The following paper was originally published in the
Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)
Santa Fe, New Mexico, April 27-30, 1998

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

Jakub Szymaszek, Andrzej Uszok, and Krzysztof Zielinski
University of Mining and Metallurgy, Kracow

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

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, 30-059 Kraków, Poland
{jasz,uszok,kz}@ics.agh.edu.pl, http://galaxy.uci.agh.edu.pl~{jasz,uszok,kz}

Abstract

This paper presents experienc gathered when implementing the localization system for an office environment in CORBA. It describes methods which enable preserving fine-grained object-oriented structure of the system and achieving efficient performance at the same time. The presented study is a practical lesson about the implementation of a scalable system oriented towards information dissemination. 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 data records. The proper usage of this duality may have substantial influence on the overall system performance. The repository is equipped with a scalable notification mechanism built around a notification dispatcher and notification tree concepts. 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]. Those systems deliver information about changes of the 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 an 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 paper relaxes this limitation and proposes a template of a CORBA repository component with the incorporated light-weight mechanisms of notification, persistency and security.

This repository component combines and refines some ideas that recently appear in component oriented software environments such as Java Beans [12, 16], San Francisco Components [3], or CORBA Component proposed by Iona Ltd and others [15]. The major innovations 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 [7] has been exploited and three different types of the repository component have been proposed.

The structure of the paper is as follows. In Section 2 Active Badge next generation project, which induces the presented study is described. The repository component template is defined in Section 3. It is an observable component with dynamic attributes and a built-in searching engine. Changes of its attributes are propagated via notification mechanism. Implementation issues of this component are described in Section 4. The notification dispatcher concept used to built this mechanism is explained and a short description of the repository persistency and security functionality is presented.

Features and scalability of the proposed notification mechanisms are then analyzed in more details. This section includes also comparison of the proposed notification mechanism with CORBA Event Services. Next, in Section 5 performance evaluation study that concerns the investigated scalability is reported ad discussed. Section 6 presents basic application utilizing information gathered by ABng system. The paper ends with conclusions.

2 The Active Badges next generation project

The system, called *Active Badges*, was originally invented and developed at Olivetti Research Laboratory, in Cambridge, UK [6] in 1990-92. It uses 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 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 [2]. 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 the 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 [1] distributed environment.

2.1 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 [8] and OrbixWeb [11]. 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 than in terms of sensors and badges. A location is a part of an environment 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 incorporates development of the *Wonder Room* location-aware users' 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 currently nearest phone or setting parameters of various home appliances, such as air-conditioning, TV sets, VCRs, light, according to the preferences of users located in the neighborhood of these appliances, period of time, etc. Such applications may be used for *personalization* of user's equipment. Systems of this type are examples of, so called, ubiquitous computing [17].

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 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 possess 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. The 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 significantly extended, in the future.

Besides these numerous relationships between entities were grasped. A state of some entities depends or even is composed of the 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 it depends upon so as it can modify functionality of this entity and of the system. For instance, when a sensor is replaced or added to some room related *ABng_Location_Descriptions* and *Views* have to be informed, which in result will change processing of the sighting. Source of changes in states of entities can be:

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

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 lists of observers 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 objects in the real ABng system with even moderate number of users.

3 Component template

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

- General templates for an entity as well as a repository can and have to be designed,
- These templates should provide support for the implementation of a mechanism eliminating the overhead related to the representation of each entity as a separate CORBA object,
- A light-weight notification mechanism for repository clients has to be invented.

In the ABng three types of entities, and in the result three types of entity interfaces have been differentiated, with:

- **static attributes**: Most of 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 adequate interface. Such an interface offers a pair of access operations to set and retrieve the value of a single attribute in which an attribute is referred by its name and a value is decoded using the IDL *Any* type. The interface allows to retrieve all attributes as a list. This approach is an example of the application of the *Dynamic Attribute* design pattern [7]. This type of an interface is provided by entities which have many attributes or these attributes are different for individual entity instances. The 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 these attributes which are common for all objects representing by entities. Additionally, it uses the *Dynamic Attribute* paradigm to provide an access to object-specific attributes. An example of an ABng object providing such an interface is *Equipment* whose instance describes a piece of office equipment. For all kinds of equipment a set of common attributes has been distinguished, such as a name, a vendor name, etc., which has been defined as explicit attributes.

