Dynamically Updatable Component-based System (DUCS)

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

ABSTRACT
We present our framework called DUCS. DUCS supports on-the-fly updates of distributed component-based applications. DUCS has a layered architecture running on top of a virtual machine extended with a support for object unload and replacement. DUCS is a meta-level framework, so the impact on the application’s implementation is minimal. It supports dynamic component replacement, state transfer among components, interface modifications, and automatic propagation of updates. The framework is expandable and can be applied to many programming languages as well as many hardware platforms.

Keywords
dynamic updates, components, reflection, state-transfer, interface adaptation

1. INTRODUCTION & MOTIVATION
The popularity of component based frameworks (e.g., EJB) as well as standardisation of communication protocols (e.g., SOAP) support production of distributed component based systems connected by the Internet. Updating such distributed applications is very challenging. First, the maintainers may not always be able to stop the application (e.g., financial services require continuous uptime). Second, some components of the applications are often used on remote nodes that connect spontaneously, may have limited network bandwidth or hardware and are therefore difficult to access and maintain when needed.

This Ph.D. project is about how we can build a distributed component-based system that can be updated dynamically. We present a general structure of our Dynamically Updatable Component-based System (DUCS). To demonstrate the viability of our ideas, we use DUCS to build a distributed group-ware application that can be extended with new functionality on-the-fly.

2. ASSUMPTIONS
We design the framework for distributed, component-based applications that are executed in virtual machines (we use Java VM). Nodes (for example clients) connect spontaneously to other nodes (for example servers). The total number of nodes does not need to be determined (e.g., client-server applications with big number of clients). Such application’s components can be executed in heterogeneous environments (e.g., server on a main-frame system, clients on PDAs) and some of the application’s components must remain operational continuously (e.g., server’s web services).

3. GOALS
The goal of this project is to provide a model for a framework facilitating dynamic updates of component based applications and build a prototype that proves the viability of the framework. The framework shall have the following properties:

- It shall facilitate update process of distributed applications by supporting dynamic object replacement, state transfer, and reflection,
- It shall support automatic propagation of updates,
- It shall not complicate the process of building the applications using it,
- It shall be able to be applied to the existing application with little or no impact on them,
- It shall be expandable and modifiable.

The goal of the prototype is to identify all (or at least major) requirements for the framework, present how to use the framework, and to evaluate its impact on the applications using it (with respect to performance, memory usage, etc).

4. FRAMEWORK DETAILS
An update is a replacement of one component by another where a semantical relationship between these two components can be defined (In our prototype we use the functional relationship.) A dynamic update is an update which is performed on executing components. A correct update process is a set of dynamic updates to a running systems, which transfer a running set of components to a state that would be achieved if the system was shut down and later restarted after the replacements of the intended components.

DUCS supports correct update process and guarantees that the updates (not all modifications) are performed correctly. A set of update layers supervises the update process, a set pluggable interface adapters handle potential version mismatches and update requests propagate the updates in the system.
4.1 Layers
Similarly to our previous work[1], DUCS separates application logic from the update logic. DUCS is a layered framework with the following layers (presented in the bottom-up order):

- **Application layer** is the layer, where applications consisting of components are built. The communication between the components goes through their interfaces, which are controlled by their meta-level component, the Configuration Manager.

- **Configuration Manager (ConfMgr) layer** is the meta-level entity responsible for performing updates on single nodes. The ConfMgr handles object updates and manages interface changes by supporting pluggable adapters.

- **Architecture Manager (ArchMgr) layer** is responsible for managing the configuration of components across many nodes. It manages addition, removal, movement and updates of groups of components. ArchMgr is only contacted by the ConfMgr when interface adapters (described later) can not handle a version mismatch.

In the current work, we have not addressed issues in this layer, but we expect it to be the necessary element of a general DUCS hence it will be analyzed in the future.

4.2 Update request
*Update requests* are used to initiate the update process on the Configuration Manager layer. Update requests include new implementations, state transfer functions, interface adapters, and (optionally) update constraints. Update requests are used to propagate the updates across the distributed application.

4.3 Interface adaptation
Because we deal with heterogeneous networks with spontaneously connecting nodes, updating applications in one atomic step is not possible. Additionally, because of hardware and bandwidth limitations, we may not always be able to perform the necessary updates. Therefore, we need to deal with simultaneous cooperation of components with different versions.

To allow updates of components’ interfaces, we introduce *Interface adapters*. Interface adapters are Configuration Manager layer’s functions that translate calls between providers and requires interfaces across different component versions. The interface adapters are pluggable during updates.

5. IMPLEMENTATION DETAILS
The application layer communicates with the ConfMgr layer using reflection on every method call. To support reflection, we need some additional functionality from the VM. This extra functionality includes reflection, unloading, and versioning of objects and is placed in the meta-VM.

We implement reflection using wrapper components based on Javassist[2] framework, which allows run-time creation of wrapper-classes (components). The object replacement will be implemented by changing object references (similar to the Gilgul approach[3]) extended with the state transfer function and the pluggable interface adapters.

6. STATUS
This Ph.D. project runs from 1st June 2003 to 31st of May 2006. Now we are in the process of building the basic elements of the framework for future prototyping. Currently, we are programming the meta-VM and incorporating reflection to ConfMgr into the component model. Next, we will expand the ConfMgr with the state transfer functions and propagation of updates using update requests.

7. CONTRIBUTIONS
The poster provides a brief description of a framework called DUCS that supports building and adapting existing applications to the updatable framework. DUCS unifies research within the area of dynamic updates: state transfer function[5], dynamic object replacement[3], use of reflection[4], integrated in a layered framework. DUCS supports building updatable distributed systems with unknown number of components using interface adapters.

In the prototype, we are extending real worlds products (Java virtual machine) with the support for component updates and reflection. The update logic is separated from the application logic so the separation of the application code gives the freedom for optimizing the update functions while keeping the application code (almost) intact. This fulfills all the goals outlined in section 3.

8. REFERENCES