Building a Scalable and Efficient Component-Oriented System using CORBA – an 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, 30-059 Kraków, Poland
{jasz,uszok,kz}@ics.agh.edu.pl, http://iris.ics.agh.edu.pl

Abstract

This paper presents experience gathered through the implementation of a CORBA-based localization system for an office environment. The localization system simultaneously preserves the fine-grained object-oriented structure of the system and achieves efficient performance. The presented study is a practical lesson concerning the implementation of a scalable, information-dissemination system. The key idea is to represent a large observable collection of objects by a repository that provides access to them both as individual CORBA objects and as data records. The proper use of this duality may have a substantial impact on the overall system performance. The repository is equipped with a scalable notification mechanism built around the concepts of a notification dispatcher and a notification tree. Fundamental features of the proposed solution are illustrated by a performance study and a representative application.

1 Introduction

Many existing information systems may be classified as information-dissemination applications [4]. Such systems deliver information about changes to a subset of data to a group of interested users. New approaches to the construction of dissemination systems, namely object-orientation and distribution, introduce the problem of system scalability. There have been few attempts to build such systems using these technologies from scratch, and there is, in general, no answer to the scalability problem. One of the most crucial design decisions for such a system is the choice of abstraction level for objects composing the system. For example, when building a CORBA-based system disseminating share prices, one must decide whether individual share prices will be represented as CORBA objects or not. Generally, this is a question of how to map objects of the system model into CORBA objects. Until now, there is a common belief that a CORBA-based system constructed from a huge number of CORBA objects is inherently ineffi-

cient. The solution presented in this paper refutes this belief and proposes a template CORBA repository component with light-weight mechanisms for notification, persistence and security.

This repository component combines and refines some ideas that have recently appeared in component oriented software environments such as Java Beans [19, 20], San Francisco Components [3], and CORBA Components (proposed by Iona Ltd and others) [16]. The major innovations of this repository are a dual form of access to repository entities that is by value or by CORBA references, and a scalable notification mechanism with built in smart proxies. Finally, the idea of dynamic attributes [13] has been exploited and three different types of repository component have been proposed.

The structure of the paper is as follows. In Section 2, the *Active Badge next generation* (ABng) project, the context of this study, is described. The repository component template is defined in Section 3. Implementation issues concerning this component are described in Section 4. Next, in Section 5, a performance evaluation study of the system's scalability is reported and discussed. Section 6 presents a basic application that uses information gathered by an ABng system. The paper ends with conclusions.

2 The Active Badge next generation project

The system, called *Active Badge*, was originally invented and developed at Olivetti Research Laboratory, Cambridge, UK [6] in 1990-92. It uses a hardware infrastructure whose 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 a poller, and distributes low-voltage power. A poller is implemented as a PC or a workstation with active sensor control software. An active badge periodically transmits an infra-red message containing a globally unique code (a badge identifier) using the defined data link layer protocol [2]. Messages are received and queued by sensors. A poller periodically polls sensors, and retrieves badge messages from sensor queues. Each badge message, including an identifier of the sensor which received the message, is forwarded to the software part of the Active Badge 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 to which badges are attached. Using this 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 the users' environments. The software for the original Active Badge system developed at ORL used the ANSAWare [1] distributed environment.

2.1 Goals of the ABng project

The ABng project focussed on the redevelopment of the Active Badge system software such that the following requirements are satisfied:

- the system is flexible and reconfigurable;

- the system separates the details of location data gathering from the application layer;
- the system provides location data filtering;
- the system ensures privacy of location data and security;
- the system enables the construction of location-aware applications.

To satisfy the first requirement, ABng uses modern component and object-oriented technologies. The system was developed using CORBA-compliant environments: Orbix [7] or, alternatively, OmniORB [12] and OrbixWeb [11]. It is based on an 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 represented using abstract notions of *location* and *locatable* objects rather than in terms of sensors and badges. A location is a part of an environment obtained by partitioning 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 monitored environment. 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 a *view*, which is a collection of 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 localization precision of a view's locatables is determined by 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 the current locations of users or 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, receive data related to these objects. Associated with every view in the system is a list of users who can access that view. As a result, only users found in the associated list have access to location data as well as to other attributes of locations and locatables contained in a particular view.

The ABng supports the *Wonder Room* location-aware user environment over the location system. This environment consists of a number of applications which control various elements of the users' equipment. Examples of such applications are redirection of phone calls to the phone nearest the called individual or setting parameters of various home appliances (such as an air-conditioner, TV sets, VCRs, lights) according to the preference of users located in the neighborhood of these appliances. Such applications may be used for *personalization* of each user's equipment.

2.2 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 software. These entities can be divided into

two main categories:

- Those closely related to the Active Badge system configuration, such as *Sensor* and *Badge* on the lowest level, and *ABng_Location_Description* and *Badge_Holder* above it,
- Those describing the office environment in which the Active Badge system was installed such as *User*, *Equipment*, *Location_Type*, *Location* and *View*, etc.

