DCUP: Dynamic Component Updating in Java/CORBA Environment

František Plášil[1,2], Dušan Bálek[2], Radovan Janeček[1]

[1] *Charles University*
Faculty of Mathematics and Physics,
Department of Software Engineering
Malostranské náměstí 25, 118 00 Prague 1,
Czech Republic
phone: +420-2-2191 4266
fax: +420-2-532 742
e-mail:{plasil, janecek}@nenya.ms.mff.cuni.cz

[2] *Institute of Computer Science*
Czech Academy of Sciences
Pod vodárenskou věží
180 00 Prague 8
Czech Republic
phone: +420-2-6605 3291
fax: +420-2-858 5789
e-mail: {balek, plasil}@uivt.cas.cz

Keywords: software component, dynamic updating, Java, CORBA, Module Interconnection Language, externalization

Abstract. In this paper, the authors present a novel architecture, called *DCUP (Dynamic Component Updating)*, which allows for dynamic component updating at run time (components are frameworks of objects). The following key problems of dynamic component updating are addressed: (1) making an update of a component fully transparent to the rest of the application, (2) transition of state from the old to the new version of a component, (3) transition of references which cross the component boundary (in both directions), (4) dynamic communication with a component provider. In DCUP these problems are addressed by a small set of abstractions with a clear separation of their functionality. In contrast with the usual believe that it is difficult to map abstraction supporting component based programming to concrete computer systems, the abstractions proposed by DCUP are very easy to map to the Java and CORBA programming environments.

1 Introduction

1.1   Updating components

It is generally believed, that in the (potentially near) future, many of the software applications will be integrated from reusable components and that there will be a market with such components. Inherent to the idea is the necessity to customize such a component in accordance to specific requirements of a particular application and to set the necessary interconnection with other parts of the application. With respect to the goal of this paper (Section 1.2), we limit ourselves to components for object-oriented environments. There are many approaches to setting the interconnections [MDK94, IB96, BAB96, FS96], but not all of them deal with customization as well as e.g. Java Beans do [JBN97]. The thing that still remains to be a challenge is the component updating with minimal human effort/interaction. This is particularly important if the components are available on the market. The issue is even more complex in the case where the updating should take place at run time, as is necessary e.g. in real-time applications.

In general, updating a component at run time inherently means disconnecting it from the other parts of the application and connecting a new version of the component back into the application. Here, two key issues arise. First, what to do if the new component does not support exactly the same interface as the old one; here connectors [BAB96] are a way to deal with the problem. Second, references into and out of the component have to be updated. The former ones are typically handled by a higher-level abstraction of the reference (like the CORBA reference [OMG95a], or the event listener [JBN97]) which can be reassigned to a target in the new component, or by introducing auxiliary objects which mediate access into the component (wrappers,

proxies). The references out of the component are usually handled as a "requires" abstraction [MDK94, BAB96]. Another key issue of component updating is the conversion of the state between old and new versions of a component. This issue is not addressed by any of the systems mentioned above i.e. [MDK94, BAB96, JBN97].

A further problem is how to control the actual update of a component and what kind of knowledge on the other part of the application is necessary. A typical approach chosen is an interactive communication via some kind of "application builder" [JBN97, FS96]; usually, an update has to be done by a qualified person aware of the structure of the framework which is being rebuilt. Another interesting approach uses the Castanet commercial product [MA96]; based on a time schedule downloaded with a particular component, it poles, at the component's provider side, for new versions of the component. However, actual updating, possible only at a file granularity level (libraries of classes, data files), should only be done while the application is not running (otherwise the update correctness is not guaranteed).

All in all, to paraphrase [MB96], the main obstacle in a large application of component based programming is the difficulty of mapping the proposed abstractions into concrete working computer systems. This is reflected by the fact that there are just a few significant commercial products available at the moment (e.g. [MA96, JBN97, KON96]).

1.2 The goal of the paper

The purpose of this paper is to present a novel architecture, called *DCUP (Dynamic Component Updating)*, which allows for dynamic component updating at run time and where components are frameworks of objects. DCUP has been designed to also support updatings which are initiated by the original component provider and are to be done transparently to the user, in an automated way.

The first goal of the paper is to point out the clear functional separation of the basic abstraction in DCUP. The choice and functionality of these abstractions were guided by the following design issues:

(a) versioning and communication with a component provider
(b) transitions among a component's versions
(c) transparency of a component's updating with respect to the rest of the application
(d) fitting both into the classical centralized and distributed environments.

Our second goal is to stress the scalability of the DCUP architecture as a pleasant consequence of the fact that these abstractions can be applied recursively.

1.3 Structure of the paper

In Section 2, an overview of the DCUP architecture is provided. More details on the key DCUP abstractions are given in Section 3. The mechanism designed to support communication with a component provider is described in Section 4. As a proof-of-the-concept, a simple application has been prototyped. Section 5 presents the key fragments of the application and summarizes experiences gained while designing and debugging the prototype application. Section 6 is devoted to related work and future intentions. Finally, the key achievements are summarized in the concluding Section 7.

