The Architecture of a Dynamically Updatable, Component-based System

Work-in-progress Report, March 2002

Robert Pawel Bialek
Department of Computer Science
University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen, Denmark
bialek@diku.dk

Abstract

On-the-fly replacement of software may require simultanous distributed updates of components. If an update changes some interfaces or protocols, the update must be performed in a globally consistent manner.

This paper describes an architecture of a dynamically updatable component-based system that unifies research within architecture definitions, architecture configuration, and dynamic software update. It proposes a layered, modular design with meta-components, being responsible for performing the updates. The system is open for architectural and implementational extensions such as update constraints, or different update functions.

1. Introduction

Updating software components in a distributed environment, depending on the scope of the update, may have different impacts on the nodes of a distributed application. If an update only causes minor changes to the component implementation, no other components are affected by such an update. On the other hand, if a component changes its interface, connections, or a protocol it uses to communicate with other components; the update requires that all the components connected to the updated one need to be updated at the same time.

In this paper, we want to present the architecture of a system model that allows dynamic software updates in a distributed environment. The model uses abstractions to distribute responsibilities of the updates. The abstractions are represented in modules, which can make it easily expandable with architectural and implementational features.

2. Background

From the research areas addressing software updates and architecture reconfigurations, we were inspired to propose a model for a distributed updatable system that clearly defines responsibilities for distributed application's updates.

In the following, we will briefly present the motivating research areas and outline the ideas that had impact on our model.

2.1. Updatable Software Models

The development of software is a long task that stretches beyond the analysis, design, implementation and test phase. While executing the software, new requirements arise, bugs are observed and to meet the new requirements, software updates are needed. One way to ease the software update process is to have an open architecture, which makes it possible to introduce new functionalities or modifications as fast as possible. One of the major design models, for open architectures, is the Component Based Model (CBM) [3]. CBM has two important properties which make it scalable and expandable: First, components are implementations and architectural abstractions at the same time, which makes the architectural changes easier. Second, extensions are made (almost) independently of other components because interaction between components is well-defined in component interfaces [1].

Because modifications to a code often affect a single part of application, CBM with its encapsuled design and isolation is a flexible model that can handle updates. Furthermore, CBM is one of the mostly used design models now a days with several implementations , e.g. EJB, CORBA, DCOM.

2.2. Dynamic Architectures

Components can be placed on different computers, easily transported (e.g. using serialization) and connected with each other in a variety of ways. To describe (inter)relations between components in a distributed system, we can use Architecture Definition Languages (ADLs). Among many ADLs, Olan [2] is a model that encapsules legacy software in components, abstracts the middleware communication into *connectors*, and allows to specify the geographical placement of components.

Nowadays ADLs are used to describe distributed applications. ADLs are describing fixed configuration, and are used to ease deployment of such applications.

Another approach, addresses the issues of dynamism in the ADLs. In [6] three levels of dynamism are introduced in ADLs: 1. Interactive dynamism (allows changing data only in fixed connections between fixed components), 2. Structural dynamism (allows adding/removing components and connections to a system), 3. Architectural dynamism (allows changing whole configurations of components).

A model supporting dynamism in ADLs is represented by MARMOL (Meta ARchitectural MOdeL) [6]. MARMOL uses a language called Pilar, which allows reifying (creating links

with meta-components) base components (called avatars) by their meta-components. Then, base components can take actions (reflect) on the basis of the meta-component behavior. All components can interact with other components on the same level, reify some avatars or be themselves reified by components on a meta-level.

Updates can be seen as dynamic operations from the architectural point of view. One update may affect the interactive dynamism (change communication methods). Update may as well add new components - Structural dynamism, or the update may address the whole architecture of a distributed application - Architectural dynamism.

Because update may affect one or more levels of the distributed applications, there must be a way to control the changes, so they do not break the distributed applications consistency.

2.3. Dynamic Software Updates

Besides the research within Dynamic Architectures, several researches address the problem of updating the software while it is running.

Generally, there are the following update methods:

Dynamic linking Upon updates of some object implementations, the new object versions can be deployed by exchanging all object references from old to new object versions. The old object can then be removed. The new versions of objects do not have to be present on the same node ([9] introduces an algorithm for linking across different spaces without compromising security). Gilgul [5], on the other hand, is an extension to Java, where all pointers to an object can be updated by a single substitution operation, which speeds up the update process.

Dynamic linking allows replacing objects dynamically and does not require any functionality from the objects. This method is mainly preferred with object that do not have persistent state.

Static State transfer uses persistent state and is used for objects that preserve the state after the update.