These entities not only encapsulate their state but also possess more or less complex functionality. Some examples are: a request to play a particular sound could be sent to a badge; a particular instance of equipment, such as an air conditioner in a given room, could be requested to change its state. This last functionality is possible thanks to the integration of the ABng system with an infra-red controlling system. Generally, it is assumed that functionality linked with a given type of entity will evolve and be significantly extended in the future.

There are numerous relationships between ABng entities. The state of a particular entity may depend upon or be composed of the states of other entities. Thus, representation of the entire state of such an entity requires that information from many other entities be gathered. The ABng layered architecture requires the separation of location descriptions, such as a room or floor, from the entity encapsulating a set of sensors installed there. Combining these two entities into one will make evolution or replacement of the current sensors with other localization capabilities impossible.

Additionally, an entity needs to be immediately informed about state changes of entities it depends upon, permitting it to modify its functionality appropriately. For instance, when a sensor is replaced or added to a particular room, related *ABng_Location_Descriptions* and *Views* entities have to be informed, which will cause a change in the processing of sightings. Entity state changes can be caused by several sources:

1. a system administrator, when updating repositories with data describing the ABng configuration and office environment – such changes are relatively rare and not bursty,
2. the movement of a locatable object – such changes usually occur very frequently,
3. changes in the state of equipment – such changes may occur frequently.

Changes must be propagated not only within the system but also to interested observers. Thus, an appropriate mechanism for managing sets of observers and their interests is necessary.

Due to all these reasons, each of the above-mentioned entities appears to be complex enough to justify its representation as a separate CORBA object. This results in a large number of independent CORBA objects in a deployed ABng system supporting even a moderate number of users.

3 Component template

After reviewing the previous Active Badge system, we factored the following into the ABng system:

- General templates for each entity, as well as for a repository, were designed,
- These templates support the implementation of each entity eliminating the overhead related to representing each entity as a separate CORBA object,
- A light-weight notification mechanism for repository clients was designed.

In the ABng, three types of entities, and as a result, three types of entity interfaces have been derived, with:

- **static attributes**: Most ABng entities have a fixed and relatively small number of attributes. Such entities are accessed via interfaces, in which each entity attribute is represented by a corresponding IDL attribute.
- **dynamic attributes**: Another approach is to treat an entity as a collection of attributes of arbitrary types and to provide an access to them via an appropriate interface. Such an interface offers a pair of accessor operations to set and retrieve the value of a single attribute in which an attribute is referred to by its name and value is encoded using the IDL *any* type. The interface permits the retrieval of all attributes as a list. This approach is an application of the *Dynamic Attribute* design pattern [13]. This type of an interface is provided by entities which have many attributes or these attributes are different for individual entity instances. An ABng example of such an entity is the *Location_Description* object, which describes a piece of an office space, such as a floor, a room or a building. These real-world objects are inherently different, and it is impossible to design a uniform set of attributes for them.
- **hybrid attributes**: This type of an interface explicitly defines those attributes which are common for all objects representing entities. Additionally, it uses the *Dynamic Attribute* paradigm to provide access to object-specific attributes. An example of an ABng object providing such an interface is *Equipment*, an instance of which describes a piece of office equipment. All kinds of equipment have a set of common attributes, such as a name, a vendor name, etc.; these have been defined as explicit attributes.

For every type of interface a corresponding template was designed. Below, the template for interfaces with static attributes is described in detail as an example.

The template of an entity interface, shown in Figure 1, defines a set of attributes: *attribute1*, ... ,*attributeN*. It also inherits from the *Entity_Comander* interface which defines its specific functionality.

Each entity possesses a unique identifier (*item_id*) inside its repository, which can be used by repository clients to refer to objects. This is an alternative to using object references for this purpose.

Besides, every entity inherits from the *Entity_Observed* interface which enables other objects to register their interest in changes to an entity's state. When the state of an entity changes, any registered parties are informed about it.

The second uniform feature is the *descr* attribute of the *Description* type, which is a structure possessing fields corresponding to the attributes of a given entity. The structure is used to obtain

```
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;
}
```

Figure 1: Entity interface template

the entire state of the entity in just one request. This feature was mainly designed for use by the notification mechanism based on smart proxies, described in the next section. A smart proxy on the client side can retrieve the whole state of the entity when it is created and then serve a local request using cached data. It will also retrieve the whole state if the cache is invalidated by the notification mechanism, which is described later on.

The next common element is the definition of the *Value_Description* structure. Like *Description* it has a field for every entity attribute but the type of this field is either the type of a corresponding attribute, providing it is not an object reference, or the *Value_Description* structure from the entity referenced by this attribute. The purpose of this approach is to enable the

return of the entity state as a set of already collected data without any references to outside objects.

