Adaptação Guiada por Políticas de Sistemas Tolerantes a Faltas Bizatinas

Miguel Pasadinhas, Daniel Porto, Antónia Lopes, and Luís Rodrigues

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
{miguel.pasadinhas, danielporto, ler}@tecnico.ulisboa.pt
LASIGE, Faculdade de Ciências, Universidade de Lisboa
malopes@ciencias.ulisboa.pt

Resumo Ataques maliciosos, erros de software ou até mesmo enganos de operadores podem causar desvios arbitrários do comportamento esperado dos sistemas. A tolerância a faltas bizantinas (BFT) engloba um conjunto de técnicas que tornam os sistemas robustos na presença de faltas arbitrárias. Vários protocolos BFT foram propostos, cada um otimizado para diferentes condições operacionais e geralmente com fraco desempenho fora dessas condições. Para colmatar esse problema, propomos o Policaby, um gestor de adaptação capaz de executar políticas de adaptação que guiam o sistema num caminho de conformidade com os seus objetivos. Estas políticas são descritas numa linguagem de alto nível, com a expressividade necessária para capturar as adaptações desejáveis neste tipo de sistemas.

1 Introdução

As faltas arbitrárias, também designadas por faltas Bizantinas [11], capturam desvios arbitrários ao comportamento esperado de um componente, sejam estes derivados de causas naturais ou resultantes de ataques premeditados por agentes que pretendem interferir na operação de um dado sistema. Pela sua abrangência, os mecanismos de tolerância a faltas Bizantinas (BFT), têm sido alvo de um interesse crescente nas últimas décadas.

Infelizmente, apesar do elevado número de protocolos que foram propostos para o efeito, não existe um protocolo BFT capaz de superar qualquer um dos restantes em qualquer ambiente de execução. Na verdade, cada solução está otimizada para cenários particulares e pode ter um desempenho fraco fora desses cenários. Por exemplo, o Zyzzyva [10] é um protocolo BFT que está otimizado para o cenário em que rede é estável, mas que necessita de executar um processo de recuperação moroso caso se suspeite que um nó especial – o líder – tenha falhado.

Por esse motivo têm sido propostos sistemas adaptativos capazes de alterar o comportamento dos sistemas tolerantes a faltas Bizantinas de acordo com as condições operacionais observadas. Trabalhos como o Aliph [2] e o Adapt [3] mostraram que a adaptação dinâmica pode melhorar o desempenho dos sistemas BFT. Contudo estes sistemas possuem um suporte muito limitado para expressar

as políticas de adaptação. Uma política de adaptação é uma especificação das adaptações que podem ser realizadas, juntamente com um conjunto de informações que caracterizam os objetivos do sistema e que guiam o processo de seleção das melhores adaptações a realizar. O Aliph implementa uma única política de adaptação estática e plasmada no código fonte do próprio sistema, não prevendo qualquer tipo de ajuste à mesma. O Adapt implementa também uma única política que consiste em abortar o protocolo BFT ativo quando as condições de execução mudam e substitui-o por um outro protocolo que se comporta melhor nessas condições. A escolha do novo protocolo é guiada pelas preferências do utilizador, contudo essa configuração é complicada e pouco intuitiva.

Apesar do número de trabalhos que aborda a adaptação dinâmica de protocolos Bizantinos ser reduzido, a literatura que descreve sistemas adaptativos em contextos distintos é bastante rica [5,7,13]. É pois fácil encontrar na bibliografia exemplos de sistemas adaptativos com mecanismos e linguagens de definição de políticas mais expressivos e flexíveis do que os suportados pelo Aliph ou Adapt. Infelizmente, a maioria desses sistemas oferece um suporte limitado para gerir sistemas replicados, dos quais os sistemas BFT são um caso particular. Em particular, verificamos que algumas das adaptações mais relevantes em aplicações BFT são difíceis de expressar nos sistemas existentes.

Neste contexto, propomos neste artigo o Policaby, um gestor de adaptação que suporta uma nova linguagem para expressar políticas de adaptação de sistema BFT. O Policaby é, ele próprio, tolerante a faltas bizantinas, executando-se em múltiplas réplicas que se coordenam para assegurar a coerência mútua. A linguagem usada para expressar as políticas de adaptação tem primitivas desenhadas a pensar nas especificidades dos sistemas replicados (como os sistemas BFT), contudo é uma linguagem genérica pelo que o Policaby pode também ser usado como gestor de adaptação de sistemas sem requisitos de tolerância a faltas.

