subscribers about changes to its state ([BMRSS96]).

[5] *Asynchronous Completion Token:* To efficiently associate state with the completion of asynchronous operations ([PHS98]).

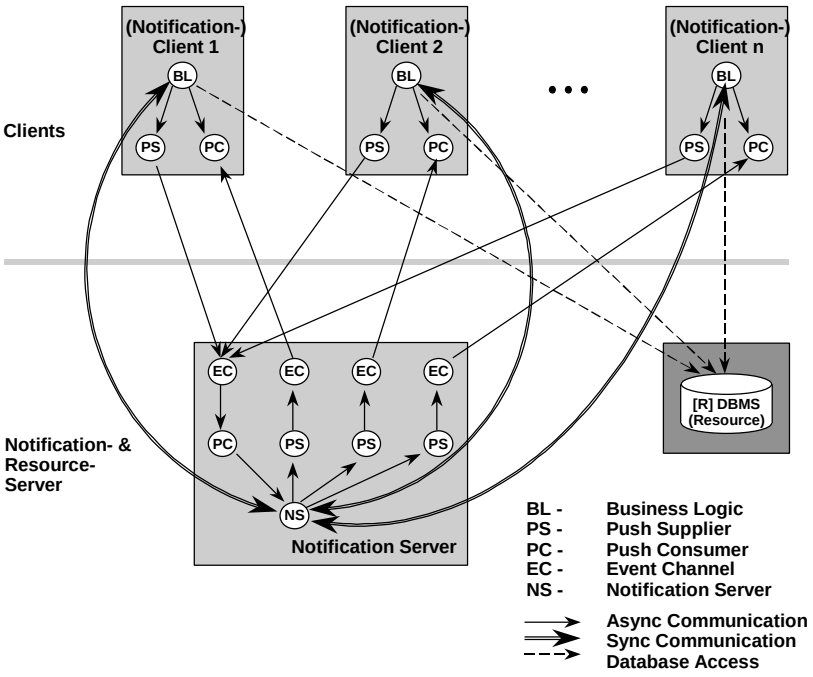

Figure 5

The following IDL interface descriptions are used for synchronous communication. A client uses the Notification Server via its NotificationServerIF and, in the other direction, the Notification Server can communicate with a client over the NotificationClientIF.

```
// NotificationServer
// IDL

module NotificationServer {

	interface ConstructionInfoIF {

		attribute CosEventChannelAdmin::DSTEventChannel channelToNS;
		attribute CosEventChannelAdmin::DSTEventChannel channelFromNS;

		#pragma class byValue ConstructionInfo
		struct byValue {   // cOOl :-)
			CosEventChannelAdmin::DSTEventChannel channelToNS;
			CosEventChannelAdmin::DSTEventChannel channelFromNS;
		};
	};

	interface NotificationServerIF {

		readonly attribute string name;

		ConstructionInfoIF::byValue registerClient (in NotificationClientIF aNotificationClient);
		void unregisterClient (in NotificationClientIF aNotificationClient);
	};

	interface NotificationClientIF {
```

```
            readonly attribute string name;

            void unregisterFrom (in NotificationServerIF aNotificationServer);
        };
};
```

Acknowledgments

Thanks are due to Steven Abell, Donald Griffin, and Brad Appleton for giving helpful hints for improvement.

References

[BMRSS96] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M.:
Pattern-Oriented Software Architecture: A System of Patterns.
Jon Wiley & Sons, 1996

[GHJV94] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.:
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1994

[OMG94] *Object Services Architecture.*
Object Management Group, December 1994

[OMG95] *The Common Object Request Broker: Architecture and Specification*
Object Management Group, July 1995

[OMG96] *CORBAservices: Common Object Services Specification.*
Object Management Group, March 1996

[PHS98] Pyarali, I.; Harrison, T.; Schmidt, D.C.:
Asynchronous Completion Token.
In: Pattern Languages of Program Design 3.
Addison-Wesley, 1998