Universidade do Minho

João Paulo da Silva Bordalo

MINHA: Avaliação realista de aplicações distribuídas num ambiente centralizado

Dissertação de Mestrado
Mestrado em Informática
Trabalho efectuado sob a orientação de
Doutor José Orlando Roque Nascimento Pereira

Outubro 2011

Agradecimentos

Dedico este espaço a todos aqueles que me ajudaram na realização da minha dissertação e é com grande orgulho e satisfação que lhes agradeço pela sua contribuição.

Começo por agradecer ao Prof. Doutor José Orlando Pereira, a sua disponibilidade, orientação e ajuda prestada pois foram essenciais para a concretização do meu trabalho.

Em segundo lugar agradeço aos alunos de doutoramento Nuno Carvalho e Filipe Campos a dedicação, apoio e indicações que me deram ao longo do meu trabalho, uma ajuda essencial em termos de informação para a dissertação.

Agradeço, a nível pessoal, à minha namorada Sara Vieira pela ajuda, incentivo e pela coragem que me foi dando no decorrer deste trabalho extenso. Aos meus pais e familiares, pelo seu imprescindível apoio e ânimo em todos os momentos da dissertação.

Aos meus amigos pela sua ajuda e motivação Paulo Ruivo, Nuno Macedo, André Batista, Daniel Cadete, Pedro Costa, Mauricio Neves e restantes amigos.

Destaco também a ajuda e disponibilidade dos meus colegas de laboratório de Sistemas Distribuídos, aos quais agradeço: Miguel Araújo, Luís Zamith, Nelson Gonçalves, Pedro Gomes e Miguel Borges.

Resumo

Nos últimos anos os sistemas distribuídos têm sofrido um crescimento exponencial. Estes sistemas, normalmente implementados na plataforma Java, são compostos por um vasto conjunto de componentes de *middleware*, os quais desempenham várias tarefas de comunicação e de coordenação. Esta tendência influência a modelação e a arquitectura de novas aplicações cada vez mais complexas obrigando a um enorme esforço e a um custo elevado na avaliação do seu desempenho. A concorrência e a sua distribuição, bem como o facto de muitos problemas só se manifestarem pela grande escala em si, não permite que a sua avaliação seja feita com recurso a simples ferramentas que não tenham em conta estas características.

Avaliação realista e controlada de aplicações distribuídas é ainda hoje muito difícil de alcançar, especialmente em cenários de grande escala. Modelos de simulação pura podem ser uma solução para este problema, mas criar modelos abstractos a partir de implementações reais nem sempre é possível ou mesmo desejável, sobretudo na fase de desenvolvimento na qual ainda podem não existir todos os componentes ou a sua funcionalidade estar incompleta.

Esta falha pode ser colmatada com a plataforma MINHA, que permite uma avaliação realista das aplicações através da combinação de modelos abstractos de simulação e implementações reais num ambiente centralizado. O ponto principal desta dissertação consiste na criação de um modelo de rede a ser usado pela plataforma de modo a incluir na avaliação variáveis como os tempos necessários para a troca de mensagens, elevando assim a precisão dos resultados gerados. Para isto é apresentado o método de calibração do modelo através do qual é possível aproximar o modelo do ambiente real. Este sistema permite reproduzir as condições de um sistema em grande escala e através da manipulação de *bytecode* Java, suporta componentes de *middleware* inalterados. A utilidade deste sistema é demonstrada aplicando-o ao WS4D, uma pilha que cumpre a especificação Device Profile for Web Services.

Abstract

In recent years, distributed systems have been suffering an exponential growth. These systems, typically implemented in Java platform, are composed by a wide range of components of middleware, which perform several communication and coordination tasks. This trend influences the modelling and the architecture of the newest applications, which are increasing complexity and requiring an large effort with high costs on the evaluation of their performance. Concurrency and distribution, as well as the fact that many problems manifest only in large scale, would not allow doing an evaluation using simple tools which do not take in account these features.

The realistic evaluation of distributed applications is still a difficult task, particularly for large scale scenarios. The use of simulation models can be a solution for this problem, but their creation based on real implementations can sometimes be impossible or undesirable, as the system can be incomplete and non functional. This problem can be solved with the MINHA platform, that allows a realistic evaluation of applications trough the combination of abstract models and the simulation of real implementations in a centralized environment. The main goal of this dissertation is the creation of a network model to be used by the MINHA platform. This model introduces new variables in the evaluation such as the needed time for message exchange, resulting in more accurate results. Furthermore, it is presented a calibration method that improves the faithfulness of the model to the real environment. This allows the reproduction of a large scale system and through java bytecode manipulation it allows the usage of pre-existent middleware components. The usefulness of this system is demonstrated by applying it to WS4D, a stack that complies with the Device Profile for Web Services specification.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Nos dias de hoje, a heterogeneidade dos mercados, a dispersão geográfica e a escala têm um impacto profundo na evolução das tecnologias de informação. Como tal, os mercados tornaram-se mais exigentes para com os sistemas de computação, forçando-os a tornarem-se mais complexos e a crescerem consideravelmente. Este crescimento aumenta, inevitavelmente, o interesse na avaliação da performance, visto que o objectivo principal é proporcionar uma melhor combinação entre o custo e a performance. Os serviços de rede de grande escala incluem comportamentos complexos difíceis de reproduzir. O seu desenvolvimento requer, portanto, uma avaliação e testes completos e realísticos, mas é sabido que nem sempre é fácil encontrar e responder aos problemas de maneira rápida e correcta.

É necessária uma avaliação detalhada da performance, da confiança dos protocolos abstractos e das suas implementações para obter a melhor combinação em diferentes ambientes. Testes reais são normalmente custosos de configurar e executar e dependem da disponibilidade do sistema por inteiro, tornando-se mais complicado quando conciliado com grandes *clusters* ou sistemas de grande dimensão. Embora possam ser usadas pequenas aplicações e cargas sintéticas (conhecidas como *toy applications* e *micro-benchmarks*) é-lhes difícil identificar pequenas interacções da semântica e dinâmica da aplicação, com o meio, assim como efeitos significativos dos testes.

A avaliação do sistema também depende da sua total disponibilidade e nem sempre tal é possível. Uma solução seria criar modelos abstractos dos componentes em falta, mas tal nem sempre é desejável, pois o interesse recai sobre a avaliação da solução real e não sobre os modelos abstractos. Quando não é possível ter todo o sistema disponível,

como acontece por vezes durante a fase de desenvolvimento, os modelos de simulação são uma forma fácil de prever a performance pois permitem imitar o comportamento real do sistema. Uma simulação centralizada de uma execução distribuída de múltiplas instâncias num único espaço de endereçamento, com um modelo de simulação discreta de eventos do ambiente, permite a análise e modificação global do estado do sistema com interferências reduzidas, permitindo também, uma avaliação precisa das implementações reais. Esta abordagem é seguida em ferramentas como SSF [1] e PlanetLab [11].

Gerir e manipular o tempo virtual tendo em conta o tempo real usado pelas implementações permite que o sistema seja ajustado de forma a representar os resultados dos sistemas reais de forma precisa. Assim, torna-se fácil obter uma substituição incremental dos modelos por implementações reais dos componentes durante a avaliação da eficiência do sistema. Existem várias ferramentas de simulação disponíveis, mas nenhuma delas admite código fonte inalterado, requerem sempre que este seja modificado de modo a ser incorporado no sistema de simulação.

A plataforma MINHA em desenvolvimento na U. Minho fornece todas as características normais da simulação, lidando automaticamente com o código fonte do sistema a ser testado através da manipulação de *bytecode* Java. Esta ferramenta permite realizar uma avaliação realista dos sistemas reais reproduzindo as condições a que estes estão sujeitos. Isto é feito combinando as implementações reais e modelos abstractos de simulação num ambiente centralizado. A simulação no MINHA consiste na utilização de eventos de simulação, utilizando uma linha de tempo virtual que permite simular o tempo gasto pela execução onde o decorrer do tempo é representado pela execução de eventos. A execução de código de simulação é feita através de um núcleo de simulação que permite controlar o acesso aos recursos do sistema.

1.1 Problema

Para o teste dos sistemas referidos, é fundamental dispor de um componente que permita considerar a rede. E como tal, para que o MINHA possa considerar este elemento na avaliação, necessita de recorrer a um modelo de simulação que represente o ambiente onde a aplicação irá executar. Uma vez que o MINHA não visa a simulação de redes, esta componente não deverá ser muito detalhada de forma a não trazer muito *overhead* à simulação, mas deverá conter detalhe suficiente de modo a reproduzir fielmente a realidade.

1.2 Objectivos e contribuições

A criação deste modelo de rede figura o objectivo principal da dissertação. Trata-se de um modelo de simulação de alto nível que reproduz um comportamento similar ao da rede onde a aplicação deverá ser executada, tendo em conta os tempos efectuados na transmissão de dados, bem como garantias dadas pelos protocolos de comunicação. Para poder obter este comportamento, o modelo deverá sujeitar-se a uma calibração sobre as suas propriedades. Esta calibração é automática e consiste na avaliação da rede real através da utilização de *benchmarks*, que recolhem os dados necessários para determinar os valores a serem aplicados ao modelo de simulação.

1.3 Estrutura da dissertação

Esta dissertação está dividida da seguinte forma: No Capítulo 2 é apresentado o levantamento do estado da arte sobre as formas de avaliação de sistemas distribuídos, mais concretamente na técnica da simulação através de eventos discretos. Neste capítulo são também apresentados vários exemplos de ferramentas de simulação. No Capítulo 3 é introduzida a ferramenta de simulação MINHA, descrevendo as suas características e funcionamento. No Capítulo 4 é apresentado o modelo de simulação de rede, referindo a adaptação das propriedades das redes reais ao modelo, desde a transmissão de dados à implementação das garantias dos protocolos de comunicação. De seguida, no Capítulo 5 é explicado como se procede à calibração dos modelos, referindo os *benchmarks* utilizados e o modo de aplicação dos valores de calibração. O Capítulo 6 apresenta o estudo de caso, onde é demonstrada a utilidade da ferramenta, através da sua aplicação ao WS4D. Finalmente, no Capítulo 7 são expostas a conclusões finais e o trabalho futuro.

Capítulo 2

Estado da Arte

Neste capítulo são apresentadas as várias técnicas de avaliação de aplicações e respectivas considerações. A correcta escolha destas técnicas permite poupar muito tempo e trabalho pois a maioria dos esforços de avaliação de sistemas tornam-se grandes projectos, que se não forem geridos correctamente, podem falhar. Dentro dessas opções destaca-se a simulação a qual apresenta várias possibilidades para ser usada, que também devem ser correctamente escolhidas de forma a não produzirem resultados inválidos que não possam ser usados.

2.1 Técnicas de avaliação

Seleccionar uma técnica de avaliação é um passo chave na avaliação da performance de um projecto. Existem muitas considerações que envolvem a selecção correcta. De acordo com o livro *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling* existem três técnicas: modelação analítica; simulação; e medição, cujas considerações são expostas na Tabela 2.1, seguindo-se as respectivas justificações:

A consideração mais importante a ter em conta é a fase em que se encontra o projecto. As medições são possíveis apenas se já existir um sistema idêntico ao proposto, normalmente usado para o melhoramento da versão de um produto. Quando se trata de algo novo, esta técnica não é útil, apenas a modelação analítica e a simulação. Ainda assim, o uso destas duas últimas técnicas é mais convincente quando se baseiam em medições anteriores.

Criterion	Analytical Modeling	Simulation	Measurement
1. Stage	Any	Any	Postprototype
2. Time required	Small	Medium	Varies
3. Tools Analysts	Computer	languages	Instrumentation
4. Accuracy	Low	Moderate	Varies
5. Trade-off evaluation	Easy	Moderate	Difficult
6. Cost	Small	Medium	High
7. Saleability	Low	Medium	High

Tabela 2.1: Critérios a selecção uma técnica de avaliação. Fonte [10].

O tempo disponível para a avaliação é também uma consideração importante. Quando há urgência na avaliação a melhor técnica a adoptar é a modelação analítica, pois a simulação demora muito tempo e as medições normalmente demoram mais que a modelação analítica mas menos que as simulações. Ainda assim, o tempo necessário para as medições é o mais variável entre as técnicas, pois neste se algo pode correr, então vai correr.

A próxima consideração refere-se às ferramentas disponíveis, que incluem habilidades de modelação, linguagens de simulação, e instrumentos de medida. As ferramentas de análise são normalmente especializadas numa só técnica, o que leva a que esta consideração dependa da capacidade da ferramenta disponível para usar.

O nível de precisão pretendida é um factor muito importante na avaliação. A modelação analítica exige que sejam feitas muitas simplificações e assunções que muito dificilmente resultam em resultados precisos. Como a simulação implementa maior detalhe e necessita de menos assunções, os seus resultados conseguem ser mais parecidos com a realidade. As medições podem também não dar resultados precisos, uma vez que normalmente os parâmetros relacionados com o ambiente são únicos para cada experiência, e ainda assim os parâmetros podem não representar a gama de variáveis presentes no mundo real.

O objectivo de todos os estudos de performance é comparar as diferentes alternativas ou encontrar valores óptimos para os parâmetros. Das várias técnicas a modelação analítica é a que fornece a melhor relação entre os parâmetros e as suas interacções. Com a simulação é possível procurar a melhor combinação de valores, mas é difícil decidir qual destas combinações é a melhor. Em relação a medições, é difícil afirmar se o melhoramento se deve a uma mudança aleatória ou a uma configuração particular.

Os testes efectuados sobre os sistemas têm os seus próprios custos. Avaliar o sistema através de medições é a solução mais custosa, uma vez que esta necessita de equipamento real, instrumentos e de tempo. O custo da simulação depende da sua capacidade de alteração das configurações, sendo normalmente aplicada a sistemas caros. Os modelos analíticos apenas necessitam de papel, lápis e tempo, sendo por isso a alternativa mais barata.

Através de medições reais de um sistema, é bastante mais fácil convencer alguém usando os seus resultados. Normalmente o uso de resultados analíticos provoca cepticismo devido à incompreensão da técnica ou aos resultados finais. Aliás, normalmente os modelos analíticos são validados através de simulações ou de medidas actuais.

Dadas a comparação entre estas considerações a que mais se destaca relativamente sobre o objectivo da abordagem apresentada por este documento é a simulação. Pois apresenta um melhor equilíbrio entre as várias considerações, principalmente a precisão dos resultados, a avaliação em qualquer fase do projecto, o custo e a negociabilidade. Por exemplo, mesmo que o sistema esteja disponível para a medição, um modelo de simulação pode ser preferível pois permite comparar as alternativas sobre uma grande variedade de ambientes e cargas de trabalho. Tal como, se o custo for um factor importante, a simulação é preferível aos modelos analíticos, uma vez que a validação dos resultados e respectiva negociabilidade são considerações que pesam mais na validação dos resultados.