2 DCUP Architecture overview

2.1 Application and components

An *application* in DCUP is a tree-like hierarchy of nested components. A *component* is a framework of objects (class instances). A subset of operations provided by a component can be accessible for its parent or sibling components via a set of interfaces.

Components are dynamically updatable; with respect to an update operation a component is divided into its *permanent part* and *replaceable part*. Orthogonally, with respect to the nature of the operation provided, the

component is divided into its *functional part* and *control part*. The respective interfaces are called *control interface* and *functional interface.*

2.2 Structure of a component

The framework devising a component has to contain two distinct objects: exactly one object supporting `ComponentManagerInterface` (Section 3.1) and exactly one object supporting `ComptentBuilderInterface` (Section 3.2); conventionally we call these objects the Component Manager (*CManager* for short) and the Component Builder (*CBuilder* for short). A CManager is the heart of the component's permanent part. Similarly, a CBuilder is the key object of the component's replaceable part.

In addition, a component can contain *functional objects* (the "plain" objects which device the functionality of the component) and, recursively, nested components (subcomponents). Each of them has its own CManager and CBuilder. A functional object is *reachable* if it supports the `Reachable` interface (Section 3.4). A reachable object can be accessed from the parent component via a *wrapper* object (Section 3.3) which is associated with it. Wrappers belong to the permanent part of the component.

In summary, a component is a framework divided, with respect to updating, into the permanent part and replaceable part (Figure 1a). The permanent part contains a CManager and wrappers of the component. The replaceable part contains a CBuilder, functional objects, and subcomponents of the component. With respect to operations provided, the component is orthogonally divided into its functional part and control part (Figure 1b). The functional part contains functional objects together with respective wrappers, and subcomponents. The control part contains CManager and CBuilder.
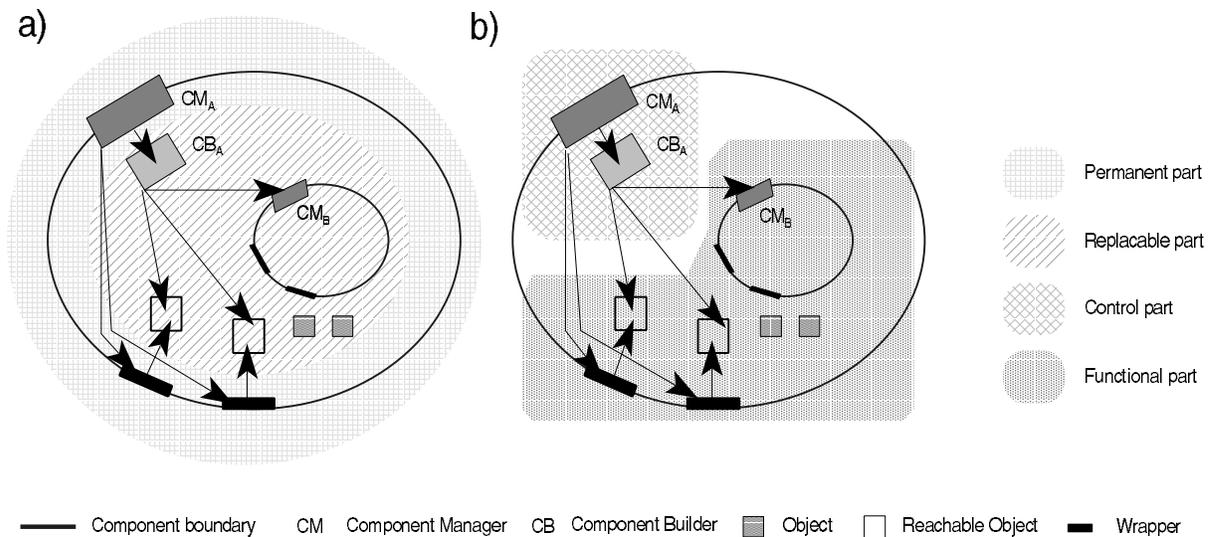

Figure 1 Structure of a component

2.3 Component building, updating, and terminating

By *updating* of a component we mean replacing its replaceable part by a new version of this part at run time. Thus, the *lifecycle* of a component is the sequence RP_1, RP_2, ... , RP_n, where RP_i is the i-th version of the replaceable part. In DCUP, each RP_i version of the replaceable part is associated with a $CBuilder_i$ (the i-the version of the Component Builder). The `CBuilder`$_i$`.onArrival()` method creates the functional objects and the subcomponents of RP_i, initializes their state (usually from the externalized state of a previous version of the component), and sets up all references among the objects in RP_i. (We say that `CBuilder`$_i$`.onArrival()` builds RP_i). The `CBuilder`$_i$`.onLeaving()` method ends all execution threads in the replaceable part of the component, and externalizes the state of its "important" objects; finally it destroys all functional objects and

subcomponents in RP_i. (We say that the *CBuilder.onLeaving() terminates* RP_i .)

