6
Conclusão

O LuaProxy, apresentado nesta dissertação, mostrou como usar descritores simples e de fácil reutilização para especificar os níveis de QoS desejados, ao custo de possivelmente não satisfazerem às necessidades mais complexas. Adicionalmente, ele se baseia em outros trabalhos desenvolvidos na plataforma LuaOrb, que, assim como este, optam pela flexibilidade da linguagem interpretada Lua para atenderem às necessidades de modificarem seu comportamento em tempo de execução. Como resultado, combinaram-se as rotinas de adaptação do cliente (implementações de QoS) com as desenvolvidas pelo provedor de serviço (dstubs) em uma proposta preliminar para o suporte à adaptação dinâmica.

Para aumentar o potencial do ambiente proposto, os dstubs e ustubs foram apresentados; sem eles, as implementações de QoS contariam apenas com os métodos do objeto remoto para executarem suas funções. Os dstubs oferecem alternativas de adaptação implementadas pelo próprio desenvolvedor do serviço, mas à priori ainda estariam atreladas à interface do servidor.

O *upload* de código, por outro lado, possibilita a ambas as extremidades da comunicação cliente-servidor executarem código projetado para interagirem entre si, constituindo uma camada sobre o *middleware*, transparente a seus usuários. Mesmo que não haja interesse nos mecanismos de adaptação dinâmica propostos neste trabalho, os ustubs ainda poderiam ser empregados, de maneira independente, para melhorarem a eficiência no uso dos objetos remotos.

6.1 Trabalhos Relacionados

Várias propostas [17, 18, 19] abordam o problema da adaptação dinâmica em contextos diferentes. O trabalho apresentado em [17] descreve um *framework* em Java para incorporar mecanismos adaptativos a aplicações de dispositivos móveis. Esse *framework* monitora a variação da disponibilidade de recursos para detectar quando a adaptação é necessária.

Essa monitoração é similar à utilizada pelo LuaProxy, mas requer que as condições sendo observadas persistam por um período de tempo configurável pelo programador. Ao contrário do LuaProxy, esse *framework* também aciona a adaptação dependendo do padrão de interação entre os componentes do sistema, observado por interceptadores.

Uma solução alternativa para a reconfiguração automática [18] consiste em um pré-processador que examina o código-fonte da aplicação e extrai informações sobre comportamentos de execução alternativos. Quando o agente de monitoração sinaliza a necessidade de adaptação, uma nova configuração é escolhida de uma base de dados.

Essa base de dados é costruída antes de o sistema ser disponibilizado (*deployed*). A aplicação é testada sob diversas condições (combinações de níveis de recursos livres) e a reação a cada opção de reconfiguração é observada. Isso permite modelar o comportamento de cada reconfiguração sobre os recursos do sistema, de modo a escolher, em tempo de execução, a melhor alternativa para se adaptar a um determinado evento.

Em sua versão atual, o LuaProxy requer que as implementações de QoS selecionem manualmente qual estratégia de adaptação adotar para tratar cada evento. Por outro lado, a base de dados citada no parágrafo anterior permite automatizar essa escolha, mas requer que o sistema seja testado *à priori*.

O modelo de programação para serviços interativos na Internet proposto em [19] utiliza mensagens ICMP [20] para propagar informação sobre disponibilidade de recursos. Essas mensagens disparam um mecanismo de *ações*, que são funções oferecidas pela camada de transporte para ajustarem parâmetros específicos desses protocolos (TCP e UDP). O conjunto de ações disponíveis é pequeno, de modo a evitar protocolos incorretos, ineficientes ou inseguros. Tomando-se o protocolo UDP como exemplo, pode-se fazer um processo "dormir" quando os eventos indicarem que não vale a pena transmitir um

pacote (como um aumento de custo ou de taxa de perdas), o que proporcionaria uma economia de energia em dispositivos móveis.

Trata-se, entretanto, de uma solução específica para um determinado conjunto de protocolos. O LuaProxy, por sua vez, foi desenvolvido sobre a arquitetura CORBA, tornando-o, independente dos protocolos de comunicação utilizados, mas, exatamente por esse motivo, impossibilitado de tirar proveito de suas peculiaridades.

Finalmente, nenhum dos trabalhos mencionados nesta seção utiliza a abordagem de *upload* e *download* de código em tempo de execução. O LuaProxy oferece esse recurso como uma opção adicional para implementar aplicações adaptativas.