2 Trabalho Relacionado

Após os protocolos BFT inicialmente propostos em [11], foram desenhados vários protocolos mais eficientes e capazes de funcionar numa gama mais abrangente de condições [6,10,8,1]. Infelizmente, cada protocolo apresenta um desempenho mais favorável em condições de execução especificas, não existindo um protocolo capaz de superar todos os restantes em todas os ambientes de execução. Por este motivo, tem sido feito em esforço em desenvolver sistemas BFT adaptativos, capazes de se adaptar ao ambiente de execução verificado. A reconfiguração dinâmica de um único protocolo, ou a comutação em tempo de execução entre diferentes protocolos, são exemplos de adaptações que podem ser concretizadas.

Um dos primeiros sistemas capaz de alterar o protocolo BFT face a variações nas condições de operação foi o Aliph [2]. No entanto, este sistema apenas concretiza uma única política de adaptação, definida de forma estática, que consiste em substituir um protocolo por outro sempre que as condições de operação assim o justificam. Neste sistema, os protocolos suportados são organizados num anel

e troca de protocolos segue a ordem definida na construção deste anel, o seja, um protocolo quando é desativado, é sempre substituído pelo protocolo seguinte no anel. As condições que definem quando cada protocolo deve ser desativado estão plasmadas estaticamente no código fonte, e capturam o conhecimento do programador sobre o desempenho de cada um dos protocolos. Assim, não existe no Aliph suporte para expressar novas políticas de adaptação do sistema – existe apenas uma que está embutida no código. Para além disso, o Aliph é um sistema monolítico, sem um gestor de adaptação externo ao sistema que é adaptado – é o próprio sistema que decide as próprias adaptações.

O Adapt [3] é um sistema BFT adaptativo que recorre a uma arquitetura de três camadas, nomeadamente: o sistema que deve ser adaptado (BFTS), um sistema de eventos (ES) que monitoriza continuamente o BFTS e um sistema de controlo de qualidade (QCS) que recebe dados do ES e decide as adaptações. Esta arquitetura é inspirada na malha de controlo MAPE-K [9], usada recorrentemente em sistemas adaptativos não monolíticos. Apesar do BFTS ser tolerante a faltas bizantinas, nem o ES nem o QCS o são. À semelhança do Aliph, as adaptações suportadas pelo Adapt são exclusivamente a troca de um protocolo BFT por outro mas, ao contrário do Aliph, a sequência de protocolos escolhida não é fixa. O Adapt introduziu um passo intermédio no qual decide qual o melhor protocolo para as condições de execução detetadas pelo ES. Para realizar essa decisão o Adapt baseia-se em métricas chave (monitorizadas pelo ES) que se consideram ser capazes de caracterizar o estado do sistema. Exemplos destas métricas são a latência e o débito do sistema. O processo de decisão passa pela construção de uma matriz que representa o desempenho esperado de cada protocolo (nas linhas) face a cada métrica chave (nas colunas). Cada entrada da matriz é calculada com o auxilio de uma função de previsão previamente treinada. O Adapt possui também um vetor que associa um peso a cada métrica chave, peso este que é fornecido pelo administrador do sistema. Assim, da média ponderada do desempenho esperado de cada protocolo resulta num valor que é considerado a utilidade desse protocolo. O protocolo com maior utilidade é escolhido para executar. As políticas de adaptação do Adapt baseiam-se, portanto, num vetor de pesos, no qual um administrador de sistema pode atribuir maior importância a determinadas métricas chave em detrimento de outras. Apesar de ser uma melhoria face às políticas não configuráveis do Aliph, este mecanismo não é muito expressivo. Dada a proliferação de fatores, é difícil prever o impacto das políticas que resultam de diferentes ponderações dos fatores, por exemplo, é difícil estimar o efeito de dar um peso duas vezes maior à latência face ao débito.

