1
Introdução

Praticamente toda aplicação, todo software desenvolvido, precisa passar por alguma correção ou evolução durante o seu período de atividade. F. Brooks [1] afirma:

> "Mais de 90% do custo de um sistema típico ocorre em sua fase de manutenção, e que qualquer parte de software de sucesso inevitavelmente sofrerá alguma manutenção."

Juntamente com a necessidade de manter um software, nasce em alguns sistemas a necessidade de uma evolução independente (atualização parcial), ou seja, capacidade de atualizar partes do sistema sem a necessidade de refletir a atualização em outras partes do sistema. Como exemplo de resposta a essa necessidade, podemos citar o estilo de arquitetura REST [2] e o protocolo HTTP [3], que permitem que servidores sejam atualizados sem obrigar a atualização dos clientes. Quando consideramos a possibilidade de existirem diferentes versões em um sistema, torna-se inevitável e necessária a realização de algum controle ou manipulação das versões dentro do mesmo.

Uma das principais vantagens comumente atribuídas à abordagem de desenvolvimento baseado em componente de software é o melhor suporte à extensibilidade independente (*independent extensibility*) [4, 5], onde o termo componente de software possui o sentido de unidade de implantação. Szyperski [6] apresenta uma definição de componente de software que é amplamente aceita:

> "Um componente de software é uma *unidade de composição* com, ao menos, *interfaces* contratualmente especificadas e *dependências de contexto* explícitas. Um componente de software pode ser implantado independentemente e está sujeito a composição por terceiros."

Usualmente, essas *interfaces* contratualmente especificadas são denominadas de *facetas*, e as *dependências de contexto* explícitas são denominadas de *receptáculos*. Entraremos em maiores detalhes sobre esses conceitos e terminologia na seção 2.1.

Em sistemas distribuídos é comum a utilização de plataformas de *middleware* para oferecer uma abstração que auxilie o processo de desenvolvimento. No nicho de plataformas de *middleware* para sistemas distribuídos que ofereçam um modelo de componentes, podemos encontrar algumas soluções já propostas pela academia e pela indústria, tais como FRACTAL [7], CCM [8], COM [9], OPENCOM [10], LUACCM [11, 12] e SCS [13]. Porém, com base na nossa experiência, não há um suporte adequado à coexistência de múltiplas versões em nenhuma dessas soluções. Quando falamos de suporte a múltiplas versões, referimo-nos à capacidade de gerir mais de uma versão de uma mesma interface. O código 1.1 ilustra um exemplo de diferentes versões de uma interface IDL *Hello*. Não existe nenhuma regra de definição ou formação de diferentes versões de uma mesma interface. Quem define que as interfaces são relacionadas é a semântica das mesmas, o que elas representam.

Código 1.1: Especificação IDL das versões da interface *Hello*.

```
1  /* Versão 1.0 */
2  module helloworld {
3    interface Hello {
4      void sayHello();
5    };
6  };
7  /* Versão 2.0 */
8  module helloworld {
9    interface Hello {
10     void sayHello();
11     void sayGoodbye();
12   };
13 };
14 /* Versão 3.0 */
15 module helloworld {
16   interface Hello {
17     void say(in string text);
18   };
19 };
```

Utilizando os modelos de componentes tradicionais, existem duas possíveis alternativas de projeto para permitir múltiplas versões de uma mesma interface: adicionar novas facetas, ou criar novos componentes para cada versão da interface que se deseja dar suporte. É importante frisar que para prover múltiplas versões através de uma dessas possíveis abordagens não basta simplesmente criar múltiplos componentes ou facetas. Analisando pela perspectiva da modelagem e organização do modelo e do seu uso, essas soluções levam à perda da identidade da abstração de componentes, e é contrária à diretiva de modularização do sistema.

Os princípios que estamos adotando para analisar a modularização são baseados nos conceitos de compreensibilidade, continuidade e ocultação de informação, apresentados por B. Meyer [14]. *Compreensibilidade* está ligada à capacidade de um leitor humano conseguir compreender um módulo sem

precisar conhecer outro, ou por precisar conhecer poucos outros. O conceito de *continuidade* é satisfeito se, dada uma necessidade de modificação em um problema de especificação, requer a modificação de apenas um ou poucos módulos. E *ocultação de informação* é uma regra de desenvolvimento baseada no critério de continuidade, onde se deseja selecionar o sub-conjunto de informações do módulo que devem ser públicas, pois são necessárias para o bom entendimento e uso do módulo, e manter as demais informações como privadas. Entraremos em maiores detalhes sobre os principais requisitos para prover suporte a múltiplas versões na seção 3.1.

O modelo de framework proposto por Ajmani [15, 16], cujo protótipo se chama UPSTART, oferece uma solução para sistemas distribuídos que permite atualizações dinâmicas [17, 18] e provê suporte à coexistência de diferentes versões de uma mesma interface, tratando a comunicação entre elas. Ele foi pensado exatamente para prover suporte à evolução dos sistemas distribuídos e gerenciar as múltiplas versões em uso no sistema, porém não é focado em modelos de componentes.

O objetivo do nosso trabalho é definir um modelo de componentes e desenvolver um *middleware* que ofereçam um suporte adequado à coexistência de múltiplas versões de interfaces de serviços em uma mesma instalação de sistema, tendo o framework UPSTART como referência. Para isso, estenderemos um modelo tradicional de componentes com novas abstrações para dar suporte a múltiplas versões de interfaces, e adaptaremos o sistema de componentes SCS para dar suporte ao modelo proposto. O novo sistema de componentes resultante se chamará SCS-MV. Com base no histórico de evolução de diferentes componentes SCS utilizados em uma aplicação real, realizaremos experimentos com a versão proposta do SCS para avaliar a eficácia do novo modelo e de sua implementação através do SCS-MV.

1.1 Estrutura do Documento

A estrutura deste documento é tal que no capítulo 2 apresentaremos alguns trabalhos relacionados à área de modelos de componentes e de suporte a múltiplas versões. No capítulo 3, apresentaremos o modelo proposto para solucionar a deficiência no suporte a múltiplas versões em modelos de componentes. No capítulo 4 apresentaremos o experimento realizado para avaliar o modelo proposto. Por fim, no capítulo 5, apresentaremos as considerações finais e os trabalhos futuros.