2.2 Simulação

A simulação é uma técnica útil e muito usada na avaliação da performance de sistemas distribuídos. Consiste, fundamentalmente, em imitar o comportamento exercido por componentes de hardware, redes de computadores, através de modelos abstractos do seu estado ao longo do tempo. Não obstante, o modelo não deverá representar toda a complexidade do sistema mas ser uma simplificação do sistema real suficientemente detalhado de forma a retirar conclusões válidas. A simulação pode ser usada para explorar as várias capacidades do sistema, adquirir novos pontos de vista sobre a eficiência e comportamento da aplicação e também fornecer alternativas a componentes não disponíveis durante a fase de desenvolvimento, permitindo, ainda assim, uma forma fácil de prever a performance do sistema, ou compará-lo com várias alternativas.

2.2.1 Características de simulação

As propriedades dos modelos de simulação variam conforme os seus objectivos, assim como o detalhe que fornecem, quanto mais detalhado o modelo mais realísticos são os resultados da avaliação. Porém, um elevado detalhe nem sempre é desejado, pois pode implicar uma maior complexidade e resultando, por vezes, em modelos intratáveis. A alternativa pode passar pela utilização de modelos mais abstractos, mas ainda que não demasiado abstractos de modo a fornecerem as características pretendidas do modelo. Ainda assim, a capacidade do sistema de simulação escalar tem sempre de ser tomada em conta, uma vez que as aplicações alvo tendem a ser cada vez maiores.

Não é só o grau de detalhe do modelo que define a qualidade dos resultados de um teste. Ao usar uma ferramenta que tem como variável o tempo de execução, ou mesmo a precisão dos valores gerados pelo modelo, as unidades das variáveis utilizadas para este efeito definem a qualidade dos resultados obtidos pelo modelo de simulação.

Sendo que os sistemas distribuídos correspondem à tendência do crescimento global, algumas ferramentas de simulação apostam na utilização de um vasto número de recursos de forma a aproximar o ambiente ao qual o sistema estará sujeito, chegando por vezes a utilizar nodos geograficamente distribuídos. Desta forma alguns factores têm de ser levados em conta como a complexidade da configuração destes recursos, ou mesmo, a sua disponibilidade. Outra desvantagem destes sistemas é a dificuldade de controlar a simulação, recolher os seus resultados e analisá-los. Desvantagem esta, ultrapassada quando usada simulação centralizada.

É habitual que haja a necessidade de proceder à avaliação do sistema durante a sua fase de desenvolvimento. Como tal, é natural nestas situações que o sistema ainda não se encontre totalmente disponível tornando-se impossível a sua análise. Esta característica torna-se, por muitas vezes, fundamental para o seu correcto desenvolvimento. A capacidade do sistema de simulação de fornecer variados modelos de simulação pode facilitar o teste da aplicação, evitando a necessidade de recorrer a outros sistemas de simulação que os forneçam, possibilitando a poupança de tempo. Isto porque, o tempo gasto na avaliação pode influenciar na aprovação da mesma, pois normalmente, um único teste não é suficiente, sendo necessário reconfigurar o sistema.

A linguagem utilizada pelas ferramentas de simulação pode restringir o conjunto de aplicações suportado, limitando-o, consoante o caso, a aplicações que implementadas na mesma linguagem. A utilização de ferramentas obsoletas, por norma influencia a

qualidade dos resultados gerados. Isto porque pode retratar situações entretanto corrigidas ou não considerar outras que tenham surgido e que não tenham sido consideradas aquando a sua criação.

2.2.2 Técnicas de simulação

Existem várias técnicas de simulação, porém apenas algumas se revelam de interesse para a simulação de sistemas computacionais: emulação, simulação baseada no método de Monte Carlo, simulação orientada a *traces* e simulação orientada a eventos discretos [10].

Emulação Ainda sendo um tipo de simulação, a emulação é mais voltada para desenhos de hardware. Quando se pretende emular um recurso todos os aspectos têm de ser considerados e consequentemente modelados. Este tipo de simulação, é normalmente usada quando a totalidade do recurso é necessária para imitar o seu comportamento, ou avaliar diferentes hardware sem ser preciso construir o parte física do sistema.

Monte Carlo Uma simulação baseada no método de Monte Carlo é uma simulação estática que não considera uma variável temporal. Esta técnica é normalmente usada para simular fenómenos probabilísticos que não mudam de características com o tempo, baseando-se em amostras aleatórias para calcular resultados, quando é difícil de obter um resultado exacto com um algoritmo determinístico.

Orientada a *traces* Este tipo de simulação utiliza *traces* como parâmetros de entrada. Estes *traces* representam um registo temporal dos eventos ocorridos durante a execução de um sistema real. Como tal, este tipo de simulação permite comparar a eficiência de diferentes algoritmos, obrigando a que o *trace* utilizado seja independente do sistema em estudo, de forma a poder ser aplicado às restantes alternativas. Esta técnica é geralmente usada para a análise ou ajustar algoritmos de gestão de recursos.

Estas simulações fornecem algumas vantagens como credibilidade e justiça na comparação dos resultados obtidos pelas diferentes alternativas dos modelos em estudo, visto que o *trace* usado preserva os efeitos sobre a carga de trabalho, permitindo uma validação fácil dos modelos bastando comparar as suas performances com a do sistema real.

Sendo determinísticos, os *traces* permitem obter uma menor aleatoriedade, ainda que

os resultados possam ser diferentes, a variância não é muito significativa, e através de algumas repetições é possível obter uma confiança estatística sobre os resultados. Esta característica, permite também analisar as vantagens de pequenas alterações no modelo.

Apesar das vantagens que o uso de *traces* fornece, este também implica algumas desvantagens. Quando surge a necessidade de aumentar o detalhe do sistema de simulação, a sua complexidade aumenta também. Assim como, o uso de *traces* mais longos, o detalhe que este proporciona também aumenta, podendo ocupar demasiado espaço tornando impossível a sua utilização.

A utilização do *trace* obriga a restrições como a limitação da carga de trabalho que este representa, se for necessário alterá-la, um novo *trace* tem de ser gerado de forma a representar a configuração da nova carga, este novo *trace* pode então deixar de ser representativo nos outros modelos, impossibilitando a sua comparação, obrigando a criar um novo *trace* compatível com todos os modelos. Mais ainda, este tipo de simulação limita o teste do sistema à carga de trabalho correspondente ao *trace*.

Eventos Nos sistemas cujos modelos de simulação se baseiam em eventos, as operações são representadas através de uma sequência cronológica de eventos, em que cada evento ocorre num instante de tempo alterando o estado do sistema. Ou seja, o tempo avança conforme os eventos são agendados para ocorrerem. Quando o sistema deixa o instante *t* e o próximo evento está agendado para acontecer apenas no instante *t'*, a linha de tempo salta do instante *t* para *t'* instantaneamente. Esta técnica permite que a quantidade de eventos, possíveis de serem simulados por um sistema de simulação, exceda qualquer outra técnica mais tradicional. Se esta técnica for utilizada com recurso a outras ferramentas que permitam a paralelização de código de simulação, pode diminuir, consideravelmente, o tempo necessário para a avaliação da aplicação em relação ao que seria gasto no ambiente real.

Todas as simulações de eventos possuem uma estrutura comum, contendo os seguintes componentes: O agendador de eventos mantém uma lista com os eventos à espera de acontecerem, sendo executado antes de qualquer evento, podendo ser chamado durante a execução de um evento para agendar um novo. Este componente tem um impacto significativo na eficiência do simulador. Para cada simulação existe uma variável global que representa o tempo simulado, denominada por relógio de simulação. O mecanismo de avanço no tempo permite avançar o tempo incrementando o tempo para o próximo primeiro evento.

A execução de um evento é definida pela sua rotina que actualiza as variáveis de estado do sistema e agenda outros eventos. De forma a que os parâmetros do modelo possam ser alterados, são usadas rotinas de entrada que carregam estes parâmetros. As rotinas de inicialização permitem definir o estado inicial das variáveis do sistema. As rotinas de *trace* permitem verificar o estado intermédio das variáveis durante a execução da simulação, ajudando na detecção de erros. No fim da simulação, é utilizado um gerador de relatórios que calcula o resultado final imprimindo-o num formato específico. O programa principal junta todas as rotinas, inicia a simulação e executa as várias iterações. Como o número de entidades na simulação muda constantemente, é necessário fazer uma gestão dinâmica da memória correndo, periodicamente, um *garbage collector*.

Este tipo de simulação é fortemente utilizado na avaliação de sistemas de grandes dimensões, abrangendo a simulação de variados recursos.

Um exemplo da utilização desta técnica é o OMNeT++ [16]. Esta ferramenta é essencialmente orientada para redes, mas possui uma vasta biblioteca para simulação de discos, sistemas de ficheiros e outros componentes de interesse. A sua desvantagem consiste na falta de um encaminhamento directo para a atribuição de componentes de *middleware* existentes.

O NS2 [8] é também um simulador de redes usado na simulação de protocolos de encaminhamento, fortemente usado em pesquisa de redes *ad-hoc*, fornecendo um suporte substancial para a simulação de protocolos TCP, *multicast* sobre redes com e sem fios. Esta ferramenta permite explorar diferentes cenários de uma forma rápida como alteração de parâmetros ou configurações, dando preferência ao tempo de iteração ao invés do tempo de execução. Ao usar uma única *thread* para executar a simulação, restringe a execução de eventos a um de cada vez, não permitindo execuções parciais. Ao usar segundos como unidade de tempo na agenda de eventos, os resultados obtidos pela ferramenta, deixam de ser tão precisos.

Outros sistemas como RacewaySSF [6], são orientados para núcleos de simulação permitindo o uso de modelos de simulação para substituir os componentes indisponíveis. Esta ferramenta faz uso de múltiplos núcleos de CPU para paralelizar o código de simulação e, consequentemente, executar os modelos mais depressa.

Tal como o RacewaySSF, JiST (Java in Simulation Time) [4], é orientado para núcleos de simulação. Este evita os gastos das *threads* nativas por cada *thread* de simulação usando continuações. Contudo, não virtualiza as APIs do Java limitando o código Java

capaz de ser executado, e como tal, não reflecte o gasto do código Java no tempo de simulação.

O sistema de simulação Neko [14], permite a reutilização dos modelos de simulação como concretizações reais. Esta característica permite que a aplicação a testar possa ser executada ainda na fase de desenvolvimento substituindo os recursos em falta pelos respectivos modelos de simulação. Apesar destas vantagens, utiliza uma API orientada a eventos e não as classes padrão do Java. Utiliza ainda um modelo simples para ajustar o custos da comunicação e da computação das operações, cujos resultados não representam o custo real da execução.

Combinar componentes reais com simulados é também frequente para a avaliação de software. ModelNet [15] é um exemplo de uma ferramenta para simulação de redes, desenhada para correr em *clusters* locais, simulando redes WAN em LAN. Esta usa um ou mais *clusters*, que funcionam como máquinas de encaminhamento, para controlar todo o tráfego da rede proporcionando um ambiente centralizado, no qual os programadores podem ver e controlar toda a execução. Esta simulação corre em tempo real com uma precisão de milisegundos. Contudo, apenas consegue avaliar sistemas completamente desenvolvidos, não suportando desenvolvimento parcial.

Um exemplo diferente é o do FAUmachine [12]. Trata-se de um sistema de virtualização capaz de simular vários componentes de hardware. É normalmente usado para componentes de sistemas operativos, e portanto não focada para sistemas distribuídos.

Uma ferramenta de simulação mais completa é a CESIUM [3], que permite o uso de recursos simulados e consegue reproduzir a execução do código real com grande precisão. Fornece virtualização de *threads* e primitivas de controlo para concorrência o que resulta em detalhe adicional quando código concorrente é simulado. A desvantagem desta ferramenta é o uso do método `currentTimeMicros()` para temporizar a execução de código, uma vez que implica acesso a recursos que podem ser susceptíveis a interferências com tarefas em segundo plano. Esta ferramenta foi desenvolvida com base na versão Java 1.0 encontrando-se agora indisponível.

UMLSim [2] é uma abordagem similar à CESIUM. Consiste num supervisor baseado em Linux, que virtualiza o Linux enquanto fornece uma linha de tempo e rede simuladas. Esta ferramenta corre uma JVM dedicada para cada máquina virtual, o que provoca um gasto significativo e limitando a escalabilidade dos testes que podem ser executados. Para além disso, o código não foi mantido e é restrito a antigas versões do Linux.

Para além da simulação e das técnicas anteriormente referidas, existe outra forma de avaliar aplicações. Estas consiste em sistemas reais de avaliação que fornecem um ambiente real que substitui o ambiente a ser usado pela aplicação.

Um exemplo tradicional deste tipo de sistemas, são as redes *ad-hoc* onde testes podem realizados. Este tipo de redes é bastante usado para testar pequenas aplicações distribuídas. Como apenas representa uma rede local configurada com um número reduzido de *hosts*, quando utilizada para testar grandes aplicações, os resultados obtidos não são muito representativos.

A ferramenta Emulab [9], é orientada à simulação de redes, fornecendo um conjunto dedicado de nodos computacionais e hardware de rede. A sua vantagem é a capacidade de assumir diferentes topologias de rede e configurações de acordo com as necessidades da aplicação, através da configuração dinâmica dos nodos e das camadas de rede. Sendo um ambiente centralizado, permite o controlo da execução na sua totalidade.

Uma abordagem mais radical deste tipo de soluções é o PlanetLab [11], que fornece uma grande plataforma para executar qualquer tipo de código. Trata-se de uma ferramenta de geograficamente distribuída composta por um conjunto de *clusters*. Esta característica permite aumentar o realismo da simulação do software ao fornecer vários nodos numa configuração espalhada mundialmente. Para isto, faz uso da noção de *virtualização distribuída*, atribuindo uma fatia dos seus recursos ao *middleware*. A noção de fatia consiste num conjunto de nodos onde a aplicação recebe uma fracção dos recursos de cada nodo na forma de uma máquina virtual. Apesar de todas esta características, a configuração e a execução de experiências no PlanetLab podem ser, só por sim, um desafio. Sistemas como o Splay [13] tornam a configuração, execução e a observação muito mais fáceis. Mas é limitada a uma ferramenta específica e desenvolvida na linguagem Lua.

2.3 Sumário

Neste capítulo são referidas as várias técnicas de avaliação das quais a simulação se mostra ser a forma mais adequada para avaliar sistemas de grandes dimensões como é o objectivo desta dissertação. Dentro desta técnica existem várias abordagens, das quais a utilização de simulação por eventos sobressai notoriamente, uma vez que as abordagens utilizadas pelo método Monte Carlo são orientadas para problemas do campo probabilístico. A utilização de *traces* limita em muito a eficiência de avaliação, impedindo a

realização de testes dinâmicos.