In a component, the updating transitions RP_i -> RP_{i+1} are controlled by its CManager. To terminate a RP_i, the CManager calls *CBuilder$_i$.onLeaving()*, loads a new version of the CBuilder class, creates a CBuilder$_{i+1}$, and builds a RP_{i+1} by calling *CBuilder$_{i+1}$.onArrival()*. More specifically, the updating transitions are determined by the *updateComponent()* method of the CManager. The basic functionality of this method can be captured by the following pseudo-code:

```
public class CManager {
  . . .
  updateComponent(String subComponentName, String builderClassFile,
                  String storeDataStoreID, String restoreDataStoreID){
    // if ComponentName is the name of a subcomponent, then delegate this call to the
    // corresponding CManager, else:
    OldBuilder.onLeaving(storeDataStoreID);
    NewBuilder = new ComponentBuilder(BuilderClassFile);
    NewBuilder.onArrival(RestoreDataStoreID);
  }
}
```

2.4 Referencing across component boundaries

To get access to the functionality of other components, functional objects may need to reference objects in these components. In DCUP, the way these references are handled depends on their direction in the hierarchy of component nesting.

a) **Downwards reference.** Only references into direct (child) subcomponents are allowed. If an object FO_C in a component C needs a reference to another object FO_S in a subcomponent S, it can get it via a *CManager$_S$.bindToReachableObject()* call. FO_S must be a reachable object. In S, as a result of the *bindToReachableObject()* operation (*binding*), a wrapper object WO is created and FO_C obtains a reference to WO. This wrapper object mediates access to FO_S from outside of S (Section 3.3). The following code fragments illustrate the idea:

```
//in C component:
    ReqReference = CManagerₛ.bindToReachableObject("ReqObjectName");
    //CManagerₛ provided by the CBuilderc
    ReqReference.doAnOperation();

//in CManagerₛ:
    bindToReachableObject(String name){
    . . .
    aWrapper = CreateCorrespondingWrapper(name);
    aWrapper.setTarget(ReachableObject);
    return aWrapper;
    }
```

b) **Upwards reference.** Basically, it is the parent component's responsibility to provide the required reference. In analogy with the *require* construct in Darwin [MDK94], a component C indicates its requirements by providing the *getRequirements()* (specifies the required references via a list of agreed names) and *provideRequirements()* (provides the corresponding references) methods. These methods are to be called by the object which creates C (typically the CBuilder of the parent component P). After the component gets all required references we say the component is *embedded*. The following fragment of the code illustrates the idea:

```
// embedding of C:
// (in the object which creates C, typically CBuilderₚ)
Requirements req = CManagerc.getRequirements();
req.supply(object);
CManagerc.provideRequirements(req);
```

2.5 Component naming

As mentioned in Section 2.1, an application is a tree-like hierarchy of components. To update, create, and destroy a component located somewhere in the hierarchy, we apparently need a support for the identification of such a component. Further, if a client of a component needs to bind to a reachable object inside the component, it has to determine the object by a name. For these purposes, two orthogonal namespace types are employed:

1) The *component hierarchy namespace*. There is only one instance of this namespace type per application. The hierarchy of components in an application is reflected by usual composed names of its components. Conventionally, the name of a component is represented by the (composed) name of the component's CManager. (The respective syntactical sugar is illustrated in Section 5.)

2) The *referencial namespace*. There is one instance of this namespace type per component. The namespace associated with a component is formed by the names of reachable objects in the component and by the "formal" names which denote the required upwards references of the component (only simple, non-composed names, form the namespace).

During an update, a new version of a component builder may change the structure of a component framework which results e.g. in a different number of reachable objects etc. If this is the case, the builder has to provide a renaming scenario (Section 3.2) to map the old names into new ones.

2.6 Updater: activation of updating

In DCUP, new versions of a component are available for downloading from a *component provider*. Two essential kinds of updating schemes are recognized: *push model* - the updating process is activated from the outside (by the *component provider*), and *pull model* - the updating process is activated from the inside of the application itself (e.g. according to an *update schedule*). Coordination of the updating process is based on the `UpdaterInterface` (Section 3.1) which has to be supported by a dedicated thread, called *Updater*, associated with a CManager. As the response to a request for an update, the Updater gets a new version of the updated component from the respective component provider, and calls the `updateComponent()` method of the associated CManager.

The Updater is allowed to update any component belonging to the CManager's naming subtree. In general, not every CManager is associated with an Updater; in a typical case, only the highest level CManager is associated with an Updater. However, should a component update request be issued by more Updaters, the requests are guarantied to be serialized by the implementation of related CManagers. More details on updating process will be given in Section 4.

3 Key abstractions - more details

3.1 Component Manager and Updater

Formally, `ComponentManagerInterface` is an empty interface which inherits from `ComponentControlInterface` and `BinderInterface`. The Component Manager implements two interfaces that separate the two main areas of its functionality. The `ComponentControlInterface` is used e.g. for creating a component, component updating, and terminating. The `BinderInterface` reflects the role of setting inter-component references. For brevity, we introduce only the most significant methods of these interfaces:

```
public interface ComponentControlInterface{
/* registering part */
  void registerEmbededManagerName(String name, ComponentManagerInterface target);
  void registerReachableObjectName(String name, Reachable target);
  // dual methods for unregistering exist
```

```
/* updating part */
  void updateComponent(String subComponentName, String builderClassFile,
                  String storeDataStoreID, String restoreDataStoreID);
  // methods for component creating and destroying exist with similar parameters
}
```

The `ComponentControlInterface` (CCI) is divided into two distinct parts. The *registering* part consists of methods which register component managers and reachable objects of the component managed by the given CManager. These methods have an input parameter `name` which uniquely identifies the object/manager within a particular namespace (Section 2.5). The *updating* part pertains to updating of a component. In `updateComponent()`, the parameter `subComponentName` is a name of the component to be updated; `builderClassFile` identifies the new CBuilder class which will be loaded and launched to build a new version of the component.

