# 2
# Trabalhos Relacionados

Neste capítulo falaremos sobre os trabalhos relacionados. Na seção 2.1 abordaremos alguns sistemas de componentes existentes, aprofundando um pouco mais na descrição do sistema SCS. Posteriormente, nas seções 2.2 e 2.3, falaremos sobre os trabalhos relacionados encontrados com suporte a múltiplas versões de componentes em sistemas distribuídos: o ICE e o UPSTART.

## 2.1
## Modelos de Componentes

As tecnologias de componentes de software, tais como FRACTAL [7], CCM [8], COM [9], OPENCOM [10], LUACCM [11, 12] e SCS [13], são semelhantes em suas abstrações de modelo, onde existe o conceito de uma unidade que oferece um serviço através de uma determinada interface (*facetas*), e uma abstração para realizar a conexão de componentes através das facetas oferecidas. Essa abstração pode ser a existência de pontos de entrada de conexão (unidades que requerem uma determinada interface - *receptáculos*), ou a existência de uma estrutura que representa a conexão (e.g. canal).

O LUACCM e o SCS possuem uma abstração semelhante ao CCM, onde existe o conceito de facetas e receptáculos, ilustrado na figura 2.1. O COM e o OPENCOM seguem também essa mesma abstração de facetas e receptáculos. Por sua vez, o FRACTAL difere um pouco deste modelo de componente. Ele define uma membrana que engloba todas as facetas (o que define o componente), mas não propõe o conceito de receptáculos, e utiliza um canal para representar a conexão de facetas.

Os modelos de componentes tipicamente disponibilizam mecanismos de manipulação, conexão e introspecção, por onde é possível acessar e conectar os componentes, além de obter descrições, em tempo de execução, das facetas disponibilizadas e das conexões realizadas.

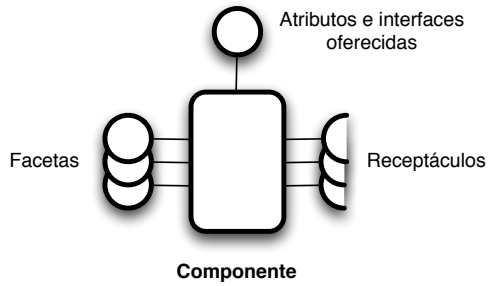


Figura 2.1: Modelo de Componente do CCM

### 2.1.1 Modelo de Componentes SCS

Esta seção não tem por objetivo descrever o modelo de componentes SCS detalhadamente, mas apresentar os conceitos principais do modelo, necessários para o entendimento dos capítulos posteriores. Maiores detalhes podem ser encontrados na dissertação de Augusto [19] e em outros documentos, disponíveis em [13].

Uma informação importante sobre o SCS é que ele possui como um requisito a necessidade de comunicação dos componentes entre diferentes linguagens, como Lua [20], Java [21], C++ [22] e C# [23]. Com essa finalidade, existem implementações do modelo nessas diferentes linguagens, mas, por questões de simplicidade, neste trabalho nós vamos nos ater apenas à implementação Lua do SCS.

#### Componentes

Um componente SCS é caracterizado, primariamente, por suas interfaces providas e requeridas, que poderão ser descobertas em tempo de execução. Interfaces providas são denominadas facetas, e interfaces requeridas são denominadas receptáculos. Um componente SCS poderá disponibilizar diversas facetas, e ter zero ou mais receptáculos, e as facetas e os receptáculo são identificados por nomes únicos dentro do componente. Receptáculos podem ser de dois tipos: um simples, que aceitará receber apenas um objeto implementador da interface requerida, e um múltiplo, que aceitará mais de um. A figura 2.2 apresenta um exemplo de componente SCS.

#### Facetas

Facetas são funcionalidades providas por um componente, na forma de métodos e atributos definidos em uma interface. Cada faceta tem uma interface bem definida, e um componente pode oferecer mais de uma faceta de uma

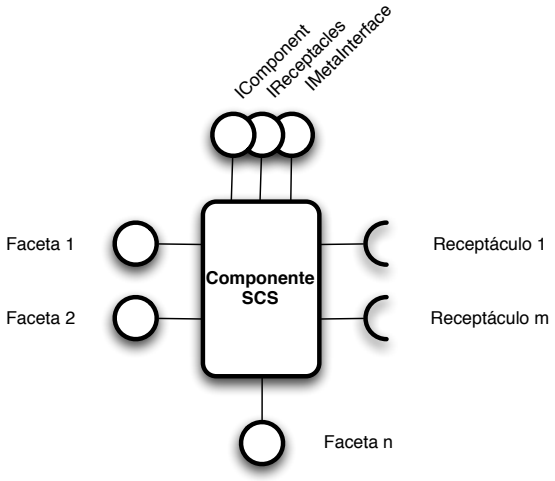


Figura 2.2: Exemplo de Componentes SCS

mesma interface. Componentes SCS utilizam CORBA [24] como tecnologia de suporte a objetos distribuídos em múltiplas linguagens, tendo assim suas interfaces definidas através de definições IDL, segundo o padrão definido pelo Object Management Group.

