Building a Scalable and Efficient Component Oriented System using CORBA – Active Badge System Case Study*

Jakub Szymaszek, Andrzej Uszok and Krzysztof Zieliński
Institute of Computer Science
University of Mining and Metallurgy (AGH)
Al. Mickiewicza 30, Cracow, Poland
tel. +48 12 6339406, fax. +48 12 6338907
e-mail: {jasz,uszok,kz}@ics.agh.edu.pl

1 Introduction

This paper presents experiences gathered when implementing the localization system for an office environment in CORBA. It describes methods which enable preserving fine-grain object-oriented structure of the system and achieving efficient performance at the same time. The presented study is a practical lesson about implementation of a scalable system oriented towards information dissemination. The key idea is to represent a collection of objects by a repository that provides access to them both as individual CORBA objects and data records. The proper usage of this duality may have substantial influence on the overall system performance.

2 Requirements of information dissemination system

Many existing information systems may be classified as information dissemination applications [8]. Those systems deliver information about changes of interesting subset of data to the group of interested users. New approaches to a construction of dissemination systems, namely object-orientation and distribution, introduce a new problem of system scalability. So far, there are few attempts to build such systems using those modern technologies from scratch and there is no general answer to the scalability problem. One of the most crucial design decisions is the choice of a degree of a abstraction level of objects composing the system. For example, when building a CORBA-based system disseminating share prices, one should decide whether individual share prices will be represented as CORBA objects or not. Generally, this is a question how to map the objects of the system model into CORBA objects. Until now, there is a common conviction that a CORBA-based system built of a huge number of CORBA objects is inherently inefficient. The solution presented in this

*This work is supported by Olivetti-Oracle Research Laboratory, Cambridge, UK

paper relaxes this limitation and proposes a template of a CORBA repository component with the incorporated light-weight mechanisms of notification, persistency and security.

3 Active Badge next generation project

The system, called *Active Badges*, was originally invented and developed at Olivetti Research Laboratory, in Cambridge, UK [3] in 1990-92. It uses hardware infrastructure which key components are infra-red sensors, installed in fixed positions within a building, and infra-red emitters (*active badges*) that are worn by people or attached to equipment. Sensors are connected by a wired network which provides a communication path to the controlling device, called poller, and distributes low-voltage power. A poller is implemented as a PC or a workstation with a sensor control software active on it. An active badge periodically transmits an infra-red message containing a globally unique code (a badge identifier) using the defined data link layer protocol [1]. Messages are received and queued by sensors. A poller periodically polls sensors, and retrieves badge messages from sensor queues. Each badge message as well as an identifier of the sensor which received the message is forwarded to the software part of Active Badges system. The software layer maintains a database that maps sensors to places in which sensors are installed and badges to users wearing these badges and to pieces of equipment which badges are attached to. Using these data the system can infer where users or pieces of equipment are currently located. The information about the current location of users and equipment is provided to various applications, such as presentation tools which display location data or applications which use location data to control users' environment. The software part of the original Active Badge system developed at ORL uses ANSAWare distributed environment.

3.1 Goals of ABng project

The ABng project (Active Badges – next generation) aims at development of a new software layer of the Active Badge system that fulfills the following assumptions:

- is flexible and reconfigurable;
- separates the details of gathering of location data from the application layer;
- provides location data filtering;
- ensures privacy of location data and security;
- enables to build systems making a user's environment location-aware.

To satisfy the first requirement, ABng uses the modern component and object-oriented technology. The system is developed in CORBA-compliant environments: Orbix [4] and OrbixWeb [7]. It is based on the object model in which all logical and physical elements of the Active Badge system (users, locations, sensors, badges, etc.) are represented as CORBA objects.

The system has a layered architecture which hides details of gathering location data. This makes it possible to replace a localization method based on infra-red sensors and emitters by another one. In ABng location data are presented using abstract notions of *location* and *locatable* objects rather then in terms of sensors and badges. A location is a part of an environment space obtained as a result of partition of the space according to an arbitrary, user-defined rule. Typically, an office space can be divided into buildings, floors, rooms, etc. A locatable is an object which can be observed by the system and whose location changes within the environment space can be monitored. A locatable can be a person or a piece of equipment, such as a computer, a printer or a book.