BinderInterface comprises of the methods for handling both downwards and upwards references (Section 2.4):

```
public interface BinderInterface {
 //downwards references
   Object bindToReachableObject(String name);
 //upwards references
   Requirements getRequirements();
   void provideRequirements(Requirements references)
   Object getOutsideReferenceNamed(String name);
}
```

As mentioned in Section 2.6, a CManager can be associated with an Updater. In `UpdaterInterface,` the `handleUpdateMessage()` method plays a key role; its task is to handle incoming update messages and to activate the update action associated with the message.

```
public interface UpdaterInterface {
  void handleUpdateMessage();
}
```

3.2   Component Builder

The functionality of the CBuilder abstraction was described in Section 2.3. The corresponding interface takes the following form:

```
public interface ComponentBuilderInterface {
  void onLeaving(String dataStoreID);
  void onArrival(String dataStoreID);
  void store(String dataStoreID);
  void restore(String dataStoreID);
  ScenarioInterface getScenario();
}
```

If, in the new version of the component, modifications to its local name space are to be done, the `getScenario()` method provides corresponding mapping between the old and the new name space.

3.3   Wrappers

As explained in Section 2.2, a wrapper mediates access to a reachable object of a component (*target*), and it is a part of the functional interface of the component. After an object, external to the component, has obtained a reference to a wrapper via the `bindToReachableObject()` operation of the component's CManager, it can call the operations of the target. A wrapper's functionality is as follows:

(a) it delegates calls to its target object, (b) it provides the component's CManager with the number of threads currently visiting the component via this wrapper, (c) it locks access to its target during the updating of the component, (d) if an operation on the wrapper's target x is to return a reference to another reachable object y inside the component, the wrapper performs the appropriate `bindToReachableObject()` operation and returns the y's wrapper.

3.4 Reachable objects

Reachable objects of a component are objects which provide the "real" functionality of a component. Reachable objects are the only objects "visible" from the outside of a component. A reachable object has to implement the `Reachable` interface and has to be also registered (Section 3.1) with the CManager of a component.

```
public interface Reachable {
  String getName();
  void setName(String name);
}
```

4 Updater - Provider interplay schemes

In both push and pull models, an update request ends by calling `handleUpdateMessage()` in the Updater of a component. In this section, we will focus on the issue where this method should be activated.

In principle two schemes emerge:

a) direct connection between application and a component provider; `handleUpdateMessage()` is called directly by the component provider;
b) the connection to a component provider is mediated by a (global) *Update daemon* which runs at the user side. (This approach is employed e.g. in [MA96].)
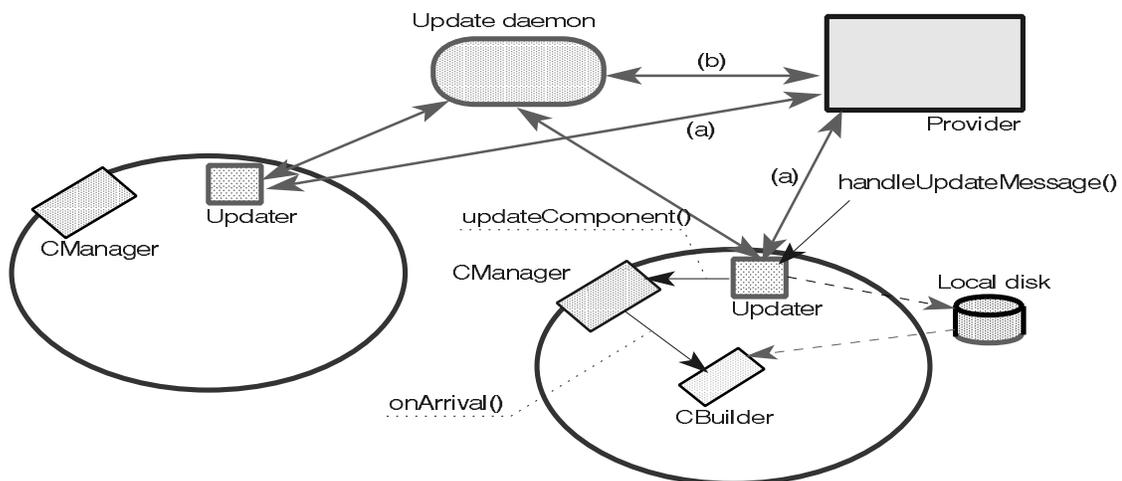

Figure 2 Updater - Provider interplay
a) direct connection
b) connection mediated by Update daemon

In general, an update daemon mediates update requests to Updaters in running applications. However, if a target application is not currently running, requests are buffered in the Update daemon. Moreover, the Update daemon can be employed as a cache. Once an Update daemon receives an updating request, it can use a special protocol to obtain all the classes necessary for an update process and prefetch/cache them before the message is delivered to the corresponding Updater. As mentioned in Section 2.6, in response to a request for an update, the Updater gets a new version of the updated component from the respective component provider. To make it easy to maintain the last version of the component classes, the Updater downloads all necessary classes to its local external storage (prefetches them). Then, the CBuilder of the component gets the new version classes from the local external storage.

