5
Conclusões

A escolha de uma linguagem de script depende de uma série de fatores, varios deles relativos à linguagem em si, outros relativos à sua implementação. Quando lidamos com cenários de desenvolvimento multi-linguagem, um aspecto que não deve ser negligenciado é o projeto das interfaces entre as linguagens. Seja estendendo a linguagem de script através de código C, ou tornando uma aplicação C extensível através de uma linguagem de script, a API oferecida pela linguagem tem um papel fundamental, muitas vezes influenciando o projeto da aplicação.

Este trabalho traçou um panorama dos problemas gerais enfrentados na interação entre código C e o ambiente de execução de uma linguagem de script. Apresentamos as formas como as APIs de cinco linguagens tratam estes problemas, indicando pontos positivos e negativos das diferentes abordagens utilizadas. Realizamos uma comparação prática do uso destas APIs através de um estudo de caso onde as linguagens de script foram embutidas em bibliotecas C exportando uma mesma interface. A implementação consiste de uma biblioteca genérica para scripting, chamada LibScript, e uma série de plugins que realizam a interface com as diferentes linguagens. Pudemos assim observar como elas tratam aspectos importantes relativos a linguagens embutidas, como a passagem de dados, chamadas de funções entre as duas linguagens, tratamento de erros e o isolamento dos ambientes de execução em aplicações.

Embora os mesmos problemas gerais, como transferência de dados, registro e chamada de funções, sejam comuns aos diferentes cenários de uso de uma API de linguagem de script, aplicações embutindo uma máquina virtual tendem a demandar mais da API do que bibliotecas implementando módulos de extensão. Este ponto é ilustrado pelas dificuldades impostas pela API de Python tanto no acesso a variáveis como no registro de funções globais; e principalmente pela complexidade da API de chamada de funções de Perl.

O fato de que a API de Python dificulta o uso de variáveis e funções globais, favorecendo o uso de módulos, pode ser justificado como uma forma de promover um modelo de programação mais estruturada. Isto é interessante

para o uso da API no desenvolvimento de módulos de extensão, uma vez que o uso de variáveis e funções globais é extremamente prejudicial nestes casos, já que poluiria o espaço de nomes das aplicações Python. Já para o caso onde a linguagem é embutida para prover suporte à execução de scripts em uma aplicação C, a ausência de uma forma conveniente para definir funções globais no espaço de nomes dos scripts é questionável.

A abordagem empregada por Perl, usando um pré-processador com o objetivo de gerar automaticamente o código para a conversão de dados na passagem de parâmetros e valores de retorno, se mostrou inadequada para o cenário envolvendo interpretadores embutidos. Embora o uso do pré-processador simplifique os casos simples de declaração de funções C, a falta de uma API bem definida para tratar a transferência de dados entre o interpretador Perl e o código C se faz perceber nos casos mais elaborados. Duas destas situações se fizeram presentes no estudo de caso: o recebimento de parâmetros *varargs* e a passagem de valores de retorno tratando múltiplos contextos de execução. Ambas exigiram manipulações de estruturas e construções de mais baixo nível, que o pré-processador tem por objetivo ocultar.

Observações interessantes resultaram da comparação da API de Java com a das demais quatro linguagens de script, uma vez que, embora possua diversas características em comum com estas linguagens, Java não seja considerada uma linguagem de script. Enquanto a tipagem estática reduz bastante a necessidade de conversões de dados explícitas no código C para tipos primitivos da linguagem, na prática a verificação de tipos para objetos e a ligação de campos e métodos acontece de forma dinâmica, já que estes têm que ser realizados em tempo de execução pela JNI. Assim, no contexto da interação de uma máquina virtual com código C, as vantagens trazidas pela tipagem estática são reduzidas. Além disso, a resolução dinâmica de campos e métodos faz com que a manipulação de objetos via C tenha diferenças sutis de comportamento em relação ao que ocorre em código Java, o que pode ser uma fonte de erros do programador.