State transfer has the following steps: a program state is saved, program is stopped, the new version replaces the old one, the state from the previous version is rolled to the new one.

IF the program can not be stopped (after a state is saved), e.g., because it is a part of critical service, the next method can be used.

Dynamic State transfer Gupta [7] introduces a state transfer function that transports a state of an executing program to its new version. The State Transfer function can be generated (almost) automatically. Hicks [8] describes a system that atomically generates such functions on the basis of two different implementations.

The state transfer function is used to update a running part of code. Both Gupta and Hicks argue that state transfer can not move a running program from any particular place in the program flow to an arbitrary other one. The programmer must be involved and inform where the state transfer should appear.

None of these methods was related to the research within Dynamic Architectures. In section 3, we will present how these methods are used in our Updatable Distributed System.

3. The Updatable Distributed System

Our updatable, distributed system is inspired by MARMOL [6] and Olan [2]. Olan encapsules legacy software and communication in classes. This gives a freedom to control program flows during updates. MARMOL, on the other hand, introduces a multi-abstractional, layered architecture, where components have well defined responsibilities on each layer.

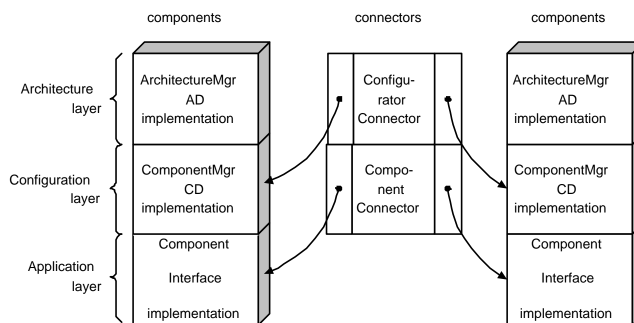

Figure 1. Three-layer model

The model consists of components and connectors. Components provide interface and implementation, connectors represent higher level abstractions encapsulating communications between components.

In our model, similarly to [12], we split the system into three layers. Each layer includes components and connectors.

components are the single entities having their implementations, and interfaces.

One component may have many objects in it. The interfaces define which objects are handling the communication with other components.

All the information about the component configuration is saved in the *component descriptor*.[1]

connectors are entities that encapsule the communication between components. It means that components do not interact directly but through the connectors that are used to save communication channel's states during updates. They are placed one layer above the components that they connect.

Connectors can be treated as components that have a well-defined function. Components on the same level can interact with each other. So, a component can signal to a

[1]This solution is similar to the EJB model, where each elements has its Home/Remote interface, and the connection between the implementation and interfaces is defined in the Bean descriptors.

connector to take some actions upon an update, e.g., to store the communication during an update of some components under it.

As we see on figure 1, the three layers in our model are:

Application layer includes components' implementation and data.

On this level we can add, remove, or update new objects and data.

Configuration layer includes Connectors (connections between components), ComponentMgr that manages the application layer and the component descriptor (CD).

On this level, we can add update-constraints, version control mechanisms and different object loading methods.

Architecture layer includes architecture description (AD), i.e., data describing components and their (inter)connections. And the ArchitectureMgr that manages the Configuration layer. The communication between ComponentMgrs is encapsuled in connectors.

This layer can be expanded with reconfiguration constraints.

3.1. Update request

Update request, sent either by a user or an ArchitectureMgr, initiates an update.

Figure 2 shows an example of an update request that includes many kinds of updates. Generally an update descriptor includes:

- *update type* - whether it is an addition, removal, or update of a new component,
- *list of updated objects* - IDs for the object classes that need to be updated.
- *new versions of the objects* - the implementations of the new object versions,
- *update method* that should be used to update each object: replace/dynamic
- *update function* which is a state transfer function that is used to update an executing object process.
- *update constrains* specifying order of (sub)updates and relations between them, e.g., Object A can be updated if version of $B >= 1.1$

```
<update_descriptor>
  <add_obj to="server01//comp1">
    <object name="C">
      <implementation>...</>
    </object>
  </add>
  <remove_obj from="server01//comp1">
    <object name="D">
    </object>
  </remove>
  <update_obj in="server01//comp1">
    <object name="A" method="replace">
      <old_version>1.0</>
      <new_version>1.1</>
      <implementation>...</>
    </object>
  </update>
  <update_obj in="server01//comp1">
    <object name="B" method="dynamic">
      <old_version>1.1</>
      <new_version>1.2</>
      <update_function>...</>
      <implementation>...</>
    </object>
  </update>
</update_descriptor>
```

Figure 2. Example of an update descriptor

The update request includes an addition, removal and two kinds of updates: update by object replacement and dynamic update which involves execution of update function. The new object implementations, as well as implementation of the update function, are included in the update request.