O ByTAM [14] é um gestor de adaptação que segue também uma arquitetura de três camadas, semelhante ao Adapt – possui um sistema gerido, um sistema de monitorização e um gestor de adaptação. O ByTAM tornou o sistema de monitorização e o gestor de adaptação tolerantes a faltas bizantinas, sendo uma melhoria face ao Adapt. Ao contrário dos sistemas vistos anteriormente, o ByTAM permite a especificação de adaptações genéricas, não estando limitado à troca de protocolos. As políticas suportadas pelo ByTAM são expressas através de um conjunto de regras na forma de evento-condição-ação, sendo que os obje-

tivos que a política pretende atingir ficam codificados apenas de forma implícita nestas regras. Para além disso, a especificação destas regras requer a codificação de uma interface em Java, pelo que as políticas têm que ser compiladas juntamente com o gestor de adaptação. Isto significa que podem ser especificadas políticas extremamente complexas (uma vez que se pode escrever código Java arbitrário) mas a um nível de abstração pouco apropriado. Para escrever políticas simples o utilizador tem que lidar com toda a complexidade do Java. Este facto combinado com a inexistência de uma API própria que facilite a escrita das políticas tornam este mecanismo difícil de utilizar na prática.

Existem na literatura várias técnicas e linguagens para especificação de políticas de adaptação mais flexíveis e expressivas do que as apresentadas nos sistemas acima. A linguagem Stitch [5,7] baseia-se na especificação de táticas (primitivas de adaptação que possuem ações e os seus impactos esperados sobre algumas métricas chave) que têm a particularidade de suportar impactos não deterministas. O não-determinismo é capturado especificando os impactos como cadeias de Markov em tempo discreto. Estratégias de adaptação são árvores de decisão em que cada ramo está protegido por uma guarda e os nós representam táticas. Estas árvores pretendem representar o processo de decisão de um administrador de sistema quando manualmente adapta um sistema. Para decidir qual a melhor estratégia a executar, o Stitch baseia-se em na média ponderada (com pesos fornecidos pelo utilizador) da utilidade de várias métricas chave. A utilidade de uma métrica é um valor entre 0 e 1, calculado por funções especificadas pelo utilizador, que mapeia valores da métrica em valores de utilidade. Este passo intermédia de calcular a utilidade das métricas permite lidar com a descrepancia nas gamas dos valores das várias métricas. A linguagem proposta por Liliana et al. [13] também se baseia na definição da adaptações que têm ações e impactos esperados. Ao contrário dos anteriores, este sistema baseia-se numa lista ordenada de objetivos para escolher a melhor adaptação a executar. Este mecanismo torna-se mais intuitivo de configurar uma vez que não é necessário descobrir quais os pesos necessários para mapear os objetivos de negócio. Em vez disso existe um mapeamento muito mais direto entre os objetivos de negocio e os objetivos especificados na linguagem.

Apesar da elevada expressividade fornecida por estas linguagens, existem algumas adaptações que se revelam difíceis ou mesmo impossíveis de expressar nestas linguagens. A título de exemplo, estas linguagens não suportam especificar uma adaptação que seleciona a réplica mais apta para ser líder do protocolo BFT. Como vamos ver mais à frente, expressar este tipo de políticas no Policaby é intuitivo devido ao mecanismo de parametrização das adaptações.

3 Linguagem

Parte fulcral do Policaby é a linguagem usada para especificar as políticas de adaptação. A linguagem foi desenhada com o objetivo de especificar políticas de adaptação para sistemas replicados, tais como os sistemas BFT. Como vimos, os gestores de adaptação existentes para adaptação de sistemas BFT não oferecem

suporte para expressar políticas de adaptação e linguagens como as descritas em [13,7], apesar de muito flexíveis, não permitem capturar de forma simples algumas adaptações relevantes em sistemas BFT. Por questões de espaço, nesta secção vamos apresentar um subconjunto da linguagem. Uma apresentação sistemática e detalhada da linguagem pode ser encontrada em [12].

3.1 Adaptações

A maioria do protocolos BFT recorrem à eleição de um líder e uma adaptação relevante consiste em trocar o processo que assume este papel. Por exemplo, se um sistema BFT estiver a usar o protocolo Zyzzyva, em regime estável a comunicação é feita apenas entre o líder e as restantes réplicas (não existindo comunicação direta entre duas réplicas não líderes). Assim, ao selecionar como líder a réplica que tem menor latência para a restantes réplicas é possível reduzir o tempo necessário para conseguir a coordenação entre os processos (que designamos por *t_ coord*). Vamos assumir que esse tempo de coordenação é inversamente proporcional à latência que o líder tem para as restantes réplicas (uma simplificação que fazemos para conveniência da exposição). Se pretendêssemos especificar esta adaptação com os mecanismos descritos em [7,5] poderíamos usar seguinte política:

```
1  define P = p: Protocolo | p.zyzzyva = true & p.activo = true
2  define l = oneOf{r:Replica | r.lider = true}
3  define novoLider = oneOf{r:Replica | r.lider = false &
4    forall t:Replica | r.latencia < t.latencia}
5
6  impactmodel mudarLider
7    size(P) > 0 -> {
8         [0.9] {novoLider.lider' = true & l.lider' = false &
9                t_coord' = t_coord * novoLider.latencia / l.latencia} +
10        [0.1] {novoLider.lider' = false & l.lider' = true}}
```

Listagem 1.1. Adaptação de mudança de líder especificada com os mecanismos descritos em [7,5].