The basic ABng concept is *view* which is a collection of some location and locatable objects, i.e. it represents a part of the environment space and a subset of objects that can be localized within this part. The precision of localization of view's locatables is equal to the size of locations belonging to the view. Within a system a number of view objects can exist, each of which can hold information concerning current locations of users of equipment belonging to different groups and provided at different levels of abstraction with different precision.

The concepts of locations, locatables and views are crucial for data filtering and protection of privacy of location data. Every application can individually decide which view and which locations or locatables, contained in the view, it is willing to observe. It can subscribe to interesting objects and, as a consequence, to receive the required data related to these objects. With every view existing in the system a list of users, who can access this view, is associated. Thus only these users have access to location data as well to other attributes of locations and locatables contained in the view.

The ABng project assumes that development of the *Wonder Room* location-aware users' environment over the location system. This environment will consist of a number of applications which control various elements of the users' equipment. Examples of such applications are redirection of phone calls to the currently nearest phone, setting parameters of various home appliances, such as air-conditioning, TV sets, VCR-s, light, according to the preferences of users located in the neighborhood of these appliances. Such applications may be used for *personalization* of user's equipment. Systems of this type are examples of, so called, ubiquitous computing systems [10].

3.2 Design considerations

After the analysis of the desired functionality of the ABng system many kinds of entities have been singled out which have to be represented in the software. These entities could be divided into two main categories:

- Closely related to Active Badge System configuration, such as *Sensor* and *Badge* on the lowest level, and *ABng_Location_Description* and *Badge_Holder* above it,

- Describing office environment in which Active Badge System was installed such as *User*, *Equipment*, *Location_Type*, *Location* and *View*, etc.

These entities do not only encapsulate their states but also posses more or less complex functionality. For instance, a request to play some sound could be sent to the badge or particular instance of equipment, such as air conditioning in the given room, could be requested to change its state. This last functionality is possible thanks to the integration with the infra-red controlling system. Generally, it was assumed that functionality linked with the given types of entities could evolve and be in the future significantly extended.

Besides these many relationships between entities were grasped. A state of some entities depends or even is composed of states of others. Thus to present the whole state of such an entity information from many other entities has to be gathered. It is for instance justified to separate description of particular part of location, such as room or floor, from entity encapsulating a set of sensors installed there. The sufficient reason for this is that description of a room or floor is universal while a set of sensors is ABng specific. Combining these two entities into one will make evolution or replacement of the ABng with other location system impossible.

Additionally, an entity should be immediately informed about the changes in states of entities depends upon so as it can modify functionality of this entity and the system. For instance, when sensor is replaced or added to some room related *ABng_Location_Descriptions* and *Views* have to be informed, what in result will change processing of the sighting. Source of changes in states of entities could be:

- a system administrator, when updating repositories with data describing ABng configuration and office environment – this changes are relatively rare and there is long time gap between them,
- a movement of a locatable object – this changes are usually very often and they quickly follow each other,
- changes in the state of equipment – this changes could be often and quickly follow each other,

Such a change should be propagated not only within the system but should be further disseminated to interested observers. Thus the appropriate mechanism for managing list of observer is necessary.

Because of all these reasons each of the singled out entities appears to be complex enough to justify its representation as a separate CORBA object. The result of such a decision is that there is a large number of independent CORBA object in a real ABng system with even moderate number of users.

4 Component template

The conclusions from the investigation of the previous system, which were applied during the system development are:

- General entity as well as entities repository templates could and have to be designed,
- These templates should provide support for implementation of a mechanism eliminating overhead related to representation of each entity as a separate CORBA objects,
- Light Observer mechanism for entities repository has to be invented.