Todo componente SCS permite a obtenção de suas facetas, através de uma interface obrigatória chamada IComponent. Desta forma, temos uma interface única que serve como representação de um componente.

### Receptáculos

No modelo SCS, componentes especificam suas dependências externas através de estruturas chamadas receptáculos. Para definir uma dependência, basta criar um receptáculo e neste especificar a interface necessária. Assim, esta dependência será resolvida logo que uma faceta implementadora desta mesma interface seja conectada ao receptáculo.

O modelo almeja facilitar a conexão entre componentes, e assim fornece outra faceta padrão, IReceptacles, que provê meios de conectar aos receptáculos componentes que implementem as interfaces especificadas. A interface IReceptacles é obrigatória. Receptáculos podem permitir a conexão de apenas uma faceta implementadora de sua interface desejada (receptáculo simples), ou várias (receptáculo múltiplo).

### Conexões

Conexões entre componentes que seguem o modelo SCS são bastante simples: basta conectar uma faceta a um receptáculo, através do método

apropriado para isto, que se chama *connect*. O único detalhe, já mencionado, é o fato de receptáculos poderem ser simples ou múltiplos. A possibilidade de se conectar e desconectar facetas a receptáculos efetivamente permite a (re)configuração dinâmica das dependências externas do componente.

A Figura 2.3 mostra componentes conectados de diferentes formas. O *Receptáculo 1* é um receptáculo múltiplo, e as facetas *Faceta 1* dos componentes *B*, *C* e *D* estão conectadas nele. Já o *Receptáculo 2* é um receptáculo simples e só possui a *Faceta 2* do componente *D* conectada nele.

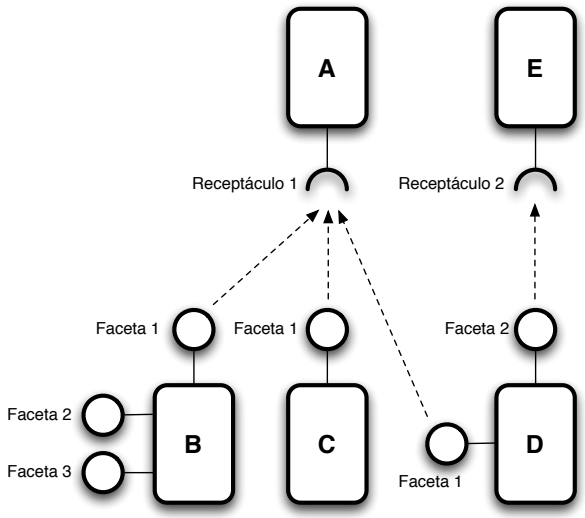


Figura 2.3: Componentes SCS conectados

### Introspecção

O modelo fornece suporte nativo à introspecção. Através de uma faceta chamada IMetaInterface, pode-se descobrir remotamente quais as facetas e receptáculos estão definidos em um componente. Esta faceta é obrigatória e conta com uma implementação padrão. Seus métodos permitem a obtenção de todo o conjunto de descrições de facetas e receptáculos.

## 2.2 Ice

O Ice [25, 26] é uma plataforma de *middleware* para objetos distribuídos, que permite o desenvolvimento de aplicações cliente-servidor, e é fortemente inspirado em CORBA [27]. Diferentemente de CORBA, o Ice disponibiliza um mecanismo para dar suporte a múltiplas versões de interfaces de objetos.

Ice permite que um mesmo objeto Ice seja composto por uma coleção de sub-objetos (nomeados de *facetas*), onde os sub-objetos possuem nomes e

podem ser de interfaces distintas (Figura 2.4). Com isso, é possível adicionar vários sub-objetos ao mesmo objeto Ice, onde cada sub-objeto representaria uma versão diferente. É possível definir uma faceta padrão para o objeto Ice. Para isso, basta definir o nome vazio para a faceta.