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

The template of an entity interface was defined as follows:

```
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;
}
```

The template 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 a 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 of an entity state. When the state is changed the 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 attributes of a given entity. This structure is used to get the whole meaningful state of the entity in just one request. This feature was mainly designed in order to be used 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 when cache is invalidated by the notification mechanism, which is described later on.

The next uniform element is a 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 to return the entity state as a set of already collected data without any references to the 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 the above mentioned one. The difference is that 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. In a interfaces with hybrid attributes occur both explicit attributes and a list of name/value pairs accompanied by access operations.

Objects built according to any of the three templates 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,
    in Entity::Description data_record)
    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);

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

The operations of a repository are a 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 the reference with values stored in the *Description* structure.
- *remove* – remove an entity denoted by the reference.
- *find* – returns a collection of references of these 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 entities descriptions is returned.

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

4 Implementation 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.

4.1 Light notification mechanism

A typical ABng application can use a lot of various information encapsulated by ABng objects. To obtain this information an application interacts with various ABng objects. To optimize these interactions, ABng implements a caching algorithm – 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 *find* method), the smart proxy of the entity object is instantiated within the application's address space and entity's descriptions is 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, in ABng 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 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.

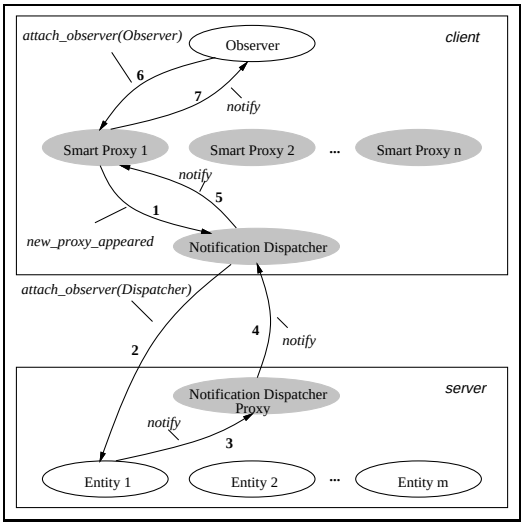

Figure 1: ABng notification mechanism

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, 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.

4.2 Persistency

States of entities have to survive 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 restored in the appropriate file.
- *Object Database Object Adapter (ODOA) [10]* – this sophisticated mechanism of achieving CORBA objects persistency uses an object database (ObjectStore [13]) 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 opens a heavy *update* transaction. The OOA used in the ABng was obtained as a specialization of the Object Database Adapter Framework [9]. 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 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.

4.3 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.

4.4 The scalability of 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 of the notification has the following structural features:

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

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

This last drawback could be overcome using replication. The client component with collection of smart proxies and notification dispatcher could be generalized as a notification component shown in Fig.2. The scalability with respect to the number of clients could be achieved by organizing the system into notification tree built of notification components. Each smart proxy has registered several notification dispatchers of the higher layer, etc. In the root node the repository of entities exists. In other nodes only smart proxies are present. The proposed architecture represents in fact a distributed collection of entities which could be highly available for large number of clients despite of geographic distribution.

The propagation of the repository entity update is marked for example in Fig.2. It is easy to see that the proposed solution has a similar scalability to notification based on multicast over IP communication protocols that in fact propagate messages down to a multicast tree. The advantage is that the notification tree does not require any multicast protocol support.

In context of this discussion it is necessary to ask about comparison of proposed solution with CORBA Event Services. There are some similarities and differences. The most important difference is that Event Service does not use a smart proxy concept so caching has to be solved in separate way. The notification component is similar to the event channel in the sense that it separates the repository as a source of events from the client. Further comparison is very much dependent on implementation details which are not defined by the OMG specification. For instance, in Iona's implementation based on a multicast protocol usage an events producer does not even know the number of notified consumers. This approach scales well but is based on the proprietary protocol and is very difficult to extend from LAN to WAN. To achieve selective notification it is necessary to define as many event channels as many sources of notification exist.

5 ABng system evaluation

The ABng software, implemented according to the design concepts presented in this paper, was subject to intensive testing in regards to its performance and scalability. The system was implemented in Orbix 2.2MT whereas clients, used in tests, were implemented in OrbixWeb 2.01. OrbixWeb 3.0 was not used as its mapping of a sequence of object references, when returned by a server is faulty: smart proxies are not created for returned references. This results in incorrect operation of the Smart Proxy Layer proposed in this paper.