Detailed investigation of update activating protocols will be a subject of our future research.

5 Example - Banking Application

5.1 Decomposition into components

This section illustrates the DCUP idea on an example. The example was implemented as a proof of the concept prototype; this section contains fragments of the code.

Let's suppose that there is a bank in which a number of tellers serve a potentially huge amount of customers. A customer specifies the desired transaction to a teller who then accomplishes the request. If there is an overdraft, he or she has to ask their supervisor for an approval. Both the teller and the supervisor access an account repository. To model the bank we introduce the following components: `Bank`, `Supervisor`, and `DataStore`. The Banking Application is implemented as the uppermost component `MainComponent`. The nesting and instantiation of these components is illustrated in Figure 4.
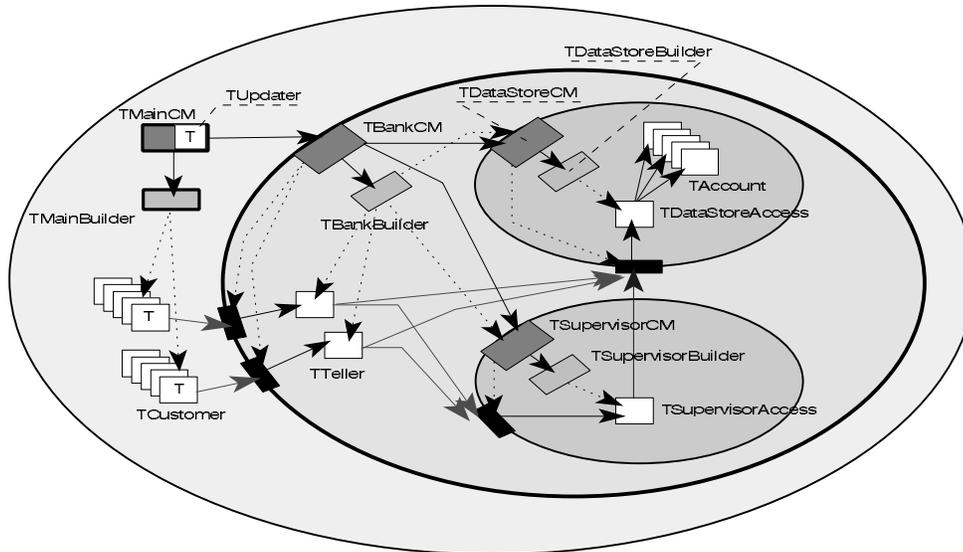

Figure 3 Structure of the application

5.2 Main component and its initialization

Starting the Banking Application means instantiating and creating the MainComponent. This is done by the following code:

```
public static void main(String []args){
  TMainCM app = new TMainCM();
  app.createComponent("", "BankingApplication.TMainBuilder");
}
```

The `createComponent()` method instantiates the `TMainBuilder` class and calls its method `onArrival()`, which then builds the whole framework of the MainComponent. So, following Figure 4, `TAppBuilder` creates the Bank component and a number of the `TCustomer` objects (implemented as threads). The `onArrival()` might look as follows:

```
// in TMainBuilder class
public void onArrival(String dataStoreID){
  BankCM = new TBankCM();
  ParentCM.registerEmbeddedManagerName("BankCM", BankCM);
  BankCM.createComponent("", "bank.Bank.TBankBuilder");
  for(int i = 0; i < NumOfCustomers; Customers[i++] = new TCustomer());
}
```

Once a `TCustomer` is created, it binds itself to a (randomly chosen) `Teller` and uses its services:

```
CI = (TellerInterface) ParentCM.bindToReachableObject("Teller3");
accNumber = CI.createAccount(1500);
CI.deposit(accNumber, 2000); CI.withdraw(accNumber, 1000); ...
```

5.3   Creating other components

Following the general scheme of creating components (Section 2.3), the *BankCM.createComponent()* method calls *BankCB.onArrival()* which then creates the (reachable objects) `Tellers`, the Supervisor component and the DataStore component.   So *BankCB.onArrival()* looks as follows:

```
// in the TBankBuilder class (instantiated as a BankCB)
public void onArrival(String dataStoreID){
  SupervisorCM = new TSupervisorCM();
  ParentCM.registerEmbeddedManagerName("SupervisorCM", SupervisorCM);
  DataStoreCM = new TDataStoreCM();
  ParentCM.registerEmbeddedManagerName("TDataStoreCM", TDataStoreCM);
  SupervisorCM.createComponent("", "bank.Supervisor.TSupervisorBuilder");
  DataStoreCM.createComponent("", "bank.DataStore.TDataStoreBuilder");
```