Esta especificação captura o impacto esperado de realizar a adaptação que consistem em mudar o líder. A linguagem permite capturar fatores não deterministas, nomeadamente a possibilidade da adaptação falhar e o líder permanecer o mesmo (nesta caso, com uma probabilidade estimada de 0.1). Nesta especificação, a política define explicitamente qual o critério para escolher o novo líder; isto é feito na linha 3, onde é definido que o novo líder será aquele que possuir uma menor latência para os restantes. No entanto, considerando que o impacto de escolher um determinado líder já está capturado na política, não deveria ser necessário explicitar o critério pata escolher o líder. Ao invés, a política deveria definir um objetivo de negócio (por exemplo, reduzir a latência), e o motor de execução da política deveria usar os impactos para escolher o melhor líder. Uma forma alternativa de atacar este problema seria escrever n adaptações, em que cada uma representava o impacto de mudar o líder para a réplica i, em que n é o número de réplicas existentes e $i \in \{1, ..., n\}$. Desta forma o líder seria escolhido implicitamente pela escolha da melhor adaptação.

O trabalho de Rosa et al. [13] introduz o conceito de adaptação de nó (*node adaptation*), uma adaptação que atua sobre um nó específico de um sistema distribuído. Em execução, cada nó vai ser testado, permitindo assim descrever com apenas uma adaptação o impacto de realizar essa adaptação tendo cada nó como alvo. Infelizmente as únicas adaptações de nó permitidas são adição e remoção de componentes de um nó específico, não dando possibilidade para a mudança de uma configuração global do sistema (como uma mudança de líder) com base numa adaptação de nó.

A linguagem que desenvolvemos para o Policaby evita estas limitações. Apresentamos agora uma possível especificação da adaptação de mudança de líder no Policaby:

```
1  Adaptation mudarLider:
2      Input:
3          r: Replica
4      Requires:
5          System.lider != r
6          System.protocolo = Zyzzyva
7      Impacts:
8          [0.90]:
9            System.t_coord *= r.latencia / System.lider.latencia
10           System.lider = r
11         [0.10]:
12           System.lider = System.lider
```

Listagem 1.2. Especificação de uma adaptação de mudança de líder no Policaby.

A especificação de uma adaptação no nosso sistema possui uma sintaxe que separa claramente as condições em que a adaptação é aplicável (com a primitiva *Requires*) e os impactos esperados (com a primitiva *Impacts*). Tal como em [7,5], continua a ser possível indicar que uma adaptação pode ter diferentes resultados, cada um com uma probabilidade esperada de acontecer. Uma vez que o impacto da adaptação depende de qual a réplica escolhida, e não queremos à partida explicitar o algoritmo para seleção do líder, introduzimos na linguagem um mecanismo de parametrização das adaptações, que generaliza os mecanismos propostos em [13]. As adaptações podem especificar parâmetros formais aos quais vão ser associadas entidades concretas durante a execução. No exemplo, a adaptação é parametrizada por um elemento r do tipo *Replica* – a réplica que será o novo líder. Quando o Policaby for decidir que adaptação mais beneficia o sistema, esta adaptação será instanciada tantas vezes quanto o número de réplicas existentes no momento. A avaliação dos impactos de cada instância resulta num estado diferente pois os impactos são especificados em função do parâmetro formal r. As adaptações podem especificar tantos parâmetros formais quanto os necessários. Por exemplo, se alguma adaptação necessitar de um par de réplicas, pode especificar dois parâmetros formais do tipo Replica. É importante referir que apenas é possível parametrizar adaptações com tipos componente (descritos na Secção 3.2) que são especificados pelo utilizador. Associado a cada adaptação existe um *script* que é o código usado para realizar a adaptação. Quando o Policaby decide uma adaptação, o *script* associado a essa adaptação é executado. O *script* é especificado usando a linguagem de programação Groovy que tem acesso às *API* fornecidas pelo Policaby para facilitar a execução das adaptações.