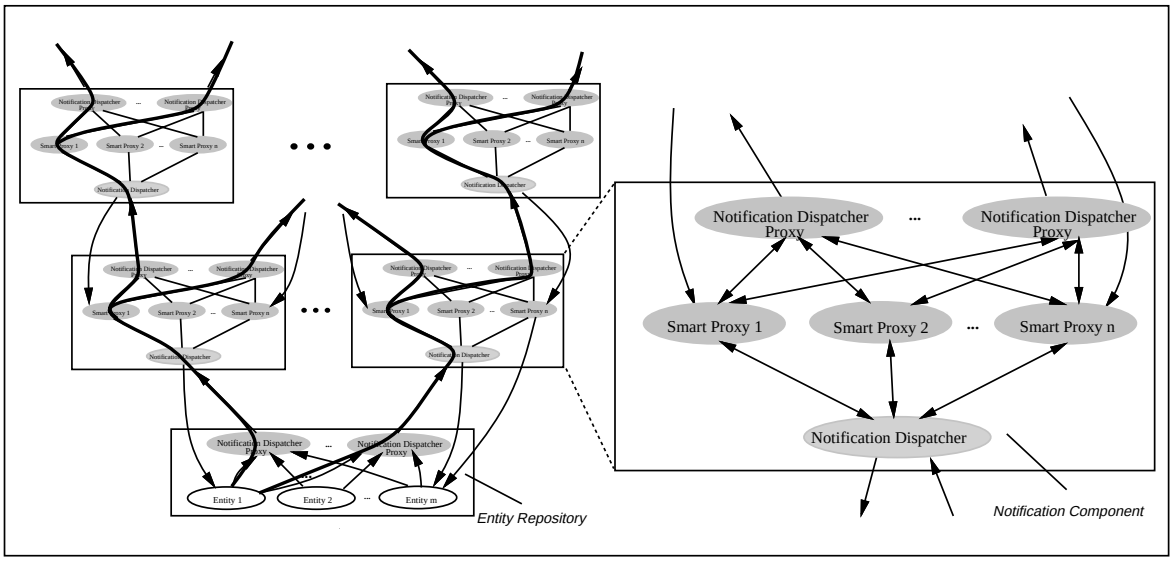

Figure 2: Notification Tree

Performance results presented in this section were obtained in the environment consisting of 15 Sun Ultra workstations and a Sun Enterprise 3000 server, connected by 2 Ethernet 10Mb switches. 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 of the system component was chosen for the test. However, 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 usual activities during the test, but were rather slightly loaded.

Performance tests were carried out according to typical scenarios occurring in majority of ABng applications i.e: retrieving of the current state of the entities in an application bootstrap and scalability of notification mechanism when numbers of applications observing changes in entities increase. Improvements of implementation of these activities 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 repository: by their values, using *find_value* and by their references, using *find*. In order to obtain values of entities, when using *find*, subsequent *get_descr* methods have to be called. Time spent in these calls executed in the OrbixWeb applet when numbers of entities in repository increase is presented in Fig. 3 and Fig. 4. All these calls were invoked with a *pattern* matching all entities in the repository. Results show that accessing entities by references is by few magnitudes more costly than accessing them by value. Thus access by references, 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.

These figures present also performance of 2 subsequent invocations of the *find* method when SPL is used. The execution time of the second *find* one is obviously almost neglectable as it happens locally. Additionally, the first call with SPL is about 40% more costly than combined *find* without SPL and *get_descr* calls. This difference is caused by the SPL construction time.

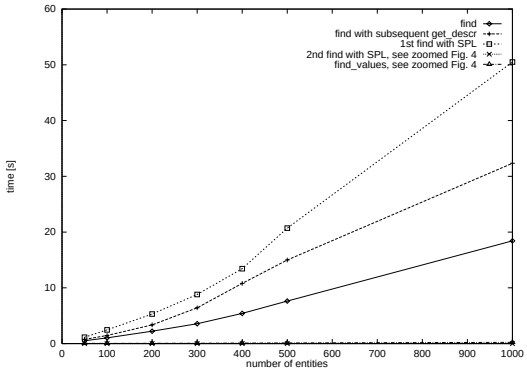

Figure 3: Performance comparison of different ways of accessing states of repository entities