After that, `BankCB` asks its subcomponents for their upwards reference requirements via calling *getRequirements()* of their CManagers. It is a call that returns a `Requirements` object which encapsulates the references requested. `BankCB` provides these references by calling *provideRequirements()* of the CManagers in the subcomponents (Section 2.4). In our example, to be more specific, `BankCB` is responsible for providing the Supervisor component with the reference to the `DataStoreAccess` reachable object:

```
  Requirements Req = SupervisorCM.getRequirements();
  DataStoreInterface DSI = DataStoreCM.bindToReachableObject("DataStoreAccess");
  Req.supply(DSI);
  SupervisorCM.provideRequirements(Req);
```

After `Tellers` are created, as they are reachable objects, `BankCB` has to register their names. This registration allows for the binding to the `Tellers` from the outside. Further, the `Tellers` are provided with references to reachable objects in the DataStore and Supervisor components:

```
  SupervisorInterface CI = SupervisorCM.bindToReachableObject("SupervisorAccess");
  for(int i = 0; i < NumOfTellers; i++){
    Tellers[i] = new TTeller();
    ParentCM.registerReachableObjectName("Teller"+i; Tellers[i]);
    Tellers[i].setSupervisor(CI);
    Tellers[i].setDataStoreReference(DSI);
  }
}
```

So this ends the *onArrival()* method of the `BankCB`. .

5.4   Updating

The Updater thread associated with `TMainCM` listens for an update message (we do not specify the source or the messages). After it receives an update message, it calls the *updateComponent()* method of its parent CManager. So, for example, the DataStore component could be updated as follows:

```
  ParentCM.updateComponent("Bank.DataStore", "bank.DataStore.TNewDataStoreBuilder",
                      "/Repository/CityBank", "/Repository/CityBank");
```

The first string parameter denotes the DataStore component within the Bank component. The second parameter represents the location of the new `TDataStoreBuilder` class and the last two parameters are store resp. restore locations for externalizing the DataStore component state. Note, that updating can be done without the tellers

and the customers being aware of it.

5.5 Experience gained from prototype implementation

At first sight, there is a lot of "manual" work while implementing CManagers, wrappers, etc. However, the code of these objects nearly always looks the same regardless of the component. Thus, it seems to be natural to design and employ some kind of Component Description Language (CDL), in which one could describe component interfaces and (potentially) specify interconnections between components similarly to e.g. Darwin [MDK94]. For a component, CManager code, Wrappers, CBuilder skeleton and component requirements can be generated automatically based on such a CDL description of the component.

For the externalization of components' states during updates, a simple stream-based mechanism was employed. Each CBuilder was associated with its dedicated directory in the local file system. A CBuilder externalized the state of all "important" objects in the component into one file in this directory. For each subcomponent, a subdirectory was assigned, and the process was recursively repeated. Again, a way to automatize the externalization of a component state during updates could be found. One of the problems here is to identify the "persistent state" of the objects involved (the subset of attributes which is worth externalizing). A remedy here might be a property-like designation of these attributes, similar to CORBA Property Object Service [OMG95b].

For debugging purposes, we took advantage of clear Updater separation from the source of update messages. In the prototype implementation, the Updater was run on event-driven bases in a graphical interactive environment (similar to Java Beans [JBN97]).

6 Related work and future intentions

To begin with, Marimba corp. has released the Castanet System [MA96], which provides users with automatic software updating. After downloading the Castanet *tuner*, the user can tune a *channel*, which represents either a software or an electronic document package. The tuner uses the pull model based on an update schedule associated with the channel. To ensure update correctness, update changes should be made while the application is not running. On the contrary, the DCUP approach allows for updating at run time and both the push and pull models of updating activation are applicable. Moreover, in Marimba, each application is closely bound to the tuner and has to be launched via this tuner. In DCUP, an application is started in a classical way, only Updaters communicate with component providers and/or with the Update daemon if the push model is employed.

There is a group of works ([FS96], [GK96], [MDK94], [IB96], [BAB96]) which focus on a way to specify the configuration of distributed applications composed of components. Essentially, all of them use some kind of configuration language (also called MIL, Module Interconnection Languages), that allows interfaces of software components to be defined, and to specify the structure of an application in terms of interface interconnections. Even though some of them allow dynamic reconfiguration as in [FS96] and [GK96], in principle, these works deal with relations among components instead of modifying the internals of the components which is the case in DCUP. A special case of a dynamic reconfigurable component system is the Java Beans in which component interconnections are adjustable dynamically in an interactive graphical environment (application builder, e.g. Java BeanBox [JBN97]); moreover, component interconnections supported by Java Beans are limited to event-based connections.

In fact the most inspiring work for DCUP has been the mobile agent concept combined with persistence [ATS96, CPV97, GV97, KPT96] and particularly [LAR96, LCH96]. However, none of these works offers a dynamic component modification.

Our future intention will aim at a formal specification of a component and at providing developers with support for the externalization of the component state (as described in Section 5.5). As for communication between an Updater and a Provider, there is no standardized protocol at present. Such a protocol should provide a means for class downloading from the Provider to the Updater's local disk, for communication

between the Updater and Update deamon, for ensuring communication security, etc.

