CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

BEng Thesis

Biblioteca de emulação para plataforma modular de simulação de sistemas ciber-físicos

Renato Oliveira

CISTER-TR-181002

Biblioteca de emulação para plataforma modular de simulação de sistemas ciber-físicos

Renato Oliveira

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

http://www.cister.isep.ipp.pt

Abstract

One of the biggest issues when developing cyber-physical systems is the ability to test them in a similarenvironment for which they were designed. The execution of these tests in the real environment mayrequire large sums of money, or, the testing cycles can be slower than the expected reduction of thesecycles.The objective of the KhronoSim project is the development of a modular and extensible testingplatform, which shall allow whoever uses it to develop intuitive closed loop real-time tests for cyberphysical systems, via software. Therefore, the usage of a full-system emulator (which is an emulator thatsimulates at such a level of detail that the software can run directly on it, without any modification) isimperative simulate the environment of the testing target as faithfully as possible. In the scope ofKhronoSim, the chosen emulator was QEMU.Considering this context, and to further approach the behavior of emulated systems to the behavior ofreal systems, this project focuses on: (1) the development of a library (QEMU-Lib) that is able tomanipulate QEMU instances, to ease its use; (2) changing QEMU 19s source code to implement a newfeature, that reduces the execution frequency of a virtual machine, known as the throttle mechanism.QEMU-Lib was implemented having into account all the enumerated requirements made by the client. Itis modular, extensible and concise, using C++ language and tested using BOOST. Using it, makes itpossible to manipulate individual QEMU instances, should it be turning an instance on or off or sendingcommands via network sockets.The throttle mechanism was implemented by altering QEMU 19s source code in the less invasive waypossible and tested to measure its functional specificities. The results show that the data curve whilsthaving this mechanism activated is linear when the exiting condition is a time constraint and thecomputational load is null. When there is some computational load, namely the ordering of a vectorusing an algorithm of n "17 log n complexity, the data curve presents a quadratic form.

© CISTER Research Center
www.cister.isep.ipp.pt

Biblioteca de emulação para plataforma modular de simulação de sistemas ciber-físicos

CISTER Research Centre

2017 / 2018

1140822 Renato Oliveira

Biblioteca de emulação para plataforma modular de simulação de sistemas ciber-físicos

CISTER Research Centre

2017 / 2018

1140822 Renato Oliveira

Licenciatura em Engenharia Informática

Setembro de 2018

Orientador ISEP: Cláudio Maia

Supervisor Externo: Luís Miguel Pinho

«à minha companheira, Soraia Mesquita, sem a qual não teria atingido este sucesso, pelo companheirismo, interesse e apoio dados não só no âmbito da Licenciatura, mas também durante o meu percurso profissional, académico e pessoal»

Agradecimentos

Agradeço ao meu orientador, Cláudio Maia e ao supervisor Luís Miguel Pinho pela disponibilidade, vontade e interesse demonstrados no que toca ao apoio que me providenciaram no âmbito deste projeto; ao CISTER e aos colegas de trabalho, pelas oportunidades proporcionadas e confiança depositada; à minha companheira, Soraia Mesquita, sem a qual não teria atingido este sucesso, pelo companheirismo, interesse e apoio dados não só no âmbito da Licenciatura, mas também durante o meu percurso profissional, académico e pessoal; aos amigos mais próximos que conheci durante este ciclo de estudos com ênfase aos colegas Daniel Gonçalves, Ivo Ferro, Flávio Relvas e Eric Amaral pelo companheirismo, competitividade saudável e, acima de tudo, pela confiança que depositamos de forma reciproca.

Abstract

One of the biggest issues when developing cyber-physical systems is the ability to test them in a similar environment for which they were designed. The execution of these tests in the real environment may require large sums of money, or, the testing cycles can be slower than the expected reduction of these cycles.

The objective of the KhronoSim project is the development of a modular and extensible testing platform, which shall allow whoever uses it to develop intuitive closed loop real-time tests for cyber-physical systems, via software. Therefore, the usage of a full-system emulator (which is an emulator that simulates at such a level of detail that the software can run directly on it, without any modification) is imperative simulate the environment of the testing target as faithfully as possible. In the scope of *KhronoSim*, the chosen emulator was QEMU.

Considering this context, and to further approach the behavior of emulated systems to the behavior of real systems, this project focuses on: (1) the development of a library (QEMU-Lib) that is able to manipulate QEMU instances, to ease its use; (2) changing QEMU's source code to implement a new feature, that reduces the execution frequency of a virtual machine, known as the throttle mechanism.

QEMU-Lib was implemented having into account all the enumerated requirements made by the client. It is modular, extensible and concise, using C++ language and tested using BOOST. Using it, makes it possible to manipulate individual QEMU instances, should it be turning an instance on or off or sending commands via network sockets.

The throttle mechanism was implemented by altering QEMU's source code in the less invasive way possible and tested to measure its functional specifities. The results show that the data curve whilst having this mechanism activated is linear when the exiting condition is a time constraint and the computactional load is null. When there is some computational load, namely the ordering of a vector using an algorithm of $n * \log n$ complexity, the data curve presents a quadratic form.

Keywords (Theme): *Cyber-physical systems, Emulation, Library, Throttle*

Keywords (Technologies): *QEMU*, *C++, BOOST*

Resumo

Um dos grandes problemas que existe no desenvolvimento de sistemas ciber-físicos é a capacidade de teste dos mesmos num ambiente semelhante para o qual são desenhados. A realização de testes em ambiente real pode requerer quantidades avultadas de dinheiro, ou então, os próprios ciclos de teste podem ser demasiado lentos para responder à tão procurada redução dos ciclos de desenvolvimento.

O objetivo do projeto KhronoSim é o desenvolvimento de uma plataforma de testes modular e extensível, que permite testar de forma intuitiva sistemas ciber-físicos em circuito fechado, via software. Para atingir este objectivo, é imperativa a utilização de um emulador full-system (isto é, um emulador que simula sistemas com um grande nível de detalhe para que o software possa correr diretamente no mesmo sem alterações) para atingir uma solução que permita simular ao máximo o ambiente do sistema a testar. No âmbito do KhronoSim o emulador escolhido foi o QEMU.

Considerando este contexto, este projeto de estágio foca-se no desenvolvimento de uma biblioteca de manipulação de instâncias de QEMU, para facilitar a sua utilização, bem como a alteração do código fonte do mesmo para introduzir uma nova funcionalidade responsável pela diminuição da frequência de execução de máquinas virtuais, denominado mecanismo de throttle, de modo a aproximar o comportamento dos sistemas emulados ao seu comportamento real.

No caso da biblioteca, esta foi implementada tendo em consideração todos os requisitos enumerados pelo cliente. Esta biblioteca é modular, extensível e concisa, implementada usando a linguagem C++ e testada à posteriori com BOOST. Através dela, é possível manipular várias instâncias de QEMU individualmente, seja ligar/desligar máquinas ou o envio de comandos via sockets de rede.

Em relação ao mecanismo de throttle, o código fonte do QEMU foi alterado para suportar esta funcionalidade, e devidamente testado para aferir as especificidades do seu funcionamento. Os resultados mostram que a curva dos dados do tempo de execução do mecanismo de throttle é linear quando a condição de paragem é um temporizador e a carga computacional é virtualmente nula. Quando o teste ao mecanismo de throttle utiliza computação via ordenação de vetores com um algoritmo de complexidade $n*\log n$, a curva dos dados é quadrática.

Palavras-chave (Tema): *Sistemas ciber-físicos, Emulação, Biblioteca, Throttle*

Palavras-chave (Tecnologias): *QEMU*, *C++, BOOST*

Índice

Índice de Figuras

Índice de Tabelas

Notação e Glossário

API *Application Programming Interface*

CISTER Centro de Investigação em Sistemas Confiáveis e de Tempo Real

CPU *Central Processing Unit*

GUI *Graphical User Interface*

HMP *Human Monitor Protocol*

I/O *Input/Output*

I&D Investigação e Desenvolvimento

ISEP Instituto Superior de Engenharia do Porto

JNI *Java Native Interface*

KVM *Kernel-based Virtual Machine*

LIB *Library*

MvC *Model View Controller*

PESTI Projeto/Estágio

QEMU *Quick Emulator*

QMP QEMU *Machine Protocol*

RAM *Random Access Memory*

TCG *Tiny Code Generator*

TCP *Transfer Control Protocol*

VCPU *Virtual Central Processing Unit*

VM *Virtual Machine*

WBS *Work Breakdown Structure*

Capítulo 1. Introdução

1.1.Enquadramento

O projeto entitulado "Biblioteca de emulação para plataforma modular de simulação de sistemas ciber-físicos" e reportado neste documento está enquadrado no âmbito da unidade curricular de Projecto/Estágio (PESTI) da Licenciatura em Engenharia Informática do Instituto Superior de Engenharia do Porto (ISEP). Este projeto foi proposto pelo CISTER, uma unidade de Investigação e Desenvolvimento (I&D) integrada no ISEP, e está enquadrado num projeto de I&D entitulado *KhronoSim* que decorre em parceria com a Critical Software S.A. e a Universidade de Coimbra.

O objetivo do projeto *KhronoSim* é o desenvolvimento de uma plataforma de testes modular e extensível, para vários setores do mercado que lidam com sistemas **ciber-físicos**, tais como os setores aeroespacial e automóvel. Entenda-se por sistema ciber-físico uma tecnologia transformativa para gerir sistemas interconectados entre os seus aspetos físicos e as suas capacidades computacionais [7].

A platforma deverá permitir às partes interessadas integrar modelos de simulação com sistemas reais, em ambiente de malha fechada, à medida das necessidades e de acordo com os cenários de teste, e permitir o controlo em tempo-real estrito, isto é, garantindo controlo total sobre o sistema controlado de acordo com restrições temporais bem definidas *a priori*.

Um dos requisitos da plataforma *KhronoSim* é suportar a capacidade de substituir componentes físicos por componentes virtuais, em função do cenário a ser testado, e de modo a permitir que o sistema sob-teste possa ser testado parcial ou totalmente. Este requisito pode ser satisfeito através da simulação/emulação de sistemas. Por conseguinte, no âmbito do projeto de PESTI, o objetivo é desenvolver e trabalhar a componente virtual. Mais propriamente, desenvolver mecanismos que permitam ter um controlo em tempo de execução sobre os componentes simulados/emulados, ou mesmo até do sistema na sua totalidade, de modo a garantir que o ambiente de teste é o mais próximo possível da realidade, e desta forma extrapolar o resultado dos testes para o sistema real.

1.2.Apresentação do Projeto/Estágio

Um dos grandes problemas que existe no desenvolvimento de sistemas ciber-físicos é a capacidade de teste dos mesmos num ambiente semelhante ao qual vão operar. Em muitos deste tipo de sistemas, a realização de testes em ambiente real pode requerer quantidades avultadas de dinheiro, ou então, os próprios ciclos de teste podem ser demasiado lentos para responder à tão procurada redução dos ciclos de desenvolvimento. Existem casos em que validar situações de erro neste tipo de sistemas implica a realização de testes destrutivos em termos de hardware, o que implica perda de tempo e aumento de custos dos projetos, como por exemplo em sistemas aeroespaciais.

Para contornar estes problemas, é possível até certa extensão, recorrer à simulação de sistemas. Para obter resultados fidedignos neste contexto, a solução escolhida para simular um ambiente de testes, deverá ser flexível, isto é, não é viável recorrer a soluções rígidas, com processos internos que podem eventualmente introduzir *overhead*, invalidando assim os resultados dos testes realizados. Posto isto, é imperativo obter uma solução que permita simular o ambiente do sistema a testar o mais próximo possível da realidade. Várias soluções existem para resolver o problema acima descrito e que passam pela utilização de ferramentas de simulação, emulação ou virtualização. Para que o leitor perceba a diferença entre cada um destes conceitos, estes são definidos nos parágrafos seguintes.

O conceito de **simulação** é definido pelo fato de todas as partes funcionais de um modelo serem substituídas na sua totalidade por sistemas virtuais [9]. Esta substituição é feita de modo a que a simulação se foque não só no comportamento exterior do sistema, mas também no estado interno do mesmo. Por conseguinte, é mais orientada para o estudo e análise de sistemas em que seja importante reproduzir o comportamento real do mesmo [9].

A **virtualização** é uma técnica de implementação de uma máquina virtual, denominada de *guest*, que actua como uma máquina real com o seu respectivo sistema operativo, num ambiente conhecido como *host* [17]. O *host* contém todos os recursos de hardware necessários para executar a máquina virtual. Na virtualização existe uma dependência no conjunto de instruções que existe entre o *host* e o guest (*i.e.,* devem ser exactamente iguais), além de que o *host* deve suportar certas propriedades de *hardware* que permitem a virtualização [17]. Esta dependência faz com que quase todas as instruções executadas pelo guest, sejam executadas diretamente pelo CPU do *host*, havendo apenas algumas que são intercetadas e executadas via software do *host.*

A **emulação** é uma técnica de implementação de uma máquina virtual num computador *host*, cujo conjunto de instruções executado pela máquina virtual poderá ser diferente do conjunto de instruções existente no computador *host* [17]. Nenhuma das instruções do guest é executada diretamente no CPU do *host, todas elas* são traduzidas via software no *host* antes de serem executadas. Por conseguinte, a emulação é menos abrangente que a simulação porque procura apenas representar o comportamento visível do sistema, sendo que muitas vezes o seu estado interno pode ser ignorado porque não influencia o seu comportamento externo. Posto isto, um **emulador** ou **simulador**, é um *software* que executa uma ou mais aplicações que foram inicialmente compiladas para um CPU com uma arquitetetura diferente da arquitetura do *host* [13].

Uma vez que o objetivo principal do projeto reportado neste relatório é substituir componentes físicos por componentes virtuais, de modo a permitir a simulação de um sistema real num todo ou em parte, sem a necessidade de controlar o estado interno do sistema, o mecanismo mais indicado para os seus objectivos é a emulação.

1.3.Apresentação da Organização

O CISTER[1] está sediado no Instituto Superior de Engenharia do Porto. Este centra a sua atividade na análise, projeção e implementação de sistemas informáticos embebidos e de tempo-real. As suas áreas principais de actuação são as seguintes: redes e protocolos de comunicação em tempo-real; redes de sensores sem fios; paradigmas e linguagens de programação passíveis de serem utilizados em sistemas de tempo-real; sistemas operativos de tempo-real; computação cooperativa e qualidade de serviço; sistemas multiprocessadores e multi-core; gestão de energia e sistemas ciber-físicos.

A nível de projetos, a instituição tem vários projetos a decorrer nas várias áreas de actuação descritas acima e várias parcerias com empresas conhecidas a nível mundial.

[1] https://www.cister.isep.ipp.pt/

1.4.Contributos deste trabalho

O trabalho reportado neste relatório tem duas grandes contribuições: (1) a criação de uma biblioteca de manipulação de instâncias de QEMU[2][13], que fornece uma interface de interação com várias instâncias de QEMU e onde é possível operar sobre as mesmas e efetuar o seu controlo através de operações como ligar e desligar, pedir informações de execução, alterar parâmetros de execução, etc; (2) a alteração do código fonte do QEMU, para introduzir uma nova funcionalidade que permite diminuir a frequência de execução de máquinas virtuais. Mais especificamente, esta funcionalidade permite aumentar o nível de controlo sobre o sistema que é emulado, dependendo das necessidades do cliente. Ainda acerca desta funcionalidade, e devido à sua natureza inovadora, é apresentado neste relatório um estudo acerca dos efeitos do funcionamento da mesma. O resultado deste estudo foi publicado e apresentado na conferência ICPS'2018 (International Conference on Industrial Cyber-Physical Systems) [10] e na conferência INForum'2018 [11]. Adicionalmente, é apresentado em apendice um método que permite controlar *cores* virtuais de máquinas baseadas em *Linux.*

[2] O QEMU é uma ferramenta de código aberto que permite a emulação e virtualição de sistemas. Como é reportado nos capítulos seguintes, o QEMU foi a ferramenta escolhida para a realização deste projeto. Deste modo, no Capítulo 4 esta ferramenta é descrita com mais detalhe.

1.5. Estruturação do relatório

No capítulo atual, foi dada uma contextualização do problema, seguida de uma apresetação da empresa e por último uma descrição dos contributos relevantes do presente trabalho.

No Capítulo 2, são descritas a metodologia de trabalho utilizada, o planeamento que se utilizou no âmbito do projeto, bem como as tecnologias utilizadas no mesmo.

No Capítulo 3, é apresentado o estado da arte na área da emulação, onde irão ser comparados alguns dos emuladores e virtualizadores existentes.

No Capítulo 4, é descrito em grande detalhe o QEMU, as suas partes constituintes e as suas peculiaridades. Esta descrição está enquadrada com as alterações necessárias para a implementação da QEMU-LIB e do mecanismo de throttle.

No Capítulo 5, são descritas a análise, implementação e testes realizados à QEMU-LIB.

No Capítulo 6, são descritas a análise, implementação e testes realizados ao mecanismo de throttle.

No Capítulo 7, são apresentadas algumas reflexões sobre os vários aspetos do projeto, englobando conclusões acerca do planeamento, da solução e do próprio aluno.

No Capítulo 8 será apresentada a bibliografia abordada no âmbito do projeto.

No 0 estará disponível a seção de anexos onde estarão todos os artefactos relevantes no âmbito do projeto que não estão representados no corpo principal do presente documento.

Capítulo 2. Contexto

Neste capítulo são abordados o problema a resolver no âmbito de PESTI, bem como as áreas de negócio relacionadas com o projeto, nomeadamente a área dos sistemas ciber-físicos, com o intuito de providenciar enquadramento acerca da área onde este projeto se encaixa, a metodologia de trabalho praticada e o planeamento pretendido.

2.1. Problema

Efetuar testes a sistemas ciber-físicos traz consigo alguns encargos[3] quer a nível temporal, quer a nível monetário. A nível temporal o encargo encontra-se na realização dos próprios testes. Devido à natureza deste tipo de sistemas, é necessário re-criar o ambiente que replica o ambiente dos sistemas que controlam, além do tempo necessário à criação dos próprios testes. A nível monetário o encargo encontra-se não só pelo tempo necessário ao desenvolvimento do sistema e do ambiente de testes, mas também, em alguns casos, pela necessidade de realizar testes que podem danificar ou mesmo destruir o sistema desenvolvido (estes últimos cenários podem ser encontrados na indústria automóvel ou aeroespacial).

Para colmatar os encargos acima mencionados, é possível substituir alguns dos componentes reais por sistemas virtuais, ou mesmo em certos casos, todo o sistema pode ser substituído. Esta aproximação na resolução do problema é teoricamente viável, todavia é necessário ter em conta que o comportamento obtido aquando da realização de testes usando esta aproximação pode não ser semelhante ao comportamento obtido ao utilizar o componente real. Por conseguinte em alguns cenários, poderá ser necessário recorrer a testes utilizando os sistemas reais. Como tal, e para minimizar o uso de sistemas reais, é imperativo que o comportamento dos próprios componentes virtuais seja o mais aproximado possível do comportamento observado no sistema real, para que se possa obter resultados fidedignos e de confiança. No entanto, este objetivo é complicado de atingir, pois existem várias nuances inerentes a equipamentos reais que são complicadas de transpor para um ambiente virtual, como por exemplo comportamentos de relógios, periodicidade de interrupções e velocidade de execução.

Como foi definido na seção 1.2, a emulação é uma forma de obter um sistema semelhante ao sistema real, ao nível funcional, pelo simples fato de que esta se preocupa com o comportamento externo demonstrado pelo sistema que emula, em detrimento de se preocupar com uma reprodução fidedigna do estado interno do mesmo. Caso isso se verificasse, estaria a ser discutida simulação ao invés de emulação. Ao recorrer à emulação de sistemas embebidos, uma das primeiras barreiras que se encontra ao tentar reproduzir um comportamento fiel ao sistema real, é o baixo poder de processamento que eles apresentam. Isto deve-se ao facto de os sistemas embebidos na sua grande maioria terem como objetivo o menor gasto possível de recursos energéticos, e do seu propósito ser a realização de tarefas que não necessitam de capacidades de processamento elevados.

Recorrendo à emulação para testar estes sistemas, verifica-se que a forma como se comportam ao nível funcional não corresponde ao sistema real. Portanto, qualquer teste efetuado num sistema emulado que se encontra neste contexto, não poderá ser utilizado para tirar qualquer conclusão acerca do sistema real. Reproduzir o comportamento funcional de um sistema real num sistema emulado requer cuidado na análise dos aspetos que constituem este comportamento. Um passo importante para atingir uma reprodução fidedigna de um sistema real via emulação é a redução da velocidade de execução.

Direcionando mais a análise do problema para a utilização do QEMU como ferramenta de emulação, é importante referir que este não é exeção, e sofre do mesmo problema, os resultados obtíveis utilizando o mesmo não são fidedignos. Isto deriva do facto da versão atual do QEMU não dispor de nenhum mecanismo de diminuição de velocidade de execução.

A velocidade de execução de instruções num ambiente de emulação no QEMU é dependente de alguns fatores, entre eles a frequência de execução do processador da máquina *host* e o tipo de sistema emulado.

Posto isto, para uma emulação fidedigna de sistemas que pela sua natureza têm menos poder de processamento, o QEMU não estaria apto no seu estado atual, sendo que este é um dos problemas que terá de ser resolvido neste projeto.