Resta portanto, a emulação e a simulação por eventos. A primeira opção trata-se de uma simulação demasiado específica com o objectivo de simular o comportamento completo de um determinado sistema. Obrigando por isso, à utilização de um modelo de simulação muito detalhado, impedindo o seu fácil reaproveitamento e flexibilidade para o teste de diferentes ambientes. A segunda opção mostrou-se ser a melhor escolha para a avaliação. Esta opção permite focar a avaliação em determinadas características do sistema, permitindo uma maior flexibilidade no detalhe da simulação através do uso de modelos que simulam os componentes do sistema.

A simulação por eventos pode ser comparada com outro tipos de sistemas de avaliação. Estes consistem em sistemas que fornecem uma avaliação mais geral do comportamento das aplicações. O objectivo é fornecer um ambiente real onde a aplicação será testada. Como exemplo, o Emulab fornece uma rede real, ainda que centralizada, que pode assumir várias configurações consoante a topologia da rede destinada à aplicação. Já o PlanetLab fornece um ambiente real dispondo de nodos de rede espalhados geograficamente, o que permite que a aplicação possa ser testa num ambiente bastante realista.

Para além do realismo adicionado à avaliação, estes sistemas requerem uma configuração bastante complicada quando comparados com sistemas de eventos de simulação, e uma maior dificuldade na gestão da aplicação.

Como visto nenhuma das soluções apresentadas fornece um equilíbrio entre as várias características da simulação. O objectivo desta dissertação, é a apresentação de um modelo de simulação de rede a ser usado pelo MINHA de modo a permitir aumentar a capacidade desta ferramenta. Este modelo tem como objectivo simular, de uma forma simples, o comportamento real de qualquer rede sem a adição de grande *overhead* ao simulador. Desta forma pretende-se responder à falta de soluções de modo a permitir uma avaliação precisa e abrangente aos vários componentes do sistema.

Capítulo 3

A Plataforma MINHA

Como visto no capítulo anterior, existem várias ferramentas que permitem testar sistemas distribuídos, contendo diferentes características que os diferencia a todos, focando-se em características específicas. Normalmente um *middleware* distribuído lida com várias instâncias dos seus intervenientes como mostra o esboço da Figura 3.1.

Neste capítulo é apresentada uma alternativa a estas ferramentas, o MINHA. Esta permite reproduzir uma execução distribuída usando apenas uma única JVM como mostra a Figura 3.2, onde as diferentes JVMs da aplicação são representadas por JVMs virtuais que executam sob o controla da única JVM.

Para isso utiliza um núcleo de simulação que permite controlar a virtualização do tempo de simulação, a execução de eventos e mediar o acesso aos recursos do sistema, eliminando as interferências na competição por recursos partilhados.

Através do uso de múltiplas *threads* e primitivas de sincronização, permite a execução de código concorrente adicionando detalhe ao modelo de simulação. Uma vez desenvolvido em ambiente Java, permite a fácil integração de aplicações que utilizem esta linguagem sem a necessidade de alterar o código fonte, uma vez que, tanto as classes da aplicação como do Java, são carregadas a partir de uma classe de inicialização costumizada que substitui as bibliotecas nativas e o *bytecode* de sincronização por referências do modelo de simulação. A execução da simulação nestas condições pretende fornecer as seguintes características:

Observação global sem interferência Como todo o processo é centralizado, é possível obter uma observação global de toda a execução e das variáveis de sistema. Como a

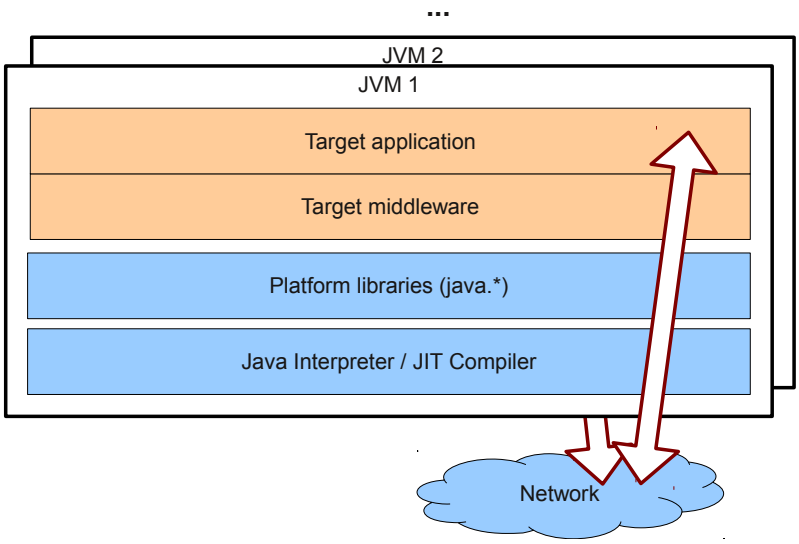


Figura 3.1: Aplicação Java distribuída.

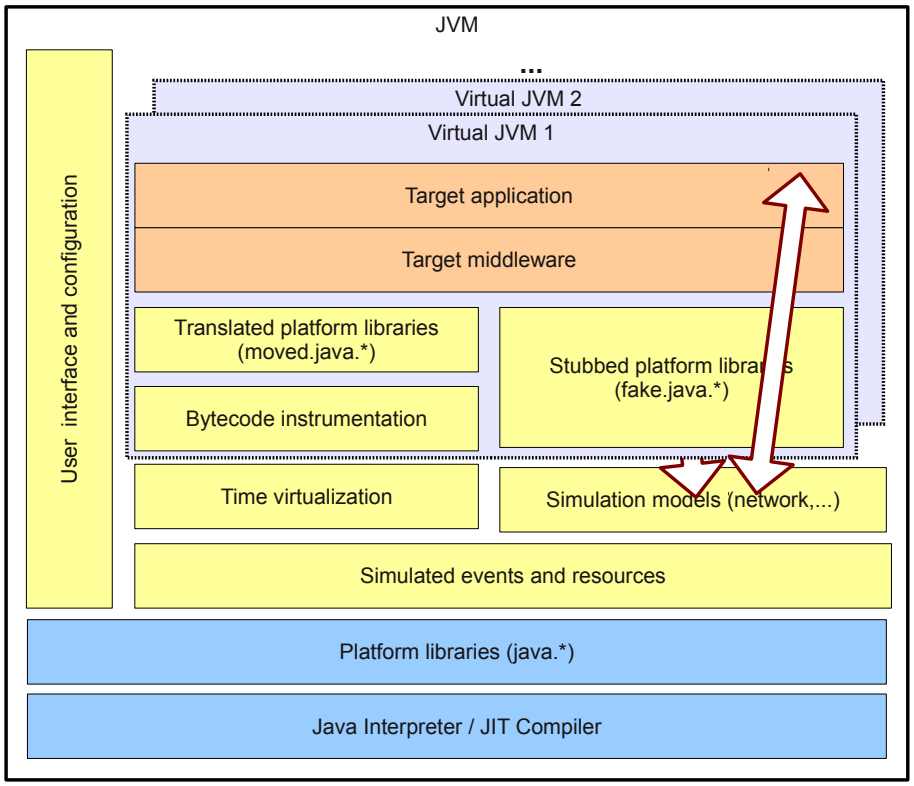


Figura 3.2: Simulação de uma aplicação Java distribuída.

instrumentação está do lado do simulador, sempre que seja preciso depurar a execução, o seu custo não é contabilizado, ao contrário da execução num ambiente real, sendo que, ainda nesta situação o tempo de execução pode ser considerado para análise.

Componentes simulados Quando em fase de desenvolvimento, é necessário avaliar o sistema para diferentes ambientes (ex. configurações de rede) e componentes de software (ex. a camada da aplicação no topo do *middleware*). Com o MINHA, estes ambientes e modelos de software podem ser substituídos por modelos de simulação, e incorporados sistemas de teste para serem automaticamente executados conforme a evolução do código.

Larga escala Aplicações de larga escala, que requeiram um grande número de recursos presente no ambiente real, tornam o desenvolvimento mais complicado. Como tal, testar durante a fase de desenvolvimento pode obrigar a que parte do sistema esteja já operacional, o que implica grandes custos já nesta fase. Algumas aplicações podem até lidar com informação sensível, o que obriga a testes exaustivos de forma a garantir a sua correcção para salvaguardar tal informação.

Análise "e se" automática Recorrendo a componentes simulados e executando o sistema com parâmetros variáveis, é possível avaliar o impacto induzido por ambientes extremos, explorando ainda condições que não seriam possíveis na prática.

Este capítulo está divido em duas secções principais: núcleo de simulação e JVM virtualizada, onde a primeira aborda a mediação do tempo de simulação, a gestão de eventos e de recursos, e a segunda explica como é feita a adaptação da API da plataforma Java ao simulador através da manipulação de *bytecode*, e como são controladas questões como código concorrente.

3.1 Núcleo de simulação

O núcleo é composto por duas camadas, onde a primeira oferece apenas primitivas de simulação por eventos e para a gestão de recursos abstractos. O segundo fornece as bases para a combinação de código real com código simulado, (i) medindo o tempo de execução e gerindo um processador simulado; e (ii) permitindo a execução sequencial de código Java eliminando a inversão de controlo que resulta da simulação de eventos.

3.1.1 Event e Resource

A simulação de eventos está organizada em duas classes. A classe `Event` é o ponto principal da simulação visto que define a evolução da execução. Cada evento corresponde a um passo na simulação, executando atomicamente num determinado tempo de simulação. Cada instância desta classe define um método `run()` onde é especificada a acção do evento, o método deste código difere de evento para evento, conforme a sua função.

Para executar, o evento tem primeiro, de ser agendado na fila da linha de tempo na posição correspondente ao seu tempo de execução. Imediatamente antes de executar, o evento é removido da lista, iniciando assim a sua execução. Um evento pode agendar-se a si próprio para um tempo futuro, como pode agendar e cancelar outros eventos mas não pode, nunca, agendar para um tempo passado. A simulação é iniciada agendando pelo menos um evento, que origina a agenda dos eventos seguintes, permitindo a evolução da simulação.

O agendamento de eventos é feita obtendo o tempo actual da linha de tempo de simulação, a qual é representada pela classe `Timeline`. Esta classe representa o tempo global de simulação, gerindo o seu progresso ao longo da execução. Para isso, utiliza uma lista ordenada de eventos, `java.util.PriorityQueue`, tal como é normalmente utilizada para simulação de eventos. Estes eventos são agendados para instantes precisos no futuro da simulação, assegurando que os eventos executam ordenadamente de acordo com o seu tempo de agenda. O tempo dito actual, é o tempo relativo ao evento que executa actualmente, isto é, o menor tempo de toda a lista. O tempo virtual avança na linha de tempo, saltando no tempo conforme os eventos são executados. Quando todos os eventos da lista forem removidos, a simulação termina.

A classe `Resource` fornece primitivas de controlo básico para os recursos do sistema. As primitivas `acquire` e `release` permitem marcar um recurso como ocupado ou livre respectivamente. Esta classe tem uma lista com os eventos que pretendem adquirir o recurso. Se o recurso estiver ocupado, o evento que o tenta adquirir é adicionado à lista onde permanece até ser o primeiro elemento da lista e consequentemente, que o recurso seja libertado. Recursos concretos, como núcleos de CPU simulados, podem ser implementados, bastando instanciar ou estender a esta classe.

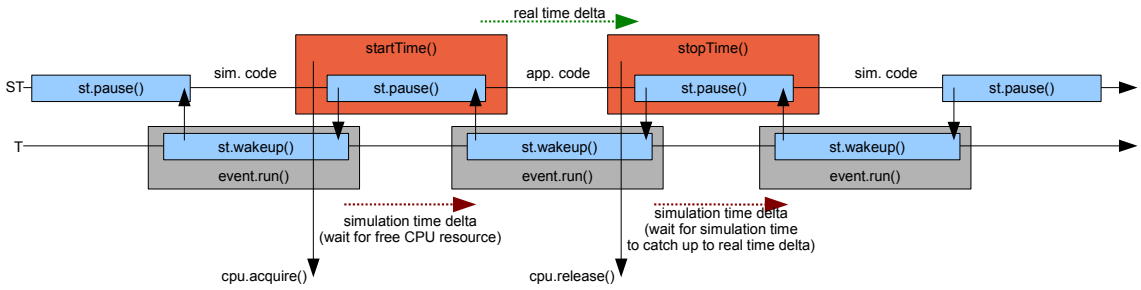


Figura 3.3: Evitar a inversão de controle e obtenção do tempo de virtualização.

3.1.2 Virtualização do tempo

A virtualização do tempo permite ao simulador executar a aplicação, da mesma forma que esta correria no ambiente real, tendo sempre em conta o tempo real. Para que isso possa ser feito é mantida uma `SimulationThread` (ST) para cada *thread* de controlo que apenas executa quando um seu evento de simulação adquire o CPU. Sendo assim possível, obter concorrência, cruzando a execução de pedaços de código de acordo com o tempo virtual. Cada ST está também associada a um recurso que representa o processador. A ocorrência de um evento surge sempre entre duas invocações do método `pause()` que bloqueiam a execução da *thread* (Figura 3.3).

Para que a ST possa voltar a avançar, antes de ser bloqueada, agenda ou guarda numa ou mais listas um evento de simulação *wake-up* que ao ser executado invoca o método `wakeup()` que desperta a execução da *thread*. A execução de código real é sempre feita entre a invocação de dois métodos: o método `startTime` marca o inicio da execução, permitindo adiá-la para um instante de simulação em que o processador está livre. Enquanto a execução decorre o seu tempo é medido e o processador é marcado como ocupado. A execução pára quando é invocado o método `stopTime`, e o tempo de simulação avança consoante o intervalo de execução.

O tempo de execução é medido com uma *thread* local que virtualiza o relógio de ciclos do CPU. Isto evita a interferência de outras *threads* que correm no mesmo *host* e permite obter um custo preciso das instruções actualmente executadas pela *thread* sobre o controlo do simulador.

3.2 Virtualização da JVM

Sendo o tempo um factor fundamental na avaliação do *middleware*, é importante que a sua reflexão na simulação seja o mais precisa possível. Para isso é preciso ter em atenção a execução de operações bloqueantes, tais como sincronização de *threads* e operações de *input/output*, as quais deverão ser evitadas. Quando são usadas devem ser traduzidas para as primitivas de simulação. Além disso, código que execute em diferentes instâncias virtuais não pode interferir directamente através de variáveis partilhadas.

Estes objectivos são alcançados reescrevendo as classes que possam levar a estes problemas e introduzindo invocações ao núcleo de simulação apropriadamente. Isto é feito através de uma classe de carregamento que usa a manipulação de *bytecode* da plataforma ASM [5].

Para criar diferentes JVMs virtuais e isoladas entre si, é usada uma instância separada desta classe de carregamento para cada uma delas. Um subconjunto de classes, que contêm o núcleo de simulação e os modelos de ambiente, é mantida de forma global, encarregando a sua inicialização à classe de carregamento do sistema. Esta configuração permite um canal de controlo que possibilita a interacção entre as JVMs virtuais.