3.2 Componentes e Instâncias

Como mostrado na listagem 1.2, as adaptações podem ser parametrizadas com parâmetros formais de tipos específicos. Esses tipos correspondem a componentes na linguagem. As componentes servem fundamentalmente dois objetivos: (1) especificação de tipos cujas instâncias podem variar ao longo do tempo e (2) especificação de tipos enumerados. Como exemplo do primeiro objetivo temos o conceito de réplica. As réplicas concretas que existem no sistema variam ao longo do tempo, sendo necessário existir um tipo para as representar. Como exemplo do segundo objetivo temos protocolos BFT. Podemos definir Protocolo como uma enumeração cujas entidades são os protocolos concretos. Na listagem 1.3 podemos ver a definição de uma componente *Replica* com atributos descritos como ou mensuráveis ou não mensuráveis. Os atributos mensuráveis têm os seus valores fornecidos por um sistema de monitorização. Vamos explorar a interação com este sistema na Secção 4. Por sua vez, os atributos não mensuráveis apenas mudam de valor como resultado de uma adaptação. Assim, devemos usar atributos mensuráveis para especificar atributos cujo valor está fora do controlo do Policaby e que devem portanto ser medidos – como a latência das réplicas – e atributos não mensuráveis sobre parâmetros sobre os quais temos controlo total. No exemplo, o custo de uma réplica foi especificado como não mensurável, significando que o custo de cada réplica é conhecido e não varia ao longo do tempo. Se imaginarmos um cenário em que o custo das réplicas varia fora do controlo do Policaby, então deveria ser usado um atributo mensurável.

```
1 Enum Protocolo Instances Zyzzyva, PBFT, Aardvark
2
3 Component Replica:
4     Params:
5         mensurable latencia: number
6         mensurable debito: number
7         custo: number
```

Listagem 1.3. Definição de Componentes e Instâncias no Policaby.

O conjunto de réplicas dos sistemas replicados pode variar ao longo do tempo (por exemplo, como resultado de adaptações de adição/remoção de réplicas). Como resposta a esse requisito, podem ser criadas e destruídas instâncias dos tipos componentes em tempo de execução. Existe, para esse efeito, uma *API* que pode ser usada nos *scripts* de adaptação, que permite criar e destruir instâncias de qualquer componente.

Para além dos atributos associados a componentes, existe também um conjunto de atributos globais do sistema. Os tipos desses atributos podem ser componentes, permitindo assim caracterizar a configuração atual do sistema (por exemplo notando qual a réplica que é líder ou qual o protocolo ativo).

3.3 Objetivos da Adaptação

Como é comum em gestores de adaptação, assume-se que os objetivos do sistema gerido podem ser especificados em função de um conjunto de indicadores de desempenho alvo (no inglês *key performance indicator*, KPI). No Policaby, todos

os atributos das componentes e atributos globais são considerados KPIs. Isto significa que tanto os impactos das adaptações como o objetivo do sistema gerido são especificados em função desses atributos. Por conveniência, os KPIs podem ser agregados em KPIs compostos. A título de exemplo, podemos especificar um KPI *custo_ total* como sendo a soma do custo de cada réplica: `KPI custo_total Is Sum Replica.custo`.

À semelhança da linguagem proposta em [13], os objetivos da adaptação são especificados recorrendo a uma lista de sub-objetivos, ordenada por ordem de importância. Esta aproximação é mais expressiva do que soluções que apenas permitem expressar um único objetivo, que consiste em maximizar uma função de utilidade, tipicamente uma média ponderada da distância a valores alvo para cada KPI. Por exemplo, podemos especificar que o custo total das réplicas deve ser inferior a 2 Euro/hora – `Goal custo_total < 2 Confidence 0.9` – ou que desejamos minimizar o tempo despendido a coordenar pedidos – `Goal Minimize System.t_coord`. Como assumimos um modelo não-determinístico para os impactos das adaptações, alguns objetivos (ditos *exatos*, isto é, que especificam uma condição de aceitação) podem especificar o grau de confiança mínimo para ser considerado cumprido. No exemplo do custo total podemos ver que especificámos uma confiança mínima de 0.9 que o objetivo seja considerado cumprido. Assim, apenas se considera que uma adaptação cumpre este objetivo se a soma das probabilidades dos ramos que cumprem o objetivo for superior a 0.9. Por outro lado os objetivos de otimização (maximização e minimização de expressões) recorrem à média do valor obtido em cada ramo ponderado pela sua probabilidade. Uma característica vantajosa da abordagem de objetivos ordenados é que fornece uma degradação graciosa caso não seja possível cumprir todos os objetivos, pois aqueles que são considerados mais relevantes são garantidos em primeiro lugar.