3.2. Performing the update

To perform the update, we need to use one of the methods presented in 2.3. Generally, updates require to go through several steps:

1. Identify the scope of the update.

   From the update descriptor, we can see if an update only affects components being under the entity responsible for the update and is a local update; or if it requires other elements to be updated and is a global update. Depending on the result, we either run the update locally or send an update request to the higher abstraction layer.

2. Prepare the application for the update.

   The application waits until the affected components are ready for the update. If it is application layer, ComponentMgr waits for the participating processes (or threads) to signal that they are in a state that is safe to perform the updates. On the configuration layer, we wait for connectors to block the communication, and for other ComponentMgrs to finish their updates. On the Architecture level, we just start the update because we only have one ArchitectureMgr. The ArchitectureMgr can queue many update requests and perform them serially.

3. Perform the update.

   Execute the update function and inform the update initiator about its result. The update function is using dynamic update techniques, and it coordinates the connectors.

4. Reflect changes by saving all the modifications to the component in the component description. The changes

made to the application configuration and components need to be reflected in the Architecture Description and Component Description, respectively.

3.3. Maintaining application consistency

Seen from the reconfiguration point of view, updates may break application's consistency on a local or global level. Solutions to maintain local consistency are present in the area of *process migration*, which generally include checkpointing, and cloning. To handle global consistency, apart from maintaining local consistency, we should maintain all the intercommunication between the components. Stuurman et.al. [11] use dedicated communication channels to do so. Another solution is presented by De Palma [10], who introduces an extended transaction model that builds on an isolation property. If an update process does not interfere with normal computational process then the update can proceed. Otherwise, it should be rolled back, which may lead to reconfiguration (update) starvation.

In our model, we propose encapsulating all the communication in Connectors. Connectors store the data during inavailability of processes being updated. Connectors only ensure applications consistency during local updates, i.e. updates that do not change the connectors interface.

Global updates, on the other hand, require involvement of ArchitectureMgr that will control execution of the whole distributed application and, if necessary, temporarily block the involved components. Such updates require that requests made to the old component (component from before the update) are serviced before the interface is changed, forcing:

1. The interacting components to stop sending new requests.
2. All the requests queued in the connectors are executed before update starts.

4. Examples

In the following, we will show two kinds of updates that present how the architecture is used to control the update process. We will first show a local update, where ComponentMgr isolates the update of its object by blocking the connector. Later, we will show a global update that involves ArchitectureMgr in administrating the connected components.

4.1. Local update

On figure 3, we see that update of an interface object is performed in the following steps (step numbers correspond to the arrow numbers in the figure and indicate the order of operations during an update)

1. An update request is sent to the ComponentMgr1, which evaluates the update type and concludes that it is a local update of the interface object.
2. The ComponentMgr1 blocks the communication to the Component1 by sending a signal to the Connector.

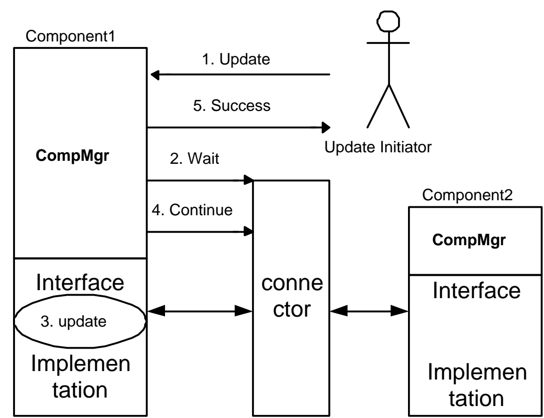

Figure 3. Local update

Nubers indicate the order of messages masspping between the components.

3. ComponentMgr1 starts the update. The update uses the update method provided in the update request.
4. After the update, the component descriptor in Component1 is updated, and the connection is restored by sending a message to the Connector
5. Finally, the update result is returned to the update initiator

We can see that Component2 is unaffected during the update of Component1.

4.2. Global Update

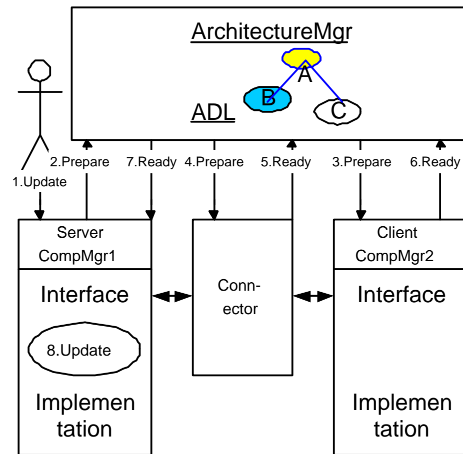

Figure 4. Global update

Numbers indicate the order of messages passing between components.