3.2.1 Bibliotecas da plataforma

A reescrita das classes não é tão simples como pode parecer, pois por razões de segurança todas as classes pertencentes ao pacote `java.*` não podem ser reescritas, sendo que algumas delas precisam realmente de ser virtualizadas. Para ultrapassar este problema, todas as referências para essas classes são substituídas sendo encaminhadas para *stubs* do pacote `fake.java.*`, que substituem as classes originais.

Este é o caso de todas as classes que contêm métodos nativos, elementos estáticos e métodos sincronizados. Os métodos nativos são particularmente relevantes, uma vez que acedem directamente aos recursos do sistema que precisam de ser controlados e quando usados pelas instâncias de simulação têm de ser mantidas dentro da sua zona segura respeitando o objectivo da simulação. Por exemplo, o método `System.nanoTime()` retorna o tempo actual do sistema em nanosegundos. Quando em ambiente de simulação, este método não retornará o valor adequado, o tempo actual da simulação, mas o do sistema. A sua tradução pode ser encontrada no pacote `fake.java.lang.System`. A substituição destas classes consiste em adicionar e converter algumas das suas operações

para primitivas de simulação. Por exemplo, de forma a evitar que a execução bloqueie na tentativa de aceder a um ficheiro durante a execução, estas acções são encapsuladas pela simulação.

A solução da reescrita de classes não pode, porém, ser aplicada a todas as classes. A classe `java.lang.String` não permite nenhuma alteração, mas como não tem nenhum membro estático que possa perder informação entre as instâncias de simulação, não representa nenhum problema.

3.2.2 Sincronização

As sincronizações de *threads* feitas através das primitivas do pacote `java.util.concurrent.*` são fáceis de lidar, bastando redireccionar para as suas homólogas no pacote `fake.java.util.concurrent.*`. O desafio que resta consiste em interceptar e traduzir as operações de *bytecode* dos monitores Java nativos e dos pares de *mutex*/*condition variable* em cada objecto. Para isto, é adicionado um `fake.java.lang.Object` a todas as instâncias criadas, poupando tempo na procura das classes que usam sincronização. Assim, é evitado o acesso directo ao sistema e o mecanismo dos monitores é substituído pela gestão de acesso a esse objecto. As operações do monitor, são direccionadas para as primitivas de sincronização da classe `Object`, nomeadamente `lock`/`unlock`,`wait`, and `notify`.

Os métodos `static synchronized` são um caso particular da sincronização na simulação. Como os membros estáticos são associados a uma única classe e não a uma instância especifica, o `Object` extra, não resolve o problema. A solução passa por adicionar um campo à sua classe que contém uma instância *singleton* de um objecto usado para a sincronização.

Esta solução, porém, é bastante custosa dado que obriga à criação de dois objectos adicionais para cada objecto da aplicação e à execução de dois eventos de simulação por cada operação de sincronização. Como as variáveis `lock` e `condition` não são usadas na maioria dos objectos da aplicação, a sua criação pode ser adiada apenas para quando forem usadas pela primeira vez. Criando uma via rápida nas operações de sincronização evita os eventos de simulação a não ser que haja contenção.

Ainda assim, esta abordagem continuará a ser impraticável em dois casos. O primeiro é referente à sincronização de operações sobre objectos que não podem ser traduzidos,

nomeadamente, `arrays` cujo bloqueio não garante que se estenda aos seus elementos. Este problema é resolvido usando uma `HashTable` que contem *objectos sombra* dos elementos que são usados quando o caminho predefinido falha. O segundo refere-se a sincronizações implícitas na construção das classes, que são necessárias para a implementação adequada dos *singletons*, que não podem ser substituídos, uma vez que a criação das classes é feita concorrentemente. A solução consiste em criar as classes de forma sequencial onde as referências são passadas correctamente.

3.2.3 Classes `moved`

Como a maioria das classes nas bibliotecas da plataforma são 100% Java, podem ser traduzidas da mesma maneira que o código do utilizador no *middleware* e aplicações, tal como acontece com as estruturas no pacote `java.util.*`. Algumas classes têm sincronização interna que precisa de ser interceptada, por exemplo a classe `java.util.Hashtable` que tem métodos concorrentes como `get` e `put`. Outro exemplo são as classes `java.io.OutputStream` e `java.io.InputStream` que bloqueiam entre escritas e leituras.

Para garantir que as operações de sincronização funcionam realmente, as primitivas de sincronização são convertidas nas primitivas de sincronização da simulação. Este trabalho evita a reescrita de classes e simplifica o trabalho do simulador que analisa e processa estas classes automaticamente. Todas estas classes passam a fazer parte do pacote `moved.java.*`, e as suas referências são actualizadas adequadamente.

3.3 Utilização

Para utilizar o MINHA para avaliar uma aplicação distribuída ou componente de *middleware* é preciso proceder à configuração de um número de instâncias de JVMs virtuais. Esta configuração é feita através da uma única linha de comandos, pela qual são passados os parâmetros de configuração usados na linha de comandos de cada instância. Estes dados são depois tratados pelo simulador, atribuindo a cada configuração uma JVM virtual. Desta mesma forma, também são passados alguns parâmetros do MINHA usados para ajustar as execuções dos componentes da aplicação.

A incorporação de uma aplicação distribuída no MINHA é uma tarefa bastante simples.

Isto porque, como só corre num único *host* a sua configuração e execução torna-se mais fácil de realizar. Se fossem necessários vários *host*, cada um deles teria de ser configurado individualmente.

No total, existem quatro tipos de argumentos que poderão ser fornecidos: as várias classes *main* da aplicação que representam as instâncias dos diferentes *hosts*, que serão associadas, cada uma, a uma JVM virtual individual. Os endereços de IP dos *hosts* que deverão ser atribuídos a cada virtual *hosts*. Um valor temporal que adicionará um intervalo entre a execução de cada instância, e o número de instâncias a executar de cada classe. Todos estes argumentos são opcionais com a excepção das classes *main* que são fundamentais para executar a simulação.

Classes

```
$ java minha.Run HelloWorld
```

Neste exemplo está a linha de comandos que executa um simples programa *HelloWorld*. Aqui, apenas uma JVM virtual é iniciada.

Endereço IP O exemplo de uma simulação de uma simples aplicação de entre cliente/servidor, na qual são passados endereços de IP como parâmetros:

```
$ java minha.Run 10.0.0.1 EchoServer, 10.0.0.2 EchoClient 10.0.0.1
```

Neste exemplo, duas JVMs virtual são criadas, uma para cada classe. A primeira executa a instância da classe `EchoServer` num nodo de simulação ao qual é atribuído o endereço de IP 10.0.0.1. A segunda executa com a instância `EchoClient` num nodo com o endereço IP 10.0.0.2. Esta classe recebe ainda, como argumento de linha de comandos, o endereço IP do `EchoServer`.

De notar que tanto a aplicação `EchoServer` como `EchoClient` são programas em Java que fazem uso da API padrão e correm directamente em JVMs não modificadas. Por exemplo, para obter o mesmo efeito sem o MINHA, era preciso correr o seguinte comando numa máquina real com o endereço 10.0.0.1:

```
$ java EchoServer
```

e o comando seguinte numa segunda máquina também real:

```
$ java EchoClient 10.0.0.1
```

Atraso de arranque No exemplo anterior pode ser necessário ter de esperar pelo arranque do servidor antes do cliente executar. O parâmetro `-d` permite adiar a execução durante os segundos indicados.

```
$ java minha.Run 10.0.0.1 EchoServer, \
             -d=3 10.0.0.2 EchoClient 10.0.0.1
```

Neste caso, o cliente apenas iniciará três segundos depois do servidor arrancar.

Instâncias Num ambiente cliente/servidor, normalmente há a necessidade de criar múltiplas instâncias da classe `EchoClient`. Estas podem ser criadas explicitamente, ou implicitamente. Quando a criação das instâncias é deixada à responsabilidade do simulador, basta indicar o número de instâncias que queremos correr. Isto é bastante prático quando é preciso correr um grande número de instâncias.

```
$ java minha.Run 10.0.0.1 EchoServer, \
             -d=3 2 EchoClient 10.0.0.1
```

Aqui, é criada uma instância `EchoServer` e duas `EchoClient`. A desvantagem desta opção é a impossibilidade de atribuir um IP especifico a cada uma dessas instâncias. E como tal, estes serão atribuídos posteriormente pelo simulador. Se for necessário especificar endereços IP para cada instância, a criação de cada uma tem de ser feita manualmente:

```
$ java minha.Run 10.0.0.1 EchoServer, \
            -d=3 10.0.0.2 EchoClient 10.0.0.1, \
            -d=3 10.0.0.3 EchoClient 10.0.0.1
```

Este exemplo, cria três JVMs virtuais, um nodo com uma instância `EchoServer` com o endereço IP 10.0.0.1, e dois nodos, com uma instância cada um, da classe `EchoClient`, as quais com endereços IP 10.0.0.2 e 10.0.0.3, respectivamente, e ambos recebendo como atributo o endereço IP do servidor.

3.4 Sumário

Neste capítulo é apresentada a ferramenta de simulação MINHA. Como foi dito anteriormente esta ferramenta tenta criar um equilíbrio entre as suas características. A simulação criada por si, é totalmente centralizada através da virtualização de JVMs, possibilitando o total controlo e observação de toda a execução. Ao proceder a toda a simulação numa única *thread* evita o peso que JVMs reais provocariam no poder de processamento, e consequentemente suporta um maior número de testes comparativamente com o uso de JVMs reais. A utilização de um núcleo de simulação permite moderar e controlar toda a execução bem como os acessos aos recursos, garantindo a sequência normal da execução. Suporta também, modelos de simulação usados para substituir componentes que possam não estar disponíveis, Como por exemplo, modelos de rede.

Capítulo 4

Modelo de Rede

Como o MINHA é uma ferramenta que visa avaliar sistemas distribuídos é necessário incluir de alguma forma esta componente da rede na avaliação e para isso, é usado um modelo de simulação baseado nas propriedades das redes reais, com o objectivo de imitar os seus comportamentos.

O objectivo deste modelo é apenas fornecer uma abstracção suficiente para permitir que a aplicação seja testada considerando os custos da comunicação resultando numa avaliada bastante precisa. Como tal, o modelo abdica do detalhe por uma melhor performance na simulação da aplicação. Características como a arquitectura da rede, o número de *routers* ou *subnets* não são consideradas, apenas são consideradas características directamente relacionadas com a transmissão de dados, como o tempo necessário para o envio de pacotes e as taxas de frequência das suas potenciais perdas.

A inclusão do modelo de rede no MINHA, obriga a que este respeite algumas das suas características. A actividade gerada na rede tem de ser feita através de eventos de simulação, permitindo assim, que a execução possa ser completamente controlada. Seguindo a filosofia adoptada no MINHA, o modelo de rede deve permitir uma simples integração com o *middleware*, sem que seja necessário alterar o seu código fonte, para isso a API da plataforma Java é mantida permitindo criar transparência na interacção com o modelo.

Este capítulo divide-se em quatro partes: numa primeira parte é explicada a arquitectura e o funcionamento do modelo de uma forma geral, as duas partes seguintes explicam o funcionamento de forma mais detalhada, mais concretamente o mapeamento e o controlo de fluxo na rede. Por último, é explicado de que forma as classes do pacote `java.net.*` são alteradas e adaptadas ao modelo de forma a fornecer os protocolos de

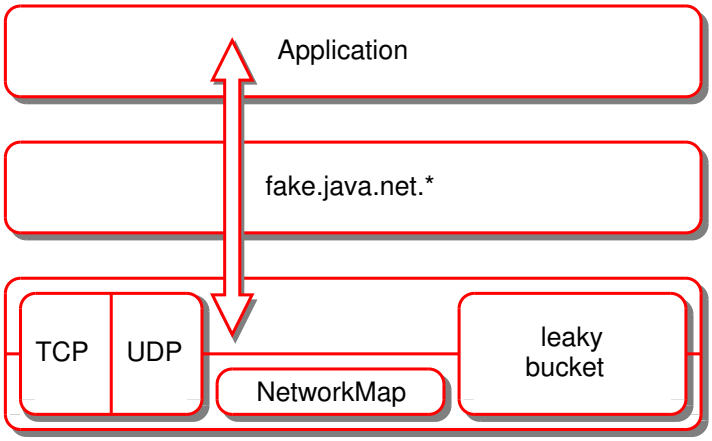


Figura 4.1: Camadas do modelo de rede do MINHA.

comunicação.

4.1 Arquitectura da rede

Na Figura 4.1 pode ser vista a arquitectura adoptada pelo modelo de rede. Este modelo representa um simples modelo de rede que visa, não simular completamente uma rede, mas apenas fornecer um modelo com abstracção suficiente para permitir que a aplicação ao ser testada com ele possa ser avaliada com bastante precisão.

A sua actividade é dividida em três camadas: a primeira representa a aplicação, a segunda camada representa a adaptação das classes do `java.net.*` nas do pacote `fake.java.net.*`, e a camada de rede que representa a gestão do modelo de rede, dividindo-se em outras três partes: os protocolos de comunicação, o mapeamento da rede e gestão do fluxo da rede.

Uma vez que este modelo é fechado, não estando sujeito a problemas de rede relacionados com o exterior, nomeadamente desconexões repentinas de *hosts*, pode ser visto como um *pipe* perfeito, no qual se existirem perdas de informação estas são devidas à configuração da rede e não a acções externas.

Quando a aplicação necessita aceder à rede, esta utiliza a API de rede do Java, sendo automaticamente redireccionada para as classes do pacote `fake.java.net.*`, que disponibiliza a mesma API, diferindo apenas na implementação dos seus métodos de forma a interagir com o modelo de rede. Através destas classes pode escolher qual o protocolo que pretende utilizar de entre TCP, UDP *unicast* ou UDP *multicast*.

Ao disponibilizar estes protocolos, o modelo precisa de garantir que as suas propriedades são garantidas, caso contrário, a utilização deste modelo não resultaria em resultados relevantes. O protocolo UDP não fornece nenhuma garantia em relação à entrega dos seus pacotes, já o TCP garante que todos os seus pacotes são entregue e pela mesma ordem pela qual são enviados. Estas garantias são fornecidas pelas classes que implementam os protocolos, e pela classe `sim.global.net.Network`, uma vez que esta controla o fluxo de transmissão de dados da rede. A transmissão dos dados é gerida pelo algoritmo *leaky bucket*, em que a ordem de transmissão interna de diferentes *streams* é determinada pelo algoritmo *round robin* de forma a assegurar justiça entre fluxos.

Para que um *socket* possa comunicar com um outro a rede precisa de saber onde este se encontra para lhe entregar a informação. Para isto, a rede precisa de fazer um mapeamento de todos os seus participantes, podendo assim verificar se determinado *socket* existe e qual o seu paradeiro.