4 Policaby

Nesta secção descrevemos o Policaby. Em primeiro lugar descrevemos a arquitetura de três camadas assumida; de seguida apresentamos o algoritmo de decisão usado pelo motor de avaliação das políticas.

4.1 Arquitetura

O Policaby é um gestor de adaptação replicado, tolerante a faltas Bizantinas, capaz de decidir adaptações que guiam um Sistema Gerido (distinto do Policaby) num caminho de conformidade com os seus objetivos. Como requisito para tomar essas decisões, deve existir um Sistema de Monitorização capaz de monitorizar algumas métricas chave sobre o Sistema Gerido. Esse conhecimento gerado pelo Sistema de Monitorização caracteriza o ambiente de execução do Sistema Gerido, permitindo ao Policaby selecionar as melhores adaptações face a esse ambiente.

O fluxo de informação é o seguinte: o Sistema de Monitorização usa sensores para recolher métricas chave sobre o Sistema Gerido; essas métricas são tratadas pelo Sistema de Monitorização e enviadas para o Policaby; com esse

conhecimento, o Policaby decide se existe uma adaptação que melhore o desempenho do sistema; por fim, caso exista uma adaptação a ser executada, essa decisão é comunicada ao Sistema Gerido. As métricas enviadas pelo Sistema de Monitorização são totalmente ordenadas pelo BFT-SMaRt [4], uma biblioteca para a construção de serviços tolerantes a faltas Bizantinas. O BFT-SMaRt implementa um protocolo de tolerância a faltas bizantinas e garante que todas as réplicas recebem as mensagens em ordem total. Por se tratar de um serviço construido em cima do BFT-SMaRt, o Policaby é tolerante a faltas bizantinas. O Policaby executa o algoritmo de decisão de adaptações periodicamente, sendo que essa execução é despoletada por uma mensagem vinda de um temporizador (proporcionado pelo Sistema de Monitorização). Novamente, esta mensagem é totalmente ordenada pelo BFT-SMaRt. A ordem total de entrega de mensagens garantida pelo BFT-SMaRt garante que todas as réplicas do Policaby recebem todas as mensagens pela mesma ordem e como resultado todas as réplicas tomam a mesma decisão.

4.2 Motor de Avaliação das Políticas

O Policaby é composto por duas partes fundamentais: (1) um analisador que transforma especificações escritas na linguagem descrita na Secção 3 em objetos internos e (2) um motor capaz de usar esses objetos para decidir adaptações que guiam o sistema gerido ao encontro dos seus objetivos. O motor corre periodicamente e resulta na execução de uma adaptação. Existe sempre no sistema uma adaptação *NOP* que não realiza qualquer ação. Isto permite que, se nenhuma das restantes adaptações melhorar o sistema, nenhuma ação seja tomada.

O algoritmo de decisão do motor de avaliação das políticas é o seguinte. Em primeiro lugar, é criado um conjunto S contendo a adaptação *NOP*. Depois, cada adaptação é instanciada com base dos seus parâmetros formais e cada instância é colocada no conjunto S. A instanciação de uma adaptação corresponde à associação dos seus parâmetros formais com instâncias concretas das componentes associadas. Por exemplo, a adaptação *mudarLider* da listagem 1.2 seria instanciada n vezes, uma para cada réplica existente. De notar que se uma adaptação possuir mais do que um parâmetro formal, todas as combinações são geradas. No final deste passo, todas as combinações possíveis de instâncias de adaptações estão em S. Neste ponto, são verificadas as condições de aplicação das adaptações (especificadas com a primitiva *Requires*) e as instâncias de adaptações que não cumpram todas as condições são removidas de S. As adaptações presentes em S correspondem agora a todas as instâncias de adaptações válidas para a decisão. Para cada uma é gerado um estado esperado, avaliando os impactos da adaptação. Resta apenas verificar qual o estado esperado mais favorável de acordo com os objetivos. Assim, começando pelo objetivo mais prioritário (que é o que foi definido mais acima nas políticas), o conjunto S é validado contra cada objetivo. Os elementos de S que não estejam em conformidade com o objetivo são removidos, até que S contenha apenas um elemento ou os objetivos acabem. No final destes testes, se S contiver apenas um elemento, então essa é a adaptação mais vantajosa. Caso S contenha vários elementos, considera-se que

todas as adaptações presentes são equivalentes, e uma aleatória é escolhida. Caso nenhuma adaptação consiga realizar um determinado objetivo, esse objetivo é descartado e todas as adaptações presentes em S passam automaticamente ao próximo objetivo. Desta forma os restantes objetivos podem ser usados como fator de desempate.