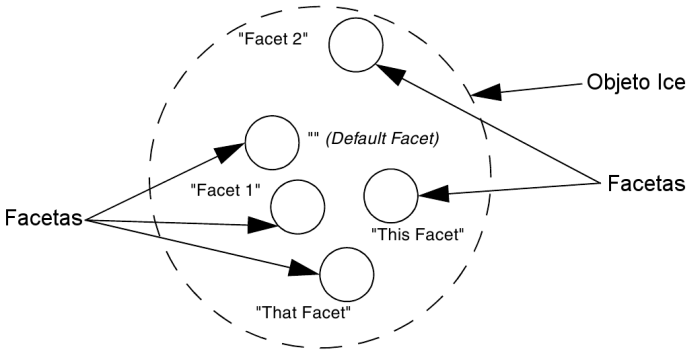


Figura 2.4: Objeto Ice

Um dos propósitos das facetas é permitir a criação de novas versões de aplicações de uma maneira mais limpa, sem comprometer a compatibilidade com versões antigas já implantadas no sistema. Mais importante ainda, a decisão de qual faceta utilizar é feita em tempo de execução ao invés de ser em tempo de compilação, implementando uma espécie de ligação tardia, tornando o acoplamento mais fraco.

Entretanto, apenas com as *facetas*, não é possível resolver o problema de versionamento. Ainda cabe ao desenvolvedor decidir como versionar os sub-objetos, para que o sistema permaneça legível e mantenha as consistências semânticas. Na documentação do Ice [26] são realizadas algumas considerações enfatizando que o uso das *facetas* é o primeiro passo para prover uma solução para o suporte a múltiplas versões, e que não é uma solução por si só.

## 2.3 Upstart

O Upstart [15, 16] é um protótipo de um modelo de atualização de software automática para sistemas distribuídos. Nele, o sistema é modelado como uma coleção de nós, que se comunicam através de chamadas remotas de métodos.

O Upstart assume que os sistemas alvos da implantação do seu modelo precisam necessariamente ser robustos. Com isso, o modelo assume que o sistema precisa saber lidar com falhas de comunicação, falhas arbitrárias em nós, e que os nós devem ser capazes de se recuperar após falhas, restaurando seus estados e reingressando ao sistema.

Nesta seção abordaremos algumas características importantes desse trabalho:

- Objetos com estado persistente;
- Objetos de simulação e seu encadeamento;
- Função de transformação.

### 2.3.1 Estado Persistente

Uma característica muito marcante no Upstart é a necessidade dos objetos possuírem um estado persistente. Com isso, quando um nó que havia falhado ou terminado intencionalmente é recuperado, o nó recarrega a parte do estado persistido.

Essa característica é explorada pelo mecanismo de atualização dinâmica do sistema. Existem várias estratégias e cuidados necessários para se realizar atualizações dinâmicas [28, 29], mas a opção de utilizar o estado persistente no Upstart foge de alguns desses problemas, como a identificação do momento para realizar a atualização. Quando a atualização está para acontecer, o sistema finaliza o nó, realiza as atualizações necessárias na representação do estado persistente do nó, e reinicia o mesmo já atualizado. Ele tira proveito do fato dos nós possuírem o estado persistente para forçar a finalização do nó, e com isso garantir que não existe nenhum código sendo executado, para que o processo de atualização não gere um estado inconsistente ou ocasione algum erro. Mais detalhes sobre o processo de atualização dinâmica, que foram suprimidos dessa seção pois fogem do foco deste trabalho, podem ser encontrados no apêndice C.

### 2.3.2 Objetos de Simulação

O Upstart introduz o conceito de objeto de simulação (*SO - simulation object*). *SOs* são adaptadores definidos pelo responsável pela atualização para possibilitar a comunicação entre nós que executam em diferentes versões. A simulação é necessária porque a atualização em sistemas distribuídos pode ocorrer de uma forma assíncrona (nem todos os nós são atualizados ao mesmo tempo), o que permite que nós em versões antigas se comuniquem com os nós atualizados, e vice e versa. Sendo assim, existem os *SOs* futuros e passados, onde os futuros simulam versões futuras à versão corrente do nó, e os passados simulam versões anteriores à versão corrente.

Em um dado momento, é possível que um nó possua uma lista de *SOs* futuros e uma lista de *SOs* passados. É importante frisar que essas listas de

*SOs* são encadeadas, de tal maneira que um *SO* passado repassa a chamada para o *SO* passado seguinte a ele, até que se chegue à versão corrente. E o *SO* futuro repassa a chamada para o *SO* futuro anterior a ele, até que se chegue à versão corrente. Esse comportamento é importante, porque isso evita que se reimplemente todos os *SOs* quando se realiza a atualização de um nó, pois eles não estão amarrados à versão corrente do nó. A figura 2.5 ilustra o encadeamentos *SOs*.