4.2 Gestão de endereços e ligações

Em qualquer rede, para que seja possível estabelecer uma ligação entre dois pontos é necessário saber onde estes se encontram. Para possibilitar a sua localização é necessário manter um registo de todos os intervenientes na rede, identificando-os com um identificador único de modo a permitir a sua distinção. O identificador utilizado é o seu endereço de rede. Para uma melhor compreensão, o endereço de rede, IP:porto, será referido apenas como *endereço*. Qualquer referência específica a cada um dos seus elementos será devidamente referida. O registo consiste em mapear todos os elementos da rede, facilitando a sua organização e o controlo do seu acesso.

O mapeamento no modelo de rede do Minha é da responsabilidade da classe `sim.global.net.NetworkMap`, esta classe utiliza várias estruturas de dados para guardar diferentes tipos de referências necessárias ao mapeamento da rede, permitindo verificar se um determinado endereço está registado na rede e se existe uma ligação entre dois elementos.

Durante o arranque do Minha, os vários *hosts* representados por JVMs virtuais tentam efectuar o seu registo na rede. Os seus endereços podem ser atribuídos como parâmetro no arranque da simulação. Se não forem fornecidos, os endereços podem ser atribuídos pelo simulador, escolhendo endereços IP que não estejam em uso. Os endereços atribuídos pelo sistema são da gama: $10.X.X.X$, onde $X = [0, 255]$. Depois de adquirido, o *host* pode

então registar-se na rede. Em relação ao endereço MAC este não pode ser passado como parâmetro, sendo automaticamente atribuído pelo MINHA no arranque da simulação. Tal como para os *hosts*, o registo dos *sockets* consiste em armazenar o seu endereço num dos *maps* existentes para o registo dos *sockets*, o *map* onde é registado depende do protocolo a que este pertence.

No caso particular dos *sockets* TCP, é preciso estabelecer uma conexão para comunicar entre dois *sockets*. Para isso, o *socket* que pretende iniciar a comunicação tem de fornecer à classe `NetworkMap` o endereço do *socket* de destino. Com este endereço, a classe consulta o *map* que contém os *sockets* registados, obtendo a referência do *socket* de destino e enviando-lhe o pedido de conexão. Entretanto o destinatário aceita a conexão e envia essa informação à fonte.

Quando um pacote é enviado para a rede leva consigo o endereço do destinatário. Tal como no caso anterior, é através deste que a classe `NetworkMap` descobre onde se encontra o *socket* a quem se destina o pacote.

4.3 Fluxo da rede

A principal função de uma rede é permitir a troca de informação entre os seus intervenientes. De forma a assegurar que esta troca é feita com sucesso, é necessário controlar o acesso aos recursos da rede. No caso do modelo de rede, o acesso à rede é também controlado, permitindo limitar a largura de banda disponível conforme a configuração do modelo. Como tal, se a utilização da largura de banda chegar ao limite definido, o acesso à rede passa a ser condicionado impedindo os pacotes de serem transmitidos. No modelo de rede do MINHA o controlo do fluxo é protagonizado pela classe `sim.global.net.Network`, onde o controlo de acesso é gerido pelo algoritmo *leaky bucket*, em que a ordem da transmissão interna de diferentes *streams* é determinada pelo algoritmo *round robin* de forma a assegurar justiça entre fluxo. O recurso ao algoritmo *leaky bucket* permite garantir as propriedades do protocolo TCP e ao mesmo tempo simular o tratamento dado aos pacotes UDP. De forma a poupar recursos do sistema, o controlo da transmissão de dados é feito periodicamente, quando existem dados a ser enviados.

A transmissão de pacotes é limitada por uma taxa de transmissão pré-definida, dependendo da largura de banda que se pretende simular como disponível. Se a largura de banda for inferior à taxa de transmissão, somente o número de bytes correspondente ao

espaço disponível é transmitido. Os restantes bytes, ficam retidos num *buffer* à espera de largura de banda suficiente para serem enviados. Este *buffer* é limitado, e quando este atinge o seu limite o tratamento que é dado aos pacotes que entretanto vão chegando depende do seu protocolo: se o pacote for do tipo UDP, é automaticamente perdido sem a possibilidade de ser recuperado, se for do tipo TCP é armazenado numa `LinkedList` de forma a garantir as propriedades do protocolo correspondente. Se na chegada de um pacote TCP a lista não estiver vazia, este é lhe inserido em vez de entrar no *buffer*. A utilização desta lista permite respeitar as propriedade do protocolo TCP, evitando que os pacotes se percam e assegurando que estes não sofrem nenhuma reordenação. Quando um conjunto de pacotes consegue largura de banda suficiente para a sua transmissão, o espaço que ocupa no buffer é libertado, e os primeiros pacotes da lista são adicionados ao buffer. Depois de transmitidos, o espaço que estes pacotes ocupam na largura de banda, é também libertado dando espaço a uma nova transmissão.

Em ambientes reais a transmissão da informação não é imediata. Como tal, o modelo aplica atrasos, que podem ser configurados, na transmissão dos dados assim que ganham espaço de largura de banda e são enviados. Desta forma, é possível simular a transmissão de informação numa rede real.

4.4 Protocolos de comunicação

Diferentes ambientes exigem diferentes soluções nas garantias da transmissão de dados. Estas garantias são fornecidas pelos vários protocolos de comunicação, que definem com clareza a especificação das suas propriedades e as garantias a que estes se propõem. Este modelo de rede suporta dois destes protocolos, TCP e UDP, em que o UDP se divide em *unicast* e *multicast*.

Para que estes protocolos possam ser usados no modelo de rede do MINHA, o pacote `java.net.*` foi utilizado como base da implementação do pacote `fake.java.net.*`, aproveitando grande parte do código das suas classes. As diferenças consistem essencialmente na alteração dos seus construtores e de métodos mais complexos, nomeadamente, os bloqueantes e os que acedem à rede.

O acesso à rede, no modelo do MINHA, representa a passagem da execução de código da aplicação para código pertencente ao simulador, e como tal, considerado código de simulação. Os eventos de simulação que ocorrem nestas classes agem como notificadores

de eventos específicos, como a chegada de um novo pacote e alterações do estado da rede. Estes são normalmente criados antes do bloqueio da execução de forma a poderem ser agendados quando estes eventos ocorrem, de forma a retomar a execução.

A classe `fake.java.net.AbstractSocket`, que contém todos os métodos e variáveis que são comuns a todos os *sockets*, é estendida por todas as classes de *sockets*. Por exemplo, quando um qualquer *socket* é criado, esta classe verifica o seu endereço de forma a garantir que este se encontra dentro dos valores correctos. O uso desta classe é bastante útil, pois permite criar acções generalizadas sobre *sockets* abstraindo-as do protocolo a que pertencem, e como tal, se for necessário adicionar um tipo extra de *socket*, estas acções continuarão a assumi-lo.

Esta secção é dividida em duas partes: a primeira explica de que forma as classes referentes ao protocolo TCP foram adaptadas, e a segunda refere como o mesmo processo foi feito para o protocolo UDP.

4.5 Protocolo TCP

O protocolo TCP permite trocar mensagens entre dois *sockets* garantindo que estas são todas entregues ao destinatário pela ordem que foram enviadas. No MINHA foi criada a classe `fake.java.net.Packet` para encapsular dados em trânsito, a qual implementa a interface `sim.global.net.PacketInterface`. Esta interface permite a introdução de novos tipos de pacotes TCP, bastando que as suas classes a implementem. Todos estes pacotes são marcados com um número sequencial (*sn*) que identifica a sua ordem, e através deste é confirmada a sua ordenação.

Como estes pacotes podem ter diferentes funções são distinguidos conforme o seu tipo:

- `Data`: para transportar informação
- `Ack`: para confirmar entrega de determinado pacote
- `Close`: para notificar o fecho do canal de comunicação

No Java, a utilização do protocolo TCP é feita através de duas classes: `java.net.Socket` e `java.net.ServerSocket`. Do lado do MINHA, estas classes são substituídas

respectivamente pelas `fake.java.net.Socket` e `fake.java.net.ServerSocket`. As restantes classes do pacote `java.net.*` são usadas normalmente sem a necessidade da sua alteração nesta fase. Em ambos os casos, os métodos mais básicos foram mantidos, apenas os que efectuam acessos à rede sofreram alterações.

Estas alterações dependem da classe a que os métodos pertencem, devido às diferentes funções das classes. A classe `Socket` tem a normal função de um *socket*, enviar e receber informação através da rede. Já a classe `ServerSocket`, tem apenas a função de aceitar as conexões pedidas pelas instâncias da classe`Socket`.

Os construtores destas classes foram ligeiramente alterados. As alterações consistem apenas na forma como se associam a um endereço específico e como efectuam o registo na rede. Para se associarem a um endereço, este tem de estar associado ao *host* ao qual pertencem. Se o endereço IP e porto não foram passados como parâmetros, o endereço IP é obtido através do seu *host*, assim como o porto garantindo que está disponível. O registo na rede, é feito automaticamente pelo construtor, que passa o endereço juntamente com a referência do seu *socket* à classe `NetworkMap`, que efectua o registo como foi referido na Secção 4.2.

ServerSocket Na classe `ServerSocket` o método que sofreu grandes alterações foi o método `accept`. A entrada neste método corresponde à passagem para código de simulação, o qual passa a funcionar da seguinte forma:

Tratando-se de código de simulação, a primeira tarefa a realizar é parar a contagem do tempo real através do método `SimulationThread.stopTime`. De seguida a execução é bloqueada pelo método `SimulationThread.pause` enquanto aguarda a chegada de pedidos de conexões, que quando chegam agendam um evento de simulação na *time-line* do simulador, o qual retoma a execução da thread. Já em execução, o método envia a confirmação da conexão ao *socket* de origem através da classe `NetworkMap` (Secção 4.2). Finalmente, a contagem do tempo real é retomada recorrendo ao método `SimulationThread.startTime`.

Socket Na classe `Socket` foi o método `connect` o mais alterado uma vez que, tal como na classe anterior, também faz acessos à rede:

A sua execução começa por parar o tempo real, visto também se tratar de código de simulação. De seguida um pedido de conexão é enviado a um `ServerSocket` através da

classe `NetworkMap`, a qual descobre o *socket* através do endereço fornecido, entregando-lhe o pedido. Enquanto aguarda pela confirmação da conexão, a *thread* que invoca este método, bloqueia-se à espera. Quando este chega é agendado um evento que a desperta, retomando a contagem do tempo real.

No Java, a classe `Socket` faz uso de dois *streams* para o envio e recepção de dados, as quais são representadas pelas classes `java.io.OutputStream` e `java.io.InputStream`, respectivamente. Como se tratam de classes bloqueantes, fazem parte do conjunto das classes `moved` referidas na Secção 3.2.3. Para ultrapassar este problema, foram criadas duas novas classes `fake.java.net.SocketOutputStream` e `fake.java.net.SocketInputStream`, as quais extendem respectivamente as anteriores. Dado que alguns métodos são herdados, como os que realizam acessos à rede, é preciso substitui-los para que haja interacção com a rede.

Ambas as classes têm um *buffer* onde são guardados os pacotes a ser enviados e recebidos que aguardam enquanto não são processados. Estes *buffers* permitem controlar a janela TCP, já que o seu tamanho corresponde ao tamanho da janela. Como tal o número de pacotes em transito, sem confirmação de entrega, são os que estão armazenados nestes buffers. Para melhorar a recepção dos pacotes, o *buffer* de leitura foi implementado como *buffer* circular. Os construtores destas classes recebem como parâmetro o *socket* ao qual foram associados.

SocketOutputStream A implementação desta classe consiste fundamentalmente na substituição do método `write` herdado. Esta alteração é necessária para possibilitar o envio de pacotes através do modelo de rede. Este método procede da seguinte forma:

Tal como nos métodos anteriores, os quais representam a passagem para código de simulação, e o tempo real é parado. De seguida o tamanho dos dados a enviar é verificado para assegurar que não excede o tamanho do buffer, se exceder é lançada uma excepção. Entretanto, é criado um pacote contendo os dados a enviar e o seu *sn*. Para que o pacote seja enviado, tem de haver espaço no *buffer* para ser guardado enquanto espera pela sua confirmação de entrega. Se este estiver cheio, a execução da *thread* é bloqueada, senão, o pacote é enviado para a rede através da classe `Network` (Secção 4.3). Quando o pacote é entregue o *socket* recebe um *acknowledgement* (*ack*), e o espaço ocupado por esse pacote no *buffer* é imediatamente libertado. Se entretanto, a execução foi parada, a recepção do *ack* provoca o agendamento do um evento de simulação de forma a que a execução possa prosseguir, tentando enviar os restantes pacotes, e prosseguindo, também, o tempo

de simulação.

SocketInputStream Esta classe substitui o método `read`, o qual permite receber pacotes vindos da rede, e cuja execução decorre da seguinte forma:

Primeiro é parada a contagem do tempo real, seguindo-se o bloqueio da execução até a chegada de um novo pacote. Quando este chega, a execução é retomada através de um evento de simulação. O *sn* do pacote é verificado de forma a garantir que não houve perdas ou reordenações de pacotes durante a transmissão. Antes do pacote ser passado ao *socket*, é enviado o *ack*, e posteriormente retomada a contagem do tempo de simulação.

4.6 Protocolo UDP

O protocolo UDP consiste no envio de mensagens sem fornecer nenhuma garantia acerca da sua entrega. Na plataforma Java a informação é enviada através de pacotes representados por instâncias da classe `java.net.DatagramPacket`. Uma vez que esta classe em nada interfere na utilização do modelo de rede, foi integralmente utilizada.

O protocolo de comunicação UDP está divido em dois outros protocolos: *unicast* e *multicast*, ambos suportados pelo modelo de rede do MINHA. O protocolo *unicast* é usado para comunicações entre apenas dois *sockets*, enquanto o *multicast* é usado para comunicação em grupo. É utilizada uma classe para o uso de cada um destes protocolos.

UDP *unicast* O protocolo *unicast*, na plataforma Java, é utilizado através da classe `java.net.DatagramSocket`. Tal como para o protocolo TCP, esta classe é substituída pela sua similar pertencente ao pacote `fake.java.net.*`, mantendo assim a mesma API alterando apenas os métodos, que acedem à rede, internamente.

O construtor foi alterado pelo mesmo motivo que os das classes TCP, e de forma similar. Os métodos mais críticos da classe `java.net.DatagramSocket` são o método `send` e `receive`. O método `send` permite o envio de informação pela rede para um destino específico. Como este protocolo não garante entrega dos seus pacotes, não necessita de aguardar pela confirmação da entrega dos mesmos. E portanto, a sua execução procede-se da seguinte forma:

Como se trata da passagem para código de simulação, o tempo real pára através do método `SimulationThread.stopTime`. De seguida é criado um pacote com a infor-

mação a enviar o qual é enviado para a rede, através da classe `Network` (Secção 4.3), juntamente com o endereço do *socket* de destino. E finalmente, o tempo real é retomado novamente com o método `SimulationThread.startTime`.