5 Avaliação

O Policaby permite especificar uma gama de políticas muito mais abrangente do que sistemas como o Adapt ou o Aliph. Os mecanismos como parametrização de adaptações, não-determinismo associado às adaptações e adaptação baseada em objetivos ordenados dotam a linguagem com a capacidade de especificar políticas de adaptação que não são permitidas nesses sistemas. Como contrapartida, o processo de escolha de adaptações torna-se computacionalmente mais exigente.

A título de exemplo, uma adaptação que seja instanciada 10 vezes e que, por via do não-determinismo, possua dois ramos alternativos, corresponde a 20 estados possíveis com uma única adaptação. A escalabilidade do sistema torna-se portanto um aspeto merecedor de atenção.

Para aferir a escalabilidade do sistema, medimos o tempo desde que o Policaby inicia o processo de decisão até que decide qual a melhor adaptação, numa réplica com processador Intel Core i7 860 (2.80GHz), com 8GB de RAM DDR, disco SSD de 280GB e sistema operativo Debian 8.5. Para cada configuração testada, o tempo apresentado corresponde ao valor médio de 100 experiências.

Nesta experiência a política tem n adaptações (variando n nas várias configurações testadas) com dois ramos (com probabilidades 0.3 e 0.7) e cada ramo especifica três funções de impacto. Para além disso, esta adaptação é parametrizada por duas componentes, das quais existiam 4 instâncias, resultando em cada adaptação ser instanciada 16 vezes. Metade dos objetivos usados são exatos e a outra metade de otimização. À exceção de uma configuração, todas as instâncias de adaptação passavam todos os objetivos. Isto corresponde ao pior caso para o Policaby, uma vez que nenhum objetivo filtrou instâncias de adaptação e consequentemente, todas as instâncias de adaptação foram ser testadas contra todos os objetivos.

Na Figura 1 podemos ver os resultados experimentais. O eixo horizontal codifica o número de instâncias de adaptação testadas e o eixo vertical o tempo de execução em milissegundos. Cada série com linha cheia corresponde a um número diferente de objetivos. Podemos ver que, para um número fixo de objetivos, aumentar o número de instâncias de adaptação aumenta o tempo de decisão linearmente. O mesmo se verifica com o aumento do número de objetivos (com número de instâncias de adaptação constante). A série que usa 0 objetivos permite entender o tempo gasto para instanciar as adaptações e gerar os estados resultantes de avaliar as funções de impacto. Assim, conseguimos entender que maior parte do tempo do processo de decisão é gasto na avaliação dos objetivos. Fizemos também um teste com 32 objetivos que rejeitavam (em média) 15% das instâncias de adaptação testadas. Este caso corresponde à série a tracejado

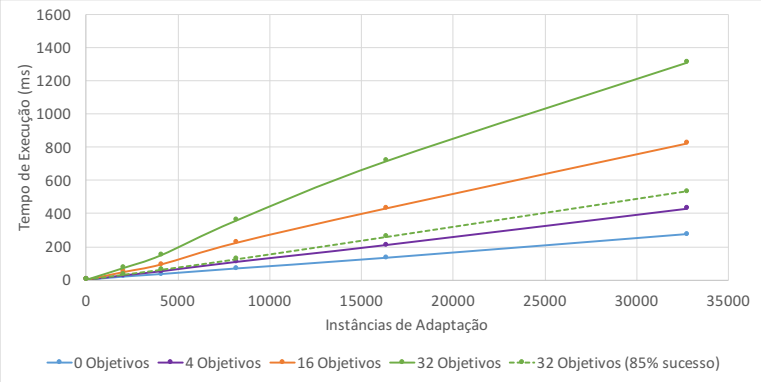

Figura 1. Tempo de decisão de uma réplica.

no gráfico e simula um caso de uso mais real, em que nem todas as adaptações conseguem cumprir todos os objetivos. Comparando os resultados de ambas as séries com 32 objetivos, podemos ver que mesmo um taxa de rejeição baixa de 15% permite melhorias significativas no tempo de decisão (superior a 50%), conseguindo o Policaby decidir uma adaptação entre mais de 32.000, tendo em conta 32 objetivos, em cerca de meio segundo.