Finally, there is the *Pattern* structure which is built in a recursive way. It contains *Pattern* structures 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 entity type.

The templates for interfaces with dynamic or hybrid attributes (not presented here) are very similar to that for static attributes. In a dynamic interface the only attribute is a sequence of name/value pairs and there are two additional methods to set and retrieve a single value. An interface with hybrid attributes has both explicit attributes and a list of name/value pairs accompanied by the accessor operations.

Objects built according to any of the three templates are stored in repositories which also possess a generic interface, presented in Figure 2.

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

   Entities find_and_attach(
    in Entity::Pattern data_record_pattern,
    in Observer obs,
    out Entity::Descriptions descritions);
}
```

Figure 2: Repository interface template

The operations of a repository are as follows:

- *add* – creates and adds a new entity to the repository. The initial state of the created object is determined by the contents of the *Description* structure passed as an argument. This method returns the object reference of the new entity.
- *update* – replace the state of the entity denoted by a reference with values stored in the *Description* structure.
- *remove* – remove an entity denoted by a reference.
- *find* – returns a collection of references of those entities which match the criteria specified in the *Pattern* structure passed to the operation.
- *find_values* – returns a collection of the states of the entities matching the given criteria. In other words, this operation returns the matching object by value rather than by reference.
- *find_and_attach* – works like the *find* method but additionally registers the observer (*obs*) for all returned entities in a repository. It also possesses an *out* parameter by which the sequence of entity descriptions is returned.

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

Although the *update* and *remove* operations possess an argument of an object reference type, object references are not directly used to identify entities that are to be updated or removed, respectively. In general, one cannot rely upon comparison of two object references to yield equality. A comparison of two different object references may yield inequality, even though they may point at the same object. To avoid this problem, an entity's reference is used to obtain the entity's unique identifier, which is returned by invoking the item_id() method and is used for indexing.

4 Implementation of component facilities

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

4.1 Light-weight notification mechanism

A typical ABng application uses a significant amount of information encapsulated by a number of ABng objects. To obtain this information an application interacts with the appropriate ABng objects. To optimize the performance of these interactions, ABng implements a caching algorithm – i.e. a Smart Proxy Layer (SPL), based on the smart proxy mechanism available in Orbix and OrbixWeb. When the application obtains an entity reference (for instance by executing the repository's *find* method), the smart proxy of the entity object is instantiated within the

application's address space and the entity's attribute values are cached. When the application enquires about an attribute value, this value is retrieved from the cache and no remote call is 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, a notification mechanism based on the Observer [5] 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 marks its cache as invalid. The next query by the application for 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, since for every smart proxy registration a corresponding proxy object in the server containing the entity would be created. To solve this problem, a notification dispatching mechanism is employed. This is depicted in Figure 3 and explained below.

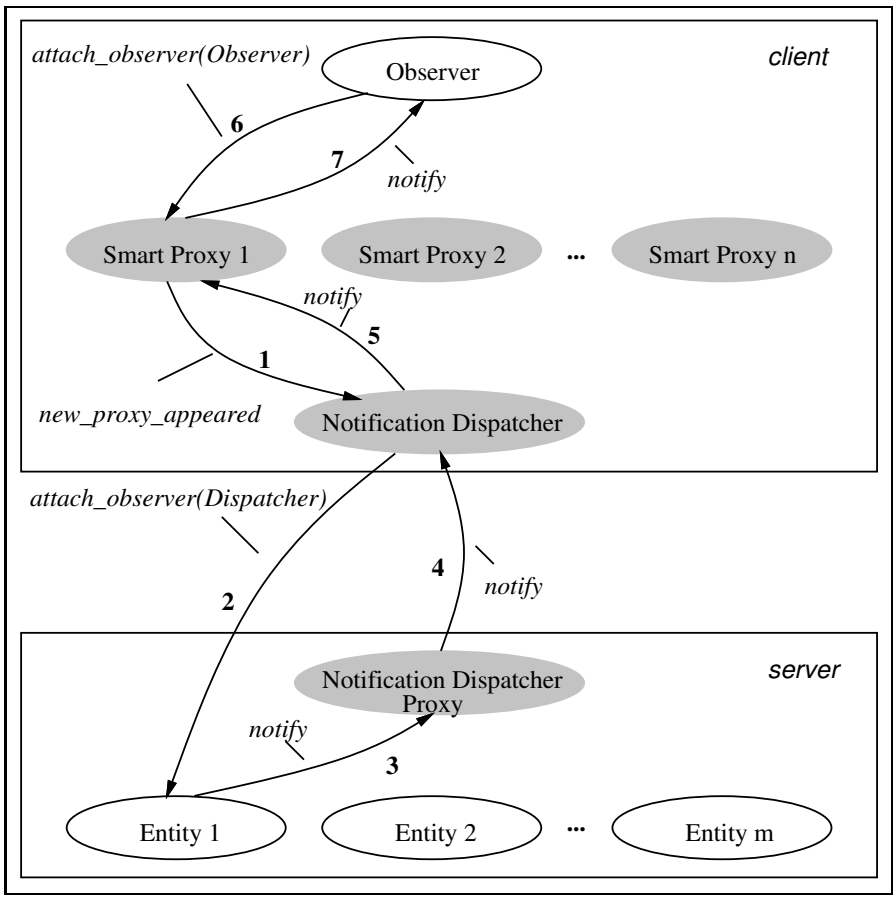

Figure 3: ABng notification mechanism

When a proxy representing the first entity from a given repository is instantiated within an application, an object, called the *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 3). 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 the 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 an entity changes, the entity notifies all notification dispatchers via their proxies (arrow 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.