Por sua vez, o método `receive` consiste em receber os pacotes enviados:

Tal como no caso anterior, o tempo real é parado. A sua execução é bloqueada, através do método `SimulationThread.pause`, até à chegada de um novo pacote. Assim que este chega é agendado um evento de simulação na *timeline* de simulação para retomar a execução, e retomando, consequentemente, o tempo de simulação.

UDP *multicast* O protocolo UDP *multicast* é suportado pela classe `java.net.Multicast`, sendo que no MINHA, esta classe é substituída pela `fake.java.net.Multicast`. Para que este protocolo funcione num ambiente fechado, como é este modelo de rede, é usada a classe `sim.global.net.MulticastSocketMap`. Esta classe faz a gestão dos grupos *multicast*, como o registo dos *sockets* e sua remoção dos grupos, como a verificação da existência de um registo num determinado grupo, e o reenvio das mensagens a todos os elementos do grupo.

Como esta classe extende a `DatagramSocket`, herda os seus métodos, utilizando-os para enviar, indicando o endereço *multicast*, e receber mensagens do grupo. Por omissão, a todas as instâncias desta classe é associada a interface de rede `eth0`.

4.7 Sumário

Neste capítulo é explicado o funcionamento do modelo de rede do MINHA, mais concretamente de que forma as propriedades assumidas por este foram implementadas. Aqui é retratado o modo como é feita a gestão da rede, desde o seu mapeamento ao controlo do fluxo dos dados que a atravessam. Este capítulo, explica também como é que os protocolos de comunicação TCP e UDP foram integrados e de que forma as suas propriedades são garantidas.

Apesar do modelo se basear nas propriedades de redes reais, não as representa na sua totalidade, uma vez que o objectivo não é criar um modelo real de rede, como fazem determinados sistemas como o NS2 e ModelNet, mas apenas permitir que os sistemas distribuídos possam ser avaliados tendo em conta apenas os normais atrasos da rede, e não situações invulgares. Ainda assim, a simples aplicação do modelo não é suficiente,

é necessário proceder à calibração dos tempos médios gerados na rede destinada à aplicação. A execução desta tarefa é apresentada de seguida no próximo capítulo.

Capítulo 5

Calibração

De modo a fornecer uma simulação correcta e realista, capaz de imitar o ambiente alvo, é preciso calibrar os modelos I/O. Esta tarefa é conseguida pela computação de um conjunto de *micro-benchmarks* que gera parâmetros específicos para serem aplicados a esses modelos.

A simples incorporação do modelo de rede não é suficiente só por si para uma simulação correcta de uma rede em particular. Apesar das características de rede, da criação de *sockets* e o envio e recepção de mensagens, funcionarem correctamente, uma aplicação distribuída deverá executar sobre uma rede que estará sempre sujeita a atrasos na transmissão de dados. Dado isto, é necessário calibrar o modelo, ajustando as primitivas da rede, como o tempo e o limite de transmissão de pacotes e controlo das suas perdas.

Para que o modelo possa reproduzir estes tempos, é necessário adicioná-los ao modelo. De maneira a fazê-lo, são usados dois *benchmarks* de rede, *flood* e *round-trip*. Estes são aplicados à rede alvo, determinando os custos necessários para atravessar a rede. Estes valores são então aplicados ao modelo de rede para que possa simular os tempos gastos pela rede real. A calibração está dividida em duas fases: o cálculo dos valores de calibração; e a respectiva validação.

Uma vez que o tempo consumido pelo CPU com cada operação está dependente do hardware disponível, tal como o tráfego de mensagens é dependente das características de rede existente, os valores de calibração obtidos são válidos apenas para o hardware e sistema operativo onde são obtidos.

Nesta dissertação, todos os testes foram feitos em duas máquinas cada uma com a seguinte configuração: 64-bit Ubuntu Server 8.04.4 Linux, dois 12 core AMD Opteron™

Processor 6172, 2.1GHz, 128 GB RAM, 64-bit Sun Microsystems Java SE 1.6.0_24 conectados numa rede com largura de banda de 1Gbps.

5.1 Benchmarks

Para que a calibração possa ser feita é preciso conhecer o tempo gasto pela rede para enviar e receber determinada mensagem. Para obter essa informação são utilizados dois *benchmarks* de rede: *flood* e *round-trip*. Em ambos *benchmarks*, são trocadas 500.000 mensagens com tamanhos variando entre 1 e 5000 bytes.

Todos os testes dos *benchmarks* e cálculo dos atrasos são controlados por pacotes externos, `calibration.results.*`. No fim dos testes, este pacote gera um ficheiro de configuração usado pela MINHA para ajustar automaticamente os valores do modelo de rede.

Flood O *benchmark flood*, consiste na escrita e envio de mensagens para um outro *host*. Sempre que um pacote é enviado, o seu tamanho e o tempo actual do sistema é guardado. O mesmo acontece no destino, quando recebe a mensagem. De todos os pacotes, apenas 80% são tidos em consideração, removendo-se os primeiros e os últimos 10%. Isto porque, no início da execução as conexões são estabelecidas entre as entidades, e porque o tráfego da rede é ainda insuficiente para gerar resultados relevantes.

O cálculo da diferença entre os tempos de escrita de cada mensagem, permite a determinação da quantidade de tempo usada pelo CPU para escrever pacotes, e assim calcular o gasto médio de escrita:

$$\begin{aligned} TotalTime &= \sum^{n} time_n \\ OverheadAvarage &= \frac{TotalTime}{n} \end{aligned} \tag{5.1}$$

A média da largura de banda da escrita também pode ser calculada somando todos os bytes transmitidos e dividindo-os pela variação de tempo:

$$
\begin{aligned}
TotalBytes &= \sum^{n} bytes_n \\
\triangle t &= t_n - t_1 \\
BandwithAverage &= \frac{TotalBytes}{\triangle t}
\end{aligned}
\tag{5.2}
$$

Seguindo o mesmo raciocínio, os mesmos dados podem ser calculados, desta vez para a leitura de pacotes.

RoundTrip O *benchmark roundtrip*, consiste no envio de um conjunto de pacotes para um determinado *host* que os reencaminha de volta à origem. Os pacotes são enviados um a um pelo utilizador, e quando chegam, são retornados imediatamente para a fonte. O *host* que as envia espera até que cada resposta chegue, antes de enviar a próxima. No fim, o tempo da viagem e o respectivo tamanho do pacote são guardados, de forma a dividir o processo da rede da análise dos dados. Como no *benchmark* anterior, só 80% desta informação é usada, removendo-se 10% dos primeiros pacotes e dos últimos.

Este *benchmark*, determina o tempo necessário para um pacote ser enviado e ser devolvido de volta ao *host*, isto é, o tempo necessário para realizar uma *round-trip*. A média dos tempos de cada execução, por byte, é usada para validar o modelo de rede, uma vez que permite a comparação entre execuções reais e simuladas. Esta média é obtida dividindo a soma do tempo das viagens, pelo total de bytes recebidos:

$$
RoundTripAverage = \frac{\sum^{n} time_n}{\sum^{n} bytes_n} \tag{5.3}
$$

O atraso da rede pode ser calculado dividindo a média por dois e, removendo os custos de leitura e escrita:

$$
Latency = \frac{RoundTripAverage}{2} - ReadCost - WriteCost \tag{5.4}
$$

5.2 Valores de calibração

Os valores de calibração são usados para aumentar a precisão do modelo de rede do MINHA, que tem por base os tempos de atraso da rede real. Os seus cálculos consistem na execução de ambos *benchmarks* e extraindo o tempo gasto de leitura e escrita e o uso da largura de banda. Para que o modelo tenha maior precisão, os custos de rede dependem do tamanho das cargas devido aos diferentes comportamentos da rede quando lidam com diferentes cargas.

Calibração TCP

O cálculo dos valores de calibração para o protocolo TCP é feito do seguinte modo:

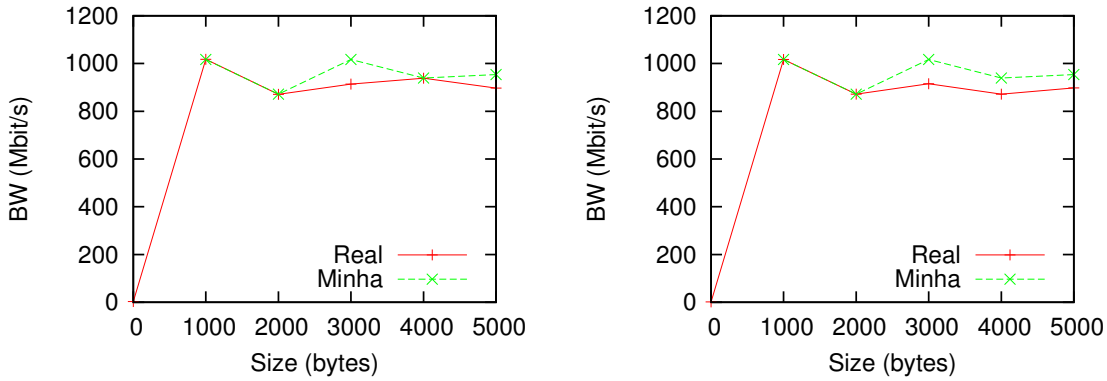
Primeiro, ambos *benchmarks* são executados ambos na rede real e no MINHA, cujos resultados são obtidos como foi anteriormente explicado. As Figuras 5.1, e 5.2 mostram esses resultados.

Na Figura 5.1 são mostrados os resultados para os *overheads* da escrita e da leitura e da largura de banda. Como pode ser visto, os valores dos testes não calibrados são muito próximos dos resultados reais. Isto deve-se ao facto da largura de banda do modelo de rede ter já sido configurada. A Figura 5.2 mostra que os resultados do *benchmark round-trip*. Neste teste, os resultados são muito diferentes. Isto deve-se ao facto do modelo de rede não estar calibrado, e por isso não existe nenhum atraso aplicado à rede resultando numa latência baixa.

Os atrasos da rede obtidos são usados para calcular os valores necessários para aplicar no modelo de rede do MINHA para simular a rede real. Os valores de calibração real são determinados por um pacote externo, `calibration.results.*`. Estes são usados para compensar a diferença entre a rede real e o modelo de rede. O valor para a calibração da escrita é obtido calculando a diferença entre o gasto real da escrita e o gasto simulado da escrita, dividido pela carga actual:

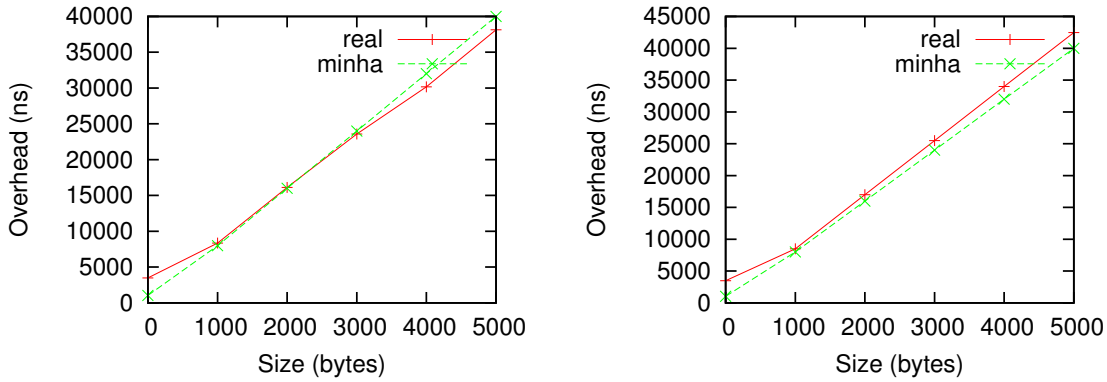
$$WriteDelay = \frac{RealWOverhead - \text{MINHA}WOverhead}{payload} \tag{5.5}$$

A calibração da leitura é calculada da seguinte forma: primeiro a diferença entre os resultados reais e simulados do *round-trip* e da variação de tempo gasta pelo teste real, é dividida pela carga da mensagem, depois este valor é divido por dois, e finalmente o



(a) Largura de banda de leitura TCP.

(b) Largura de banda de escrita TCP.



(c) *Overhead* de leitura TCP.

(d) *Overhead* de escrita TCP.

Figura 5.1: Gráficos não calibrados.

atraso de escrita é removido.

$$ReadDelay = \frac{\frac{realRTTime - realRTDeltaTime - minhaRTTime}{payload}}{2} - WriteDelay \tag{5.6}$$

Como se vê na Figura 5.2, os atrasos da rede são muito diferentes entre ambos os testes. O valor a usar na calibração da latência é dado pela diferença entre os resultados do *round-trip* real e do MINHA, somada aos resultados do MINHA. Esse valor não deve ser estático de modo a aumentar a precisão do resultado, dependendo assim dos tamanhos específicos das mensagens. Para isso, é usada uma função que recebe como parâmetro o tamanho do pacote, que se baseia nas funções resultantes da técnica da regressão polinomial aplicada aos dados reais e do MINHA. Este processo consiste em encontrar a função de regressão polinomial para cada teste, da forma das Equações 5.7, calculando a difer-

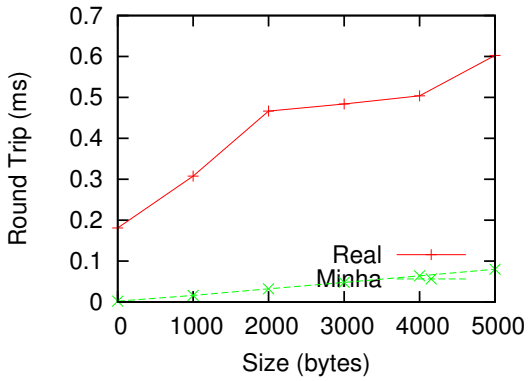


Figura 5.2: *round-trip* TCP não calibrado.

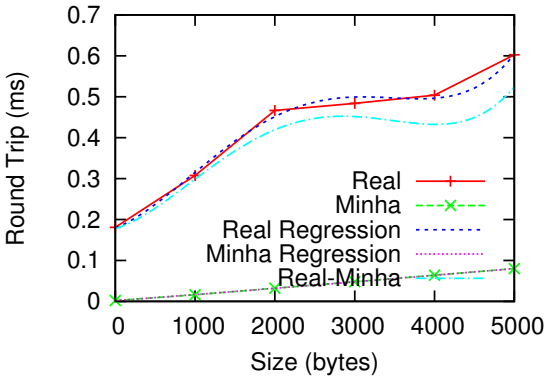


Figura 5.3: Regressões de *round-trip* TCP.

ença entre ambas as funções, Equação 5.8. Esta função é chamada de função de latência.

$$\begin{aligned} real(x) &= arx^4 + brx^3 + crx^2 + drx + er = 0 \\ minha(x) &= amx^2 + bmx + cm = 0 \end{aligned} \tag{5.7}$$

$$\begin{aligned} arx^4 + brx^3 + crx^2 + drx + er &= amx^2 + bmx + cm \\ arx^4 + brx^3 + (cr - am)x^2 + (dr - bm)x + (er - cm) &= 0 \\ latencyFunction(x) = arx^4 + brx^3 + (cr - am)x^2 + (dr - bm)x + (er - cm) & \end{aligned} \tag{5.8}$$

A Figura 5.3 mostra ambas as funções de regressão, assim como a função que representa a diferença entre os seus resultados chamada de *Real-Minha*.