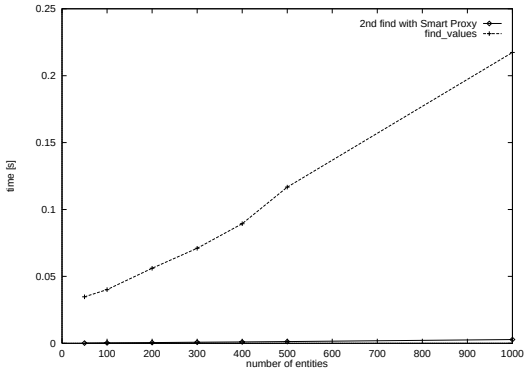

Figure 4: Performance comparison of different ways of accessing states of repository entities, zoomed

5.2 Analysis of SPL construction cost

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

1. acquiring of entity references by executing the *find* method,
2. creating of smart proxy for each of the acquired reference,
3. registering of the notification dispatcher for each of the references,
4. retrieving of entity descriptions by executing the `get_desc` 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 ones include as many remote invocations as many are acquired references. In Fig. 5 the total SPL construction time as well times sent in individual steps are presented.

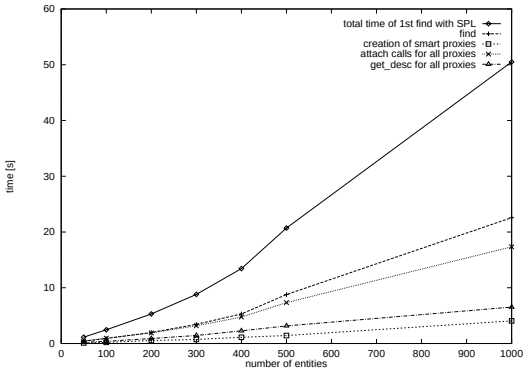

Figure 5: Time consumption by basic steps of SPL construction

These results show that majority of the time is used for remote invocation. Most of these invocations could be canceled by usage of the *find_and_attach* repository method, which eliminates necessity of remote calls from the third and fourth steps. This reduces the SPL construction time by 45%. However, this approach additionally 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 observer (parameter *obs*) initiated to the notification dispatcher. The returned sequence of entities values (out parameter *descriptions*) is used for filling smart proxy caches.

5.3 Notification time

The method used to notify observers registered in the repository about its changes are asynchronic *oneway* operations invoked successively on the observers. The impact of growing number of entity observers on the notification time is presented in Fig. 6.

The obtained results show that the time of executing *update* on an entity is almost independent of a number of its observers. The total time of informing all observers however increases by roughly 3 [ms] for each additional observer. It means that in one second only 330 successive notification calls

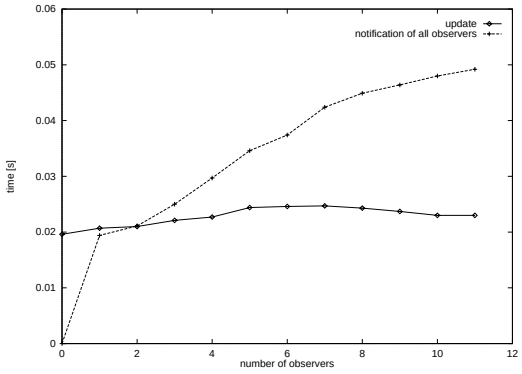

Figure 6: Impact of growing number of observers on entity update and notification times

can be executed (obviously this number depends on the performance of a computer server). Thus, in the case of the used hardware, the product of a number of observers (`Obs`) and an average number of events per second (`Evn`) could be at most 330:

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

In the case of the ABng system only the second source of events from these distinguished in Section 2.2, namely changes in location, can cause intensive stream of events. Each badge generates a new sighting every 10 [s], however statistically less than 4% of them carry meaningful information (every 4 minutes) i.e.: changes in location or clicking on one of 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 then one hundred observer applications, when an adequately fast server computer is used. Moreover, filtering of events by the usage of the *View* concept, presented earlier in this paper, further reduces the stream of events.

To scale the system additionally it is necessary to apply: some multicast protocol, the notification tree proposed in this paper or a combination of these two approaches. In the case when a multicast protocol is used, for instance by the usage of the Iona's implementation of the Event Service, a number of observers in the formula `Obs` is equal to 1 and a 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 of this approach is a 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 WAN, which is very rarely configured for multicast.