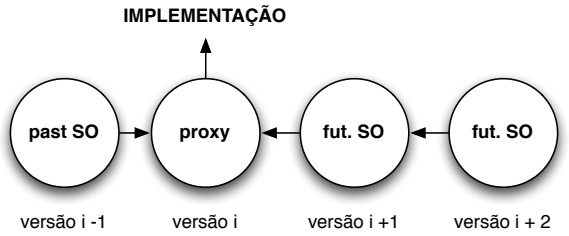


Figura 2.5: Encadeamento dos Objetos de Simulação

Os objetos de simulação são responsáveis por realizar a adequação das chamadas recebidas na sua versão para a versão seguinte ao seu encadeamento, podendo inclusive possuir um estado próprio, que é complementar ao estado do nó. Nesta adequação, ele precisa realizar o mapeamento dos parâmetros recebidos para a versão alvo. No caso de chamadas a funções que foram descontinuadas, o *SO* fica responsável por gerar uma exceção sinalizando essa condição.

A infra-estrutura se encarrega de garantir que existe um *SO* para cada versão ativa, instalando dinamicamente *SOs* futuros para novas versões e mantendo *SOs* passados enquanto as versões antigas não forem desativadas.

### 2.3.3 Função de Transformação

As Funções de Transformação (*TFs - transform functions*) são procedimentos definidos pelo atualizador, para converter o estado persistente dos nós da representação atual para a representação requerida pela nova versão. Dado que os *SOs* também podem possuir um estado, a função de transformação também fica responsável por gerar os estados dos *SOs* durante as atualizações sobre os nós.

## 2.4 Considerações Finais

Durante o desenvolvimento desta pesquisa não encontramos nenhum trabalho que se alinhasse com os requisitos de suporte a múltiplas versões

utilizando o modelo de programação baseado em componentes.

Comentamos neste capítulo que existem várias tecnologias de componentes que basicamente possuem os mesmos conceitos de modelagem. Fazendo uso desse modelo tradicional de componentes, existem duas alternativas de projeto para permitir múltiplas versões de uma mesma interface: (A) adicionar novas facetas, ou (B) criar novos componentes para cada versão da interface que se deseja dar suporte. A Figura 2.6 ilustra essas alternativas para as versões de uma interface *Hello*.

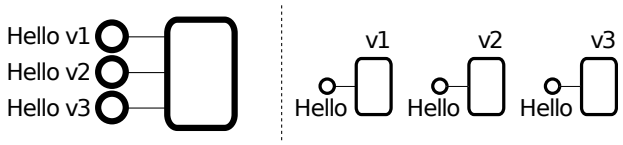


Figura 2.6: Abordagens mais comuns para prover versionamento de interfaces com o modelo tradicional de componentes de software.

Analisando pela perspectiva da modelagem e organização do modelo e do seu uso, constata-se que essas possíveis alternativas apresentam limitações. A alternativa A ocasiona a perda da identidade da abstração de facetas, pois esta deixa de representar um serviço do componente e passa a representar apenas uma versão específica desse serviço. E, se é adicionada uma faceta para cada versão do serviço oferecido, e se existe uma dependência atrelada ao serviço, então também é necessário incluir um receptáculo para cada versão da interface dependente. Com isso, o componente deixa de ter as suas dependências bem especificadas, pois não se sabe mais se ele requer que se realize conexões em todos os seus receptáculos ou em apenas alguns deles. Ambas as alternativas também são contrárias à diretiva de modularização do sistema. Os conceitos de compreensibilidade, continuidade e ocultação da informação sugerem que conceitos relacionados estejam organizados em um ou poucos módulos, e que se esconda algumas informações dentro dos módulos, para favorecer a compreensão e a manutenção dos módulos. E nenhuma dessas alternativas contribuem ou atendem a esses conceitos de modularização.

Extrapolando esse cenário para o caso de um sistema onde seus componentes oferecem várias versões de suas interfaces, evidencia-se claramente que o uso dessas alternativas tradicionais tende a tornar caótica a representação dos componentes e das arquiteturas (composições) das aplicações. A figura 2.7 ilustra um exemplo de componente neste cenário extrapolado. Nota-se que, com a inclusão de novas versões de seus serviços, o componente vira um aglomerado de facetas, onde não é expresso nenhum tipo de relacionamento entre as diferentes versões das interfaces, a não ser pelos nomes atribuídos às mesmas.