Para estes testes, foram escolhidos os graus 4 e 2, para os resultados reais e de simulação, respectivamente. Os graus dos polinómios são opcionais e usualmente, quanto maior forem, maior é a precisão da função. Contudo, isto nem sempre é desejável, uma vez que o ganho de precisão é muito pequeno quando comparado com o aumento da complexidade.

Depois de determinados todos os valores, o ficheiro de configuração é gerado com todos os valores importantes da calibração. Um exemplo deste ficheiro pode ser encontrado no Apêndice A. Neste exemplo, pode-se ver o atraso da escrita e da leitura. O tamanho da largura de banda não é um valor calculado, uma vez que precisa de ser especificado manualmente. As variáveis `network.delay.*` representam os coeficientes da função de compensação.

Todos os atrasos gerados pela calibração são aplicados apenas ao código de simulação, os quais são carregados durante o arranque da execução. O atraso de escrita é aplicado sempre que um *socket* tenta enviar um pacote para a rede, multiplicando o seu tamanho pelo valor de calibração. Isto também é realizado para a leitura do pacote, cujo atraso é aplicado depois deste chegar ao receptor e antes deste o tentar ler. A função da latência é usada quando o pacote chega ao destino, assim que este é entregue.

5.2.1 Calibração UDP

O modelo de rede MINHA suporta, para além do protocolo de rede TCP, o UDP *unicast* e o protocolo *multicast*. Apesar deste protocolo estar actualmente a trabalhar com o simples transporte de pacotes, devido à dificuldade da calibração destes atrasos na MINHA, e dada a importância de outras tarefas, esta característica foi adiada para trabalho futuro.

Para que o modelo de rede possa reproduzir a taxa de perda dos pacotes UDP, numa rede específica, duas medições têm de ser realizadas: (i) a perda dos pacotes durante a transição; e (ii) na chegada ao destino. Na Figura 5.4, os rótulos *transmission* e *reception* identificam os lugares dessas medidas, respectivamente.

(i) As perdas dos pacotes durante o tráfego de redes é já suportada pelo modelo. Como mencionado na Secção 4.3, quando a largura de banda da rede está ocupada e o buffer está cheio, os pacotes são automaticamente perdidos. Neste caso, a solução passaria por definir um limite mínimo para a perda dos pacotes em *transmission*, e perdendo todos os pacotes enquanto não existisse largura de banda suficiente.

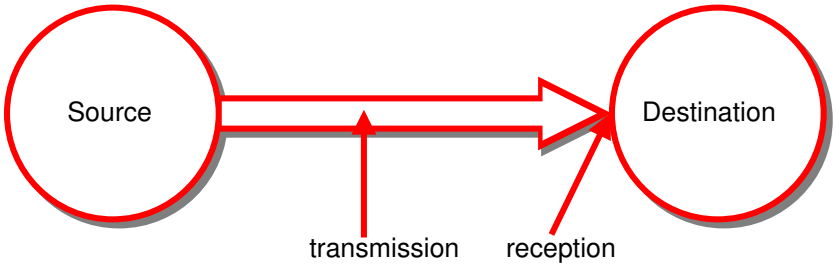


Figura 5.4: Perdas UDP.

O problema aqui, está na obtenção dessa taxa de perda, uma vez que para a determinar seria necessário utilizar um *sniffer* de rede, que executando ao nível do sistema operativo, calcularia essa taxa de perda durante a transmissão. Ainda que determinada, esta taxa teria de ser dividida sob o conjunto total de pacotes, e não apenas sob um intervalo específico de execução.

(ii) O segundo ponto, representa as perdas devidas ao *buffer* de leitura no lado do receptor. Se o buffer estiver cheio à chegada de um pacote este é imediatamente perdido. A solução seria também aplicar aqui um limite de perdas. Contudo, como no caso anterior, a taxa de cálculo em *reception* deveria ser realizada por um *sniffer* externo. Se esta taxa fosse determinada, seria aplicada sobre as mensagens entregues, antes de serem lidas.

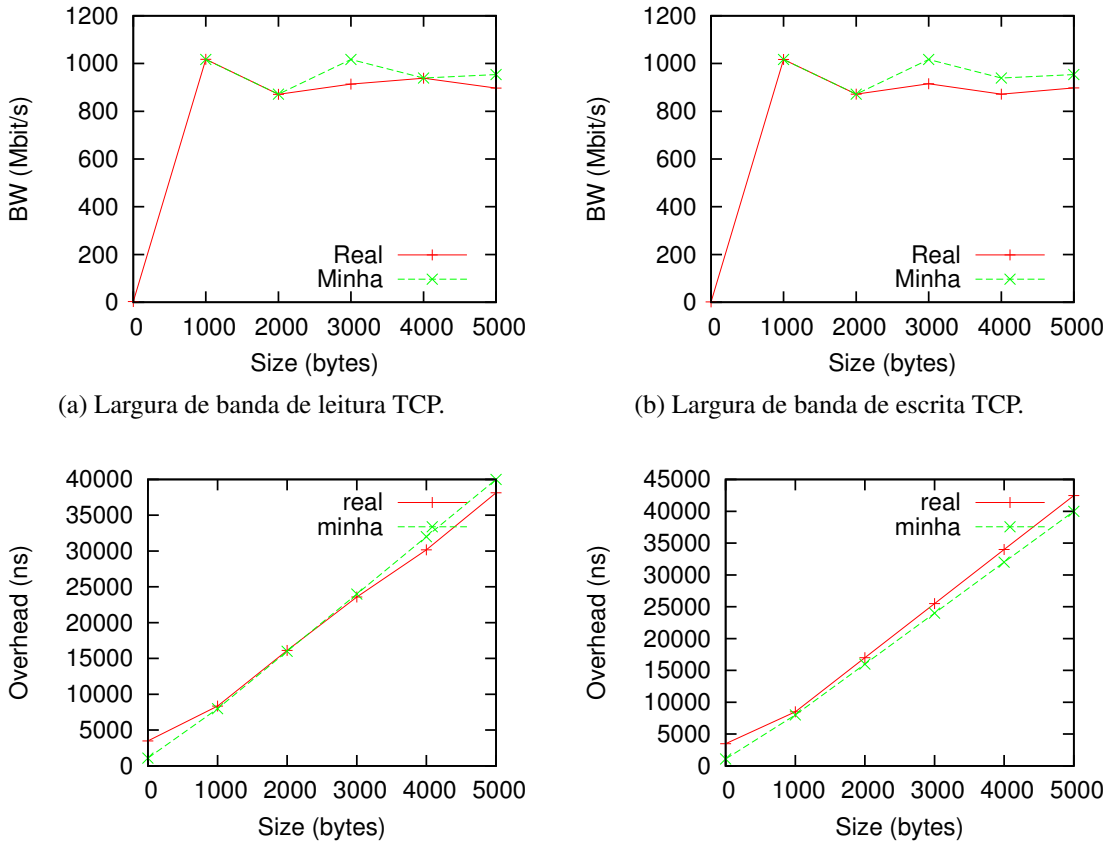
5.3 Validação

Para assegurar que foi feita uma correcta calibração, e que os atrasos aplicados aproximam o modelo da rede ao ambiente real, é necessário executar novamente ambos *benchmarks* no Minha, e comparar os resultados com a rede real.

Nas Figuras 5.5 e 5.6, pode ver-se que os resultados do Minha são quase idênticos aos da rede real. É de notar que as Figuras 5.5a e 5.5b são bastante parecidas com os gráficos não calibrados (Figuras 5.1a e 5.1b). Isto deve-se à boa qualidade dos resultados anteriores cuja calibração pouco afectou.

A Figura 5.6 mostra como a latência da rede do Minha reproduz praticamente os mesmos valores que a rede real.

Estes resultados permitem validar o modelo de calibração da rede para o hardware



(a) Largura de banda de leitura TCP.

(b) Largura de banda de escrita TCP.

(c) *Overhead* de leitura TCP.

(d) *Overhead* de escrita TCP.

Figura 5.5: Gráficos calibrados.

usado, e consequentemente, mostra que o modelo de rede pode ser usado para simular uma rede real.

5.4 Sumário

Neste capítulo, é explicado como o modelo de rede pode ser calibrado para o protocolo de comunicação TCP, assim como os passos necessários para o efectuar. Ao longo do procedimento da calibração, foram utilizados dois *benchmarks*. É também explicado o porquê de não ser considerada a calibração do protocolo UDP. Para suportar a validade do procedimento de calibração e a qualidade do mesmo, são mostrados resultados intermédios da calibração, e no fim resultados finais que comprovam a precisão da avaliação

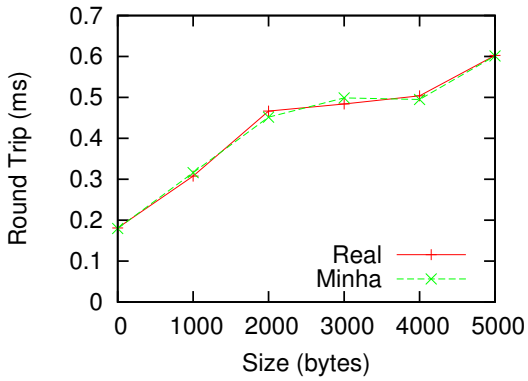


Figura 5.6: *round-trip* TCP calibrado.

de aplicações distribuídas.

No próximo capítulo, o modelo de rede MINHA é usado com um sistema de grandes dimensões, onde o estudo de caso é utilizado para validar o MINHA como uma ferramenta útil para grandes aplicações.

Capítulo 6

Estudo de Caso: WS4D

O estudo de caso destinado a avaliar MINHA com *middleware* de grande dimensão consistiu num cenário *publish/subscribe* que recorre a Web Services. Estas aplicações têm como objectivo lidar com várias centenas de dispositivos, em que um *publisher* notifica um grande número de dispositivos que subscreveram um tópico particular.

Este cenário é um desafio, visto que obter um grande número de dispositivos independentes é complicado. Além disso, as suas configurações individuais e o controlo do seu funcionamento exigem muito tempo, o que torna a avaliação de aplicações muito custosa. O objectivo principal deste teste é ilustrar como o MINHA pode facilmente testar grandes aplicações, permitindo configurar e controlar a execução das mesmas em todos os dispositivos de forma automática e centralizada. Este estudo de caso é adequado porque ajuda a avaliar a performance do MINHA e os seus pontos fortes num contexto real. O resultado é também muito dependente do desempenho da rede, sendo por isso um teste particularmente bom do modelo de rede e da sua calibração.

O capítulo está dividido em três secções principais: o *middleware*, onde é explicado o núcleo do contexto do *middleware*; a aplicação, que descreve o procedimento de teste; e os resultados do teste e respectiva interpretação.

6.1 *Middleware*

O cenário considerado é baseado na especificação OASIS Devices Profile for Web Services (DPWS) [7]. Esta foi desenvolvida para permitir o uso de Web Service em dispos-

itivos com recursos limitados. Para isso foi definido um conjunto mínimo de protocolos que os dispositivos devem implementar para permitir uma comunicação e interoperabilidade bem definidas através de Web Services, sem restringir implementações mais completas. Este perfil especifica as seguintes funções:

- Descrever um Web service
- Descobrir dinamicamente um Web service
- Estabelecer ligações seguras com um Web service
- Subscrever e receber eventos de um Web service

Esta especificação baseia-se em várias outras especificações de Web Services:

- WS-Addressing para o endereçamento de mensagens

  Define uma especificação para comunicar informação endereçada entre Web Services.

- WS-Policy para a definição e troca de políticas

  Esta especificação permite que um Web Service anuncie as suas políticas, por exemplo, para que um serviço possa ser indicado como compatível com o DPWS.

- WS-Discovery e SOAP-over-UDP para descoberta de serviços e dispositivos

  O WS-Discovery permite ao cliente descobrir serviços e dispositivos na rede, nomeadamente através do modo *ad-hoc*, usando tráfego *multicast* seguindo o *standard* SOAP-over-UDP, ou através do modo *managed* que recorre a SOAP/HTTP *unicast* dirigido a endereços do serviço conhecidos durante a execução ou pré-configurados.

- WS-Security para gerir a segurança

  O WS-Discovery usa o WS-Security para gerar assinaturas criptográficas que possam ser usadas por um cliente para verificar a integridade de mensagens não encriptadas.

- WS-Transfer / WS-MetadataExchange para a descrição de dispositivos e serviços

  A especificação WS-Transfer define como transferir informação sobre dispositivos e serviços, como representações baseadas em XML, usando a infra-estrutura do Web Service.

Esta especificação foi desenvolvida para funcionar em combinação com o WS-Addressing e o WS-Policy e define como os metadados podem ser embebidos em endereços WS-Addressing, ou como podem ser tratados como recursos HTTP ou WS-Transfer permitindo assim a sua obtenção.

- WS-Eventing para gerir subscrições de eventos

  Todas as implementações compatíveis com DPWS têm que suportar WS-Eventing, que define um modo de entrega, Push Mode, que consiste num simples serviço de mensagens assíncronas, usado para transmitir eventos aos clientes. Define também como as subscrições podem ser feitas, e como podem ser aceites, assim como o controlo da chegada de eventos do lado do *subscriber*, quando eventos chegam.

  Visto ser criada para dispositivos com recursos limitados e geralmente usada em redes *ad-hoc*, esta especificação não suporta os *brokers* usuais para a distribuição de eventos. Isto requer que todas as interacções entre Web Services sejam feitas directamente entre si.

  Por outro lado, a família WS-Notification suporta o uso de *brokers*, sendo um *standard* mais completo que o WS-Eventing, visto ser orientado a dispositivos com mais recursos. Apesar dessa limitação, WS-Eventing incorpora um mecanismo de filtro flexível na especificação base, favorecendo implementações leves e cenários de distribuição de muitos para um, podendo também ser estendido para suportar esta funcionalidade. Por esta razão foi a especificação escolhida para o *standard* DMTF WS-Management [18] e DPWS.

Existem várias implementações do DPWS, e cada vez mais as empresas estão a adoptalo nos seus produtos. Por exemplo, foi incluído nos mais recentes sistemas operativos Microsoft, Windows Vista, Windows Embedded CE, e Windows 7, estando por isso disponível na maior parte dos computadores pessoais e vários dispositivos, como *set-top boxes*. No projecto SIRENA, foi usado o primeiro desenvolvimento e implementação de uma pilha DPWS embutida para dispositivos e ferramentas associadas. O objectivo principal do projecto de investigação SOCRADES é implementar uma plataforma de projecção, execução, e gestão para sistemas de automação industrial, ao nível do dispositivo e da aplicação, onde o DPWS pode proporcionar os blocos de construção básicos para atingir este objectivo.