From the beginning, DCUP architecture has been devised in such a way that allows seamless porting to a distributed environment. So, one of the future steps will be porting DCUP to the CORBA environment. For that matter, we have identified two potential solutions to this problem (Figure 4). The first one regards the whole CORBA server process as one component which has its CORBA objects playing the role of component's reachable objects. The CManager, CBuilder, and (potentially) Updater abstractions are specialized CORBA objects running within the same server. This solution is based on employing delegation-based approach for associating an object implementation with a particular interface (e.g. TIE-approach in IONA's Orbix [ORBIXa, ORBIXb]), therefore the wrapper functionality can be implemented by the TIE objects (Figure 4a).
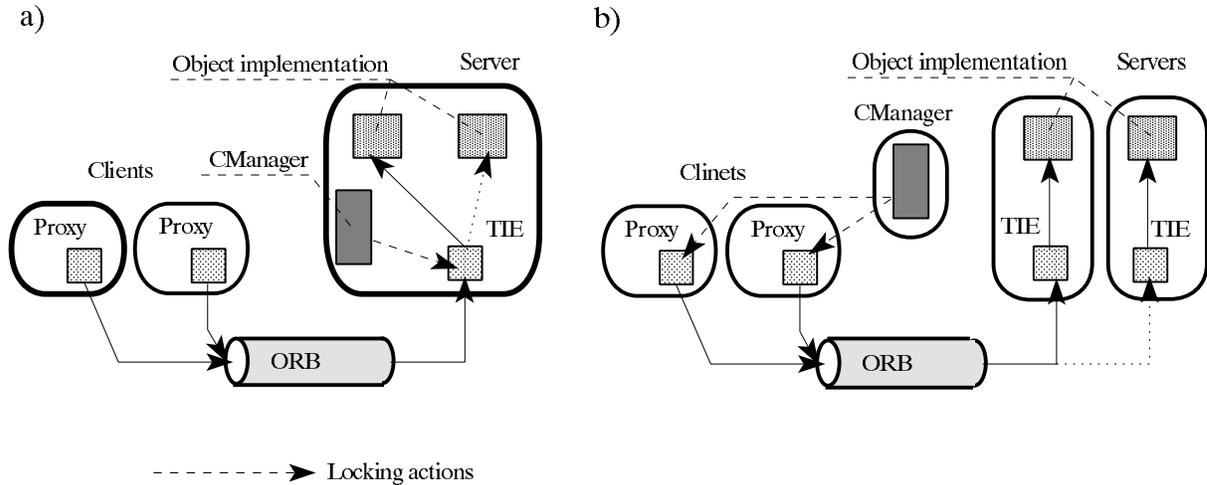

Figure 4 Porting DCUP to CORBA environment

The second solution is based on distributing of single component into several isolated CORBA processes. The replaceable part of the component composes one CORBA server process. In this case, the role of wrappers is played by client side proxies, which are controlled by the CManager captured in the standalone CORBA server process (Figure 4b).Another future intention is to investigate how to easily combine DCUP with an existing transactional service. Further, a DCUP component resembles to a JavaBeans in the version called Glasgow, described in [JBG97]. As the Glasgow supports nesting of components, the DCUP seems to be also easily portable to this environment. Note, however, that JavaBeans do not support dynamic component updating.

7 Conclusion

This paper presents a novel architecture, called *DCUP (Dynamic Component Updating)*. In this architecture, components are frameworks of objects. Component updating can be initiated by the original component provider and performed transparently to the user in an automated way.

With respect to the goals articulated in Section 1.2, DCUP achievements can be summarized as follows:
(a) Versioning and communication with a component provider is the role of the Updater abstraction. As for initialization of updating, two scenarios are supported - the push and pull models; in other words, the initiative for updating of a component can be originated either at the application or provider side.
(b) Transitions among a component's versions are coordinated by the cooperation of the CManager and CBuilder abstractions. In principle, CBuilder is associated with the life of a particular version of a component, while CManager persists over the whole lifecycle of the component.

(c) Transparency of a component's updating with respect to the rest of the application is achieved by limiting the updating effect only to a subtree of the tree-like hierarchy of the components. Thus, as the granularity of updating is a component, a component which at a lower-level in the component hierarchy can be updated (this includes recursively all its subcomponents) without effecting the remaining parts of the application. In a component, the use of its functional objects (reachable objects) from the outside of the component is mediated via automatically created wrappers (as results of binding operations). To support data integrity during updating, the wrappers can lock temporarily access to the component in such a way that the updating takes place only after the all "visitors" to the component have left it.

(d) Fitting both into the classical centralized and distributed environments. As DCUP wrappers resemble proxies, and as for accessing reachable objects name-based binding is used, it is very easy to map an distributed, e.g. CORBA resp. RMI - based [RMI96], application onto the DCUP abstractions. For example, a component can be a CORBA resp. RMI server; the corresponding CManager can assure the transition from an old to a new version of the server (Section 6)

With respect to updating, the DCUP architecture scales well thanks mainly to two reasons: First, granularity of updating being a component and components can be nested. Thus the part of the application which is to be a subject to an update is scalable. Second, the DCUP architecture can be easily applied in an distributed (CORBA/RMI) environment; thus the updating granularity scales well also in such a distributed environment.