Outro dos problemas que existe aquando a utilização do QEMU como ferramenta de emulação é a integração das suas funcionalidades com outros módulos e aplicações, sendo que até à data não existe qualquer aplicação de funcionalidades facilitadores de integração do QEMU com os mesmos. O QEMU

não é de todo um emulador *user-friendly* e tem uma curva de aprendizagem acentuada, característica que irá ser suavizada com realização deste projeto.

Espera-se no fim do projeto, obter uma versão alterada do QEMU que permita a redução da velocidade de execução de sistemas com arquitetura ARM, para tornar possível obter resultados fiáveis quando forem submetidos a testes, e obter também uma biblioteca que permita manipular instâncias de QEMU, construído de uma maneira modular e que seja relativamente intuitivo de integrar com outros módulos ou aplicações.

Adicionalmente, é esperado que exista a possibilidade de controlar *cores* virtuais de um sistema com a plataforma *Sabrelite*[3] de modo a que seja possível providenciar mais controlo de fluxos de testes.

Espera-se também, entender a influência que este mecanismo de redução de velocidade tem nos relógios do QEMU, bem como efetuar estes mesmos testes em ambiente multi-core e documentar os respetivos resultados.

Apesar deste ser um projeto centrado na área da investigação, é sem dúvida um projeto onde está envolvida engenharia de soluções. O livro *Engineering Reasoning* oferece-nos uma panóplia de perguntas que, ao respondermos, facilitar-nos-á o desenvolvimento de uma solução para o nosso problema na medida em que irá ajudar a compreender e discutir o mesmo. No Anexo 3, irão ser dadas respostas a algumas destas perguntas, sendo que pela natureza do projeto, certas perguntas serão omitidas por não fazerem sentido nesse contexto.

[3] O *Sabrelite* é uma *plataforma* de desenvolvimento *low-cost* que possui o poderoso processador *I.MX-6Quad.* https://boundarydevices.com/product/sabre-lite-imx6-sbc/

2.2.Áreas de Negócio

Como já foi explicado no ponto 1.2 a área de negócio na qual se insere este projeto é a área dos sistemas ciber-físicos. O termo "sistemas ciber-físicos" refere-se a uma nova geração de sistemas com capacidades físicas e de processamento integradas, que interagem com humanos através de várias formas [20]. A habilidade de interação e expansão de capacidades com/do mundo físico através de computação, comunicação e controlo é um fator chave para futuros desenvolvimentos de tecnologia. Existe um leque de aplicações em diferentes domínios industriais que utilizam sistemas ciber-físicos no seu dia-a-dia, nomeadamente a indústria aeroespacial, automóvel, transportadora, química, manufatura, entretenimento, entre outras.

O gráfico seguinte descreve o interesse na área dos sistemas ciber-físicos nos últimos cinco anos, à escala mundial.

Figura 1 - Interesse na área dos sistemas ciber-físicos

Data source: Google Trends (www.google.com/trends).

Como se pode aferir, os picos de interesse têm vindo a crescer, sendo que existe uma previsão de crescimento ainda mais acentuada ao longo dos próximos anos.

Os sistemas ciber-físicos têm tido cada vez mais referências pelo mundo, são uma área em crescimento, especialmente nos Estados Unidos e na Alemanha, tal como está representado na figura abaixo. Apesar de haver um crescimento maior nestes países, com o surgimento da Indústria 4.0, este crescimento alastrou-se por todo o mundo.

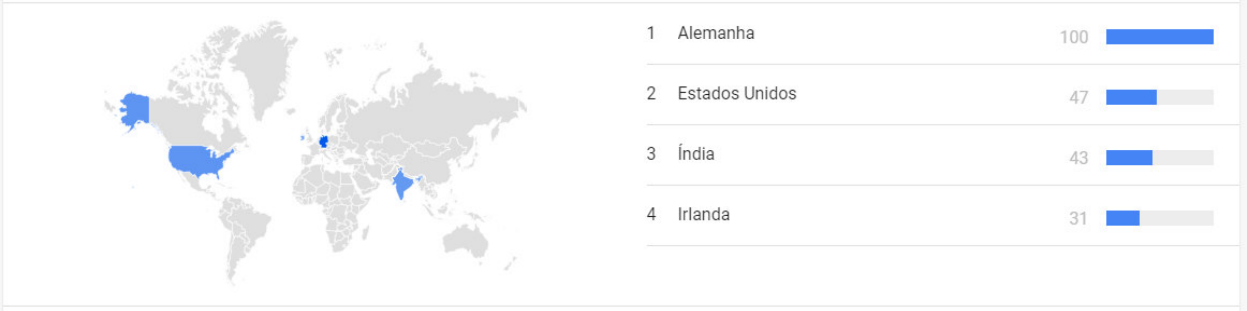


Figura 2 - Interesse na área dos sistemas ciber-físicos por região

Data source: Google Trends (www.google.com/trends).

Na figura seguinte estão representadas as áreas geográficas que demonstram mais interesse na área dos sistemas ciber-físicos. Claramente, a Alemanha é o país mais interessado de momento nesta área, na sua grande maioria por causa da indústria automóvel e manufactura, pelas quais é sobejamente conhecida [8].

Como já foi referido, a indústria automóvel não é a única indústria em que os sistemas ciber-físicos têm um papel fundamental, existem muitas áreas que podem ser influenciadas e melhoradas pelos mesmos, tornando às vezes a visualização da *big picture* algo complicada. Na figura abaixo estão representadas as áreas de aplicação dos sistemas ciber-fisicos.

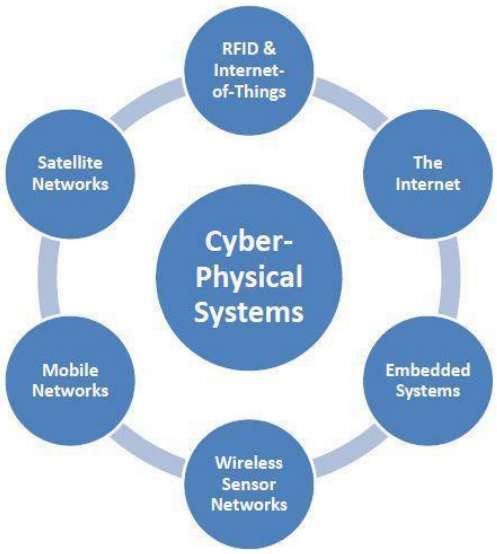


Figura 3 – Aplicação dos sistemas ciber-físicos

Data source: Research Gate (https://www.researchgate.net/)

Como se pode observar pela imagem, é claramente uma área com várias aplicações e potencial de impacto tremendo para o nosso futuro, razão pela qual tem sido continuamente explorado e alvo de investimento.

2.3. Metodologia de trabalho

Neste projeto, foram usados vários repositórios privados a pedido da organização, tanto na fase de desenvolvimento da biblioteca, como na alteração do código fonte do QEMU e nos respetivos programas de teste às funcionalidades. Seguiram-se standards de nomenclatura de *commits* e tentou realizar-se sempre os mesmos com uma periodicidade constante.

2.3.1. Análise de requisitos

No que toca à análise de requisitos recorremos ao *FURPS+* para ajudar a compreender alguns requisitos (funcionais e não funcionais) do projeto. Antes de mais, é importante perceber o que é o *FURPS+ e* qual a sua importância no âmbito da classificação de requisitos.

O *FURPS+* é um sistema de classificação criado por *Robert Grady* enquanto desempenhava funções na *Hewlett-Packard* [5]. Este sistema classifica os requisitos recorrendo a uma divisão em 5 categorias [5], nomeadamente:

- Funcionalidade (Functionality)
- Usabilidade (Usability)
- Confiabilidade (Reliability)
- *Performance*
- Suportabilidade (Supportability)

O "+" contido no acrónico representa os requisitos não funcionais do problema, tais como:

- Requisitos de design
- Requisitos de implementação
- Requisitos de interface
- Requisitos físicos

Olhando mais detalhadamente para cada uma destas categorias é possível perceber que: (1) os requisitos de funcionalidade geralmente representam as funcionalidades principais do produto; (2) a

usabilidade preocupa-se com a estética e consistência da interface do utilizador; (3) a confiabilidade preocupa-se com características como a disponibilidade do sistema, fiabilidade dos cálculos e a capacidade do mesmo de recuperação de erros; (4) a performance tem a ver com características tais como envio de informação, tempo de resposta, tempo de inicialização e tempo de término de execução; (5) a suportabilidade preocupa-se com características como a testabilidade, adaptabilidade, manutenção, compatibilidade, configurabilidade, instalabilidade, escalabilidade e localizabilidade.

Em relação às categorias existentes na área dos requisitos não funcionais, é possível aferir que: (1) requisitos de design especificam restrições no desenho do sistema; (2) requisitos de implementação apresentam restrições na escrita do código, como linguagens de programação ; (3) requisitos de interface especificam sistemas extrâneos com os quais deve ser efetuada comunicação, ou colocar restrições nessa comunicação; (4) requisitos fisicos estão ligados com restrições físicas como a utilização de *hardware* especifico ou limites de peso.

2.3.2. Metodologia de trabalho utilizada

Em relação à metodologia de trabalho, devido à natureza do projeto, não foi utilizado *Scrum*, pelo simples facto de o projeto de PESTI ser de pequenas dimensões e por ser um projeto de I&D.

Em termos de desenvolvimento de software, utilizou-se uma metodologia *Waterfall,* onde os requisitos foram identificadados e o software desenvolvido e testado. A metodologia de desenvolvimento *Waterfall* tem como valores base:

- Ser um modelo de desenvolvimento sequencial;
- Requisitos devem estar claros antes de avançar para a fase de design;
- Testes são levados a cabo após o código estar completamente desenvolvido;
- As fase de desenvolvimento são levadas a cabo por ordem;
- Cada fase de desenvolvimento tem prazos para a realização de tarefas;
- A documentação e testes são efetuados após implementação;
- Os defeitos são encontrados em fases tardias de desenvolvimento;

Apesar desta metodologia ter algumas desvantagens, as vantagens que dá ao projeto asseguram que ela se mantém uma das metodologias mais utilizadas mundialmente [2].

Nas fases de investigação utilizou-se uma metodologia de simulação[4], onde se tentou criar vários cenários de execução para permitir aferir o comportamento do software realizado nesses cenários. Em vários casos foi necessário voltar à fase de desenvolvimento de software porque este não teve os resultados ou comportamentos esperados na componente de simulação.

Em relação a testes, foram realizados tanto na fase de desenvolvimento de software, como na fase de investigação para aferir resultados, procurando em ambos os cenários efetuar testes relevantes.

2.3.3. Reuniões

No que toca a reuniões com o orientador, estas tiveram uma periodicidade semanal, nas quais se abordavam o estado actual do projeto, a discussão de tarefas a decorrer e resultados obtidos nessa semana, seguida de uma discussão acerca do trabalho futuro.

Na tabela abaixo estão representadas as reuniões mais relevantes que ocorreram no decorrer do projeto.

Tabela 1 - Reuniões efetuadas

Data	Membros	Assunto	Localização
15/02/2018	Renato Oliveira; Cláudio Maia; Luís Pinho; Staff da CRITICAL SOFTWARE	Apresentação da plataforma *KhronoSim*	Instalações da CRITICAL SOFTWARE (Coimbra)
1/03/2018	Renato Oliveira; Cláudio Maia; Luís Miguel Pinho	Reunião de inicio de estágio	CISTER

[4] A metodologia de simulação é pouco utilizada quando comparada ao caso de estudo, como forma de superar a dificuldade de realização de experiências [18]. A simulação é viável no contexto de "descoberta" [18].

21/03/2018	Renato Oliveira; Cláudio Maia;	Demonstração da QEMU-Lib; Discussão de requisitos do mecanismo de *throttle*	CISTER
14/04/2018	Renato Oliveira; Cláudio Maia;	Demonstração de mecanismo de throttle; Discussão de requisitos do *Sabrelite*	CISTER
06/06/2018	Renato Oliveira; Cláudio Maia;	Demonstração do controlo de *cores* do *Sabrelite*	CISTER

Em relação a reuniões diretas com o cliente, existiu uma, com o intuito de apresentar o projeto *KhronoSim* e onde o cliente teve lugar para especificar alguns requisitos que até então estavam ainda abstratos.

2.4.Planeamento

A fase de planeamento e o próprio ato de planear são extremamente importantes para o sucesso de qualquer projeto, especialmente em projetos de I&D, nos quais por vezes é incerto se os objectivos serão atingidos. Como tal, ter objetivos bem delineados e tentar respeitá-los ao máximo irá facilitar o desenrolar do projeto, maximizar a performance e eficiência da equipa.

Para este projeto, foram identificadas com a ajuda do orientador várias milestones que foram prontamente compiladas e documentadas num gráfico de Gantt. Na figura abaixo estão representadas estas milestones a negrito bem como algumas sub-tarefas que consideramos pertinentes.

Name	**Work**
Develop QEMU-Lib	**15d**
Study communication mechanisms	5d
Implement library	5d
Test and document library	5d
Throttle Mechanism	**15d**
Study possible approaches for throttling	5d
Implement selected mechanism in QEMU-Lib	5d
Test and document mechanism	5d
Multicore support	**23d**
Perform Sabrelite board experiments	16d
Study current multicore alternatives supported by QEMU (optional)	7d
Write progress report	2d
M1 - Progress Report	
Write paper about developed work	15d
Write final report	17d
M2 - Report Delivery	

Figura 4 - Milestones e sub-tarefas do projeto

Note-se que é complicado seguir um planeamento à risca, mas utilizando uma metodologia de trabalho correta é sempre possível adaptar o planeamento às necessidades do projeto de uma forma dinâmica, sendo que algumas durações que constam na figura acima podem facilmente ser alteradas conforme as necessidades.

Foi possível seguir e respeitar o planeamento, à exceção de algum desfasamento que ocorreu nomeadamente nos testes e documentação dos efeitos do mecanismo de ***throttle***[5] nos tempos de execução. Demorou mais do que o esperado porque foi um processo em que se necessitou de muito rigor. Existiu também algum desfasamento no estudo do suporte *multicore,* nomeadamente na compilação do *Sabrelite*.

A tarefa opcional que se encontra documentada seria levada a cabo caso tivesse sobrado tempo após a realização do projeto, algo que não se verificou.

[5] Nome dado ao mecanismo de controlo da velocidade de execução no contexto do QEMU.

Capítulo 3. Estado da arte

No presente capítulo, serão apresentados os requisitos identificados no que toca à resolução do problema. Para um desenvolvimento saudável do projeto, é necessário abordar requisitos em duas frentes distintas, nomeadamente, requisitos que: (1) se referem à escolha do emulador a utilizar; (2) dizem respeito ao mecanismo de throttle; uma vez que é importante utilizar esses requisitos como base para o desenvolvimento do projeto.

3.1. Escolha do Emulador

É importante referir que a utilização do QEMU foi um requisito do cliente. Todavia, é imperativo perceber o processo de decisão que esteve na base desta escolha, pelo que neste capítulo é efetuada uma comparação entre emuladores candidatos.

Vários requisitos foram tidos em consideração aquando da escolha do QEMU como ferramenta de emulação para o *KhronoSim, nomeadamente*: (1) capacidade de emulação *full-system;* (2) capacidade de emulação de múltiplas arquiteturas; (3) licença de software utilizada; e finalmente, (4) capacidade de emulação de sistemas multi-processador.

O primeiro requisito, dita que o emulador deve ter capacidade de emulação *full-system.* Um emulador com capacidade *full-system,* é um emulador que simula um sistema computacional a um grande nível de detalhe, para que o várias camadas de *software (*sistema operativo*, middleware, etc,)* possam correr diretamente no mesmo, sem alterações [9]. Este requisito é importante porque o objetivo é substituir alguns componentes físicos do sistema por componentes emulados, como tal, continua a ser imperativa uma simulação desses componentes com um nível de detalhe significativo.

O segundo requisito, dita que o emulador deve permitir a emulação de múltiplas arquiteturas. O objetivo é cobrir o maior leque de arquiteturas possíveis, visto que o objetivo do *KhronoSim* é a realização de testes de sistemas ciber-físicos em malha fechada, com suporte para a emulação de vários tipos de sistema, caso não seja possível aceder aos sistemas físicos a testar.

O terceiro requisito refere-se ao facto de necessitarmos de um emulador que esteja sobre uma licença de distribuição que nos permita utilizar o mesmo num sistema proprietário sem existir a obrigação de ceder o código-fonte.

O quarto e último requisito dita que o emulador escolhido deve suportar o paradigma multi-processador. Isto deve-se ao facto de vários sistemas embebidos serem dotados destas capacidades, como tal, nunca seria possível obter comportamentos aproximados do *hardware* real utilizando um emulador com capacidades exclusivamente *single-core.*

O objetivo da análise do estado da arte, no que diz respeito ao conjunto de requisitos relativos à escolha do emulador, é perceber quais os emuladores actualmente no mercado que preenchem os mesmos. Desta forma, iremos proceder à análise de alguns emuladores candidatos com o intuito de assertar se estes são viáveis para a implementação da solução. Os emuladores que consideramos candidatos são o Bochs, o Gem5, o Emul8 e o QEMU.

3.1.1. Bochs

O Bochs, é um emulador portátil escrito em C++ e suporta a emulação de processadores, memória, discos, monitor, ethernet, BIOS e outras peças de hardware comum [18]. Todavia, uma das limitações inerentes ao Bochs é o leque de arquiteturas que pode emular, nomeadamente as arquiteturas IA-32 e x86-64 IBM. Em comparação com o QEMU, este último é superior pelo simples fato de ter um leque muito mais alargado de arquiteturas que pode emular. Bochs pode ser compilado para suportar SMP (multiprocessamento simétrico) para que possa emular processadores que suportem até 254 *threads.* Todavia, essa funcionalidade ainda é experimental. O Bochs não utiliza *threading,* o que leva a uma execução sequencial mesmo que o *host* tenha capacidades multiprocessador. Não suporta de igual forma tradução dinâmica nem virtualização. Em detrimento disso, cada instrução x86/x86-64 é interpretada. Durante a fase de execução, o Bochs adopta uma postura *lazy* no que toca ao atualizar *flags* e registos, isto é, apenas os atualiza quando necessário [6]. Na tabela abaixo podemos verificar o encaixe do Bochs nos requisitos identificados:

Tabela 2 - Encaixe do Bochs nos requisitos identificados

Capacidade de emulação full-system	✓
Capacidade de emulação de múltiplas arquiteturas	-
Licença de distribuição adequada	✓
Capacidade de emulação de sistemas multi-processador	-

Descartamos a utilização do Bochs porque este não se encaixa nos requisitos identificados. Posto isto, passamos à análise do gem5.

3.1.2. gem5

O gem5 é um simulador orientado para a investigação de arquiteturas de computadores. Este utiliza um sistema de memória orientado ao evento, o que permite uma flexibilidade para se conseguir proceder à modelação de hierarquias de cache multi-nível. Isto permite estudar o impacto da utilização de diferentes tipos de memória. O gem5 emula um leque de arquiteturas reduzido, nomeadamente as arquiteturas Alpha, ARM, SPARC e x86. Apesar de suportar ARM, não suporta arquiteturas específicas. Este aspecto é uma limitação dado que no KhronoSim existe o requisito de suportar o *Sabrelite*[6], que é a arquitetura alvo de estudo do corrente projeto.

Apesar de suportar simulação *full-system,* algumas das arquiteturas possíveis de ser emuladas, requerem que o gem5 esteja a correr sobre arquiteturas compatíveis, como por exemplo a simulação da plataforma *Alpha* requer que o gem5 esteja assente sobre *hardware* com disposição *little-endian.*

[6] Apesar do *Sabrelite* ser dotado de uma arquitetura ARM, a emulação do mesmo não é suportada pelo gem5

Na tabela abaixo podemos verificar o encaixe do Gem5 nos requisitos identificados:

Tabela 3 - Encaixe do Gem5 nos requisitos identificados

Capacidade de emulação full-system	✓
Capacidade de emulação de múltiplas arquiteturas	-
Licença de distribuição adequada	✓
Capacidade de emulação de sistemas multi-processador	✓

Devido a estas limitações, nomeadamente o reduzido leque de arquiteturas que simula e restrições de hardware para a simulação de certas plataformas, a utilização do simulador gem5 é inviável.

3.1.3. Emul8

O Emul8 é outra alternativa possível no âmbito deste projeto. Este é indicado para utilização aquando o desenvolvimento de um sistema embebido, facilitando tarefas de debug e reprogramação. Não seria um substituto de um sistema embebido físico, uma vez que o objetivo da utilização do Emul8 é agilizar o processo de desenvolvimento do sistema. A utilização do Emul8 permite um controlo do ambiente de desenvolvimento, no sentido em que é possível monitorizar e influenciar os componentes do mesmo a um nível quase impossível de reproduzir aquando a utilização de *hardware* real.

O Emul8 é limitado no leque de arquiteturas que emula, nomeadamente ARM Cortex-A/M, SPARC, PowerPC e x86 – ainda em desenvolvimento. Na tabela abaixo podemos verificar o encaixe do Emul8 nos requisitos identificados:

Tabela 4 - Encaixe do Emul8 nos requisitos identificados

Capacidade de emulação full-system	✓
Capacidade de emulação de múltiplas arquiteturas	-
Licença de distribuição adequada	✓
Capacidade de emulação de sistemas multi-processador	✓

Como o Emul8 suporta apenas arquiteturas que são utilizadas em sistemas embebidos, este não se encaixa nos requisitos especificados.