Thus an uniform template of all entities interfaces was defined as follow:

```
interface Entity: Entity_Observed, Entity_Commander {

    struct Description {
        Type1 attribute1;
            .
        TypeN attributeN;
    };

    typedef sequence<Description> Descriptions;

    struct Value_Description {
        Type1::Value_Description attribute1;
            .
        TypeN::Value_Description attributeN;
    };

    typedef sequence<Value_Description> Value_Descriptions;

    struct Pattern {
        boolean is_any;
        Type1::Pattern attribute1_pattern;
            .
        TypeN::Pattern attributeN_pattern;
    };

    readonly attribute Repository_Item_Id item_id;
    readonly attribute Description descr;
    readonly attribute Type1 attribute1;
            .
    readonly attribute TypeN attributeN;
}
```

Each type of entity possesses its specific set of attributes: *attribute1*, ... ,*attributeN*. It also inherits from interface *Entity_Comander* which defines its specific functionality.

Each entity possesses a unique identifier *item_id* inside the repository, which can be used by outside clients instead of an object reference.

Besides, every entity inherits from the *Entity_Observed* interface which enables other objects to register in the entity its interest in changes of the entity state. When the state is changed registered parties are informed.

Second uniform feature is a *descr* attribute of *Description* type, which is a structure possessing fields matching every attribute of the given entity. This structure is used to get the whole meaningful state of the entity in just one request. This functionality was mainly designed in order to be used by the smart proxy mechanism. A smart proxy on the client side can retrieve the whole state of the entity when it is created and then serve local request using cached data. It will also retrieve the whole state when cache is invalidated by the notification mechanism, which is described later

The next uniform element is a definition of a *Value_Description* structure. It is similar to the *Description* structure as it has a field for every entity attribute but the type of this field is either the type of the attribute, if it is not a reference, or the *Value_Description* structure from the entity referenced by this attribute. The purpose of this structure is to enable returning of the entity state as already collected all describing it data without any references to the outside objects.

Finally, there is a *Pattern* structure, which is built, in a recursive way, of *Pattern* structure for simpler data types. The *is_any* field denotes if the *value field* is meaningful or not. The *Pattern* structure is used to specify searching criteria for the given type of entity.

Objects built according to the template are stored in repositories which also possess a generic interface:

```
interface Entity_Rep : Entity_Rep_Observed {

    typedef sequence<Entity> Entities;

    readonly attribute Entities entity_list;

    Entity add(in Entity::Description data_record)
                        raises(Duplicate_Data_Record);

    void remove(in Entity object_to_remove)
                        raises(Unknown_Object_Ref);

    Entities find(in Entity::Pattern data_record_pattern);

    Entity::Value_Descriptions find_values(in Entity::Pattern data_record_pattern);

    void update(in Entity object_to_update, in Entity::Description data_record_pattern)
                        raises(Unknown_Object_Ref, Duplicate_Data_Record);
}
```

This interface enables operation on entity both using their object references as well as their state and local id:

- *Creation and addition* of a new entity to the repository with state described using *Description* structure. This method returns the object reference of the new entity.
- *Removing* of the existing entity by specifying its reference.
- *Searching* of entities matching specified criteria denoted by an adequate *Pattern* structure. The return value is either a sequence of object references (*find*) or a sequence of *Value_Description* structures (*find_values*).

The entity repository interface inherits from *Entity_Rep_Observed* interface, enabling registration of interest in the arbitrary collection of entity. This facilitates registration of observer in the large number of entities. A state of every entity contained in the repository is made persistent by the persistency mechanism used in this repository. The access to the entity is guarded by the security service.

5 Implemetation of component facilities

Every ABng component offers mechanisms for asynchronous notification about changes of components attributes, for life cycle control, and security. They are examined below.