6 Conclusões e Trabalho Futuro

Neste artigo apresentámos o Policaby, um gestor de adaptação robusto com uma nova linguagem para expressar políticas de adaptação. Este linguagem suporta especificação de adaptações parametrizadas, permitindo assim escrever adaptações que implicitamente selecionam as melhores instâncias como argumentos. A linguagem suporta também um mecanismo que permite especificar diversos ramos (com probabilidade associada) para uma adaptação, permitindo assim lidar com a incerteza do impacto da adaptação. Para selecionar os objetivos o Policaby baseia-se numa lista ordenada de objetivos que o sistema gerido deve cumprir. Tanto quanto é do nosso conhecimento, o Policaby é o primeiro gestor de adaptação que lida com não-determinismo das adaptações capaz de escolher adaptações com base numa lista de objetivos. Em trabalhos futuros devem ser explorados mecanismos que permitam ao Policaby escolher uma sequência de adaptações em vez de uma única adaptação.

Agradecimentos: Este trabalho for parcialmente suportado pela Fundação para a Ciência e Tecnologia (FCT) através dos projectos com referências PTDC/ EEI-SCR/ 1741/ 2014 (Abyss) e UID/ CEC/ 50021/ 2013.

Referências

1. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable byzantine fault-tolerant services. SIGOPS Oper. Syst. Rev. 39(5), 59–74 (Oct 2005), http://doi.acm.org/10.1145/1095809.1095817
2. Aublin, P.L., Guerraoui, R., Knežević, N., Quéma, V., Vukolić, M.: The next 700 bft protocols. ACM Trans. Comput. Syst. 32(4), 12:1–12:45 (Jan 2015), http://doi.acm.org/10.1145/2658994
3. Bahsoun, J.P., Guerraoui, R., Shoker, A.: Making bft protocols really adaptive. In: Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium. pp. 904–913. IPDPS '15, IEEE Computer Society, Hyderabad, India (2015), http://dx.doi.org/10.1109/IPDPS.2015.21
4. Bessani, A., Sousa, J.a., Alchieri, E.E.P.: State machine replication for the masses with bft-smart. In: Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 355–362. DSN '14, IEEE Computer Society, Atlanta, GA (2014), http://dx.doi.org/10.1109/DSN.2014.43
5. Cámara, J., Lopes, A., Garlan, D., Schmerl, B.: Adaptation impact and environment models for architecture-based self-adaptive systems. Sci. Comput. Program. 127(C), 50–75 (Oct 2016), http://dx.doi.org/10.1016/j.scico.2015.12.006
6. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. 20(4), 398–461 (Nov 2002), http://doi.acm.org/10.1145/571637.571640
7. Cheng, S.W., Garlan, D.: Stitch: A language for architecture-based self-adaptation. J. Syst. Softw. 85(12), 2860–2875 (Dec 2012), http://dx.doi.org/10.1016/j.jss.2012.02.060
8. Clement, A., Wong, E., Alvisi, L., Dahlin, M., Marchetti, M.: Making byzantine fault tolerant systems tolerate byzantine faults. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation. pp. 153–168. NSDI'09, USENIX Association, Boston, Massachusetts (2009), http://dl.acm.org/citation.cfm?id=1558977.1558988
9. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
10. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative byzantine fault tolerance. ACM Trans. Comput. Syst. 27(4), 7:1–7:39 (Jan 2010), http://doi.acm.org/10.1145/1658357.1658358
11. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. 4(3), 382–401 (Jul 1982), http://doi.acm.org/10.1145/357172.357176
12. Pasadinhas, M.: Policy-Based Adaptation of Byzantine Fault Tolerant Systems. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa (Oct 2017)
13. Rosa, L., Rodrigues, L., Lopes, A.: Goal-oriented self-management of in-memory distributed data grid platforms. In: Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2011). Athens, Greece (Nov 2011), (short paper)
14. Sabino, F.: ByTAM: a Byzantine Fault Tolerant Adaptation Manager. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa (Sep 2016)