Ao comparar as APIs, consideramos apenas as suas interfaces, fazendo uma análise qualitativa da usabilidade de cada uma da perspectiva do programador C, e não uma análise quantitativa das suas implementações. O custo de desempenho adicionado pelo código que realiza a ligação entre duas linguagens, por exemplo, não pode ser desprezado. Muitas decisões de projeto de uma API são influenciadas por requisitos da implementação como restrições de portabilidade ou desempenho. Por exemplo, o tratamento automático de controle de escopo de `VALUE`s em Ruby, varrendo a pilha de C, traz grande conveniência para o programador, mas reduz a portabilidade da implementação.

Merece comentário também a disparidade entre as linguagens no que concerne à disponibilidade de documentação. Java, Python e Lua possuem extensa documentação, tanto para a linguagem como para as suas APIs para C. Para estas linguagens, pudemos basear largamente nosso estudo e a implementação dos exemplos para o estudo de caso na documentação fornecida. A documentação de Ruby relativa à sua API de C é mais escassa; em (Thomas 2004) é coberta apenas parte da API pública. Precisamos fazer uso de funções não documentadas para tarefas fundamentais como liberar referências globais registradas via C. Durante o desenvolvimento do plugin Ruby no estudo de caso, consultamos freqüentemente o código-fonte de Ruby para compreender os aspectos que não são cobertos pela documentação do comportamento das suas funções públicas. A documentação da API de C de Perl também é incompleta, espalhada através de diversas *man pages* incluídas na sua distribuição e em certos casos desatualizada. Para compreender os diversos protocolos envolvidos no uso prático da API de Perl, precisamos recorrer ao código-fonte de aplicações que fazem uso dela.

O equilíbrio entre simplicidade e conveniência é outro tema recorrente ao compararmos as APIs. A extensa API de Python, contendo 656 funções públicas, contrasta com as 113 funções expostas pela API de Lua (79 na API *core*, 34 na API auxiliar). Em diversas situações, funções na API de Python abreviam duas, três ou até mais chamadas, como no caso de funções poderosas como `Py_BuildValue` e `PyObject_CallFunction`, tornando o código C sucinto e legível. A abordagem defendida por Lua é a de uma API minimalista, oferecendo mecanismos sobre os quais funcionalidades mais elaboradas possam ser construídas. De fato, em (Ierusalimschy 2006) é apresentada uma função C equivalente a `PyObject_CallFunction` usando a API de Lua.

Ruby exporta 530 funções em seu cabeçalho e Perl 1209, mas como apenas uma pequena fração destas é documentada, torna-se difícil avaliar o tamanho da "API pública" destas linguagens e quantas destas são apenas funções para uso interno expostas nos seus cabeçalhos[1]. Isto mostra também que a documentação não é relevante apenas enquanto material de apoio para o desenvolvimento, mas também indica o quão bem definida é uma API.

A API de Java é bem documentada como a de Python e Lua, mas o número de funções exportadas não é um bom parâmetro para comparações com as demais APIs porque, em função dos tipos estaticamente definidos, muitas funções possuem uma variante para cada tipo primitivo. Java exporta sua API como uma estrutura contendo ponteiros para função; 228 funções ao

[1]Algumas funções são marcadas como sendo de uso interno, mas a maioria não possui qualquer indicação.

todo são exportadas nesta estrutura.

Outro aspecto que pôde ser observado neste trabalho é que a consistência da API depende largamente da consistência da linguagem que ela expõe. Construções onde a linguagem tem pouca ortogonalidade, como o tratamento de blocos em Ruby ou as diferenças nos tratamentos de valores escalares e arrays em Perl, acabam por aumentar a complexidade da API da linguagem e demandam tratamento específico por parte do programador no código C.

Como possibilidades de trabalhos futuros, este trabalho pode ser estendido através do estudo de outros aspectos de APIs de linguagens de script. Um foco possível é o impacto de desempenho de diferentes projetos de API em aplicações multi-linguagem. Outro é a relação entre o projeto de uma máquina virtual e o de sua respectiva API. Além disso, outra perspectiva de trabalho é a continuação do desenvolvimento da biblioteca LibScript. Possibilidades incluem adicionar novos plugins, revisar a sua API e exercitá-la embutindo a biblioteca em aplicações reais. LibScript e os quatro plugins implementados são software livre e estão disponíveis para download em `http://libscript.sourceforge.net`.