Posto isto, partimos para a análise do último objeto de análise no âmbito deste projeto, no que toca ao estudo de emuladores candidatos, nomeadamente, o QEMU.

3.1.4. QEMU

O QEMU (Quick Emulator) é um emulador de processadores com dois modos de operação: emulação *full-system* e *user-mode.* Pode ser instalado no *Windows, Linux* e *MacOs X* e pode emular as seguintes arquiteturas: x86/x86 64, PowerPC, Sparc, ARM, etc. A última revisão saiu em 07/10/2017, nomeadamente a versão 2.10.1. O QEMU é um projeto open-source, lançado sobre a Licença Pública Geral GNU versão 2. Através da emulação *full-system* é possível emular um *guest* completo, como definido no Capítulo 3.

Na emulação *user*-mode, o QEMU consegue executar programas compilados para um CPU específico, num CPU diferente, traduzindo chamadas ao sistema durante a execução. Uma das limitações deste modo é que nem todas as chamadas ao sistema são suportadas.

Por defeito, o QEMU utiliza tradução dinâmica binária para o código nativo, com suporte a código auto-modificável, sendo que os modos de tradução irão ser abordados na seção 4.6. Atualmente, o QEMU suporta arquiteturas SMP até 255 CPUs, dependendo das arquiteturas *host* e *guest.* No modo de emulação *full-system,* ele dispõe de uma única *thread (*a não ser nas implementações *multi-thread* recentes do QEMU), ainda que os CPUs do *guest* sejam emulados em paralelo. Mais detalhes sobre os detalhes internos do QEMU serão apresentados no Capítulo 4. Na tabela abaixo podemos verificar o encaixe do QEMU nos requisitos identificados:

Tabela 5 - Encaixe do QEMU nos requisitos identificados

Capacidade de emulação full-system	✓
Capacidade de emulação de múltiplas arquiteturas	✓
Licença de distribuição adequada	✓
Capacidade de emulação de sistemas multi-processador	✓

Como se pode verificar na tabela 4, o QEMU preenche todos os requisitos. Como tal, o QEMU é uma solução viável no âmbito do projeto *KhronoSim*.

3.2.Mecanismo de Throttle

No que toca ao mecanismo de throttle, identificamos que este deve: (1) permitir controlar a velocidade de execução de uma máquina virtual; (2) ser determinístico e não alterar o comportamento funcional do sistema, quando submetido ao mesmo tipo de tarefas; (3) ser utilizável em *runtime* e quando necessário.

Relativamente ao primeiro requisito, como a descrição do mesmo indica, o mecanismo deve permitir o controlo da velicidade de execução de uma máquina virtual. Por controlo entenda-se diminuição da velocidade de execução quando esta está a executar à velocidade máxima, e aumento da velocidade caso não esteja.

O segundo requisito, dita que a máquina virtual deve demonstrar um comportamento semelhante entre execuções de diferentes instâncias da mesma tarefa, quando submetida à mesma percentagem de *throttle.* Por comportamento semelhante entenda-se a obtenção de um tempo de conclusão de tarefas semelhante (dentro de um intervalo) ao obtido noutra qualquer execução. Ainda em relação a este requisito, ele dita que o mecanismo não deve alterar o comportamento funcional do sistema. Por isto entenda-se que o mecanismo deve ser extrâneo ao sistema. O sistema deve continuar a operar sem qualquer noção de que está sobre influência de uma alteração da velocidade de execução.

O terceiro e último requisito infere que o mecanismo deve ser utilizável enquanto a máquina virtual está ativa, para que seja possível manipular a velocidade de execução em tempo de execução. Isto é necessário porque diferentes sistemas podem requerer diferentes configurações de teste, e como tal, é importante dotar o sistema da capacidade de adaptação aos mesmos através da atualização de parâmetros de execução.

Tendo em conta os requisitos acima apresentados, e a escolha do QEMU como ferramenta de emulação a utilizar no âmbito do projeto, no próximo capítulo são apresentados os detalhes internos do QEMU, de modo a permitir ao leitor perceber o trabalho desenvolvido no âmbito deste projeto de estágio.

Capítulo 4. QEMU

Tal como apresentado no capítulo anterior, para emulação de sistemas foi selecionado o QEMU. Neste capítulo, vamos detalhar este emulador para permitir ao leitor entender as suas características, bem como a sua organização estrutural interna, o que permitirá obter uma melhor compreensão acerca do presente trabalho.

Antes de mais, dado que o código-fonte do QEMU é muito extenso e complexo, neste capítulo apenas serão abordadas as características mais importantes no âmbito deste projeto. Na figura seguinte está representada a arquitetura lógica do QEMU.

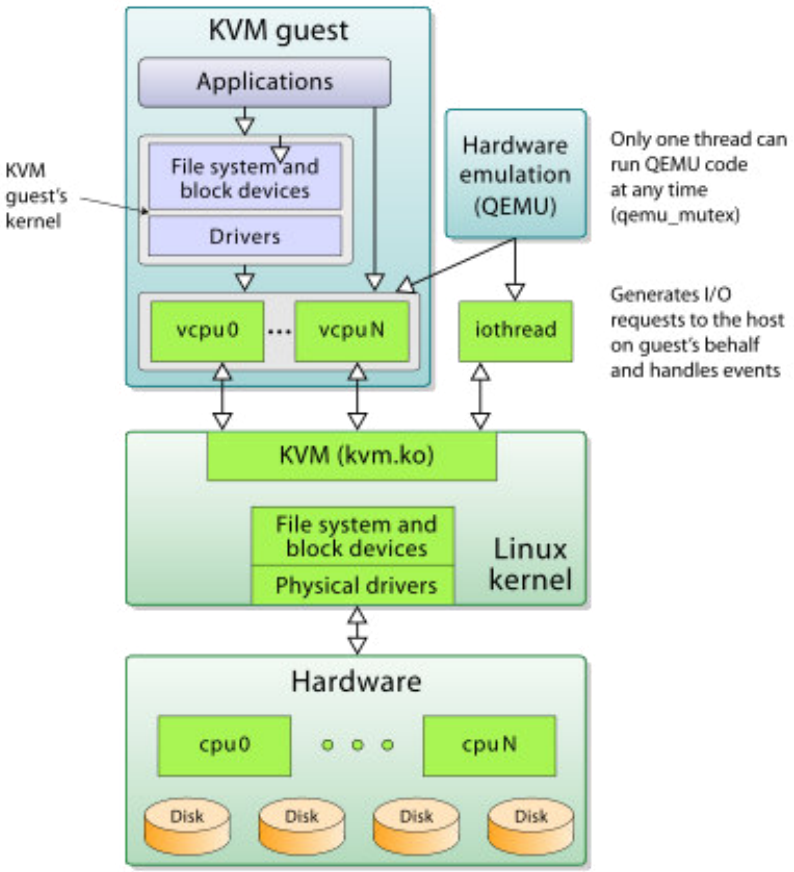


Figura 5 - Arquitetura lógica do QEMU

(Data Source: https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine)

Podemos aferir através da figura acima, que para a utilização do QEMU com o KVM ativo é necessário este estar assente sobre um *kernel* de *Linux.* O *kernel* providencia uma interface (denominada por *kvm.ko* na imagem) que consegue comunicar com o QEMU. Ainda na mesma camada, encontram-se os *drivers* físicos e o sistema de ficheiros. Na camada mais abaixo, estão os componentes físicos do sistema que corre o QEMU.

Analisando a camada superior, verifica-se que o QEMU dispõe de *CPU's* virtuais que estão encarregues de tratar qualquer instrução das máquinas virtuais. Ainda nessa camada, existe o sistema de ficheiros da máquina emulada, bem como os *drivers* da mesma. Como está representado, aplicações correm com recurso a um *CPU* virtual (ou *VCPU)*, ao sistema de ficheiros e outros dispositivos.

Lateralmente a esta camada, encontra-se a *Input/Output Thread* cujo objetivo é gerar pedidos e tratar eventos à/da máquina que corre o QEMU em nome da máquina emulada. Está também representado na imagem a parte de emulação de *hardware* que comunica com a *I/O Thread.* Esta é detentora de um semáforo *mutex* global cujo objetivo é restringir a execução de código do QEMU a uma e uma só *thread.*

Posto isto, passa-se à análise mais específica de cada uma destas partes do sistema.

4.1.Input/Output Thread

O QEMU dispõe de uma *thread* dedicada para eventos de I/O. Esta alteração surgiu em versões mais recentes do QEMU de modo a criar uma divisão lógica entre eventos de *Input/Output* (I/O) e outros eventos, e assim atingir uma maior performance devido à existência de uma *thread* dedicada a operações de I/O.

Pela sua natureza, ela não usa o semáforo *mutex* global do QEMU, como tal, dispositivos bloqueadores não utilizam este *mutex* global (que iria causar uma degradação de performance significativa para todos os outros dispositivos). Utiliza, portanto, um semáforo dedicado para estes dispositivos de *I/O* bloqueadores o que deixa o semáforo global livre para eventos não bloqueadores. Esta alteração aumenta muito a performance da máquina virtual, mesmo sem utilizar o KVM [14].

Tudo que seja escrita/leitura de discos, apresentação de imagem no monitor e inputs de teclado e rato, são eventos de I/O que são processados no âmbito desta *thread.* Devido à sua natureza, qualquer manipulação desta *thread* requer cuidado devido aos eventos de interrupção, uma vez que se for manipulada de forma descuidada, existe um risco de obter um sistema emulado instável.

4.2.VCPU Threads

As *VCPU Threads* são extremamente importantes, pois são estas que estão encarregues de correr o código do *guest* emulado. O número de *threads* é altamente dependente do sistema que se está a emular. Normalmente, o QEMU faz mapeamento de um para um entre *VPCU Threads* e *cores* físicos.

As *VCPU threads* partilham algum código com a *I/O Thread,* todavia usam o *mutex* global para salvaguardar zonas críticas de leitura e escrita e consumição de eventos.

Apesar de partilharem alguns recursos com a *I/O thread,* não são bloqueadas pela mesma, o que permite um certo grau de independência e aumenta a performance do QEMU.

Estas *threads* estão encarregues de todo o processamento não *I/O* do *guest,* todavia, tem de existir algo que monitorize todas estas *threads* e trate os eventos das mesmas, sendo que o componente do sistema encarregue disso é o *main-loop,* que irá ser explicado na próxima seção. Estão também encarregues de manipular os relógios do QEMU, seja incrementar ou pausar de acordo com as necessidades. Os relógios utilizados pelo QEMU são abordados no seção 4.4.

Outro ponto importante a abordar acerca das *VCPU Threads* é a sua forma de processamento de dados. Estas *threads* estão dependentes do CPU/arquitetura que as processa. Isto significa que, dependendo do sistema a emular, os *internals* destes blocos de código serão diferentes uma vez que é uma das *features* do QEMU suportar vários CPU e arquiteturas.

Posto isto, é importante referir que não se irão, nem fará sentido, fazer alterações a um nível interno a estes blocos de código devido à sua complexidade e diversidade. De igual forma, existe uma diferença de performance entre a utilização do *KVM* em detrimento da execução sem *KVM*.

4.3.Main-Loop

O *main-loop* do QEMU é um bloco de código crucial para o bom e correto funcionamento do emulador. Este componente tem os *signal handlers* que vão capturar e tratar os sinais recebidos da *I/O thread* e

das *VCPU threads.* Dependendo do tipo e do emissor do sinal, este irá ser tratado por *handlers* diferentes.

Podemos considerar o *main-loop* como o próprio nome indica, o *loop* de execução principal do QEMU, uma vez que é este que oferece uma camada para todas as *threads* pertinentes comunicarem entre si, bem como, oferecer segurança a condições críticas via semáforos *POSIX.*

Outro aspeto demasiado importante para não descrever são os *callbacks* processados por este componente. Estão designados no código fonte do QEMU por *bottom-halves.* Estes são *callbacks* muito leves cuja invocação, garantidamente, não irá ser atrasada (de acordo com o que está descrito na definição da estrutura). Isto permite que certas operações em que não existe necessidade de aceder a zonas críticas, mesmo nas *I/O threads,* possam executar sem esperas, o que aumenta também a performance do emulador.

Obviamente, também fornece funções para criar, escalonar e cancelar estes *callbacks,* sendo que no processo de cancelamento existe uma *race condition* com o processo de execução da mesma *bottom-half,* daí cada estrutura deste tipo ser detentora de um *mutex*. Não se pode falar em *callbacks* sem referir assincronismo, que é suportado pelo QEMU e esta implementado neste *main-loop* para obter ainda mais performance do que uma execução síncrona.

O comportamento e composição de estruturas deste bloco, é dependente da arquitetura e sistema operativo que está a ser emulado, pelo que é complicado tirar conclusões acerca do código sem especificar uma arquitetura.

4.4. Relógios/Timers do QEMU

Os relógios são a base de qualquer sistema de computação, uma vez que os ciclos de computação de um CPU se baseiam em ciclos de relógio para operar. Como tal, é importante abordar a forma como estes se integram no QEMU. Mais ainda, os relógios têm um papel fundamental no âmbito deste projeto, pois o estudo do seu comportamento, perante diferentes condições de processamento de dados, é um dos objetivos deste projeto. Nomeadamente, o estudo do impacto da diminuição da frequência de execução dos relógios utilizados pelo QEMU. Antes de mais, é importante referir a diferença entre relógios e *timers*. Um relógio é um componente que regula indiretamente o tempo e velocidade de um sistema.

Um *timer* é um caso específico de um relógio, utilizado para medir intervalos de tempo e geralmente gerar algum tipo de ação. Este tem obrigatoriamente de se basear num relógio para conseguir operar.

Na figura abaixo está representado um diagrama da implementação dos timers do QEMU.

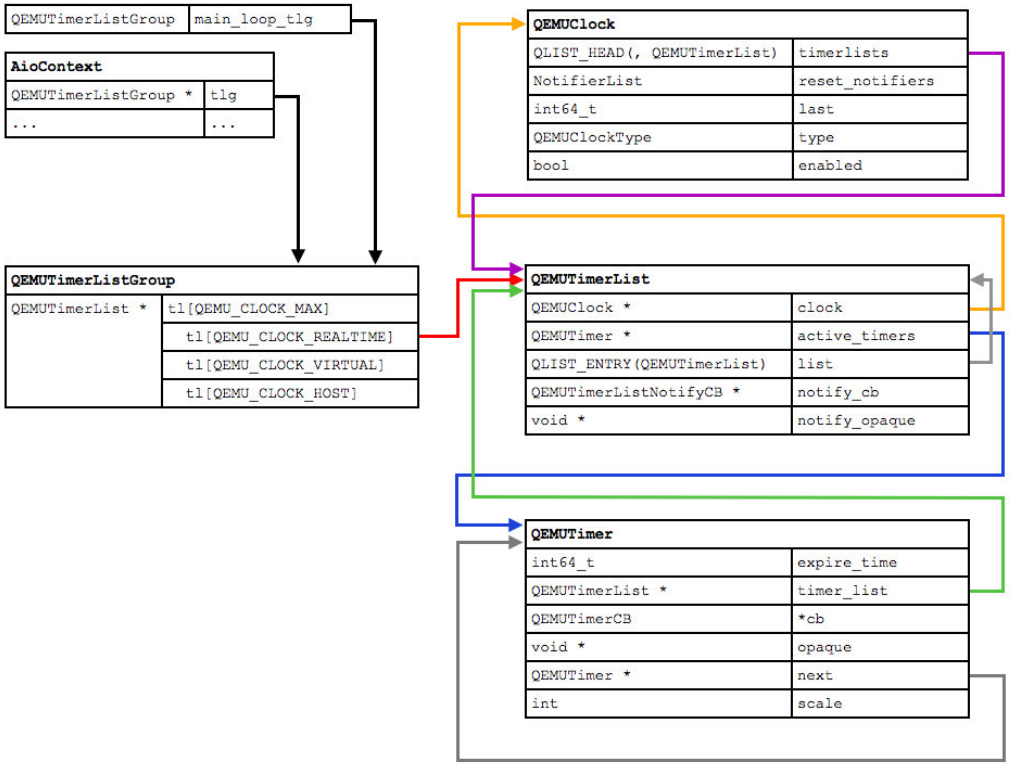


Figura 6 - Diagrama dos timers do QEMU

(Data Source: http://blog.alex.org.uk/2013/08/24/changes-to-qemus-timer-system/)

A estrutura *QEMUTimerList* contém uma lista de *timers* que estão atualmente a correr. Cada relógio tem a responsabilidade de se manter atualizado com todas as *QEMUTimerLists* desse tipo de relógio.

O *QEMUTimerListGroup* é uma estrutura que armazena elementos do tipo *QEMUTimerList*, tantos quanto o número de tipos de relógios distintos. De momento, existem dois componentes que usam o *QEMUTimerListGroup:* o *main_loop_tlg* e o *AIOContext.* O *main_loop_tlg* acede às listas de *timers* que executam no *main-loop. O AIOContext* contém *timers* separados que correm durante operações bloqueantes. Isto é importante porque o *AIOContext* tem como responsabilidade a notificação de eventos que ocorrem, pelo que ter *timers* a correr durante operações bloqueantes pode ser útil no sentido em que guarda informação acerca do contexto e/ou da duração do bloqueio.

Em relação às estruturas *QEMUTimer* e *QEMUClock* são as que detém o papel de relógio e *timer* no âmbito do QEMU.

Passando agora aos tipos de relógio existentes no QEMU, eles são três, na versão do QEMU datada ao início do projeto e que foi escolhida para desenvolvimento (Fevereiro, 2018), sendo que existiu uma atualização no dia 12 de Março de 2018 que adicionou um quarto relógio. No contexto deste relatório são apenas abordados os três tipos de relógio existentes à data de começo do projeto.

Os três tipos de relógio do QEMU são o *QEMU_CLOCK_REALTIME, QEMU_CLOCK_VIRTUAL* e *QEMU_CLOCK_VIRTUAL_RT*, todos a operar na unidade dos nanosegundos. O *QEMU_CLOCK_REALTIME* está encarregue de manter o tempo real. Está em constante execução mesmo que a máquina virtual esteja parada, pelo que deve ser utilizado para coisas que não tenham dependência do estado da máquina virtual. O *QEMU_CLOCK_VIRTUAL* executa apenas quando a máquina virtual também está a executar, pelo que é interessante explorar no âmbito deste projeto, nomeadamente aquando o estudo do mecanismo de throttle. Por último, temos o *QEMU_CLOCK_VIRTUAL_RT* que é semelhante ao relógio explicado imediatamente acima, sendo que difere num único aspeto. Se o QEMU estiver a funcionar em modo *icount* (modo que usa o número de instruções que consegue executar por segundo de tempo para simular uma execução determinística, e como tal simular um relógio) este relógio é utilizado para incrementar o *QEMU_CLOCK_VIRTUAL* quando os processadores emulados estão inativos.

Essencialmente, no modo *icount* se não existirem instruções o *QEMU_CLOCK_VIRTUAL* não consegue contar o tempo, tendo de se basear no *QEMU_CLOCK_VIRTUAL_RT.*

É de notar que usando o modo *icount* se fica impossibilitado de utilizar qualquer utilização de paradigmas *multi-threaded* ou até mesmo a *KVM-API,* uma vez que o *icount* está diretamente conectado à tradução de código e os mecanismos descritos acima não podem envolver tradução, apenas interpretação direta.

A título de curiosidade, o quarto relógio é denominado por *QEMU_CLOCK_HOST* e é utilizado quando o objetivo é emular uma fonte de tempo real, isto é, está ligado ao *host* e reflete as mudanças que ocorrem no relógio do mesmo.

É importante referir que os relógios mencionados acima são os existentes numa perspetiva de desenvolvimento. No âmbito deste projeto é importante também fazer uma breve abordagem ao conceito de relógio no contexto de utilizador avançado do QEMU.

Essencialmente, existem dois tipos de acesso a relógios que determinam os valores internos dos mesmos, que são o *vm-clock* e o *kvm-clock.* Por defeito, o QEMU utiliza o *kvm-clock*. Este relógio é baseado no relógio da máquina *host*, sendo que o QEMU faz pedidos periódicos para se manter atualizado a níveis temporais. Isto é especialmente interessante se o objetivo for sincronizar algo entre a máquina *host* e a máquina *guest.* Dependendo do objetivo do utilizador, este pode configurar o QEMU para utilizar uma das duas fontes de relógio.

No contexto deste projeto o objetivo é descentralizar a contagem e obtenção de informações temporais do *host* uma vez que uma das necessidades do projeto é diminuir a velocidade de execução do *guest*. Esta diminuição terá de respeitar as velocidades relativas de incremento de relógios, isto é, a velocidade de incremento tem de ser proporcional à diminuição de velocidade de execução. Entenda-se por incremento do relógio a adição que se faz ao valor contido na estrutura do mesmo cujo objetivo é guardar o tempo. Para isso pode-se recorrer ao *vm-clock* dado que é um relógio cujo contexto se limita apenas à máquina emulada, isto é, é essencialmente um relógio independente.

4.5.Módulo de abstração de CPUS

Como existem vários tipos de CPU que podem ser emulados, faz sentido existir um local onde é mantido o código comum e independente do CPU que se está a emular. Dependendo do CPU que se está a emular os comportamentos internos serão diferentes. Para lidar com o código comum existe o módulo de abstração de CPUS, que é um bloco de código onde está localizado o código comum de escalonamento de tarefas, ativação e desativação de CPUS e quaisquer outras funções relacionadas com obtenção de informação dos CPUS. Este bloco de código também está encarregue de algumas manipulações de relógios e timers uma vez que é aqui que temos a possibilidade de manipular o estado de um CPU, seja pausar ou continuar a execução, e como tal, faz sentido que estas ações disparem o mais rapidamente possível efeitos nos timers e relógios.