5.1 Light notification mechanism

A typical ABng application can use a lot of various information encapsulated by ABng objects. For instance, let us consider a viewer application which presents location data. It displays information about locatable users (such as a user's name, a position, etc.) and information about rooms in which users are currently located (e.g. a room number or a room name, a floor number, the nearest telephone number, etc.). To obtain this information the viewer interacts with various ABng objects. To optimize these interactions, ABng implements a caching algorithm based on the smart proxy mechanism available in Orbix and OrbixWeb. When the application contacts an entity, the smart proxy of the entity object is instantiated within the application's address space. When the application enquires about an entity attribute for the first time, the smart proxy calls the entity object and retrieves values of all its attributes, i.e. its complete description. This description is stored in a cache maintained by the proxy and the required single attribute value is returned back to the application. When the application enquires about an attribute value again, this value is retrieved from the cache and no remote calls are performed.

If a value of an entity attribute changes, all proxies active in different applications have to be notified about this change. For this purpose, in ABng a notification mechanism based on the Observer [2] architectural design pattern (also known as Publisher/Subscriber) has been designed. Every proxy registers itself as an observer of the entity it represents. Each time, an attribute of the entity is updated, the proxy is notified and after that it marks its cache as invalid. The next application's query about an attribute value causes the proxy to contact the entity and retrieve the whole description. Registering smart proxies directly within an entity object would be very inefficient as for every smart proxy a corresponding proxy object in the server containing the entity would be created. To solve this problem the mechanism of notification dispatching is employed. This is depicted in Figure 1 and explained below.

When a proxy representing the first entity from a given repository is instantiated within an application, an object, called *notification dispatcher* associated with the repository, is created. The dispatcher will represent all proxies associated with entities contained in the given repository and dispatch notification messages to the proxies. The smart proxy does not directly subscribe itself to the corresponding entity. Instead, it calls the notification dispatcher (arrow 1 in Figure 1). The dispatcher casts the smart proxy reference to the reference of an ordinary proxy and calls the real stub of the registration method (*attach_observer*) passing its own reference as an argument (the dispatcher has to provide the observer interface). This call results in a real remote invocation on the entity (arrow 2). If the dispatcher contacts the entity server for the first time, the dispatcher proxy is instantiated within the server at the same time. Within the repository the number of existing dispatcher proxies is equal to the number of applications which contain proxies observing repository's entities. The entity stores a dispatcher reference (a pointer to a dispatcher proxy) in a registry of its observers.

When the state of the entity is changed the entity notifies all notification dispatchers via their proxies (arrows 3 and 4). On the application's side, the dispatcher obtains a reference of the updated entity, which, in fact, points to the local smart proxy. The dispatcher forwards notification to the smart proxy (arrow 5). Finally, the proxy marks its cache as invalid.

Beside smart proxies maintaining caches, inside a user's application there can be also ordinary objects which are interested in asynchronous notification about entity updates. This notification is also performed using the described mechanism. An application object,

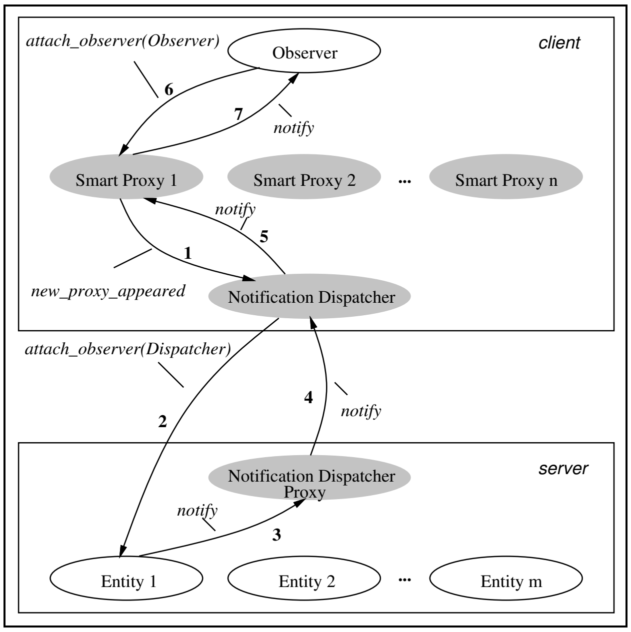

Figure 1: ABng notification mechanism

which wants to be notified about changes of an entity's state, calls the *attach_observer* operation of the interesting entity (arrow 6). This call, however, is not transmitted to the entity. It only affects a local registry of entity observers, which is maintained by the smart proxy. After the proxy is notified about entity update, it forwards this notification message to all entity observers contained within the application (arrow 7).