6.2 Trabalhos Futuros

Existem vários pontos no LuaProxy que não puderam ser devidamente explorados por falta de tempo. Alguns deles requerem mais experiência de uso para serem melhor percebidos, como a expressividade da SQDL. Por se tratar de uma proposta inicial, é muito provável que aplicações futuras deste trabalho acrescentem a ela novos recursos, com o objetivo de aumentar o número de requisitos de QoS modeláveis.

Outra linha de trabalho poderia surgir baseada nos parâmetros de QoS, possivelmente propondo estruturas de dados que descrevam suas semânticas. Em sua versão atual, esta dissertação aborda muito superficialmente a descrição dos parâmetros em uso, limitando-se a produzir expressões de qualidade para ordenar as ofertas de serviço quanto as suas conveniências. Com o tempo, expressões que reflitam mais precisamente a qualidade de um parâmetro devem ser encontradas.

Na arquitetura atual, as implementações de QoS costumam definir rotinas de adaptação independentes de outros fatores. Por um lado, tal independência facilita o desenvolvimento dessas rotinas, pois permite ao desenvolvedor se concentrar em um parâmetro específico de cada vez. Por outro lado, ela as tornaria menos capacitadas a decidirem sobre as melhores estratégias de adaptação a serem adotadas.

Por exemplo, o uso de compressão de dados para remediar uma redução

de largura de banda tende a piorar o tempo de resposta, tanto no cliente quanto no servidor. Se esse retardo já estiver próximo do limite aceitável pode ser mais interessante adotar outro procedimento de adaptação. Uma opção seria implementar rotinas que consultem manualmente os monitores quanto aos valores atuais desses parâmetros, e reagir de acordo com o intervalo no qual eles se encontram.

Acreditamos que as rotinas de adaptação das aplicações mais complexas empregariam essas informações, o que provavelmente motivou o projeto QuO [11] a definir *regiões ativas* nos contratos de QoS, baseadas nos valores obtidos dos objetos de condição do sistema (*sysconds*). Uma possível melhoria para a arquitetura das implementações de QoS seria a definição de estruturas de dados que modelariam a interdependência dos parâmetros de QoS para as rotinas de adaptação. Baseado nessa estrutura, o LuaProxy poderia acionar o mecanismo de adaptação mais adequado à combinação vigente dos parâmetros de QoS.

Um recurso interessante a ser adicionado a este trabalho seria a criação de uma entidade global, na qual as instâncias do LuaProxy se registrariam e informariam seu estado em relação à adaptação. Quando essa entidade detectasse a exportação de uma nova oferta de serviço, ela poderia sugerir às instâncias mais sobrecarregadas que migrassem para o servidor da nova oferta. Como a versão atual deste trabalho já usa um processo intermediário para acessar o *trader*, a detecção da exportação de uma nova oferta seria trivial.

Um próximo passo no trabalho com dstubs seria adicionar suporte às linguagens C e Java. Para os ustubs, seria interessante investigar como usá-los para se conseguir um certo grau de paralelismo. Normalmente os métodos CORBA são bloqueantes; o próximo método só é chamado após a conclusão do anterior. Com ustubs temporários e *callbacks*, alguns métodos poderiam executar em paralelo e, nos pontos onde um método dependesse dos resultados de outro, seria feita a sincronização.

O ideal é que essa reorganização de código seja automática e transparente. Algumas sugestões seriam o uso de pré-processadores de código ou a distribuição dinâmica do código baseado em estatísticas de acesso. O problema a ser resolvido é determinar quais trechos de código seriam transferidos a cada servidor. Como o código pode acessar os diferentes objetos de maneira intercalada, seu particionamento pode requerer uma granularidade mais grossa, sob pena de o próprio algoritmo de otimização demorar mais tempo para executar do que o que seria economizado pela mobilidade de código.

A próxima proposta aborda o problema de um ustub demorar demais

para executar sua tarefa, impedindo o servidor de aceitar outras chamadas de métodos. Uma possível alternativa seria a execução dos ustubs em co-rotinas, de modo que o código do ustub possa ceder a vez de executar (*yield*) em momentos apropriados. Infelizmente, isso não impediria um usuário malicioso de instalar um ustub que sobrecarregue o servidor intencionalmente.

Finalmente, esperamos que as sugestões aqui propostas, assim como outras a serem pensadas, sigam a linha de raciocínio clássica da plataforma LuaOrb, adicionando novos recursos mas mantendo a simplicidade e flexibilidade entre os pontos principais.