Também podemos, a partir deste bloco de código, verificar os diferentes fluxos de execução caso se use *tiny code generator* ou *KVM*, que serão explicados no próximo subcapítulo, sendo que foi crucial para perceber o funcionamento dos mesmos.

Adicionalmente, neste módulo são também definidos os semáforos referidos no ponto 4.2 e a criação das *threads I/O* e *VCPU*, bem como alguns handlers que dizem respeito a este contexto em detrimento do *main-loop.*

Apesar da criticidade de todas as funcionalidades descritas acima sobre este bloco de código, existe uma que se destaca devido ao facto desta se encaixar diretamente com um dos objetivos do trabalho. Esta é a funcionalidade de migração de máquinas virtuais. Este é o processo pelo qual o QEMU consegue mover uma máquina virtual entre *hosts* com arquiteturas diferentes, ou, adicionalmente, um *guest* para um *host*. Como é obvio, este é um processo demorado (dependente do tamanho da imagem) e não existe sem as suas nuances. Como as migrações podem ser efetuadas com a máquina virtual num estado ativo e é realizada com independência de *hardware*, tem de existir tempo para conseguir escrever dados de memória e discos em grande quantidade. Como a máquina está num estado ativo, estes dados são constantemente vítimas de modificação, pelo que a migração possivelmente tornar-se-á incompletável. Para isso, existe uma funcionalidade de *throttle*, que faz com que o CPU fique parado durante um tempo medido em microssegundos, dependendo de uma percentagem desejada.

Esta funcionalidade não está exposta, é apenas para uso interno da funcionalidade de migração, mas é uma excelente candidata para estudo e implementação do mecanismo de *throttle,* que é um dos objetivos do trabalho descrito neste relatório.

4.6.Modos de tradução de instruções

Vamos abordar nesta seção os modos de execução de instruções suportados pelo QEMU: o *Tiny Code Generator (TCG)* e o *Kernel Based Virtual Machine (KVM).*

O *TCG* é o que é o utilizado por defeito no QEMU. De uma forma sucinta, o *TCG* traduz as instruções do *CPU guest* em instruções do *CPU host*. Este processo traduz-se num overhead significativo, acabando por afetar a performance da máquina virtual, tornando-a mais lenta. As instruções do *CPU guest* são escritas numa linguagem intermédia denominada por *TCG-Ops* que depois é compilada para instruções do *CPU* do *host.*

O *KVM* para poder ser utilizado, requer que o *CPU* emulado tenha a mesma arquitetura que o *CPU* do *host,* como foi referido no Capítulo 4. Ele corre todo o código *user-mode* e algum código *kernel-mode*

diretamente no *CPU* do *host*, utilizando emulação para o restante código *kernel-mode* e todo o código *real-mode*.

Por si só, o *KVM* não efetua qualquer operação de emulação. Todavia, este expõe uma interface que um *host* pode utilizar para preparar o espaço de endereços de um *guest*, alimentar operações de I/O simulado e mapear a saída de vídeo do *guest* para o *host.*

O *KVM* ficou mais conhecido para emulação de máquinas com baseadas em *Linux* com virtualização assistida por *hardware,* em detrimento de plataformas *Windows* e *MacOs*, para o qual existe o *Hardware Execution Manager (HAXM).*

No âmbito deste projeto, vai ser dado mais ênfase na utilização do *KVM* porque vamos trabalhar com emulação de máquinas baseadas no *kernel* do *Linux.*

4.7.Monitor do QEMU

O monitor do QEMU é um dos componentes mais importantes na realização deste projeto. Este monitor é uma espécie de consola que tem como propósito providenciar uma interface de interação com uma única instância de QEMU, que está abstraído do sistema que está a ser emulado. Devido à sua natureza, só está disponível quando a instância de QEMU está a executar.

No âmbito deste projeto, é importante perceber bem o funcionamento do monitor porque um dos seus objetivos é a criação de uma biblioteca que permite a um utilizador interagir com várias instâncias de QEMU (o que inclui adicionar suporte para interagir com o mecanismo de *throttle)*. Ora se já existe algo nativo ao QEMU que suporta essa interação, faz todo o sentido estudarmos o funcionamento da mesma.

```
ETEXI

    {
        .name       = "q|quit",
        .args_type  = "",
        .params     = "",
        .help       = "quit the emulator",
        .cmd        = hmp_quit,
    },

STEXI
@item q or quit
@findex quit
Quit the emulator.
```

Figura 7 - Comando em Haxe para sair do Emulador

O monitor aceita alguns comandos que permitem obter várias informações sobre o *guest* quer acerca do sistema operativo quer máquina virtual. Especificamente, o monitor funciona através da combinação das linguagens de programação *Haxe* e *C.* Os blocos de código escritos em *Haxe e* contidos no ficheiro *hmp-commands.hx* são utilizados para definir os comandos disponíveis ao monitor. Na figura acima está representada a definição do comando *quit*, que é responsável por terminar a emulação, escrito em *Haxe.* Este bloco de código é dotado de vários atributos. A tabela abaixo irá relacionar a estrutura do bloco de código com o propósito de cada um dos atributos.

Tabela 6 - Estrutura de definição de um comando monitor

Atributo	Função
.name	Nome para invocação no monitor
.args_type	Tipo de varáveis enviado por argumento
.params	Argumentos
.help	Texto apresentado como ajuda a este comando
.cmd	Nome da função invocada internamente que

	contém a implementação do comando

Foi referido que o atributo *.cmd* é usado para invocar uma função interna. Essas funções estão definidas no ficheiro *hmp.c* que contém também a sua implementação. Existem tantas definições de funções quanto o número de comandos disponíveis para o monitor.

O monitor é um de dois métodos de comunicação com o QEMU, sendo que o outro será explicado no próximo capítulo.

4.8.QEMU Machine Protocol

O *QEMU Machine Protocol* ou *QMP* é, como o nome indica, um protocolo de comunicação com o QEMU. Este protocolo é baseado em *JSON* e permite enviar comandos para uma instância de QEMU de forma intuitiva. Para além de *JSON* para a interpretação dos comandos, este protocolo usa *sockets* de rede para realizar a conexão entre a instância de QEMU e um cliente, sendo que o cliente pode ser um terminal via *telnet* ou uma aplicação exterior.

Para se poder enviar comandos e fazer pedidos à instância de QEMU via *QMP* é necessário ativar as capacidades *QMP* através do envio do comando `"{"execute": "qmp_capabilities"}".`

A instância de QEMU irá então responder que está disponível para receber outros comandos. Normalmente, as respostas vêm dotadas de um *timestamp* relativo ao recebimento da resposta. A partir deste momento podemos enviar qualquer comando suportado, tal como pausar ou continuar a máquina virtual, pedir informação de discos, entre outros. Existe uma panóplia de comandos QMP[7] que oferecem funcionalidades variadas.

Os comandos estão divididos entre comandos regulares e comandos de *query*, sendo que os regulares normalmente alteram o estado da máquina virtual enquanto que os comandos *query* são usados para fazer pedidos.

[7] https://qemu.weilnetz.de/doc/qemu-qmp-ref.html

O objetivo deste capítulo foi dar a conhecer ao leitor a arquitetura do QEMU, bem como, as interações entre os diferentes componentes. Teve também como objetivo providenciar um conhecimento mais técnico acerca do funcionamento do mesmo, sendo que servirá como base para perceber a lógica aplicada na resolução dos problemas propostos. No capítulo Capítulo 5 irá ser abordada a resolução do primeiro problema, nomeadamente, a implementação da QEMU-Lib.

Capítulo 5. Qemu-Lib

5.1.Visão Geral

Tal como se pode depreender da descrição feita do QEMU no capítulo anterior, o QEMU é um emulador muito poderoso que permite emular diferentes processadores e arquiteturas. Como tal, traz com ele uma curva de aprendizagem acentuada[8]. Uma das limitações que contribui para esta acentuação no que toca à aprendizagem é o facto de os inputs para o QEMU serem feitos através de linha de comandos. Isto é, não existe qualquer *GUI* que nos permita interagir com o mesmo, quer seja para iniciar o emulador ou para controlar uma dada instância de QEMU.

Outra das limitações é a incapacidade de controlar instâncias singulares, pois podemos querer enviar inputs para as mesmas de maneira isolada. Já referimos no capítulo 4.7 que cada instância de QEMU é detentora de um monitor que permite interagir com a mesma. Mais uma vez, este controlo tem uma curva de aprendizagem significativa e é limitador, no sentido em que estamos dependentes dos comandos destinados ao monitor e não podemos usufruir do *QEMU Machine Protocol (QMP).* Uma forma de resolver as limitações acima identificadas seria criar uma *API* que interagisse com o *QMP* de uma forma direta. No entanto, isto iria trazer problemas em termos temporais, no sentido em que se tinha de aprender toda a sintaxe dos comandos *JSON* bem como a própria escrita dos comandos que às vezes é moroso.

Para colmatar estas limitações, no âmbito do *KhronoSim*, o CISTER ficou responsável pela criação de um módulo que pudesse interagir com instâncias de QEMU de modo a ser possível criar e manipular instâncias, independentemente do protocolo de comunicação. Este módulo deverá suportar ambiente gráfico e ambiente de consola.

[8] O QEMU tem uma documentação escassa e a que existe é confusa e aplica-se a um contexto geral. Não existe qualquer explicação sobre certas funcionalidades, muito menos acerca da sua utilização e integração. Como tal, muito do progresso obtido foi através de uma série de tentativas e assunções.

No diagrama de atividade abaixo, pode ver-se um *mockup* muito simplificado do que se deseja atingir com a implementação desta biblioteca.

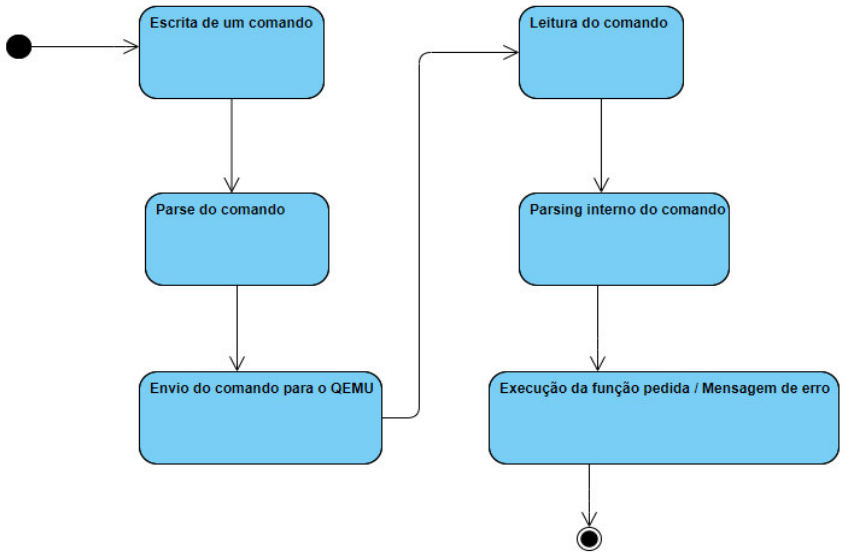


Figura 8 - Diagrama de atividade de um envio de comando para o QEMU

Apesar da simplicidade, existem conceitos que ainda são abstratos nesta fase, nomeadamente: (1) o tratamento de comandos por parte da biblioteca; (2) como comunicar com o QEMU; (3) a noção de independência de uma instância de QEMU. Neste capítulo irão ser esclarecidas e especificadas essas abstrações com o intuito de encontrar uma solução viável para o problema.

5.2.Análise

Nesta subseção vai ser efetuada uma análise aos requisitos da Qemu-Lib e proposta uma solução candidata que será implementada à *posteriori*.

A granularidade com que é apresentada a análise é ao nível de processos, mais especificamente ao nível da interação entre classes, em detrimento da divisão por casos de uso, sendo que estes também serão obviamente apresentados ao leitor na subseção seguinte.

5.2.1. Definição de casos de uso

Nesta subseção estão descritos os casos de uso identificados de acordo com os requisitos funcionais do projeto. Na figura abaixo está representado o diagrama de casos de uso que foram identificados na fase de análise.

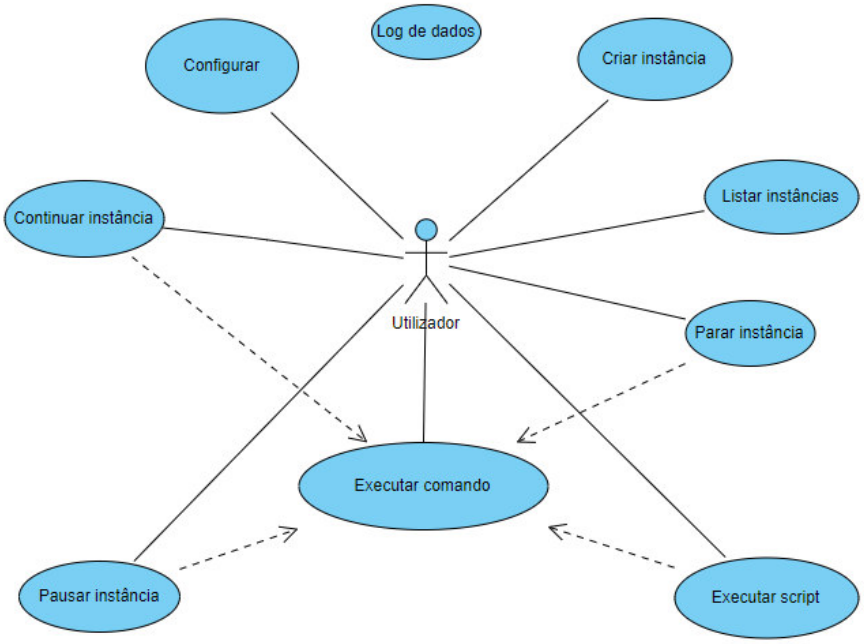


Figura 9 - Diagrama de casos de uso

Pode verificar-se que existe grande dependência de um cenário em especial estar funcional, que é o cenário "Executar comando". O bom funcionamento deste é fulcral e abre muitas funcionalidades do QEMU ao utilizador. Na tabela seguinte estão descritos os casos de uso identificados.

Tabela 7 - Descrição dos casos de uso

Caso de uso	Ator	Descrição
Criar instância	Utilizador	O utilizador deve poder criar instâncias independentes de QEMU.
Listar instâncias	Utilizador	O utilizador deve poder listar as instâncias de QEMU.
Executar comando	Utilizador	O utilizador deve poder executar comandos em instâncias de QEMU.
Log de dados	Sistema	O sistema terá de fazer o *log* de dados relevantes (entrada em funções, erros, etc)
Configurar	Sistema/Utilizador	O sistema deve ser configurável via ficheiro de configurações

Posto isto, passamos à aplicação do *FURPS+* como ferramenta de análise dos requisitos identificados[9].

Na figura seguinte está representada a análise aos requisitos que foram identificados na fase de análise.

[9] O *FURPS+* está definido na seção 2.3.1.

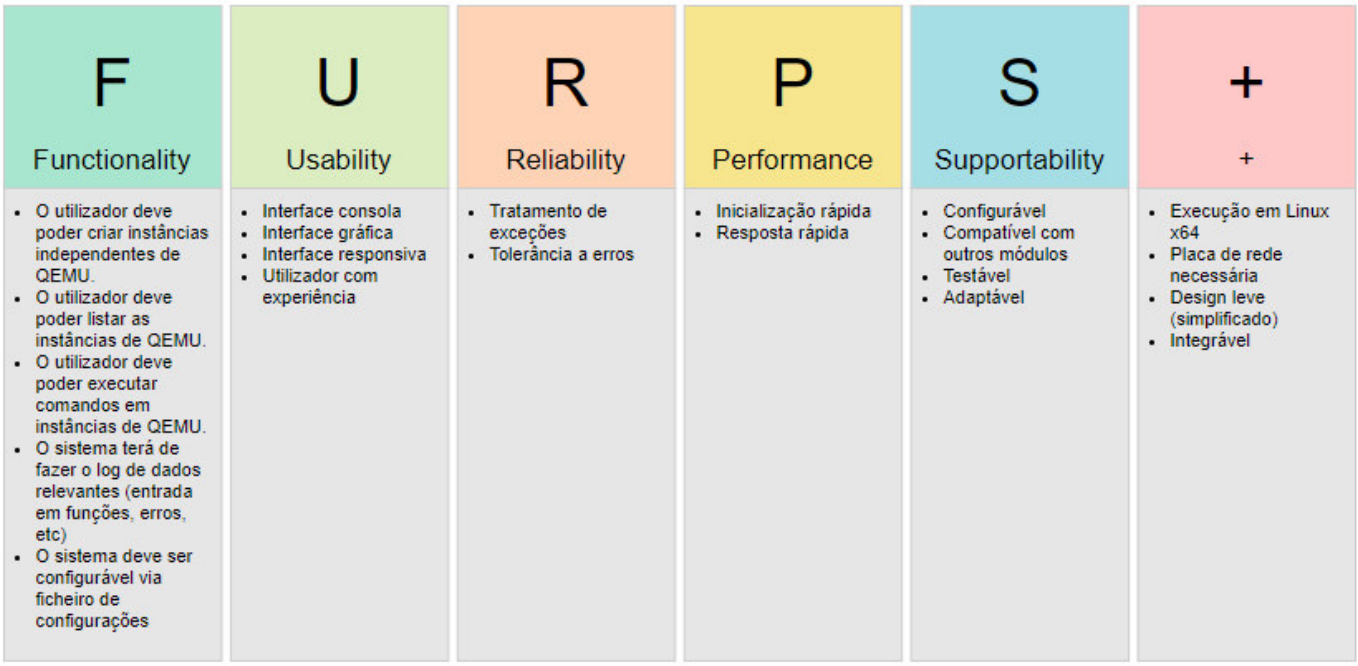


Figura 10 - Análise requisitos via FURPS+

5.2.2. Métodos de comunicação com o QEMU

O ponto fulcral desta biblioteca é a comunicação independente com várias instâncias de QEMU. Deste modo, o protocolo de comunicação utilizado tem um papel fundamental.

Existem várias formas de comunicar com processos, nomeadamente: (1) áreas de memória partilhada; (2) *pipes;* (3) filas de mensagens; (4) sinais *POSIX;* entre outros menos relevantes. À exceção dos sinais POSIX (que foi a primeira forma de comunicação abordada no âmbito deste projeto), estas formas de comunicação são classificadas como inviáveis. Isto deve-se ao facto de que é necessária uma reação a um comando enviado. Como não existe controlo sobre o processo para onde é enviado este comando, não é necessário estar a escrever em memória partilhada ou *pipes* porque não é possível consumir essas mensagens.

Posto isto, e como referido, os sinais *POSIX* foram o protocolo de comunicação inicial. Isto deveu-se ao facto de a gestão dos eventos (captura, execução) ser efetuada pelo sistema operativo, o que é indicado num cenário onde não se tem controlo sobre o processo. No entanto, esta forma não é nativa ao QEMU

e é dependente do sistema operativo. Mais ainda, para além de ser necessário criar os sinais, é ainda necessário criar o bloco de código que trata este sinal (*handler)* e controlar depois o processo da maneira requisitada. Adicionalmente, para cada tipo de arquitetura que queríamos emular e para cada conjunto de parâmetros teria de ser criada uma funcionalidade. Isto ia dar origem a um código desnecessariamente extenso.

Escusado será dizer que isto é extremamente limitador, pois as operações que podemos executar sobre um processo via sistema operativo são extremamente limitadas. Ainda assim, esta abordagem permitiu entender que tinha de existir alguma maneira mais natural de controlar estas instâncias, pelo que teve de ser descartada no início do seu desenvolvimento.

Na figura seguinte, está representada a criação de um *signal handler* e de uma função de inicialização de uma máquina de arquitetura *i386*.

```
/**
    Starts a i386 virtual machine receiving a disk name and iso name as params.
**/
pid_t start_i386(char * disk_name, char *iso_name)
{
    char * exec_cmd = (char *) malloc(strlen(I386_START)+strlen(disk_name)+strlen(iso_name)+ sizeof(char));

    sprintf(exec_cmd, I386_START, disk_name, iso_name);

    pid_t pid =fork();

    if(pid==0)
    {
        pid_t son_pid = getpid();

        set_stop_signal_handler();

        system(exec_cmd);

        free(exec_cmd);
        return son_pid;
    }
    return NULL;
}


/**
    Signal handler.
**/
void sig_catch(int sig_number) {
     printf("Caught signal number %d.\n",sig_number);

     char * stop_cmd = (char *)malloc(strlen("kill -STOP")+4);

     sprintf(stop_cmd,"kill -STOP %d",getpid());

     printf("%s\n",stop_cmd);


     system(stop_cmd);

     free(stop_cmd);
}
```

Figura 11 - Abordagem inicial de controlo do QEMU

É de salientar que a figura representa uma fase inicial do projeto onde não existia qualquer noção de independência de instâncias, este bloco de código estaria ligado a uma e uma só instância.