Existem também alguns projectos, como o Service-Oriented Architecture for Devices (SOA4D), gerido pela Schneider Electric, ou o Web Services for Devices (WS4D) [17],

gerido pela MATERNA e pelas universidades de Dortmund e Rostock, que disponibilizam pilhas DPWS implementadas em diferentes linguagens de programação.

Em particular, foi seleccionado o WS4D Java Multi Edition DPWS Stack (JMEDS), visto ser compatível com a edição *standard* (J2SE) como a micro (J2ME CLDC) da plataforma Java, suportando por isso uma grande quantidade de dispositivos. Para tal, a plataforma JMEDS disponibiliza as suas próprias alternativas para algumas classes existentes na API J2SE, como estruturas de dados (por exemplo vários tipos de utilidades como algumas funções matemáticas que não existem no CLDC, manipulação de strings ou operações de logging); e *thread pools* e *locks* de sincronização. Além disso, usa uma biblioteca de *parsing* de XML externa.

O estudo de caso foi implementado na versão 2 beta 3a do WS4D JMEDS disponível para J2SE, que suporta a versão 1.1 do *standard* DPWS, permitindo a implementação das entidades *Client* e *Device*, que obedecem a esta especificação.

6.2 Aplicação

A aplicação *publisher/subscriber* consiste em correr um cenário de notificação de eventos, em que um evento é gerado por um dispositivo produtor, e propagado pela rede até um certo número de dispositivos consumidores. A acção de propagação é executada usando o WS-Eventing. Para poderem receber notificações, todos os dispositivos estão configurados para contactar o *publisher* para subscrever operações de notificação. O *publisher* disponibiliza um serviço de eventos com uma operação de notificação que gera uma mensagem para transmitir novos valores de temperatura.

A aplicação tem um dispositivo de gestão responsável pela execução, que conhece todos os elementos da rede que participam na execução, os quais informa sobre o *publisher* e sobre o fim da execução, controlando estatísticas. Para possibilitar a análise, quando uma notificação é distribuída pelo *publisher*, o instante da emissão é guardado. Quando a notificação é recebida pelos *subscribers*, a primeira coisa a ser feita, no tratamento dos eventos, é medir o tempo que esta demorou a chegar. A medição é feita em nanosegundos.

Os testes consistiram em 10 execuções para a quantidade avaliada de dispositivos, onde cada execução consiste na emissão periódica de 60 eventos com um intervalo de 5 segundos.

A execução começa por iniciar o dispositivo de gestão, o *publisher* e todos os *sub-*

scribers. O gestor escuta as mensagens *Hello* do WS-Discovery para perceber se todos iniciaram correctamente. O gestor envia aos *subscribers* o destino do *publisher* para que o possam subscrever e informar que pode começar a distribuir eventos. Estes eventos são disseminados periodicamente pelo *publisher*, e quando acaba, o gestor é notificado. O gestor informa os *subscribers* acerca do ficheiro onde podem escrever os instantes de tempo.

6.3 Resultados e discussão

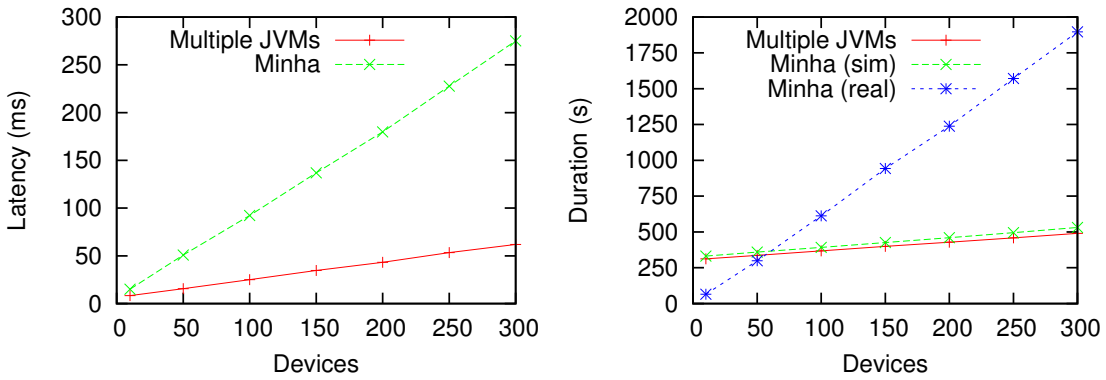
Para que os resultados do MINHA possam ser avaliados e confrontados com um cenário real seria necessário obter dados de execução de um cenário similar num ambiente real para permitir a comparação. Para isso, seria necessário obter 300 dispositivos independentes e configurá-los um por um. Mas como já foi dito, obter um número tão elevado de dispositivos e efectuar a respectiva configuração é bastante complicado. Para contornar este impasse, optou-se por correr o teste fora do simulador, mas ainda apenas numa máquina, na qual cada dispositivo é representado por uma JVM independente, e em que a rede se resume ao endereço *localhost*. Este teste é denotado por *Multiple JVMs*.

Os resultados obtidos consistem em avaliar a latência de entrega dos eventos, e o tempo utilizado nas várias execuções. Estes consistem na média aritmética de todas as execuções para cada configuração, Figura 6.1. Para as medidas de latência, as primeiras 10 interacções foram descartadas para minimizar o efeito de compilação JIT do Java, embora isto também oculte o atraso da conexão TCP.

A Figura 6.1a apresenta o intervalo entre a emissão de uma mensagem pelo *publisher* e a sua recepção pelo *subscriber*. É possível ver a evolução da latência de acordo com o aumento do número de dispositivos, em ambos os ambientes de teste. Como se pode verificar, executar simplesmente vários dispositivos em JVMs independentes, ainda que na mesma máquina, não produz o mesmo efeito quando usado o MINHA.

Quando é usado um grande número de dispositivos, o efeito destes sobre a latência é desvalorizado pelo teste *Multiple JVMs*. Assim como os CPUs destinados aos *subscribers* são usados para fazer a disseminação em paralelo, o que não aconteceria numa configuração real com os mesmos dispositivos.

Na Figura 6.1b é mostrada a relação entre o tempo total gasto em cada uma das execuções relativamente ao número de dispositivos utilizados. Aqui são mostrados três tipos



(a) Latency

(b) Duration

Figura 6.1: WS4D performance e utilização de recursos.

de valores: o primeiro corresponde ao teste *Multiple JVMs*, o segundo e o terceiro ao teste com o MINHA, em relação ao tempo simulado gasto e ao tempo real total para correr toda a simulação, correspondentemente.

- *Multiple JVMs*

  Representa o tempo real da execução sem o MINHA, onde os testes demoraram cerca de 300 segundos.

- MINHA (sim)

  Mostra os testes com o MINHA, que calculam o tempo de execução dentro da simulação, que é quase idêntico ao tempo gasto no teste *Multiple JVMs*.

- MINHA (real)

  Contudo, visto o MINHA usar uma linha de tempo simulada, (i.e. não espera pelos eventos, pelo contrário, avança o relógio para o próximo instante significativo) consegue terminar mais cedo para 50 dispositivos, mas conforme o número de dispositivos vai crescendo o tempo necessário também aumenta.

Como tal, é possível verificar que, apesar da performance do MINHA ficar a baixo da conseguida usando JVMs independentes, os resultados obtidos pelo MINHA conseguem ser melhores. Apesar disso, o MINHA gasta muito menos memória, para uma execução de 300 dispositivos, que *Multiple JVMs*. Como demonstrado na Tabela 6.1, que apresenta a

Execution mode	RAM (GB)
Multiple JVMs	25.4
MINHA	5.7

Tabela 6.1: Média do uso de memória para 300 dispositivos.

soma da memória usada por todos os 302 processos envolvidos nas múltiplas JVMs, e a memória usada pelo único processo que corre o MINHA.

6.4 Sumário

Para uma grande quantidade de dispositivos, o MINHA consegue acompanhar os tempos realizados pela execução que recorre a várias JVMs, no entanto não consegue manter um tempo útil na execução dos testes. Esta desvantagem é vista como sendo temporária, uma vez que é esperado que o MINHA possa ser melhorado usando simulação paralela, em vez do seu simples núcleo de uma só *thread* concorrente. Mesmo assim, estes testes não permitem testar realmente o MINHA a um grande nível de escalabilidade. Isto é devido à configuração do ambiente de teste, visto o tamanho máximo dum sistema que pode ser simulado é geralmente limitado pela memória disponível.

Apesar disso, este caso de estudo permite concluir que o MINHA em conjunto com o modelo de rede consegue produzir resultados tão bons ou melhores quando usadas múltiplas JVMs individuais.

Capítulo 7

Conclusões

A avaliação de sistemas distribuídos pode ser feita de diversas formas mas todas elas deverão ter como objectivo a apresentação de resultados que correspondam à realidade e não apenas uma simples aproximação. No entanto, esta tarefa é bastante complicada de efectuar, pois implica que as diversas características envolventes no sistema sejam tidas em consideração.

A plataforma MINHA permite dar resposta a este problema ao possibilitar uma avaliação centralizada e como tal totalmente controlada. A sua avaliação consiste na teoria de eventos de simulação o que introduz a componente temporal na avaliação de uma forma eficaz e bastante precisa. Ao admitir a incorporação de modelos de simulação permite que todas as características necessárias possam ser adicionadas facilmente à avaliação.

O objectivo desta dissertação consiste na criação de um modelo de simulação de rede a ser usado pelo MINHA de modo a inserir uma componente fulcral na execução de sistemas deste tipo, a rede. Este modelo visa simular a rede na qual o sistema deverá executar. Para tal, terá de ser capaz de imitar o seu comportamento de forma eficiente.

Este objectivo é conseguido através da criação de um modelo de rede de alto nível o qual permite simular o comportamento de uma qualquer rede. Este modelo fornece uma gestão da rede idêntica à realidade assim como um controlo sobre o fluxo de dados da rede, possibilitando a utilização de três protocolos de comunicação: UDP *unicast* e *multicast*; e TCP. Ainda que realista, o modelo não considera uma implementação muito detalha de forma a não introduzir muito *overhead* à simulação. Através do uso de um calibrador automático, desenvolvido especialmente para este modelo, o modelo de rede não se limita a uma única configuração de rede. Este calibrador permite ajustar a execução

do modelo de simulação de modo a que possa ser usado para a simulação de qualquer ambiente.

Para comprovar o correcto funcionamento deste modelo de simulação foram efectuados dois tipos de testes. O primeiro consiste na utilização de dois *micro-benchmarks* que permitiram testar, numa primeira etapa, a calibração do modelo numa simples configuração de rede. O segundo teste consistiu na aplicação de um caso de estudo consistindo na utilização do WS4D, o que permitiu validar a capacidade do modelo de rede de simular o ambiente de rede, através da respectiva calibração.

7.1 Publicações

No contexto desta dissertação parte do trabalho foi incorporado num artigo, actualmente aceite, que será ser publicado no final deste ano:

- Nuno A. Carvalho, João Bordalo, Filipe Campos and José Pereira. Experimental Evaluation of Distributed Middleware with a Virtualized Java Environment. In ACM Middleware for Service-Oriented Computing (MW4SOC).

7.2 Questões em aberto

Nesta dissertação é apresentado um modelo de simulação de redes a ser aplicado à plataforma Minha. No entanto este modelo não aborda toda a API do Java, como por exemplo o pacote `java.nio.*`. Seria, portanto, interessante poder adicionar este pacote ao já suportado pelo modelo permitindo uma maior variedade de aplicações que podem ser testadas com o Minha. Outro componente que seria interessante adicionar era a capacidade de simulação de acessos aos sistemas de ficheiros. Isto pode ser feito através da criação de um modelo de simulação que, tal como o modelo de rede, deverá recorrer a um calibrador automático que configure as propriedades do modelo de modo a simular o comportamento do ambiente.

Apêndice A

Network calibration: configuration file

```
# nanoseconds
network.writeCost=0
# nanoseconds
network.readCost=0
# bit/s
network.networkBandwidth=1000000000
# NetworkDelay
network.delay.real_degree=4
network.delay.minha_degree=2
network.delay.real_coef=179181.393310797,75.95146695852742, \
    0.10094977229675428,-4.5838812038688005E-5,5.202275715143598E-9
network.delay.minha_coef=1719.0068433125239,14.766913062113613, \
    1.8682245640602526E-4
```

Bibliografia

[1] *Scalable Simulation Framework API Reference Manual*, Mar. 1999.

[2] W. Almesberger. umlsim - A UML-based simulator. In *Linux.Conf.Au*, 2004.

[3] G. Alvarez and F. Cristian. Applying simulation to the design and performance evaluation of fault-tolerant systems. In *16th Symposium on Reliable Distributed Systems (SRDS'97)*, 1997.

[4] R. Barr, Z. Haas, and R. van Renesse. JiST: An efficient approach to simulation using virtual machines. *Software–Practice and Experience*, 35(6), May 2005. Reprinted in Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks, Jie Wu, editor.

[5] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[6] J. H. Cowie, D. M. Nicol, and A. T. Ogielski. Modeling 100,000 nodes and beyond: Self-validating design. *Computing in Science and Engineering*, 1999.

[7] D. Driscoll and A. Mensch. Devices Profile for Web Services Version 1.1. Technical report, July 2009.

[8] K. Fall and U. C. Berkeley. The ns manual (formerly ns notes and documentation). *Facilities*, 1(3):1–431, 2010.

[9] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the Emulab network testbed. In R. Isaacs and Y. Zhou, editors, *USENIX Annual Technical Conference*, pages 113–128. USENIX Association, 2008.

[10] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*, chapter 24, Introduction to Simulation. John Wiley & Sons, Inc, New York, NY, USA, 1991.

[11] L. Peterson and T. Roscoe. The design principles of PlanetLab. *SIGOPS Oper. Syst. Rev.*, 40:11–16, January 2006.

[12] S. Potyra, V. Sieh, and M. D. Cin. Evaluating fault-tolerant system designs using FAUmachine. In *Proceedings of the 2007 workshop on Engineering fault tolerant systems*, EFTS '07, New York, NY, USA, 2007. ACM.

[13] SPLAY Project.
http://www.splay-project.org/.

[14] P. Urban, X. Defago, and A. Schiper. Neko: a single environment to simulate and prototype distributed algorithms. In *Information Networking, 2001. Proceedings. 15th International Conference on*, pages 503 –511, 2001.

[15] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36:271–284, December 2002.

[16] A. Varga. OMNeT++. In K. Wehrle, M. Günes, and J. Gross, editors, *Modeling and Tools for Network Simulation*, pages 35–59. Springer, 2010.

[17] Web Services for Devices (WS4D) Project.
http://www.ws4d.org/.

[18] Web Services Management (WS-Management) Standard.
http://www.dmtf.org/standards/wsman.