In addition to smart proxies maintaining caches, a user's application can also contain ordinary objects which are interested in asynchronous notification regarding entity updates. This notification is also performed using the described mechanism. An application object that wants to be notified about changes to an entity's state calls the *attach_observer* operation of the entity of interest (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. An application which uses this mechanism must be linked with a library containing smart proxies and dispatchers.

In the general case, it may happen that caches become stale and applications end up using stale data. There might be several reasons of such a situation:

- **Delay** – updates may be delayed if the processing power or communication bandwidth of the system are exceeded. Generally, this can be solved by improving system efficiency of providing more of the oversubscribed resources. This is not typically a problem in ABng, however, since changes (e.g. movement of people) occur relatively rarely.

- **Losses** – oneway CORBA invocations, which are used in the implementation of the notification mechanism, do not generally guarantee request delivery. However, most existing ORBs, including Orbix and OmniORB used in the ABng system, employ the IIOP protocol implemented over TCP/IP and ensure reliable request delivery of oneway calls.

- **Inconsistency** – a cache coherency problem may occur as cache refreshing is not simultaneous and atomic. This is a well known problem [21], which may happen when an application uses caches in different processes. However, an implementation of a cache coherency protocol in a distributed CORBA environment would introduce substantial overhead. The current implementation of the notification mechanism does not address this problem, since ABng belongs to the class of applications which require only weak-consistency.

4.2 Persistence

Entity state must survive rebooting of the system. This persistence can be implemented using different approaches. However, a heavy and cumbersome mechanism could have a tremendous impact on efficiency and scalability of the system. In the ABng, two versions of persistence 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 restored in the appropriate file.
- *Object Database Object Adapter (ODOA) [10]* – this sophisticated mechanism of achieving CORBA object persistence uses an object database (ObjectStore [14]) to save separate objects as well as collections of objects. Each repository is a root for a collection of entities. However, when a 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 opens a heavy-weight *update* transaction. The ODOA used in the ABng was obtained from Iona as a specialization of the Orbix Database Adapter Framework [9]. Its special features enable multi-threaded implementation of servers as well as instrumentation of the ODOA during the compilation of the program, with names of methods (together with interface names) requiring the creation of *update* database transactions. For other methods, light-weight *read-only* transactions are created.

The persistence mechanism used in a given repository is determined during compilation (the possibility of postponing this to execution time is being investigated). There is no restriction that all repositories in the running system have to use the same persistence mechanism.

4.3 Security

Access to entities is granted based on the chosen view; thus it effects many other objects in other repositories associated with the given *View* object. The authorization server connected with a *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 a view or the contents of a view is changed (a location or a locatable is removed from the view), the authorization server informs repository caches about the change. The same notification mechanism as described in the previous section is employed here.

4.4 The scalability of the notification mechanism

The light-weight notification mechanism described in Section 4.1 seems to be promising for dissemination of information in a large community of clients distributed in the network. In this section the issue of its scalability is further analyzed.

The proposed solution for notification has the following structural features:

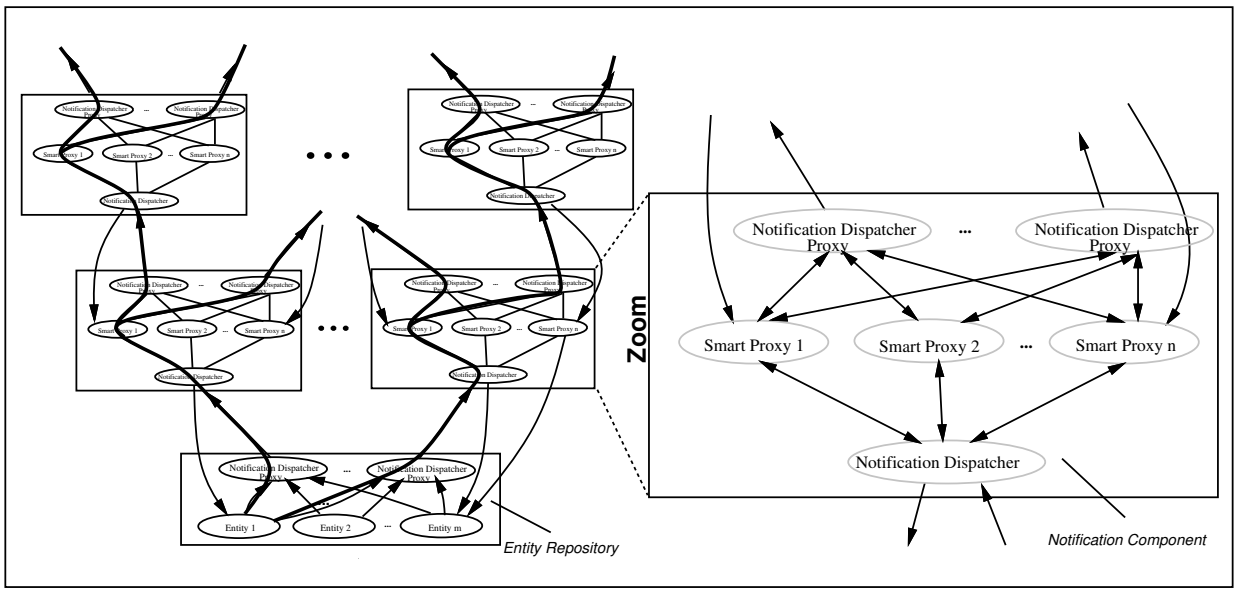

Figure 4: Notification Tree