Acknowledgements

References

[ATS96]   M. Mira da Silva, M. Atkinson: Combining Mobile Agents with Persistent Systems: Opportunities and Challenges, In Proceedings of 2nd ECOOP Workshop on Mobile Object Systems, Linz, July 1996

[BAB96]   L. Bellissard, S. B. Atallah, F. Boyer, M. Riveill: Distributed Application Configuration, In Proceedings of ICDCS '96, IEEE CS Press 1996

[CH93]   D. J. Chen, S. K. Huang: Interface for Reusable Software Components, In Journal of OO Programming Languages, January 1993

[CO97]   D. D. Corkill: Countdown to Success: Dynamic Objects, GBB, and RADARSAT-1, Communications of the ACM, May 1997 / Vol. 40, No. 5

[CPV97]   A. Carzaniga, G. Picco, G. Vigna: Designing Distributed Application with Mobile Code Paradigms, In Proceedings of 19th International Conference on Software Engineering, Boston, 1997

[DER96]   L. Deri: Droplets: Breaking Monolithic Applications Apart, IBM Research Division, Zurich, April 1996

[FS96]   H. Fossa, M. Sloman: Implementing Interactive Configuration Management for Distributed Systems. In Proceedings of 3rd International Conference on Configurable Distributed Systems (ICCDS), Annapolis, Maryland, May 1996

[GK96]   K. M. Goudarzi, J. Kramer: Maintaining Node Consistency in the Face of Dynamic Change, In Proceedings of 3rd International Conference on Configurable Distributed Systems (ICCDS), Annapolis, Maryland, May 1996

[GV97]   C. Ghezzi, G. Vigna: Mobile Code Paradigms and Technologies: A Case Study, In Proceedings of 1st International Workshop on Mobile Agents (MA), Berlin, April 1997

[IB96]   V. Issarny, Ch. Bidan: Aster: A Framework for Sound Customization of Distributed Runtime Systems, In proceedings of ICDCS '96, IEEE CS Press 1996

[JBG97]   JavaBeans "Glasgow" specification, http://splash.javasoft.com/beans/glasgow.html

[JBN97]   JavaBeans 1.0 Specification, http://splash.javasoft.com/beans/spec.html
[KON96]   KONA, http://kona.lotus.com
[KPT96]   J. Kleindienst, F. Plasil, P. Tuma: Lessons Learned from Implementing the CORBA Persistent Object Service, In Proceeding of OOPSLA '96, IEEE CS Press 1996
[KPT+96]   J. Kleindienst, F. Plasil, P. Tuma: What We Are Missing in the Persistent Object Service, Presented at the OOPSLA '96 Workshop on Objects in Large Distributed and Persistent Software Systems; available at http://nenya.ms.mff.cuni.cz
[LAR96]   D. Lange, Y. Aridor: Agent Transfer Protocol, White Paper, http://www.trl.ibm.co.jp/aglets
[LCH96]   D. Lange, D. Chang: Programming Mobile Agents in Java, White Paper, http://www.trl.ibm.co.jp/aglets
[MA96]   Marimba: The Castanet System, http://www.marimba.com
[MB96]   V. Marangozov, L. Bellisard: Component-Based Programming of Distributed Applications, Presented at 3rd CaberNet Radicals Workshop,
            http://www.twente.research.ec.org/cabernet/research/radicals/1996/papers/comp-marangozov.html
[MDK94]   J. Magee, N. Dulay, J. Kramer: Regis: A Constructive Development Environment for Distributed Programs, In Distributed Systems Engineering Journal, September 1994
[MR97]   M. Mira da Silva, A. Rodrigues da Silva: Insisting on Persistent Mobile Agent Systems, In Proceedings of 1st International Workshop on Mobile Agents, MA '97, Berlin, April 1997
[MTK97]   J. Magee, A. Tseng, J. Kramer: Composing Distributed Objects in CORBA. In Proceedings of the Third International Symposium on Autonomous Decentralized Systems (ISADS), Berlin, April 1997
[NT95]   O. Nierstrasz, D. Tsichritzis (eds.) : Object-Oriented Software Composition, Prentice Hall, 1995
[OH97]   R. Orfali, D. Harkey: Client/Server Programming with JAVA and CORBA, John Wiley & Sons, USA, 1997
[OMG95a]   Common Object Request Broker Architecture and Specification Revision 2.0, OMG 97-2-25, 1995
[OMG95b]   Object Property Service, OMG TC Document 95-6-1, June 1995
[OMG97a]   Persistent State Service, version 2.0, Request For Proposal 97-5-16, May 1997
[ORBIXa]   Orbix, Programmer's Guide, IONA Technologies Ltd. Dublin, 1994
[ORBIXb]   Orbix, Advanced Programmer's Guide, IONA Technologies Ltd. Dublin, 1994
[RMI96]   RMI - Remote Method Invocation, http://java.sun.com:80/products/jdk/1.1/docs/guide/rmi, 1996