Após reunião com o orientador e posterior comunicação com a empresa, verificou-se de facto que esta abordagem seria inviável. Verificando-se essa situação, estudamos outras abordagens, sendo que existiu

uma entre as demais que se destacou por ser nativa ao QEMU, nomeadamente o *QEMU Machine Protocol,* que já foi explicado na seção 4.8.

O QEMU pode ser inicializado com um servidor de *QMP* ativo, num endereço e porta definidos. A partir daí podem ser enviados comandos *JSON* para essa combinação endereço/porta, e esses serão interpretados pelo *QMP* e as devidas ações tomadas. Esta abordagem, para além de ser nativa, já nos traz a independência de instâncias que é requisitada, visto que cada instância de QEMU irá dispor de um servidor de *QMP* com o qual podemos interagir independentemente.

Com as vantagens que o *QMP* oferece, seja facilidade de integração ou até a panóplia de comandos, ele foi escolhido como método de comunicação principal com o QEMU.

Na figura abaixo está representada uma visão do que se pretende implementar. É de salientar que a visão possivelmente mudará na fase de implementação.

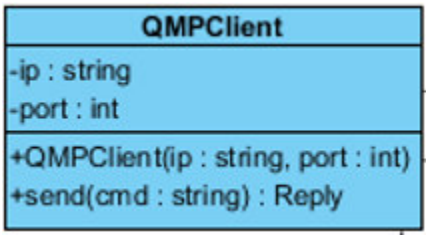


Figura 12 - Visão pretendida para o cliente QMP

5.2.3. Tratamento de comandos

Como foi referido, o *QMP* é um protocolo de comunicação baseado em *JSON.* Devido a esse facto, tornar-se-ia complicado e maçudo enviar comandos extensos para as instâncias de QEMU, e, uma vez que os comandos têm uns prefixos partilhados entre eles, em vez do utilizador ter de se lembrar desses prefixos ou de colocar sintaxe *JSON* em cada comando, verificou-se a necessidade de tratamento de comandos simplificados inseridos pelo utilizador em comandos *JSON* que o *QMP* entenderá.

Posto isto, aferimos necessária a criação de um componente na biblioteca que faça exatamente isso, receba um comando, adicione os prefixos necessários em *JSON* e devolva o comando completo que poderá posteriormente ser interpretado pelo *QMP.*

Com isto pretende-se melhorar e facilitar a experiência do utilizador, tornando a utilização da biblioteca mais intuitiva.

5.2.4. Conexão com o servidor QMP

O protocolo de comunicação *QMP* funciona abrindo um servidor *tcp* numa combinação endereço/porta. Como tal, para a nossa biblioteca conseguir comunicar com o servidor, é imperativa a implementação de funcionalidades que permitam comunicação *tcp,* numa perspetiva de cliente.

Portanto, faz sentido a criação de um módulo dedicado a este propósito: criar um cliente *tcp* cujo intuito será comunicar com uma instância de QEMU. Note-se que cada instância de QEMU será também um cliente *QMP* uma vez que queremos atingir a independência de instâncias.

Podemos recorrer ao *BOOST*[10] para atingir este objetivo, mas preferimos uma implementação mais *hands-on* onde irá ser escrito o código de criação do cliente bem como as validações necessárias, sendo que isso traz várias vantagens, entre elas a possibilidade de especialização de funcionalidades de acordo com necessidades que se apresentem. Vamos, portanto, utilizar a estrutura de dados *Socket AF_INET.*

5.2.5. Logging de dados

Numa biblioteca deste tipo, é imperativo manter *logs* das operações efetuadas, sejam problemas, erros, ou até mesmo informação relevante sobre a execução de funcionalidades.

Existem vários módulos de *Logging* disponíveis para *C++*, todavia, como queremos o máximo de controlo do fluxo de execução e não existiu qualquer restrição, implementar-se-á um módulo de *Logging* escrito no âmbito deste projeto.

As funcionalidades deste módulo serão nada mais que escrever num ficheiro de texto sempre que requisitado. Claro está, convém a utilização do padrão *Singleton* para ser possível chamar as funcionalidades do *Logger* sem ter de instanciar vários objetos. A utilização deste padrão poderá parecer facilmente redundante, porque é possível simplesmente usar funções e varáveis estáticas, mas essa solução tem um leque de desvantagens em alguma situações, nomeadamente que as funções

[10]https://www.boost.org/

estáticas não podem ser virtuais, o que invalidaria toda a abertura ao poliformismo que desejamos no desenho da biblioteca [1].

A limpeza destas variáveis também se torna complicada, uma vez que nesta linguagem a única noção nativa de *garbage collection* que existe é no conceito de *smart pointers,* sendo que o objetivo destes é providenciar toda a funcionalidade dos apontadores, adicionando mais uma panóplia de novas funcionalidades, entre elas o *garbage collection.*

Posto isto, pensou-se em 4 níveis de Logging que serão enumerados na tabela abaixo:

Tabela 8 - Níveis de Logging da biblioteca

Acrónimo	Descrição
INFO	Apenas informação relevante, p.ex. na criação de uma instância.
CRITICAL	Problema crítico, p.ex. tentar abrir uma instância não existente.
DEBUG	Opcional, para ajudar no desenvolvimento.
ERROR	Em caso de erro, p.ex. impossibilidade de comunicar com a instância.

Esta funcionalidade permitir-nos-á um melhor entendimento do fluxo de execução da biblioteca, e, em caso de erro, uma noção do sítio onde existiu esse problema e sugestões de resolução.

Na figura abaixo está representada a visão do que se pretende implementar.

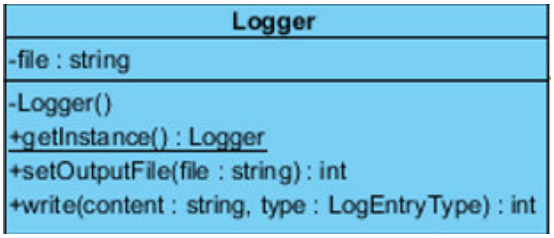


Figura 13 - Visão pretendida para o Logger

5.2.6. Configuração

Existem algumas peculiaridades que podemos necessitar de configurar de antemão, nomeadamente caminhos para o QEMU, número de portas de clientes para as instâncias e caminhos para o ficheiro de *Logging.*

Claramente, esta necessidade propõe a criação de um componente que faz o tratamento de configurações. Entenda-se como tratamento a leitura, escrita e aplicação das configurações em vigor na nossa biblioteca.

Claro está, é também necessária a inclusão de um ficheiro de configurações que é regido por diretrizes definidas por nós. Estas diretrizes são nada mais do que prefixos que este módulo de configuração está à espera de encontrar ao ler o ficheiro de configuração. A falha por parte do utilizador de respeitar estas diretrizes irá resultar em aplicação das configurações por defeito, fato que pode causar entrave aquando a execução da biblioteca, seja por conflitos de portas ocupadas ou caminhos de executável. Temos, todavia, de assumir que o ficheiro de configuração está no local correto.

5.2.7. Instância de QEMU

Até aqui, referimos várias vezes o termo 'instância de QEMU'. Uma instância de QEMU é um conceito que engloba toda a lógica da execução de uma máquina virtual, bem como a sua noção de independência. Visto que tratamos isso como um conceito, tem forçosa e logicamente que ser tratado como tal num contexto de design.

Posto isto, propõe-se a criação de um componente que será exatamente isto: um conceito logico de uma execução de QEMU, na qual disponibilizamos uma interface de comunicação com o mesmo ao exterior.

A instância de QEMU é que vai usufruir das funcionalidades de tratamento de comandos e conexão com o servidor *QMP,* visto que, como já foi referido, queremos independência de instâncias, portanto temos de aplicar esses conceitos ao nível da mesma. Mais ainda, esta independência requer a definição de um identificador único. No âmbito deste projeto, decidiu-se que este identificador seria uma variável inteira, não necessitando de qualquer tipo de persistência de dados acerca da mesma finda a execução da aplicação.

Este identificador permite referências unicamente à instância desejada, providenciando a independência lógica requisitada.

Na figura abaixo pode verificar-se uma tentativa de representação do componente em termos de implementação.

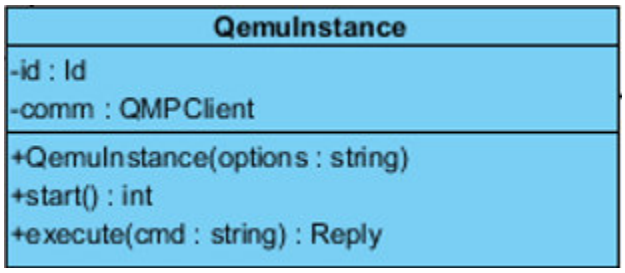


Figura 14 - Visão de implementação da QemuInstance

5.2.8. Módulo de controlo de instâncias

Já foi explicada uma noção de independência de instâncias de QEMU. Contudo, tem de existir um elemento centralizador, a partir do qual possamos operar estas instâncias.

Este componente irá agregar instâncias de QEMU e terá a responsabilidade de providenciar forma de criar/desligar instâncias, enviar comandos para cada uma ou várias instâncias, carregar configurações, listar instâncias em execução e até a captura de comandos indesejados. Devido à sua natureza, será implementado com recurso ao padrão *Singleton.*

Simultaneamente, este componente será encarregue de disponibilizar uma interface para o exterior, seja para uma GUI ou outra camada de abstração como o *Java Native Interface.* Entenda-se por disponibilização de uma interface, a possibilidade das suas funcionalidades serem invocadas por terceiros. Podemos aferir que este será o ponto de entrada na nossa biblioteca.

Na figura seguinte representamos as funcionalidades pretendidas para este ponto de entrada.

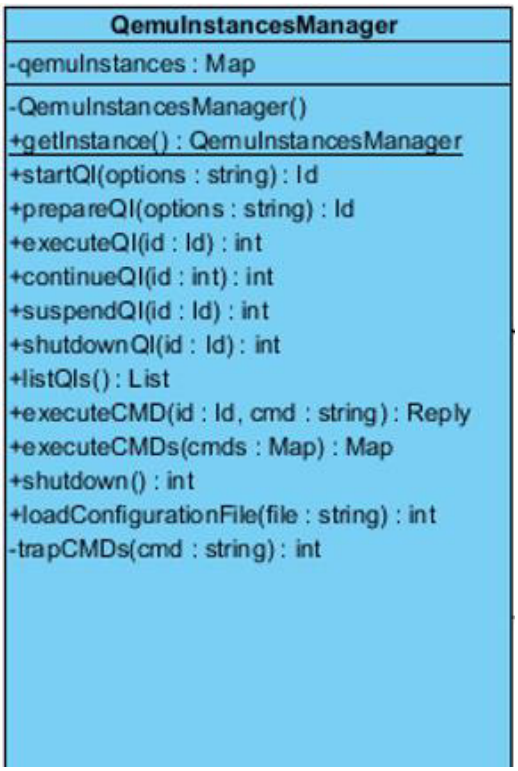


Figura 15 - Visão de implementação do QemuInstancesManager

5.2.9. Visão global da solução

Na figura abaixo está representada uma alternativa de implementação do sistema na sua globalidade. Na figura estão representadas as cardinalidades das relações, composições e agregações.

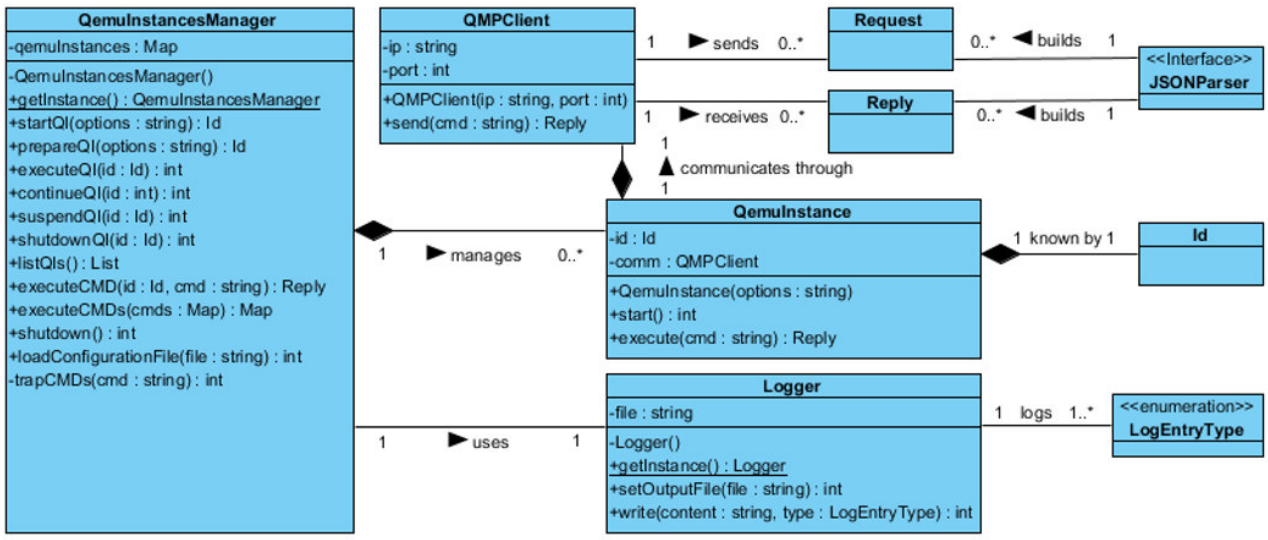


Figura 16 - Proposta biblioteca QEMU

Existem alguns conceitos como o Id da instância QEMU que apesar de já se ter decidido ser um inteiro, quisemos representar como um conceito separado, pois esta biblioteca tem de ser construída de forma modular e com possibilidade de mudanças, e deste modo queremos abstrair o mais possível o tipo de Id.

A representação do componente que trata os comandos em comandos *JSON* é uma interface porque pode ser interessante aplicar polimorfismo e obter várias classes distintas, encarregues de tratar estes comandos, para colmatar diferentes necessidades. Imaginemos que temos um utilizador mais avançado e um com conhecimentos básicos. O utilizador com mais conhecimentos pode querer já usar comandos menos simplificados enquanto que o utilizador com menos conhecimentos pode querer utilizar comandos mais simplificados ao início, e, com a sua familiarização, alterar numa fase posterior.

O mesmo se passa na enumeração dos *logs,* pode ser interessante deixar em aberto os tipos de logs que podemos utilizar, sendo que apesar de estar representado como um conceito, o *LogEntryType* pode ser alterado para uma simples enumeração na classe *Logger.*

A figura acima representa uma abordagem inicial, todavia após mais estudo da solução necessária, e mais especificidade de requisitos e da linguagem, procedeu-se ao desenho de um novo modelo de domínio que representa os conceitos identificados e as ligações entre os mesmos.

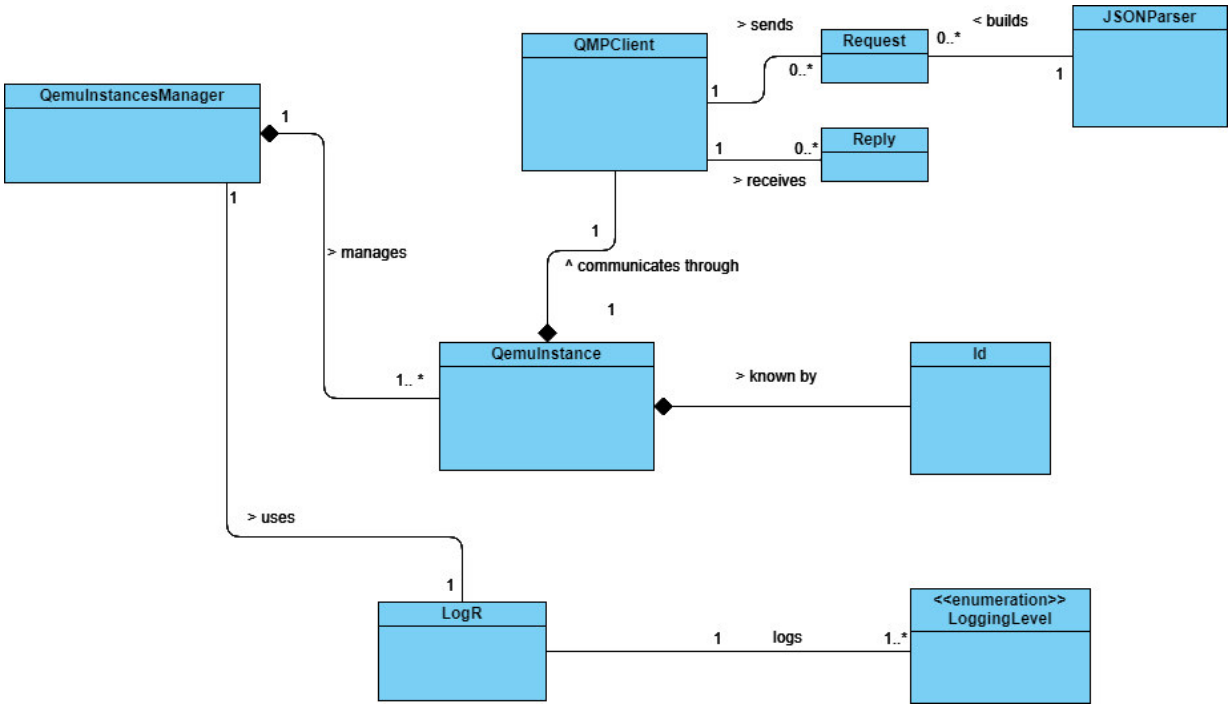


Figura 17 - Modelo de Domínio

Neste modelo de domínio pode aferir-se que existiram alterações nomeadamente na forma como o QMPClient interage com o JSON parser aquando do envio de um comando. A resposta não vem do JSONParser, mas sim do QEMU, pelo que existiu a necessidade de refletir isso no *design*.

Na figura abaixo está representada uma vista de processos com granularidade de aplicação, para apoiar na compreensão da relação entre os conceitos durante o processo de envio de um comando.

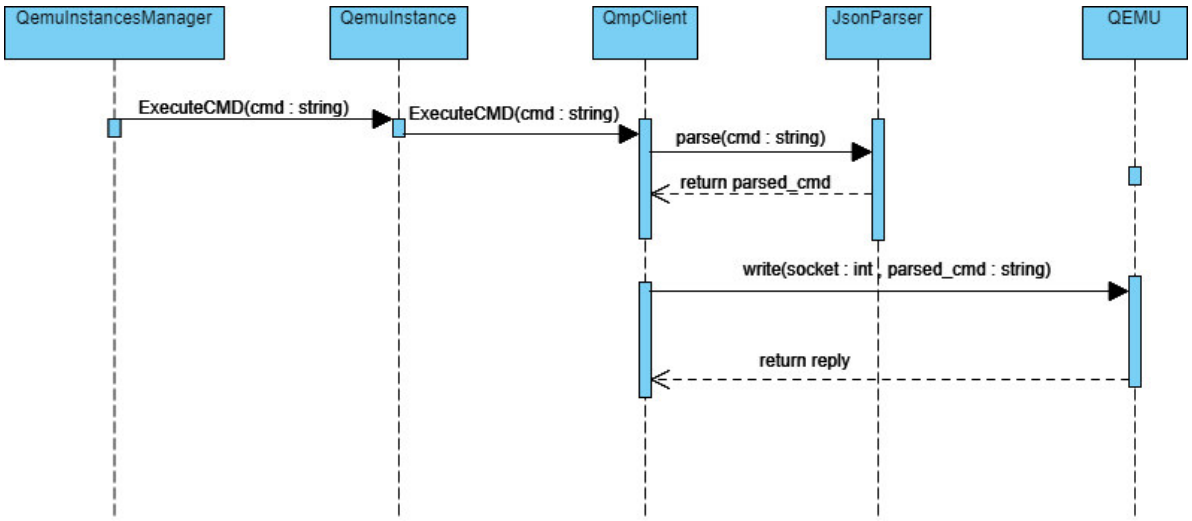


Figura 18 - Vista processos c/ granul. Aplicação

Como isto é uma biblioteca, não contém um UI acoplado, portanto a forma como está desenhada a aplicação é semelhante ao MvC (*Model View Controller)*, admitindo que a camada controladora é o componente que terá de ser escrito aquando a integração da biblioteca com um eventual módulo uma vez que separa os conceitos de domínio do mesmo. Qualquer UI ou módulo irá apenas conhecer e interagir com esta camada.

5.3.Desenvolvimento da solução

Esta seção seguirá a mesma estrutura da seção anterior. Iremos abordar os componentes que compõe a biblioteca um a um ao nível de implementação.

É de salientar que existe uma possibilidade que tenham existido mudanças em relação ao que foi analisado, e sempre que tal se verificar, essas mudanças irão ser refletidas nesta seção.

5.3.1. Métodos de comunicação com o QEMU

Começamos por apresentar a estrutura da classe utilizando o seu ficheiro *header.*

```
class QMPClient{

//default constructor
public: QMPClient();

//parametrized constructor
public: QMPClient(string ip, int port);

//string that stores the ip address
private: string ip_address;

//int that stores the port
private: int port;

//boolean value to check if its connected
private: bool isConnected;

//private file descriptor for the communication socket
private: int socket_fd;

//TODO: should(?) change return type to Reply
public: string send(string cmd);

//Reads reply after a send command
private: string read_reply();

//private method to avoid connecting everytime we want to send
private: void connect_to_server();

public: ~QMPClient();

};
```

Figura 19 - Ficheiro .h do QMPClient

Verificam-se prontamente algumas diferenças perante o que foi apresentado na análise. Adicionamos uma variável para verificar se a conexão já estava realizada. Isto porque se optou por manter a conexão aberta após a primeira conexão para poupar recursos e minimizar conexões e limpeza de portas desnecessárias.