- The notification dispatcher is a CORBA object which represents a collection of smart proxies.
- The smart proxies are not CORBA objects and may be notified using local method invocations.
- The collection of repository entity smart proxies in the client space is represented in the repository by a single notification dispatcher proxy. It saves a significant amount of memory and makes the space occupied by the repository server independent of the number of entities in the collection and the global number of existing proxies.
- A client may not be aware that a notification dispatcher is used. Its activity is completely transparent to the client even from the programming point of view.
- The number of notification dispatcher proxies in a repository server is dependent only on the number of clients in the system which contain entity smart proxies for entities in that server.
- The proposed notification mechanism is selective, which means that only notification dispatchers that have registered smart proxies corresponding to the changed entity in the repository are notified.

It is necessary to point out that the existence of a notification dispatcher does not influence the scalability of the system in terms of the number of observer clients.

This last issue can be addressed using replication. A client component with a collection of smart proxies and a notification dispatcher could be generalized as a notification component

shown in Figure 4. Scalability with respect to the number of clients can then be achieved by organizing the system into a notification tree built from notification components. Each smart proxy has registered several notification dispatchers from the next layer up in the tree. In the root node the repository of entities exists. In other nodes only smart proxies are present. The proposed architecture represents a distributed collection of entities which could be highly available for a large number of clients despite geographic distribution.

The propagation of a repository entity update is marked in Figure 4. It is easy to see that the proposed solution has similar scalability to notification based on multicast over IP communication protocols that propagate messages down a multicast tree. The advantage is that the notification tree does not require any multicast protocol support.

In the context of this discussion it is necessary to compare the proposed solution to the CORBA Event Service [15] and the CORBA Notification Service [18]. The most important difference is that these services do not use a smart proxy concept, so caching has to be solved in a separate way. The notification component is similar to an event channel in the sense that it separates the repository as a source of events from the client. Filtering of events is provided by a notification component through the attachment of an observer to any subset of fine-grain entities. In the Event Service, to achieve selective notification, it is necessary to define as many event channels as there are event sources; in the case of ABng this will create an overwhelming numbers of channels. However, the new Notification Service addresses this problem through the powerful concept of the filter object. Further comparison is very much dependent on implementation details which are not defined by the OMG specifications. For instance, in Iona's implementation of the Event Service based on the use of a multicast protocol [8], an event producer does not even know the number of notified consumers. This approach scales well but is based on a proprietary protocol and is very difficult to extend from LAN to WAN.

5 ABng system evaluation

The ABng software, implemented according to the design concepts presented in this paper, was subject to intensive testing to determine its performance and scalability. The system was implemented using Orbix 2.2MT; the clients used in the tests were implemented using OrbixWeb 2.01.

Performance results presented in this section were obtained in an environment consisting of 15 Sun Ultra workstations and a Sun Enterprise 3000 server, connected by two 10Mb Ethernet switches. The ABng system components were running on the server. All other programs were executed on separate workstations, so all CORBA invocations went through the network. One particular system component was chosen for the test. However, the results are representative of all of them as they were implemented using the same C++ template. All tests were repeated 100 times and average values were calculated. The workstations were used for normal activities during the test, but were rather lightly loaded.