Figure 4 shows steps during a global update:

1. ComponentMgr1 receives an update request and concludes that it is a global update
2. ComponentMgr1 asks ArchitectureMgr to prepare the update

3. ArchitectureMgr, on the basis of the architecture description ADL, prepares all the involved components, and
4. Connectors.
5. After the preparation request, connector confirms that it is ready to perform the update, and
6. Component2 confirm that it is ready
7. Then, the ArchitectureMgr informs the initiating ComponentMgr1 that all involved components and connectors are ready.
8. ComponentMgr1 performs the update
9. Finally ComponentMgr1 informs the ArchitectureMgr and *update initiator* about the update's result.
10. ArchitectureMgr informs the components that the update is finished and that they can continue with their normal execution.
11. Update initioator is informed about the update result

5. Discussion

Component in our model communicate with each other. In this paper, however, we have not discussed the concepts of communication between components neither on the same nor different abstraction layers. We suggest though to use asynchronous communication, which supports flexibility and autonomy of the components. The security issues in the system can be solved using authentication and cryptography techniques.

Updates of component interfaces are very challenging. They may force the whole distributed application to freeze during the update process, which may not be acceptable. Furthermore, interface updates change the application semantics, which may introduce consistency problems between different versions of components. This raises a fundamental question: whether changing application semantics at runtime is possible? We believe that global updates are possible and practical in cases where programmer updates both ends of the updated interface, e.g. pairwise update of client and server component. We have not analyzed the possible application areas, or semantical constrains, in which interface updates are possible. We believe that our model can be expanded with advanced control and update management tools, which can handle on-line semantical changes.

Granularity of updates is an other topic that has not been discussed. One update request may include many sub-updates, which may be related to each other. This introduces the need for specifying relationships between sub-updates (For example, updates of objA may happen after objB has been added), and range of one atomic update (whether an sub-update may be performed alone or together with other sub-updates). The update constrains have not been analyzed in this paper but because neither ComponentMgr nor ArchitectureMgr have been restricted in the functionality, they can be expanded with virtually any kind of update rules and constrains. The only restriction is that ComponentMgr and ArchitectureMgr are responsible for local and global updates, respectively.

6. Conclusion

The contribution of this paper is a proposal for an updatable system that unifies research within architecture definitions, architecture configuration, and dynamic software update areas in one modular system. The responsibilities of every module are well defined. The system is open for improvements in every layer, so additions in areas from architecture reconfiguration to pointer update techniques can be easily implemented. Updates can be initiated at any layer and depending on the updates' scope, they will be propagated throughout the system.

We plan to address the issues presented in the previous section in the nearest future. We will especialy focus on analysis of update protocol that supports updating interface objects.

7. Status

This paper presents the ideas underlying the author's Ph.D. project concerning remote and dynamic update of component based distributed systems. We will start building a prototype late summer 2002.

References

[1] F. Bachman. Technical concepts of component-based software engineering, May 2000.

[2] R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J. Vion-Dury. Architecturing and configuring distributed applications with olan, 1998.

[3] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Analyzing software architectures for modifiability, 2000.

[4] J. E. Cook and J. A. Dage. Highly reliable upgrading of components. In *International Conference on Software Engineering*, pages 203–212, 1999.

[5] P. Costanza. Transmigration of object identity: The programming language gilgul. *http://citeseer.nj.nec.com/483031.html*.

[6] C. E. Cuesta, P. de la Fuente, and M. Barrio-Solorzano. Dynamic coordination architectur through the use of reflection. *SAC 2001, Las Vegas, NV*, ACM 1-58113-287-5/01/02:134–140, 2001.

[7] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, February 1996.

[8] M. W. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23, 2001.

[9] J. Kempf and P. B. Kessler. Cross-address space dynamic linking. *SMLI TR-92-2*, 1992.

[10] N. D. Palma, P. Laumay, and L. Bellissard. Ensuring dynamic reconfiguration consistency. 1999.

[11] S. Stuurman and J. van Katwijk. On-line change mechanisms the software architectural level. *SIGSOFT 98,ACM l-58113-108-9/98/0010...*, 1998.

[12] M. Wermelinger and A. Lopes. A graph based architectural (re)configuration language.