Existem também funções de leitura do *socket* que não constam da análise. Naquela fase, as respostas do *QMP* seriam ignoradas, mas após consulta com o orientador e cliente, verificou-se que estas seriam pertinentes e como tal a sua implementação foi realizada.

Em relação ao envio de comandos para o QEMU, a próxima figura representa esta ação.

```
string QMPClient::send(string cmd){

    if(this->isConnected == false)
    {
        connect_to_server();
    }
    if(this->isConnected == true){
        Json_Parser parser;
        string parsed_cmd = parser.parse(cmd);
        write(this->socket_fd,parsed_cmd.c_str(), parsed_cmd.size());
        read_reply();
        return parsed_cmd;
    }
    return "";
}
```

Figura 20 - Envio de comando para o QEMU

Existem algumas validações para verificar se a conexão foi feita. Pode verificar-se também que o comando recebido por parâmetro é tratado pelo *JsonParser* cuja implementação será explicada mais abaixo.

Após ter o comando tratado, basta enviá-lo para o descritor do socket e ler a resposta logo de seguida, visto que é instantânea, e mesmo que não fosse, a leitura de *sockets,* nesta implementação, é uma operação bloqueante.

A implementação da conexão *tcp* segue o padrão para a mesma que é criar o descritor do *socket,* criar uma estrutura do tipo *sockaddr_in* do tipo *AF_INET,* fazer as devidas atribuições aos campos da estrutura e depois usar a chamada ao sistema denominada por *connect.* Por último, realizam se as diferentes atribuições necessárias às variáveis definidas na classe *QMPClient.* A figura abaixo irá apresentar essa funcionalidade.

```
void QMPClient::connect_to_server()
{
    int socket_fd = socket(AF_INET,SOCK_STREAM,0);
    if(socket_fd <0)
    {
        cout << "Error opening socket.\n";
    }
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    const char * addr_const_char = this->ip_address.c_str();

    in_addr_t in_addr = inet_addr(addr_const_char);
    if (INADDR_NONE == in_addr)
    {
        cout << "Invalid address.\n";
    }
    addr.sin_addr.s_addr = in_addr;

    addr.sin_port = htons(this->port);

    if(connect(socket_fd,(struct sockaddr *) &addr,sizeof(addr))<0)
    {
       cout << "failed to connect\n";
    }
    else
    {
        this->socket_fd = socket_fd;

        this->isConnected = true;
    }
}
```

Figura 21 - Conexão ao servidor QMP

5.3.2. Tratamento de comandos

No cliente *QMP* já vimos o conceito de tratamento de comandos em ação. Nesta seção vamos apresentar a implementação sucinta deste componente.

Como já foi referido, existem vários prefixos que são comuns a vários comandos. Como tal, começou-se por identificar essas situações e definir no cabeçalho da classe essas macros.

```
#define JSON_SEPARATOR ","
#define JSON_BEGINNING "{ \"execute\":"
#define JSON_QUOTES "\""
#define JSON_ARGUMENTS  ", \"arguments\": {"
#define JSON_CMD_LINE "command-line"
#define JSON_HMP "human-monitor-command"
#define JSON_COLON ":"
#define JSON_ENDING "}"
```

Figura 22 - Macros Json_Parser

Para além de suporte a comandos *QMP*, existiu a preocupação de suporte de comandos *human-monitor* porque alguns destes podem ter propriedades únicas ao nível da sintaxe. Adicionalmente, o mecanismo de throttle cuja implementação será detalhada no capítulo seguinte, irá ser chamado via comando *human-monitor.*

Na figura seguinte está representada a estrutura da classe de tratamento de comandos *Json.*

```
class Json_Parser{

//empty constructor
public: Json_Parser();

//public parser method, parses a command like "qmp_capabilities" and adds all the JSON tags
public: string parse(string command);

//public parser method with different separator, parses a command like "qmp_capabilities" and adds all the JSON tags
public: string parse(string command,string separator);

//private method that does the real parsing, supports human monitor commands
private: string parse_p(string command, string separator);

//utility method to split a string by a separator
private: vector<string> split(string data, string token);

public: ~Json_Parser();

};
```

Figura 23 - Estrutura da classe Json_Parser

Abaixo é representado um comando *QMP* para desligar uma máquina e o respetivo comando que podemos enviar à biblioteca para efetuar a mesma operação, para se verificar a simplicidade comparativa entre ambos.

O comando *QMP* teria de ser estruturado assim:

```
{ "execute": "quit" }
```

O comando simplificado seria simplesmente 'quit'. Ao escalarmos a escrita destes comandos para as dezenas ou centenas podemos ver o tempo poupado bem como o evitamento de erros de sintaxe por falta de algum dos símbolos chave.

Como foi referido também é possível enviar comandos *human-monitor.* Um comando deste tipo teria obrigatoriamente de estar estruturado desta forma:

```
{ "execute": "human-monitor-command", "arguments": { "command-line": "info
                                    kvm" } }
```

Este comando permite obter informações acerca do KVM. Na biblioteca, de forma simplificada o comando ficaria da seguinte forma:

```
human-monitor-command, info kvm
```

Adiciona-se o prefixo *human-monitor-command* para prevenir ambiguidades porque existem comandos disponíveis simultaneamente no *QMP* e no monitor do QEMU.

Na figura abaixo apresenta-se a implementação da funcionalidade de tratamento de comandos.

```
string Json_Parser::parse_p(string command, string separator){
    string complete_command = JSON_BEGINNING;
    if(command.find(JSON_HMP)!=string::npos)
    {
        vector<string> split_command = split(command,separator);
        complete_command += JSON_QUOTES + split_command[0]+ JSON_QUOTES + JSON_ARGUMENTS + JSON_QUOTES + JSON_CMD_LINE + JSON_QUOTES + JSON_COLON;
        for(int i=1;i<split_command.size();i++)        {
            complete_command += JSON_QUOTES + split_command[i] + JSON_QUOTES;
        }
        complete_command += JSON_ENDING;
        complete_command += JSON_ENDING;
        return complete_command;
    }
    complete_command += JSON_QUOTES + command + JSON_QUOTES + JSON_ENDING;
    LogR::log(LogR::INFO,complete_command,__FUNCTION__,__FILE__);
    return complete_command;
}
```

Figura 24 - Tratamento de comandos

Facilmente se consegue verificar que existe um tratamento mais complexo para comandos do tipo *human-monitor.*

Adicionalmente, conseguimos ver neste bloco de código uma chamada à função responsável pelo *logging (*função *log)* que servirá para nos dar informações acerca dos comandos tratados.

Existe a possibilidade de tratar comandos com separadores diferentes. Por acaso utilizamos a ‘,’ nesta implementação, mas a funcionalidade está construída de forma extensível, pelo que poderíamos trocar por qualquer outro símbolo. A própria funcionalidade de dividir uma *string* por um separador também teve de ser implementada, não é suportada pelo *C++* standard.

5.3.3. Logging de dados

Como já foi referido, o componente de *Logging* foi implementado de raiz, pelo que é possível manipular de acordo com as necessidades específicas do projeto.

Na figura seguinte apresenta-se a estrutura desta classe via ficheiro *header.*

```
class LogR{

//defines the logging level to use when logging
public: enum LoggingLevel { INFO, DEBUG, ERROR, CRITICAL };

//file path where logs will be stored
public: static string file_path;

//to log the current time at the log event
private: static std::time_t time_now;

//logs a command into the defined file
public: static void log(LoggingLevel level, string content, string function, string file);

//sets the file path
public: static void set_log_file(string file_path);

//translates enum in string : cpp does not support by default
private: static string get_enum_string(LoggingLevel level);

};
```

Figura 25 - Ficheiro .h da classe LogR

Esta classe tem funcionalidades de escrita em ficheiros, uma variável que irá conter o tempo atual (para efeitos de *Logging)* e a enumeração dos níveis de *Logging* que já foram explicados e se mantiveram inalterados desde a fase de análise.

É de salientar que implementamos este componente baseando-o no padrão *Singleton* para que seja possível fazer *Logging* a partir de qualquer componente.

A figura abaixo representa a parte pertinente da implementação desta classe.

```
string LogR::get_enum_string(LoggingLevel level)
{
    switch(level)
    {
        case LogR::INFO:
            return "INFO";
            break;

        case LogR::CRITICAL:
            return "CRITICAL";
            break;
        case LogR::DEBUG:
            return "DEBUG";
            break;
        case LogR::ERROR:
            return "ERROR";
            break;
        default:
            return "UNDEFINED";
            break;
    }
}

void LogR::log(LoggingLevel level, string content, string function, string file_detected)
{
    ofstream file;
    time_now  = std::time(nullptr);
    file.open(LogR::file_path,ios::out | ios::app);
    if(file.is_open())
    {
         file << std::put_time(std::localtime(&time_now), "%y-%m-%d %OH:%OM:%OS") <<
         "("+LogR::get_enum_string(level)+ "): " + content + "\n\t : in function: " + function + ", "
         + file_detected +"\n";
    }

    file.close();
}
```

Figura 26 - Implementação da classe de Logging

A primeira funcionalidade (*get_num_string)* permite obter o valor da enumeração dos níveis de *Logging* em formato *string* visto que o *C++* não permite obter os valores de variáveis de enumeração neste formato, por defeito.

A segunda funcionalidade (*log)* é essencialmente o propósito deste componente, permite-nos escrever no ficheiro formatando ao nosso gosto. No ficheiro, uma entrada ficará no seguinte formato:

$$(ERROR): 22: 04: 20: \ * ERROR\ DESCRIPTION * \ in\ function\ XXXX\ in\ line\ XX$$

5.3.4. Configuração

A configuração é aplicada no arranque da aplicação. Caso não seja encontrado um ficheiro de configuração, esta irá arrancar com os valores por defeito, sendo que fica praticamente inutilizável devido à necessidade de um executável válido. Por defeito, ele procura o ficheiro de configuração na pasta do executável.

Na figura abaixo está representada a estrutura da classe. Esta classe foi implementada recorrendo ao padrão *Singleton.*

```
class Configuration{

public:
    Configuration();

    static string QEMU_PATH;
    static string QMP_TERMINATION_1;
    static string QMP_TERMINATION_2;
    static string LOG_FILE_NAME;
    static string LOG_FILE_PATH;
    static int STARTING_PORT;
    static int read_configuration(string config_path);
    static vector<string> split(string data, string token);

};
```

Figura 27 - Estrutura da classe de configuração

Existem quatro configurações que necessitam de ser feitas para um funcionamento saudável da biblioteca que podem ser consultadas na tabela seguinte.

Tabela 9 - Configurações disponíveis

Configuração	Descrição
QEMU_PATH	Caminho para o QEMU
LOG_FILE_NAME	Nome do ficheiro de log
LOG_FILE_PATH	Caminho do ficheiro de log
START_PORT	Porta inicial do servidor QMP

Como tal, apresenta-se na figura abaixo a implementação da leitura do ficheiro de configuração.

```
int Configuration::read_configuration(string config_path)
{
  string line;
  ifstream config (config_path.c_str());
  if (config.is_open())
  {
    while ( getline (config,line) )
    {
        if(line.find("QEMU_PATH")!=string::npos)
        {
            vector<string> data_path= split(line,"=");
            Configuration::QEMU_PATH = data_path[1];
            continue;
        }
        if(line.find("LOG_FILE_NAME")!=string::npos)
        {
           vector<string> data_log= split(line,"=");
           Configuration::LOG_FILE_NAME = data_log[1];
           continue;
        }
        if(line.find("LOG_FILE_PATH")!=string::npos)
        {
           vector<string> data_log_path= split(line,"=");
           Configuration::LOG_FILE_PATH = data_log_path[1];
            LogR::log(LogR::INFO, data_log_path[1] ,__FUNCTION__,__FILE__);
           continue;
        }
        if(line.find("START_PORT")!=string::npos)
        {
           vector<string> data_port= split(line,"=");
           Configuration::STARTING_PORT = atoi(data_port[1].c_str());
           continue;
        }
    }
    config.close();
  }

    else{ return -1; }

    return 0;
```

Figura 28 - Implementação de leitura da configuração

Após leitura do ficheiro de configuração, fazem-se as respetivas atribuições aos membros estáticos da classe.

5.3.5. Instância de QEMU

A noção de instância de QEMU é um dos aspetos cruciais da biblioteca. Até aqui falamos de noção de independência de cada instância, devido à necessidade de envio de comandos separados ou em conjunto.

Na figura abaixo está representada a estrutura da classe *Qemu_Instance.*

```
class Qemu_Instance{

//constructs a QI taking in the system_init information
public: Qemu_Instance(string system_init);

//qemu instance identifier -> this may be changed to a value object if needed
private: int id;

//the system of the QI -> x86_64-softmmu for instance
public: string system_init;

//QMP client instance used to communicate with this instance
private:  QMPClient comm;

//executes a command in this instance, returns int for now, will be changed to a "Reply"
public: int execute_cmd(string system, string options);

//starts this QI
public: int start_machine();

public: ~Qemu_Instance();

};
```

Figura 29 - Estrutura da classe Qemu_Instance

O variável '*system_init*' é importante. Esta guarda o valor que diz respeito ao executável de QEMU que irá ser arrancado, bem como parâmetros adicionais, como por exemplo imagens de disco, quantidade memória *RAM*, entre outros.

Pode verificar-se também a existência de um cliente *QMP* para ser possível enviar comandos para o servidor *QMP* que é criado aquando o arranque da máquina via parâmetros específicos. É privado porque não será necessário referenciar o mesmo fora do contexto da instância.

O '*execute*_cmd' também é importante para conseguirmos executar comandos. É uma camada de abstração para não termos de referenciar o cliente *QMP* fora deste contexto.

A funcionalidade '*start_machine*' é a única que temos relativamente à instância, sendo que existe pelo simples facto de antes da máquina estar ligada, não existe qualquer servidor *QMP* para enviarmos comando de inicialização. É também a razão pela qual não existe mais nenhum comando em relação à

instância, sendo que qualquer operação desejada como desligar ou reiniciar a máquina, será enviada para o *QMP.*

Na figura abaixo, podemos ver a implementação desta classe.

```
//initializes communication with default values
Qemu_Instance::Qemu_Instance(string system_init): comm()
{
    this->system_init= system_init;
}

int Qemu_Instance::execute_cmd(string command,string options){
    string result = comm.send(command);
    if(result=="")
    {
        return 1;
    }
    return 0;
}

int Qemu_Instance::start_machine()
{
    const char * init = this->system_init.c_str();
    pid_t pid;
    pid=fork();
    if(pid==0)
    {
        system(init);
        exit(0);
    }
}
```

Figura 30 - Implementação da classe Qemu_Instance

O cliente *QMP* é inicializado *inline* aquando a construção do objeto *Qemu_Instance.* A máquina é iniciada recorrendo ao *fork* do processo principal, seguido de uma chamada ao sistema para execução da *string* de inicialização, que irá conter o caminho para o executável e respetivos parâmetros.

5.3.6. Módulo de controlo de instâncias

Esta classe é a que está encarregue de agregar instâncias de QEMU bem como operar sobre elas. É também aqui que se faz grande parte dos *logs,* mais especificamente quando se cria ou se envia qualquer comando a uma instância de QEMU.

Na figura seguinte podem verificar-se as funcionalidades principais desta classe.

```
//Starts a qemu instance
public: int start_QI(string options);

//Builds a qemu instance
public: int prepare_QI(string options);

//Executes a qemu instance
public: int execute_QI(int id);

//Continues a stopped qemu instance
public: int continue_QI(int id);

//Suspends a running qemu instance
public: int suspend_QI(int id);

//Shuts a qemu instance down
public: int shutdown_QI(int id);

//lists all available qemu instances
public: list<Qemu_Instance> list_QIs();

//Executes a qemu instance
public: int executeCMD(int id, string cmd);

//Executes cmds taking in a map as param
public: void execute_cmds(map<int,string> cmds_map);

//Shuts down this instance
public: int shutdown();

//Loads configuration file taking the path as param
public: int load_configuration_file(string file);

//Traps unwanted cmds
private: int trap_cmds(string cmd);

//index of the next built qemu index in the map
private: int map_iterator_index;
```

Figura 31 - Estrutura da classe Qemu_Instances_Manager

A função *prepare_QI* é extremamente importante, serve, como o nome indica, para preparar uma instância de QEMU. Usa grande parte dos valores das variáveis do ficheiro de configuração para construir o parâmetro de inicialização. Após isso, a instância de QEMU é inserida numa estrutura do tipo mapa cuja chave é um inteiro (uma referência à instância, mas abstraída da mesma) e o valor é a instância em si.

O resto das funções referentes à execução de comandos, inicialização da máquina, entre outros, são apenas chamadas às funções da classe *Qemu_Instance* uma vez que já temos as máquinas preparadas e inseridas no mapa.

5.3.7. Interface de utilizador

No âmbito da implementação desta biblioteca foi escrita uma UI em consola em *C++* muito rudimentar. Permite utilizar as funcionalidades todas da biblioteca de maneira intuitiva. No entanto, foi criada uma GUI em *Java* e a integração da biblioteca com a mesma foi requisitada para provar a modularidade da biblioteca.

Como já foi abordado, usou-se o *Java Native Interface (*JNI). Isto requer a criação de uma interface de funções nativas do lado do *Java* e uma abstração do ponto de entrada da biblioteca que é o *Qemu_Instances_Manager.*

Cada chamada ao *JNI* cria um contexto novo, pelo que existiram problemas em manter o mesmo. Por exemplo, imaginemos que se criava uma instância, quando se tentava referir a essa instância para a começar ou terminar era impossível fazê-lo porque o contexto era diferente.

Este problema ultrapassou-se utilizando apontadores para as referências. Abaixo estão representadas algumas funcionalidades desta camada de abstração.

```
JNIEXPORT jlong JNICALL Java_model_JniCppIntegration_CreateMGR(JNIEnv *env, jobject){
    Qemu_Instances_Manager * mgr = new Qemu_Instances_Manager();
    return (long)mgr;
};

JNIEXPORT jint JNICALL Java_model_JniCppIntegration_Start(JNIEnv *env, jobject, jstring options, jlong ptr)
{
    Qemu_Instances_Manager * mgr = (Qemu_Instances_Manager*)ptr;
    const char * path;
    path = env->GetStringUTFChars(options,JNI_FALSE) ;
    jint result = mgr->start_QI(path);
    return result;
};
```

Figura 32 - Exemplo de chamadas a funções C++ a partir do Java

Pode verificar-se na primeira função que se cria uma instância do *Qemu_Instances_Manager* e depois se retorna o apontador desse objeto para o *Java.* A partir daí enviamos este apontador por parâmetro para todas as funcionalidades e, dentro da função o *cast* é feito para o objeto desejado, como se pode verificar na função *Start.*

Por defeito, já existe um mapeamento específico para tipos primitivos do *C++* para tipos de dados em *Java* como por exemplo o *Jint.* É de salientar que o tipo *string* em *Java,* ou *Jstring* no contexto *JNI* é especialmente complicado de converter para uma *string* em *C++.*

Operações normalmente simples tanto em *C++* como em *Java* de manipulação de listas são especialmente complicadas como se pode aferir pela figura abaixo.

```
JNIEXPORT jobjectArray JNICALL Java_model_JniCppIntegration_ListQI(JNIEnv *env, jobject, jlong ptr)
{
    Qemu_Instances_Manager * mgr = (Qemu_Instances_Manager*)ptr;
    jobjectArray ret;
    list<Qemu_Instance> list = mgr->list_QIs();
    int list_size = list.size();
    int i;
    ret= (jobjectArray)env->NewObjectArray(list_size,
    env->FindClass("java/lang/String"),
    env->NewStringUTF(""));
    for(i=0;i<list_size;i++) {
        env->SetObjectArrayElement(ret,i,env->NewStringUTF(list.front().system_init.c_str()));
        list.pop_front();
    }

    return ret;
};
```

Figura 33 - Manipulação de listas via JNI

Também tiveram de ser efetuadas alterações do lado do *Java* para assegurar a compatibilidade com a biblioteca como está representado nas figuras abaixo.

```
/**
 * A private constructor to ensure the singleton pattern.
 */
private CppManager() {

    System.load("/home/renato/Documents/qemu-lib/src/cpp/src/out/libqemu_instances_manager.so");
    CppCaller = new JniCppIntegration();
    managerPointer = CppCaller.CreateMGR();
}
```

Figura 34 - Inicialização da biblioteca em Java

Na figura acima é realizado o carregamento da biblioteca e a criação de um *manager* que serve como ponto de entrada na biblioteca.

```
@Override
public ExecutionResult prepareInstance(Command options) {
    System.out.println(options.instruction());
    int val = CppCaller.Prepare(options.instruction(),managerPointer);
    CppExecutionResult result = new CppExecutionResult("Instance Prepared.");
    return result;
}
```

Figura 35 - Chamada à função prepare

Na figura cima está demonstrada a chamada da função *prepareInstance* que serve para preparar uma instância de QEMU. Esta função faz uma chamada à função *prepare_instance* da biblioteca. Note-se que é preciso enviar o apontador do *manager* nessa chamada.