Performance tests were carried out according to typical scenarios occurring in a majority of ABng applications, i.e. the retrieval of the current state of a set of entities in an application bootstrap and the scalability of the notification mechanism when the number of applications observing changes in these entities increases. The implementation improvements proposed in the

paper – SPL and notification – were also evaluated.

5.1 Costs of accessing repository entities

There are two different ways of accessing entities in a repository: by their values, using *find_value* and by their references, using *find*. In order to access the fields of entities obtained using *find*, subsequent *get_descr* methods have to be called. The time spent in these calls executed in the OrbixWeb applet when the number of entities in the repository increases is presented in Figure 5 and Figure 6. All these calls were invoked with a *pattern* that matched all entities in the repository. The results show that accessing entities by reference is several orders of magnitude more costly than accessing them by value. Thus access by reference, in spite of its many superior features, has to be used carefully and often combined with access by value, as in the application presented in Section 6.

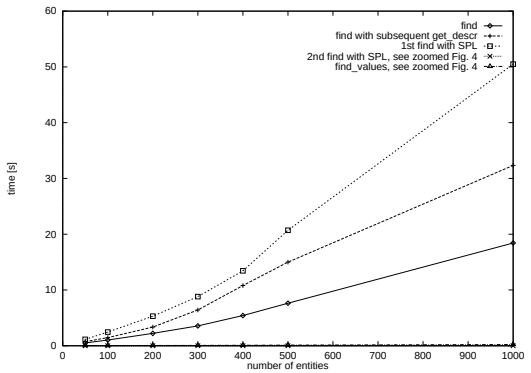

Figure 5: Performance comparison of different ways of accessing the states of repository entities

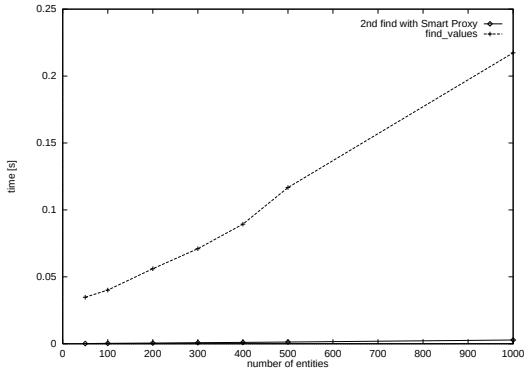

Figure 6: Performance comparison of different ways of accessing the states of repository entities, zoomed

These figures also show the performance of subsequent invocations of the *find* method when the SPL is used. The execution time of the second *find* is almost negligible, as it happens locally. Additionally, the first call with the SPL is about 40% more costly than the combination of *find* without SPL and *get_descr* calls. This difference is caused by the SPL construction time.

5.2 Analysis of the SPL construction cost

The process of constructing the SPL was divided into four basic steps:

1. acquiring entity references by executing the find method,
2. creating a smart proxy for each of the acquired references,
3. registering the notification dispatcher for each of the references,
4. retrieving entity descriptions by executing the *get_descr* method for each smart proxy.

All of these steps, except the second one, include remote CORBA invocations. The first step includes one remote invocation, the third and fourth steps include as many remote invocations as there are acquired references. In Figure 7, the total SPL construction time as well as times spent in individual steps are presented.

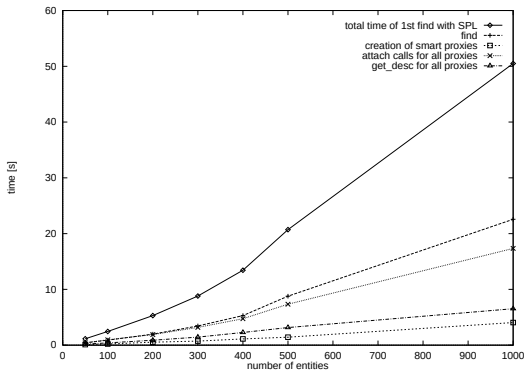

Figure 7: Time consumption by basic steps of SPL construction

These results show that the majority of the time is used for remote invocation. Most of these invocations could be eliminated by using the *find_and_attach* repository method, which eliminates the necessity of remote calls from the third and fourth steps. This reduces the SPL construction time by 45% and requires construction of a smart proxy for the repository. In this smart proxy a *find* call is replaced by a *find_and_attach* call, with the observer (parameter *obs*) initialized to the notification dispatcher. The returned sequence of entity values (out parameter *descriptions*) is used for loading the smart proxy caches.

5.3 Notification time

The method used to notify observers registered in the repository about entity's changes are asynchronous oneway operations invoked successively on the observers. The impact of a growing number of entity observers on the notification time is presented in Figure 8.

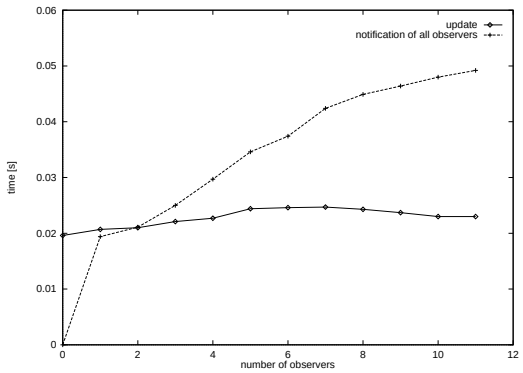

Figure 8: Impact of growing number of observers on entity update and notification times

The results show that the time required to execute an *update* on an entity is almost independent of the number of observers. The total time necessary to inform all observers increases by roughly 3 [ms] for each additional observer. This implies that in one second only 330 successive notification calls can be executed (obviously this number depends on the performance of the server and its network interfaces). Thus, in the case of the hardware used in our configuration, the product of the number of observers (Obs) and the average number of events per second (Evn) can be at most 330:

```
Obs * Evn <= 330
```

In the case of the ABng system, only the second source of events from those described in Section 2.2, namely changes in location, can cause an intensive stream of events. Each badge generates a new sighting every 10 [s]; however, statistically less than 4% carry meaningful information (every 4 minutes) – e.g. changes in location or clicking on one of the badge buttons. Only such events are reported further to observing applications. By applying the formula to these figures we can expect that the system will scale up to 500 badges and more than one hundred observer applications when a server with the processing power of ours is used. Moreover, filtering of events through the use of the view concept, presented earlier in the paper, further reduces the stream of events.

To scale the system additionally it is necessary to apply: a multicast protocol, the notification tree proposed in this paper, or a combination of these two approaches. If a multicast protocol is used, for instance by using Iona's implementation of the Event Service, the number of observers

in the formula `Obs` is equal to 1 and the number of meaningful events can reach 330 per second. By adding the notification component to the system this figure may be scaled further. The only disadvantage to this approach is the delay introduced by the notification component in the delivery of events to its observers. Moreover, the notification component is necessary when the system has to be extended geographically over a WAN, which is very rarely configured for multicast.

6 A representative ABng application

The most basic ABng application is the ABng viewer, called Jabba. The primary function of the viewer is to present a list of users with their current locations (Figure 9). Similarly, a list of equipment can be displayed. For every object, a number of its attributes are presented (e.g. a user name, a user address, a location name, location phone numbers, etc.). To perform these tasks the viewer has to collect a lot of information which is distributed among various repository servers. Additionally, information presented to the user has to be refreshed after any of the relevant repository objects change one or more attribute values. A change may reflect the current location of a user or a piece of equipment or other attributes, such as a user's address or a location description.

To work efficiently the viewer has to maintain a local copy of relevant information, i.e. to cache values of object attributes and to update values in caches after their originals are modified. In the first, ANSA-based version of the Active Badge location system, the viewer, called *xab*, was a very sophisticated and huge application and most of the *xab* code was related to maintaining caches. In the ABng viewer caching is implemented by the smart proxy layer. This has three major advantages:

- The code related to caching is completely separated from the application code. The application is not responsible for updates to the local copies of information.
- The application code is not aware that any caching algorithm is performed. It is completely transparent to the application. If the application wants to obtain an attribute value after being informed of a change (e.g. in order to redisplay it on the screen), it just calls the operation on the remote object which holds that attribute. However, the call does not actually generate a remote call on the object; it is caught by the smart proxy layer and served locally.
- The smart proxy layer is universal and can be reused in any ABng application.

In the bootstrap of Jabba, the hybrid approach to retrieving data from repositories was employed. First, all entities are retrieved by value, so their attributes can be very quickly displayed. In the background, references to these entities are acquired and the SPL is built. The construction of the SPL takes a considerable amount of time, however it is transparent to the user. After the SPL is constructed, it very efficiently supports the Jabba functionality.