Entendemos que essa falta de expressividade sobre os relacionamentos entre as diferentes versões dos serviços oferecidos por um componente prejudica:

- o desenvolvedor do componente, pois este precisa incluir receptáculos para cada versão compatível do serviço que o componente depende, podendo ser mais de uma versão para cada serviço dependente;
- o configurador do sistema, pois este precisa realizar as conexões individuais para cada uma das diversas versões existentes nos componentes para compor o sistema;
- o administrador do sistema, pois este passa a ter um maior número de facetas e conexões para administrar, além de não ter uma visão mais abstrata da arquitetura do sistema sendo administrado.

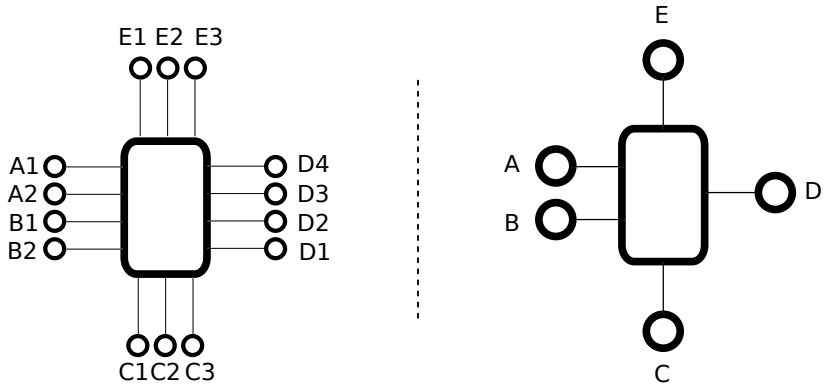


Figura 2.7: Exemplo da extrapolação das várias versões de serviços e a analogia se as versões estivessem encapsuladas (modularizadas) de acordo com o serviço prestado

Além disso, utilizando qualquer uma dessas alternativas, o desenvolvedor acabará enfrentando outros obstáculos. Um deles é a gerência da sincronia entre os estados das diferentes versões. De forma geral, o estado da instância do componente pode ser exclusivo por faceta, ou pode ser por componente e compartilhado por por todas as facetas. Independente da estratégia de implementação do estado adotada, ao optar por ter componentes diferentes para cada versão, fica a cargo do desenvolvedor do sistema garantir que as informações estejam sempre atualizadas em todos os componentes. Pois, independente de qual versão da interface está sendo utilizada, todas as versões deveriam dar respostas consistentes, e essas respostas podem ser influenciadas pelo estado do componente. Isso também ocorre se o desenvolvedor optar por adicionar no componente uma faceta para cada versão, e as facetas possuírem seus próprios estados não compartilhados na instância do componente. Para

resolver esse problema de sincronia de estado, o desenvolvedor precisaria desenvolver um mecanismo para sincronizar os estados de todas as possíveis versões cada vez que um estado fosse alterado.

Ao adotar a abordagem de adicionar facetas para cada versão, o desenvolvedor vai se deparar também com um problema de conflito de nomes. Isso ocorre pois precisa atribuir nomes de facetas diferentes para cada versão da interface que oferecer, tendo que manter os nomes antigos para ter compatibilidade com os clientes antigos.

Então, para manter a identidade da abstração de componentes, uma boa modularização do sistema e a coesão das informações, entendemos que uma faceta precisa continuar a ser a representação do serviço e não apenas uma versão específica do serviço.

Encontramos dois trabalhos na área de suporte a múltiplas versões: Ice e Upstart. O Ice atende parcialmente o problema. Ele permite que um mesmo objeto Ice seja composto por uma coleção de *facetas*, com essas *facetas* podendo implementar diferentes interfaces. Esse é o primeiro passo para dar suporte a múltiplas versões. Porém, ainda cabe ao desenvolvedor do sistema definir todo um modelo para fazer uso dessa funcionalidade mantendo o código legível e consistente semanticamente. Além disso, como o Ice não adota a abstração de componentes de software como uma entidade de primeira classe em seu modelo de programação, ele não trata de questões de versionamento relacionadas com receptáculos e conexões entre componentes, que comprometem uma visão mais abstrata da arquitetura de um sistema baseado em componentes de software.

Já o Upstart apresenta uma solução completa para prover o suporte a múltiplas versões de serviços em sistemas distribuídos. Ele permite a comunicação entre diferentes versões de uma mesma interface ao incluir o conceito de objetos de simulação, que podem simular versões passadas ou futuras da interface corrente do nó. Mas, assim como o Ice, o Upstart não adota a abstração de componente de software, e assim também não apresenta uma solução completa para esse modelo de programação. Apesar disso, como será apresentado no próximo capítulo, o Upstart serviu como inspiração para a solução proposta neste trabalho.