5.4.Testes e experiências

Em termos de testes e após consulta com o cliente, verificou-se que não era necessário algo muito extenso visto que o código está bem protegido, isto é, apenas existe modificador de acesso publico no que é necessário. Não obstante, foram realizados testes aos diversos componentes do sistema. Note-se que algumas funcionalidades são impossíveis de testar, dado que são operações internas ao QEMU e única forma que existe de saber se aquela funcionalidade faz de facto o que é pedido é implementar testes ao nível do QEMU, e isso está fora do contexto deste projeto. Foram, portanto, realizados testes ao tratamento de comandos, criação de instâncias, destruição de instâncias, execução de comandos, etc. Nestes casos, usou-se o tamanho do mapa como referência e valores de retorno para execução de comandos, sendo que no contexto de testes, um comando é bem-sucedido quando é enviado para o *QMP* sendo que depois ele interpretá-lo-á e poderá não ser válido no contexto do QEMU.

Para a criação dos testes, recorreu-se ao BOOST. Foi criado um executável com os testes, manualmente. Com o BOOST, existe a possibilidade de criar *test suits* que é uma forma de organizar os testes.

Na figura abaixo estão representados alguns dos testes implementados.

```
BOOST_AUTO_TEST_SUITE(json_parser_test_suite)

BOOST_AUTO_TEST_CASE(ensure_command_is_parsed) {
  Json_Parser test_parser;
  BOOST_CHECK_EQUAL(test_parser.parse("stop"),"{ \"execute\":\"stop\"}");
  BOOST_CHECK_EQUAL(test_parser.parse("human-monitor-command,t_cpu 20",","),"{ \"execute\":\"human-monitor-command\", \"arguments\": {\"command-line\":\"t_cpu 20\"}}");
}

BOOST_AUTO_TEST_SUITE_END()



BOOST_AUTO_TEST_SUITE(qemu_instances_manager_test_suite)

BOOST_AUTO_TEST_CASE(ensure_instance_is_created) {
  Qemu_Instances_Manager manager;
  manager.prepare_QI("test");
  BOOST_CHECK_EQUAL(manager.list_QIs().size(),1);
}

BOOST_AUTO_TEST_CASE(ensure_instance_gets_added_to_list) {
  Qemu_Instances_Manager manager;
  manager.prepare_QI("test");
  manager.prepare_QI("test2");
  manager.prepare_QI("test3");
  BOOST_CHECK_EQUAL(manager.list_QIs().size(),3);
}



BOOST_AUTO_TEST_SUITE_END()



BOOST_AUTO_TEST_SUITE(configuration_test_suite)

BOOST_AUTO_TEST_CASE(ensure_configuration_file_is_read) {
  Configuration::read_configuration("./config_test.txt");
  BOOST_CHECK_EQUAL(Configuration::STARTING_PORT,0000);
  BOOST_CHECK_EQUAL(Configuration::QEMU_PATH,"TEST");
  BOOST_CHECK_EQUAL(Configuration::LOG_FILE_NAME,"TEST");
  BOOST_CHECK_EQUAL(Configuration::LOG_FILE_PATH,"TEST");
}

BOOST_AUTO_TEST_SUITE_END()
```

Figura 36 - Testes com BOOST

Em termos de experiências foram realizadas várias execuções e demonstrações, tanto na UI em *C++* como na GUI em *Java* o que ajudou muito no despiste de erros e posterior resolução. Adicionalmente, o projeto em execução foi apresentado aos parceiros de projeto e o balanço do mesmo foi positivo.

5.4.1. Visão da solução obtida

Na figura abaixo está representado o diagrama de classes da solução obtida, bem como as relações de composição e agregação entre as classes.

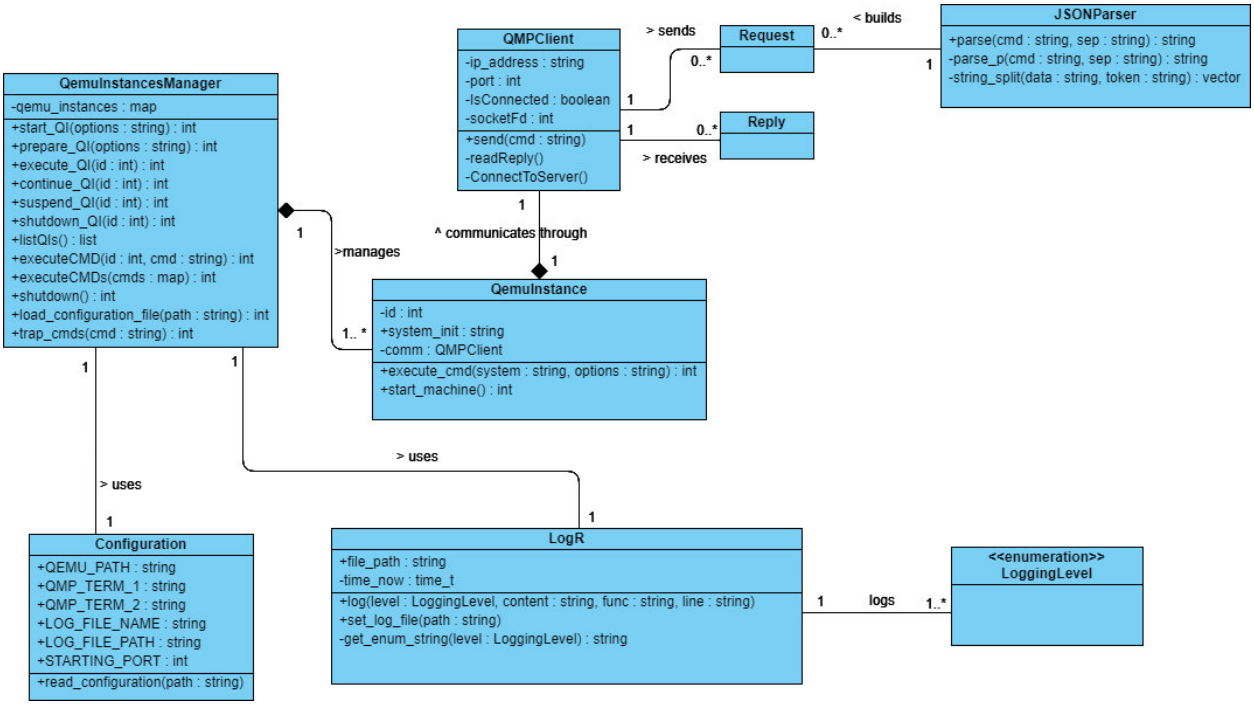


Figura 37 - Diagrama de classes QEMU-LIB

Capítulo 6. Throttle

O mecanismo de *throttle* é a parte mais inovadora e ambiciosa do projeto. Já foi explicado anteriormente que o QEMU não dispõe desta funcionalidade nativamente, pelo que tem de ser implementada directament no seu código-fonte. Como tal, uma análise cuidada ao código existente é fulcral[11].

Neste capítulo é apresentada a análise realizada ao código-fonte do QEMU, o desenvolvimento da funcionalidade de throttle e os testes realizados sobre o mesmo.

6.1.Análise e Modelação

Já foram explicados no Capítulo 4 os blocos de código do QEMU pertinentes para a implementação do mecanismo de *throttle*. Como a funcionalidade consiste na redução da velocidade de execução de *CPUs*¸ faz sentido analisar o bloco de código responsável por operá-los. Assumindo que é possível implementar esta funcionalidade, faz parte dos requisitos a sua exportação como um comando monitor, de modo a se utilizável quando necessário.

A análise começou pelos ficheiros *cpus.c* (encarregue de código geral de *CPUs), hmp.c* (encarregue de implementar funções que podem ser chamadas pelo monitor e *hmp-commands.hx* (encarregue de fornecer uma interface de comandos para o monitor e mapear com as funções do ficheiro anterior). Para entender melhor o funcionamento dos semáforos e das *I/O threads,* também teve de ser analisado o ficheiro *main-loop.c*.

6.1.1. Código geral de CPUs

O código contido no ficheiro *cpus.c* é extenso (2060 linhas de código). Por conseguinte, no contexto deste relatório, vai ser apenas abordada a parte mais importante.

[11] Muita da informação acerca do QEMU foi obtida por recompilação e execução, bem como pesquisas em *mailing-lists* dos desenvolvedores do QEMU.

Após pesquisa e análise, verificou-se que existe uma função de *throttle* já implementada. Esta função é utilizada internamente em situações de migração de máquinas virtuais entre diferentes hosts, possivelmente de arquiteturas diferentes. Essencialmente, as frequências dos *CPUs* baixam para as instruções poderem ser traduzidas entre hosts/arquiteturas.

É de salientar que no contexto deste projeto o comportamento esperado é aquele que simula de maneira mais realista o sistema real, isto é, ao baixar a frequência do *CPU,* todo o sistema terá de refletir essa mudança, incluindo os relógios e temporizadores. Na figura seguinte representamos o código desta funcionalidade.

```
static void cpu_throttle_thread(CPUState *cpu, run_on_cpu_data opaque)
{
    double pct;
    double throttle_ratio;
    long sleeptime_ns;

    if (!cpu_throttle_get_percentage()) {
        return;
    }

    pct = (double)cpu_throttle_get_percentage()/100;
    throttle_ratio = pct / (1 - pct);
    sleeptime_ns = (long)(throttle_ratio * CPU_THROTTLE_TIMESLICE_NS);

    qemu_mutex_unlock_iothread();
    g_usleep(sleeptime_ns / 1000); /* Convert ns to us for usleep call */
    qemu_mutex_lock_iothread();
    atomic_set(&cpu->throttle_thread_scheduled, 0);
}
```

Figura 38 - Código principal do mecanismo de throttle

Observando o código da função acima, podemos tirar as seguintes conclusões:

- O throttle opera segundo uma percentagem;
- A partir da percentagem, obtém-se um rácio;
- Através dessa informação, calcula-se o tempo de inatividade;

- Essa inatividade é aplicada via *microsleep* (função *g_usleep()* que faz adormecer a execução do programa e cujo *input* é especificado em microsegundos) libertando o semáforo para outras operações.

Conseguimos também aferir via adição de código de *debug* que esta função está constantemente a ser chamada, todavia, como a percentagem de throttle é 0, ela não tem qualquer efeito na execução da máquina virtual.

Na figura abaixo pode verificar-se a rearmação do temporizador cujo objetivo da função de callback é colocar a função de *throttle* no processador, para posterior execução. A função representada na figura acima faz a alteração de uma *flag (throttle_thread_scheduled)* que é utilizado como verificação na função da figura abaixo, para ver se nesse *cpu,* vai ser necessário escalonar a função de *throttle.*

```
static void cpu_throttle_timer_tick(void *opaque)
{
    CPUState *cpu;
    double pct;

    /* Stop the timer if needed */
    if (!cpu_throttle_get_percentage()) {
        return;
    }
    CPU_FOREACH(cpu) {
        if (!atomic_xchg(&cpu->throttle_thread_scheduled, 1)) {
            async_run_on_cpu(cpu, cpu_throttle_thread,
                             RUN_ON_CPU_NULL);
        }
    }

    pct = (double)cpu_throttle_get_percentage()/100;
    timer_mod(throttle_timer, qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL_RT) +
                                CPU_THROTTLE_TIMESLICE_NS / (1-pct));
}
```

Como todas estas funções não são chamadas caso a percentagem de *throttle* seja 0, é importante conseguirmos criar um comando que permita chamar a seguinte função.

```
void cpu_throttle_set(int new_throttle_pct)
{
    /* Ensure throttle percentage is within valid range */
    new_throttle_pct = MIN(new_throttle_pct, CPU_THROTTLE_PCT_MAX);
    new_throttle_pct = MAX(new_throttle_pct, CPU_THROTTLE_PCT_MIN);

    atomic_set(&throttle_percentage, new_throttle_pct);

    timer_mod(throttle_timer, qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL_RT) +
                                        CPU_THROTTLE_TIMESLICE_NS);
}
```

Figura 39 - Função que muda pct de throttle

A interface que providencia os comandos monitor é denominada por *hmp-commands.hx.* Para além da definição do comando nessa interface, é necessário mapeá-lo numa função que será criada no ficheiro *hmp.c.* Essa função terá como objetivo chamar a função representada na Figura 39. Para se ter acesso a um contexto onde essa função poderá ser chamada, terá de ser incluído o ficheiro *cpus.c* no ficheiro *hmp.c.*

Posto isto, partiu-se para o estudo dos algoritmos que constituem o mecanismo de *throttle.* Inicialmente foi extraída uma função matemática utilizando a nossa interpretação desses algoritmos, representada abaixo na sua versão simplificada:

$$T = \left(\frac{C}{Cs}\right) * (Ts)$$

O C representa o tempo de computação, ou seja, a duração da tarefa. O Cs representa o *clock step,* isto é, a duração do temporizador que chama a função de *throttle.* Este valor cresce em função da percentagem de *throttle*. Ao dividir os dois, conseguimos obter o número total de disparos de temporizadores que ocorreram durante a execução. O parâmetro Ts refere-se ao tempo de sleep, ou tempo de inatividade, calculado através dos algoritmos acima, tendo como base uma percentagem de *throttle.* Ao conhecer a duração do tempo de inatividade, podemos multiplicar esse pelo número de temporizadores que disparam durante a execução C, obtendo assim T, ou seja, o tempo de inatividade que ocorreu durante C.

Analisando a função acima através da utilização de valores de *input* teóricos, é possível obter uma curva que deverá ser semelhante à curva obtida em testes ao mecanismo efetuados à *posteriori*. Estes valores de *input* referem-se a uma percentagem de *throttle* e como *output* a duração de uma tarefa hipotética é obtida. Nesta experiência, a percentagem de *throttle* varia em intervalos de 0.1, de 0 até 0.9. Esta entrada, ou tempo hipotético de duração de tarefa, começa em 1 e vai duplicando, até atingir o valor de 384.

Ao inserir estes valores, obteve-se a curva representada na figura seguinte:

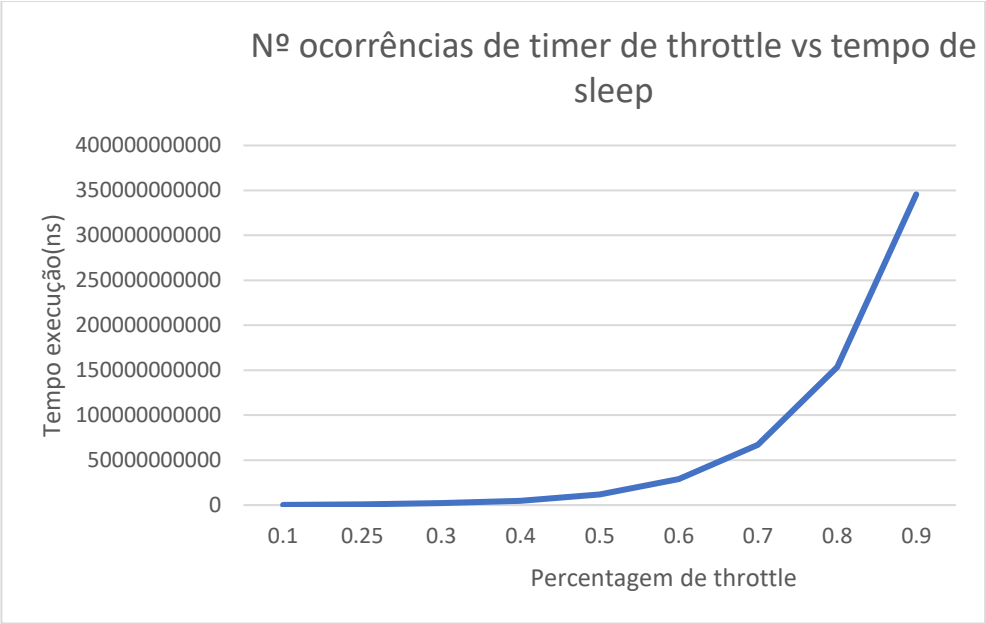


Figura 40 - Gráfico que representa estudo teórico do mecanismo

Pode verificar-se a partir da figura, que a curva começa a ser extremamente acentuada para valores de *throttle* acima de 0.5. Este resultado será utilizado como base aquando da realização dos testes com tarefas reais e (expectávelmente) deverá ser semelhante à curva obtida na figura acima.

6.2. Desenvolvimento da solução

No que toca ao desenvolvimento da solução, esta seguiu um processo de tentativa-erro, visto que a documentação é escassa.

Por conseguinte, este processo envolveu a alteração de pequenas partes do código e subsequente compilação, verificando-se os efeitos posteriormente.

Como foi referido no ponto 6.1, o objetivo é criar um comando que possibilite a chamada da função representada na Figura 39.

Decidiu-se realizar uma aproximação do interior para o exterior, sendo que definimos a função no ficheiro *hmp.c* e só depois o comando monitor no ficheiro *hmp-commands.hx.* Na figura seguinte é apresentada a primeira destas implementações

```
void hmp_throttle_cpu(Monitor *mon,const QDict *qdict)
{
    int pct = qdict_get_int(qdict, "index");
    cpu_throttle_set(pct);
    monitor_printf(mon, "CPU executing at %d%% speed.\n",100-pct);
}
```

Figura 41 - Comando interno de chamada da função de throttle

Os apontadores enviados por parâmetro são um padrão de envio de dados de forma estruturados posto em prática pelos desenvolvedores do QEMU. Mais especificamente, através da estrutura *QDict* podemos aceder a parâmetros, como por exemplo a percentagem de throttle. Este acesso não é feito de forma direta, existe também uma funcionalidade para ir buscar o valor destes parâmetros (que depende do seu tipo), e funciona através da pesquisa de chaves.

Salienta-se que no ficheiro *hmp.c* temos acesso a muitas das funções do QEMU, visto que este ficheiro serve para exportar muitas dessas funcionalidades como comandos. Uma dessas funções é precisamente a que nos interessa, nomeadamente a função *cpu_throttle_set.* Apesar de ser possível criar uma referência à mesma a partir deste contexto, não existia nenhuma função dedicada para tal. Como foi referido, esta função era apenas utilizada internamente para efeitos de migração.

Agora que a função está definida, é preciso criar uma entrada no ficheiro *hmp-commands.hx* que nos permita utilizar a mesma, representado na figura abaixo. Podemos verificar no parâmetro *.cmd* do comando abaixo a definição da função representada na Figura 39 - Função que muda pct de throttleFigura 39.

```
@item throttle_cpu
@findex throttle_cpu
ETEXI
    {
        .name       = "t_cpu|throttle_cpu",
        .args_type  = "index:i",
        .params     = "index",
        .help       = "throttles the cpu to given pct",
        .cmd        = hmp_throttle_cpu,
    },
STEXI
```

Figura 42 - Comando Haxe para invocação da função de throttle

Após definição deste comando e subsequente recompilação do QEMU, foi possível invocar o mecanismo de *throttle*, obtendo resultados visíveis de *throttling* a percentagens altas. Por resultados visíveis entenda-se a interação com o utilizador de uma forma geral. Por exemplo, a 99% de *throttle* o sistema ficava praticamente inutilizável, a interação com o utilizador era extremamente lenta.

6.2.1. Conclusões da invocação do mecanismo

Apesar dos resultados serem promissores, nada nos garante que numa perspetiva lógica os efeitos do mesmo são os esperados. Durante algum tempo acreditou-se que o mecanismo estaria completamente funcional, mas obviamente seriam necessários testes a comprovar o comportamento.

O tipo de testes que seriam interessantes aplicar eram testes que utilizassem os relógios do QEMU. Como foi referido na explicação da tecnologia, o QEMU dispõe de 3 relógios pertinentes no âmbito deste projeto.

Não existe qualquer funcionalidade nem comando de monitor que permita obter os valores dos mesmos, sendo que esta funcionalidade teve de ser implementada no âmbito do projeto.

Adicionalmente, após submissão do mecanismo a testes, caso não se verificasse qualquer efeito nos relógios, seria interessante calcular o efeito que o *microsleep* da função de *throttle* iria ter sobre os mesmos. A implementação da funcionalidade de obtenção dos valores contidos nos relógios do QEMU teve sucesso. Esta implementação seguiu o padrão descrito acima relativo à implementação de um comando monitor.

```
void hmp_get_clock(Monitor *mon,const QDict *qdict)
{
    int64_t rt;
    int64_t ct;
    //int64_t ti;
    double pct;
    double throttle_ratio;
    int64_t sleep_time_ns;
    pct = (double)cpu_throttle_get_percentage()/100;
    throttle_ratio = pct / (1 - pct);
    sleep_time_ns = throttle_ratio * 10000000;
    rt = qemu_clock_get_ns(QEMU_CLOCK_REALTIME);
    ct = qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL);
    //ti = cpu_get_clock();
    monitor_printf(mon, "Current RT Clock: %" PRId64 "\nCurrent VT Clock: %" PRId64 "\n",rt,ct);
    monitor_printf(mon, "Calculated VT Clock: %" PRId64 "\n",ct - sleep_time_ns);
    monitor_printf(mon, "Current Clock Skew: %" PRId64 "\n",ct - (ct- sleep_time_ns));
}
```

Figura 43 - Função de obtenção dos valores dos relógios

O cálculo do relógio virtual, influenciado pelo *microsleep (*manualmente) é já apresentado nesta função. Para este cálculo foi simulada a influência do tempo de *sleep* encontrado na função de throttle, subtraíndo este valor ao relógio virtual à *posteriori*. Após a obtenção do valor deste relógio teste, foi calculado o desvio em relação ao relógio com os valores inalterados.