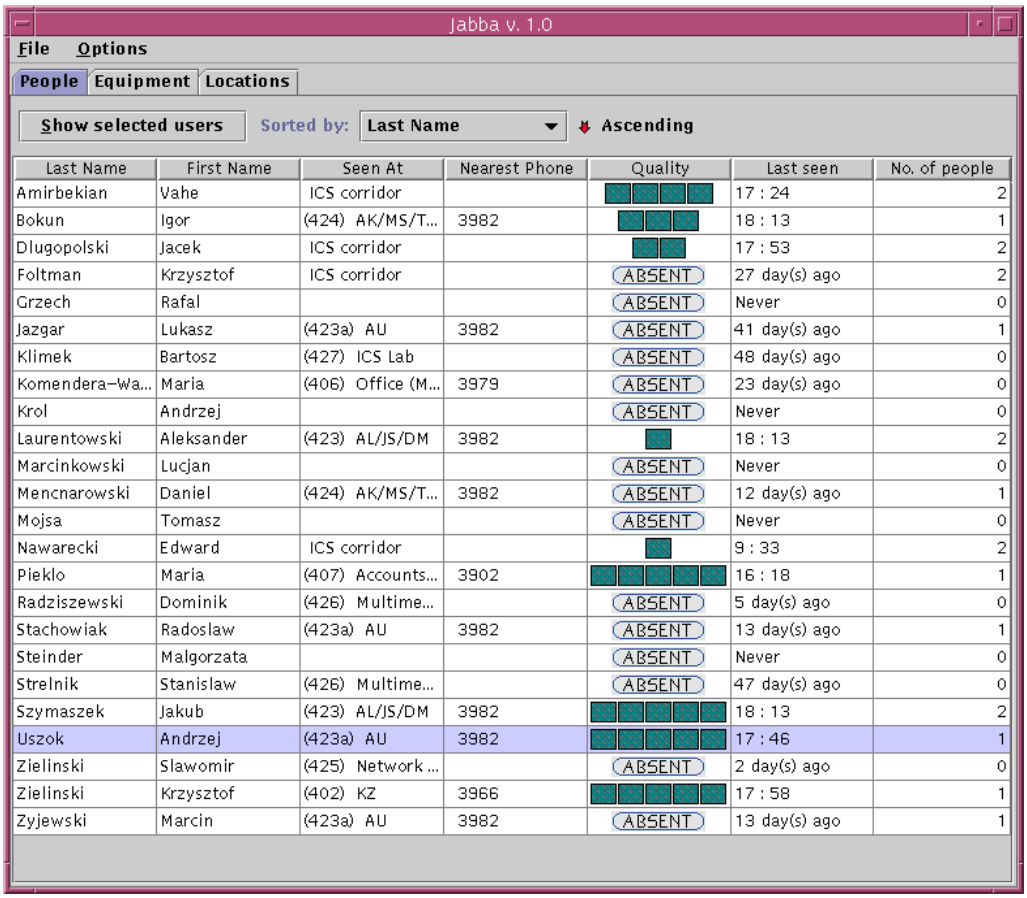

Figure 9: ABng viewer – a list of locatable users

7 Conclusion

Construction of scalable components in CORBA requires a solution to the well-known trade-off between space and simplicity of navigation in a large collection of objects, on the one hand, and system reaction time on the other. Access by CORBA references provides a conceptually clear and elegant model of access to objects in a distributed system, but when their number increases it causes unacceptable access time. On the contrary, access by value is much faster but sacrifices the ability to easily navigate in a distributed system. Our solution is to build hybrid components which combine both mechanisms. It is up to a programmer to use them correctly. The performance tests presented in the paper provide some hints as to correct use of the dual mechanisms.

A similar conclusion can be drawn with respect to the notification mechanism proposed in this paper. It works very well after the initial phase when the notification tree and smart proxies are already established but this phase takes a substantial amount of time. If a client needs fast response time, the initial values of entities should be obtained by value initially, and the notification tree should be constructed in parallel for future accesses and notification. The design and implementation in this paper have proven their correctness and scalability in a working ABng

system.

The repository component may be further enhanced by using the POA [17] approach, that provides new standard scalability mechanisms. We intend to pursue this in the future.

8 Acknowledgements

This work is supported by Olivetti-Oracle Research Laboratory, Cambridge, UK. We would like also to thank Joe Sventek for his thorough review and suggestions of extensive improvements to this article.

References

[1] APM Ltd., Cambridge UK. *ANSAware 4.0 – Application Programmer's Manual*, 1992.

[2] 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.

[3] Kathy Bohrer. Middleware isolates business logic. *Object Magazine*, 9(7):41–46, 1997.

[4] Michael Franklin and Stanley Zdonik. A framework for scalable dissemination-based systems. In *Proceedings of OOPSLA'97*, pages 94–105, October 1997.

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

[6] 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.

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

[8] Iona Technologies Ltd. *OrbixTalk Programming Guide*, July 1996.

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

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

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

[12] Sai-Lai Lo. *The OmniORB2 version 2.5, User's Guide.* Olivetti & Oracle Research Laboratory, January 1998.

[13] Thomas J. Mowbray and Raphael C. Malveau. *CORBA Design Patterns.* Willey, 1997.

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

[15] Object Management Group. *CORBAServices: Common Object Services Specification*, March 1995. last updated 07/98, OMG Report formal/98-07-05.

[16] Object Management Group. *CORBA Components, Joint Initial Submission by IONA Technologies et al.*, November 1997. OMG Report.

[17] Object Management Group. *Specification of the Portable Object Adapter (POA)*, May 1997. OMG Report.

[18] Object Management Group. *Notification Service, Joint Revised Submission by NEC System Laboratory et al.*, June 1998. OMG Report telecom/98-06-15.

[19] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects - Survival Guide*. Willey, 1995.

[20] Prashant Sridharan. *JavaBeans Developer's Resource*. Prentice Hall, 1997.

[21] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.