6 A representative ABng application

The 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 7). 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 object changes one of its attributes. A change may concern the current location of a user or a piece of equipment or other attributes, such as a user 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 the most of the *xab* code was related to maintaining caches. In 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 of the local copies of information.
- The application code is not aware that any caching algorithm is performed. It is completely transparent for the application. When the application wants to obtain an attribute value (e.g. in order to redisplay it on the screen), when it got informed of the change,

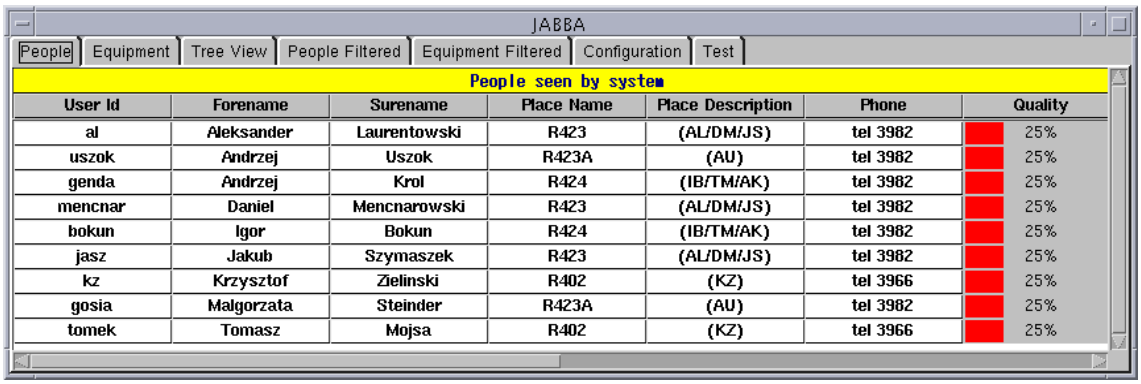

Figure 7: ABng viewer – a list of locatable users

it just calls the operation of the remote object which holds that attribute. However, the call does not come out from the client application's address space. It is caught by the smart proxy layer and served locally.

- The smart proxy layer is universal and it 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 of these entities are acquired and SPL is built. It takes considerable amount of time, however it is transparent for the user. When the SPL is built it very efficiently serves Jabba functionality.

7 Conclusion

Construction of scalable components in CORBA requires solution of well known trade-off between a space and a simplicity of navigation in a large collection of objects on the one hand and a system time of reaction which is a major scalability factor on the other hand. The access by CORBA references provides conceptually clear and elegant model of access to objects in a distributed system but when their number increases it induces not acceptable access time. On the contrary access by value is much faster but is loosing the ability of easy navigation in a distributed system. So the solution is to build hybrid components which combine the both proposed in this paper mechanisms. It is up to a programmer to use them correctly. Some hints in this matter are performance tests presented in the paper.

The similar conclusion could be drawn in 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. So for a client which needs fast response time the initial value of entities should be get by value at first place and the notification tree should be constructed in parallel for future accesses and notification.

The design and implementation solutions presented in this article proved its correctness and scalability in the working ABng system.

The presented repository component may be further enhanced by using the POA [14] approach, that provides new standard scalability mechanisms. It is our intention to follow in this direction.

8 Acknowledgments

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

References

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

[2] F. Bennett, A. Harter, *Low bandwidth infra-red networks and protocols for mobile communicating devices*, Technical Report 93.5, Olivetti Research Laboratory, Cambridge, UK (1993).

[3] K. Boher, *Middleware Isolates Business Logic*, Object Magazin, 11 (1997).

[4] M. Franklin, S. Zdonik, "A Framework for Scalable Dissemination-Based Systems", *Proceedings of OOPSLA '97* (1997) p. 94–105.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison-Wesley (1994).

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

[7] T. Mowbray, R. Malveau, *CORBA Design Patterns*, John Wiley and Sons, Inc. (1997).

[8] Iona Technologies Ltd., *Orbix 2.1 Programming Guide* (1996).

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

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

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

[12] R. Orfali, D. Harkey, J. Edwards, *The Essential Distributed Objects Survival Guide*, John Wiley and Sons, Inc. (1996).

[13] Object Design, Inc., *ObjectStore C++ API User Guide, Release 4.0.1* (1996).

[14] Object Management Group, *Specification of the Portable Object Adapter (POA)*, OMG Document orbos/97-05-15 (1997).

[15] Object Management Group, *CORBA Components, Joint Initial Submission by IONA Technologies et al.*, OMG Document orbos/97-11-24 (1997).

[16] P. Sridharan, *Java Beans, Developer's Resources*, Printice Hall (1997).

[17] M. Weiser, *Some computer science issues in ubiquitous computing*, Communications of the ACM, 6 (1993) p. 75–84.