It should be noted that the above mechanism is completely transparent to the application. The application which is assumed to use this mechanisms has to be linked with the library containing smart proxies and dispatchers.

5.2 Persistency

States of entities have to survie rebooting of the system. Their persistency can be implemented using different approaches. However, the heavy and cumbersome mechanism could have a tremendous impact on efficiency and scalability of the system. In the ABng two versions of persistency mechanism are implemented:

- *File-based* – this primitive mechanism uses a separate file to store the serialized state of each repository. It is implemented as coarse-grained, which means that when one of the entities in the repository is changed then the whole state of the repository (all entities) is stored in the appropriate file.

- *Object Database Object Adapter (ODOA) [6]* – this sophisticated mechanism of achieving CORBA objects persistency uses an object database (ObjectStore [9]) to save separate objects as well as collections of objects. Each repository is a root for a collections of entities. However, when particular entity is changed only its state is updated in the database. The disadvantage of this mechanism is that a transaction has to be created. The ODOA provided by Iona is only single-threaded and always open a heavy *update* transaction. The OOA used in the ABng was obtained as a specialization of the Object Database Adapter Framework [5]. Its special features enable multithreaded implementation of servers as well as instrumentation of ODOA, during the compilation of the program, with names of methods (together with interface names) requiring creation of *update* database transactions in the other case the light *read-only* transaction is created

The version of the persistency mechanism used in the given repository is determined during the compilation (possibility of postponing this to the execution time is now investigated). There is no restriction that all repositories in the running system have to use the same persistency mechanism, each can adopt an adequate version of it.

5.3 Security

The access to entities is granted basing on the *view* level and thus it has to have effect on many other objects in other repositories associated with the given *view*. The authorization server connected with the *View Manager* automatically grants a user access to all data about locations and locatables of the *view*. This authorization data is replicated in caches within repositories. Therefore, when a user is denied access to the *view* or the contents of the *view* is changed (a location or a locatable is removed from the view) the authorization server informs repositories caches about this changes. The same notification mechanism as described in the previous section is employed here.

6 ABng system evaluation

The ABng system is now in advanced implementation according to presented in this article assumptions. Part of it is already working and full version should be ready in February. Then the following evaluations of the systems are forseen as a comparisation of systems using light notification mechanism versus system without it:

- verification of system internal scalability – processing time of sigthing when a number of views and other entities: sensors, badges etc. are being increased.
- verification of system external scalability – notification time of changes when number of clients is being increased.

An impact of different persistency mechanisms on system performance will be as well evaluated.

References

[1] Frazer Bennett and Andy Harter. Low bandwidth infra-red networks and protocols for mobile communicating devices. Technical Report 93.5, Olivetti Research Laboratory, Cambridge, UK, 1993.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[3] Andy Harter and Andy Hopper. A distributed location system for the active office. *IEEE Network. Special Issue on Distributed Systems for Telecommunications*, 8(1), January 1994.

[4] Iona Technologies Ltd. *Orbix 2 Programming Guide*, November 1995.

[5] Iona Technologies Ltd. *Orbix Database Adapter Framework – White paper*, 1997.

[6] Iona Technologies Ltd. *Orbix+ObjectStore Adapter Programming Guide*, April 1997.

[7] Iona Technologies Ltd. *OrbixWeb 3.0 Programming Guide*, 1997.

[8] Michael Franklin and Stanley Zdonik. A Framework for Scalable Dissemination-Based Systems. In *Proceedings of OOPSLA'97*, pages 94–105, October 1997.

[9] Object Design, Inc. *ObjectStore C++ API User Guide, Release 4.0.1*, May 1996.

[10] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 6(7):75–84, July 1993.