Posto isto, foi criado o comando monitor para invocação desta função.

```
@item cpu_get_clock
@findex cpu_get_clock
ETEXI
    {
        .name       = "get_clk|get_clock",
        .args_type  = "",
        .params     = "",
        .help       = "gets_current_vm_clock",
        .cmd        = hmp_get_clock,
    },
STEXI
```

Figura 44 - Comando Haxe para a invocação da função de obtenção dos valores dos relógios

6.3.Testes e experiências

Apesar do mecanismo de *throttle* ter efeitos visíveis no que toca à interação com o utilizador, poderá não ter o efeito desejado no resto dos componentes de um sistema. Como tal, pensou-se em algumas abordagens para poder testar estes efeitos, sendo uma delas a obtenção dos valores dos relógios, para verificar se o incremento temporal nos mesmos se mantém ou se existe alguma alteração quando se aumenta a percentagem de throttle. Foram executadas várias instâncias de testes usando esta abordagem, estando um exemplo representado na figura seguinte.

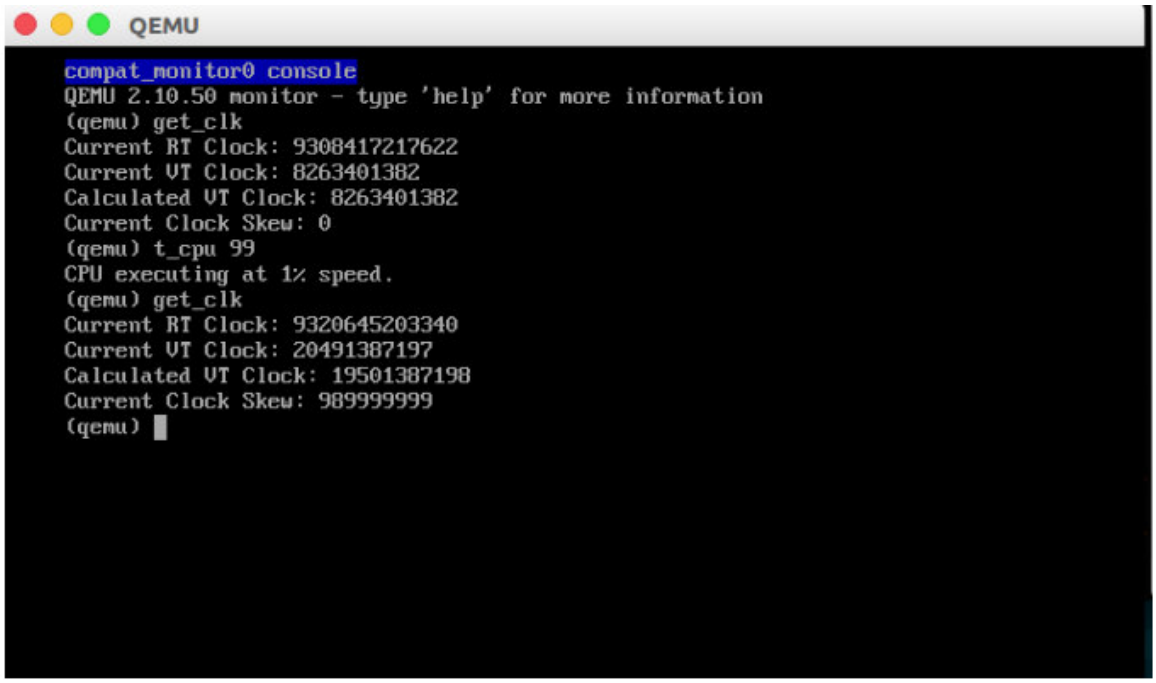


Figura 45 - Testes de influência do throttle nos relógios

Esta figura representa ação de obter os valores dos relógios *real-time* (RT), *virtual* (VT) e o relógio calculado como descrito na seção 6.2.1.

Após vários testes, verificou-se que o incremento do relógio virtual era independente da percentagem de *throttle*. Isto é um resultado negativo para o mecanismo, uma vez que o incremento devia ser mais baixo (menos *ticks* do processador deviam causar um incremento mais lento nos relógios).

A partir destes resultados é possível concluir que o mecanismo de *throttle,* apesar de ter efeitos visíveis não se está a comportar da forma esperada numa perspetiva lógica. Ainda assim, foram seguidas outras abordagens em termos de testes, antes de se partir para uma atualização do mecanismo de throttle.

Uma das abordagens propostas pelo aluno foi a implementação de uma aplicação com arquitetura cliente/servidor comunicando via *sockets* de rede.

O objetivo desta aplicação é estudar como se comporta uma aplicação a executar numa máquina com *throttling* ativo, durante X segundos, enquanto uma aplicação na máquina *host* mede também o tempo real de execução. No final da execução de ambas as aplicações, pode-se comparar os tempos de execução para avaliar qual o verdadeiro impacto do mecanismo de *throttle*.

6.3.1. Aplicação Qemu-Clock Tester

Na figura abaixo está representado um diagrama de sequência que detalha a sequência das operações da aplicação de testes desenvolvida para testar o mecanismo de *throttle,* descrita no parágrafo anterior. É importante referir que a classe *chrono*[12] faz parte do C++ *standard.*

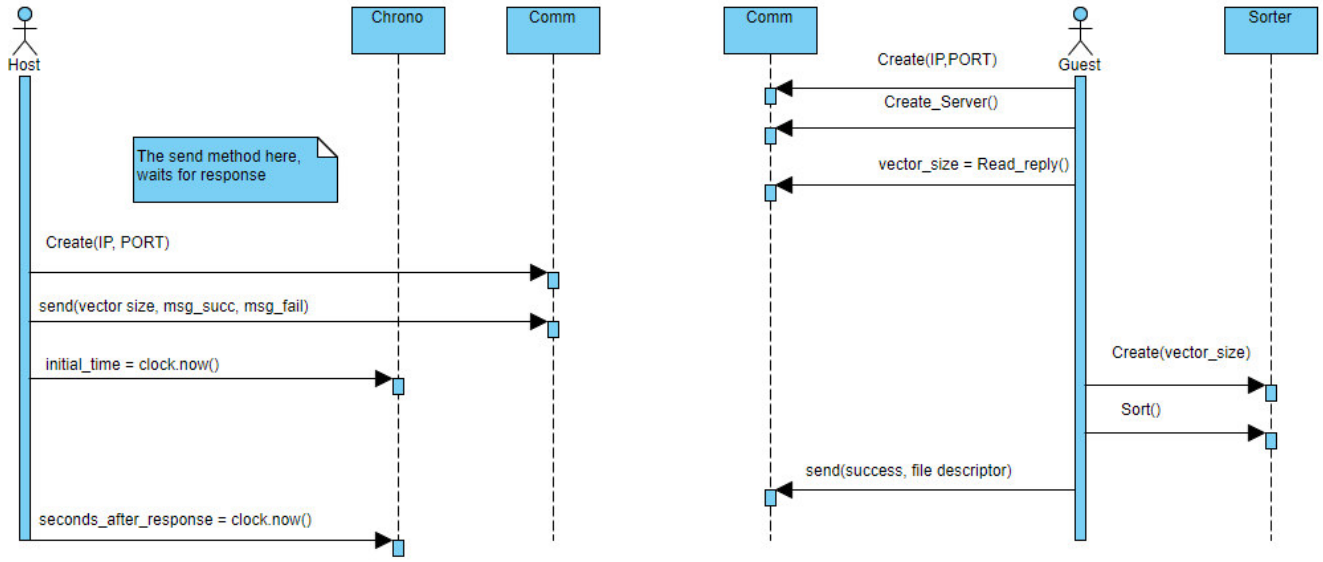


Figura 46 - Diagrama de sequência do clock-tester

[12]http://www.cplusplus.com/reference/chrono/

Essencialmente, o que se pretende com esta aplicação de testes é verificar algo semelhante ao seguinte comportamento: uma aplicação que, por exemplo, entra num ciclo cuja condição de paragem é atingir cinco segundos de execução. Esses cinco segundos, numa máquina a executar a 50% da capacidade de processamento, teria de resultar numa medição de mais de cinco segundos na máquina *host.* Para a máquina *throttled,* os cinco segundos iam permanecer, porque o incremento do tempo e a própria contagem iam estar mais lenta, todavia, com a medição de uma máquina que tivesse fora desse contexto, esperar-se-ia obter resultados acima de cinco segundos.

Esta foi a premissa que levou à implementação desta aplicação. Como tal, procedeu-se à criação de duas aplicações separadas denominadas por *host-tester* e *guest-tester.*

O propósito da aplicação que ficou encarregue de correr na máquina *host* era nada mais do que ser o cliente que liga ao servidor (*guest)*. Essencialmente, ela mede o tempo desde o momento em que se conecta ao *guest (*que é também o momento em que este começa a executar), até que recebe uma resposta do mesmo. Na figura seguinte pode verificar-se parte do código fonte da aplicação *host.*

```
int main(int argc, char* argv[])
{


    Comms com("127.0.0.1",atoi(argv[1]));
    chrono::steady_clock::time_point seconds_initial;
    chrono::steady_clock::time_point seconds_after_response;
    seconds_initial = chrono::time_point_cast<std::chrono::milliseconds>(chrono::steady_clock::now());
    com.send(argv[2],"test_success","test_failed");//received timeslice in seconds
    seconds_after_response = chrono::time_point_cast<std::chrono::milliseconds>(chrono::steady_clock::now());
    cout << "Response received...\n";
    chrono::duration<float> delta = seconds_after_response - seconds_initial;
    cout << "The requested " << atoi(argv[2]) << " vector size ordered in " << delta.count() << " seconds in the guest machine.\n";
    ofstream file;
    file.open("/home/renato/Desktop/data.txt",ios::out | ios::app);
    if(file.is_open())
    {
         file << delta.count() << "," << atoi(argv[2]) << "," << atoi(argv[3]) << "\n";
         cout << "Data saved in log file.\n";
    }
    file.close();
    return 0;
}
```

Figura 47 - Código host test

Na parte do *guest,* o seu trabalho era estar à escuta de conexões. Assim que recebe uma conexão e posterior tempo de execução por parâmetro, entrará num ciclo até atingir esse tempo de execução (numa das primeiras versões da aplicação de testes). Assim que sair desse ciclo, envia resposta para o

cliente a informar do sucesso ou falha do teste. Salienta-se novamente que esta contagem de tempo deveria ser influenciada pelo mecanismo de *throttle.*

```
chrono::steady_clock::time_point seconds_initial;
chrono::steady_clock::time_point seconds_elapsed;
seconds_initial = chrono::time_point_cast<std::chrono::milliseconds>(chrono::steady_clock::now());
seconds_elapsed = seconds_initial;
chrono::duration<float> delta = seconds_elapsed - seconds_initial;

while(delta.count() < (atof(response.c_str())))
{
    cout << "\x1B[2J\x1B[H";
    cout << "Sleeping for " << response << " seconds.\n";
    cout << "Delta:" << delta.count() <<"\n";
    cout << "\x1B[2J\x1B[H";
    cout << "Sleeping for " << response << " seconds..\n";
    cout << "Delta:" << delta.count() <<"\n";
    cout << "\x1B[2J\x1B[H";
    cout << "Sleeping for " << response << " seconds...\n";
    cout << "Delta:" << delta.count() <<"\n";

    seconds_elapsed = chrono::time_point_cast<std::chrono::milliseconds>(chrono::steady_clock::now());
    delta = seconds_elapsed - seconds_initial;
}
cout << "Sending success response back...\n";
com.send("test_success",client_fd);
```

Figura 48 - 1º Tentativa código guest test

Após vários testes com esta aproximação verificou-se que o mecanismo de *throttle* não estava a ter qualquer efeito na contagem de tempo, sendo que o tempo medido na máquina *host* era igual ao tempo pedido de execução.

Este problema podia ter origem em vários lados desde obtenção interna dos relógios por parte do *C++* a uma errada implementação do mecanismo de *throttle.* Para realizar o despiste de problemas, alterou-se a condição de paragem de tempo de execução para uma ordenação de um vetor, utilizando um algoritmo de complexidade $n * \log n$. Este algoritmo pode ser encontrado no *namespace std* do *C++*.[13]

Após a realização de vários testes usando esta aproximação verificou-se finalmente que existia uma influência sobre os tempos de execução. Uma máquina a executar a 10% da sua capacidade de processamento, demorava muito mais a ordenar um vetor do que uma máquina a 50% ou 60% da sua capacidade de processamento, como é esperado.

[13] https://en.cppreference.com/w/cpp/algorithm/sort

Isto leva-nos a concluir que o problema estaria em algo relacionado com a contagem de tempo, mas que o mecanismo de throttle está pelo menos parcialmente correto. Este problema com a contagem de tempo tanto pode ocorrer devido à forma como o *C++* obtém os relógios ou incrementa os mesmos, ou do fato do mecanismo de *throttle* ter influência na capacidade e velocidade de processamento, mas não nos relógios.

Após pesquisa mais detalhada, verificou-se que o QEMU usa por defeito o relógio do *host* como *baseline* para o relógio do *guest,* atualizando este último periodicamente. Ao buscar o valor dos relógios via comando que foi implementado, pausando a máquina (que devia parar o relógio de acordo com a documentação), voltando a executar a máquina e posteriormente buscando o valor dos relógios de novo, verificou-se que a pausa não teve efeito no relógio. Isto deve-se às atualizações assistidas pelo sistema *host.* No entanto podemos obrigar o QEMU usar um relógio alocado apenas à máquina virtual em detrimento de um relógio assistido pelo *host* via parâmetro no arranque do mesmo.

Após realização do teste explicado acima, verificou-se que o QEMU se comportou como dita a documentação: o comando de pausa de facto congelou os relógios durante esse tempo [14].

Voltou-se, portanto, à primeira abordagem de testes do mecanismo de *throttle,* cuja condição de paragem seria a contagem de tempo para se poder aferir que o mecanismo estava a ter efeito nos relógios e temporizadores.

Mais uma vez, o teste teve um resultado negativo, uma vez que o tempo continuava a passar à mesma velocidade, independentemente da percentagem de *throttle.* Este resultado prevê que o problema é mesmo na implementação do mecanismo visto que o resto das possibilidades já foram descartadas.

É normal que o mecanismo de *throttle* não influencie os relógios, o seu propósito era apenas baixar a capacidade de processamento para a realização de migrações, em detrimento duma implementação preocupada com a simulação de sistemas reais.

Como tal, tiveram de ser novamente realizadas alterações a este mecanismo. Tentaram-se várias abordagens, desde pausar as *threads* virtuais durante a duração do *microsleep* até o bloqueio de semáforos relativos a estas *threads.* Todos estes testes resultaram num congelamento do QEMU ou em total indiferença por parte do mesmo aquando da sua execução.

No entanto, existe uma instância em que se verificou efeitos nos relógios, nomeadamente quando se testou se o comando de pausa funcionava como estava documentado.

Posto isto, partiu-se para um estudo da implementação interna da funcionalidade de pausa. É uma funcionalidade extensa, todavia através da mesma foi possível verificar a existência de algumas funcionalidades que possivelmente iriam permitir o congelamento dos relógios. Após uma enorme quantidade de tentativa-erro e compilações, foi possível aferir quais destas funções eram necessárias para atingir o comportamento desejado no mecanismo de *throttle.*

A figura seguinte apresenta o código do mecanismo já com as alterações efetuadas.

```
static void cpu_throttle_thread(CPUState *cpu, run_on_cpu_data opaque)
{
    double pct;
    double throttle_ratio;
    long sleeptime_ns;

    if (!cpu_throttle_get_percentage()) {
        return;
    }

    pct = (double)cpu_throttle_get_percentage()/100;
    throttle_ratio = pct / (1 - pct);
    sleeptime_ns = (long)(throttle_ratio * CPU_THROTTLE_TIMESLICE_NS);

    cpu_disable_ticks();
    bdrv_drain_all();
    replay_disable_events();

    qemu_mutex_unlock_iothread();

    g_usleep(sleeptime_ns / 1000); /* Convert ns to us for usleep call */

    qemu_mutex_lock_iothread();

    replay_enable_events();
    bdrv_flush_all();
    cpu_enable_ticks();

    atomic_set(&cpu->throttle_thread_scheduled, 0);


}
```

Figura 49 - Mecanismo de throttle funcional

As grandes diferenças desta implementação para a anterior são as chamadas às funções `cpu_disable_ticks`, `bdrv_drain_all`, `replay_disable_events` e respetivas funções inversas. Estas funções são chamadas, depois o mecanismo de *throttle* nativo executa, e no fim o estado da máquina é reposto. As duas últimas são relacionadas com interrupções e eventos, todavia a primeira é realmente interessante. Esta funcionalidade permite desligar a contagem dos *ticks* do processador. Enquanto estes estão desativados, os temporizadores não são incrementados. Desativando os *ticks* durante o *microsleep,* garante que durante essa fatia de tempo, os temporizadores e relógios mantêm o seu estado atual. Após subsequente compilação, voltou-se a testar a influencia do mecanismo de *throttle* tendo como condição de paragem a contagem de tempo, e verificou-se que os resultados eram promissores. De fato, o tempo medido no exterior era maior do que o tempo pedido de execução.

Todavia, esse sucesso não serve como conclusão. São necessários resultados palpáveis, que possam ser medidos e estudados, sendo que, obter uma conclusão do funcionamento e do impacto do mecanismo de *throttle* tem de ser possível a partir destes resultados.

Foi, portanto, proposta uma bateria de testes a partir dos quais fosse possível chegar a conclusões acerca da influência do mecanismo de *throttle* no tempo de execução de uma tarefa, estando esses testes discriminados nas tabelas seguintes.

6.3.2. Resultados obtidos

Esta subseção vai ser dividida em duas partes, uma para cada teste apresentado.

6.3.2.1. Teste 1

Este primeiro teste consiste em verificar quanto tempo demora a ordenação de um vetor de 100000 posições utilizando um algoritmo com complexidade $\boldsymbol{\log_2(n) * n}$ tal como explicado no ponto 0. Consiste em 20 execuções por cada 10% de *throttle*, terminando aos 90%. No total, por teste, foram realizadas 200 execuções.

Na tabela seguinte podemos ver as informações relativas a este teste. Os dados completos destes testes não estão presentes neste documento (devido à sua extensão e por não trazerem valor adicional ao documento), sendo que serão apresentados de forma sucinta no âmbito deste capítulo.

Tabela 10 - Resultados teste 1

% Throttle	Média(s)	Máximo(s)	Mínimo(s)	Desvio Padrão
0	4.689	6.038	4.519	0.315962973
10	5.47095	6.201	5.363	0.175741991
20	6.17155	6.306	6.072	0.0607507
30	8.1262	8.373	7.714	0.120701118
40	10.8088	10.935	10.702	0.062405609
50	16.73	17.782	12.436	1.305378872
60	20.395	23.39	15.789	2.142271248
70	30.2596	31.835	28.412	0.871112874
80	41.25005	42.786	38.751	0.947770461
90	88.4505	99.894	75.24	6.194104499

Podemos facilmente aferir a influência do mecanismo de *throttle* no tempo de execução. É complicado tirar conclusões tendo apenas em atenção os números obtidos, portanto para este teste realizaram-se vários gráficos que permitem ilustrar o comportamento do mecanismo. O gráfico permite estudar a curva dos dados.

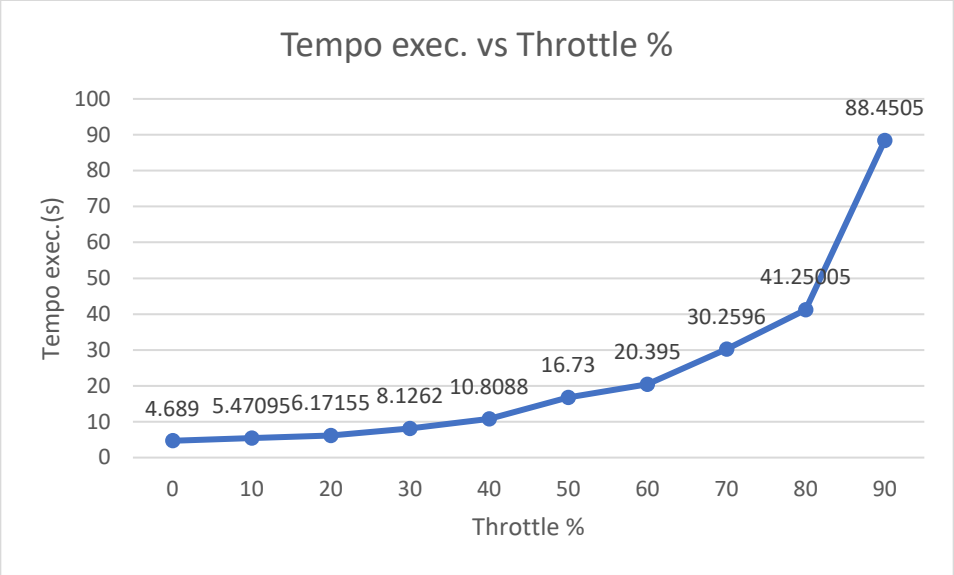


Figura 50 - Gráfico tempo de exec. vs % de throttle

Pode verificar-se que o incremento do tempo de execução não é linear, mas sim quadrático, o que é curioso. Adicionalmente, esta curva corresponde à curva obtida aquando a análise teórica do mecanismo de *throttle,* que pode ser consultada no ponto 6.1.1. Através do desvio padrão podemos tirar conclusões acerca da dispersão dos dados, pelo que, a partir deles, foi criado um gráfico de dispersão[14].

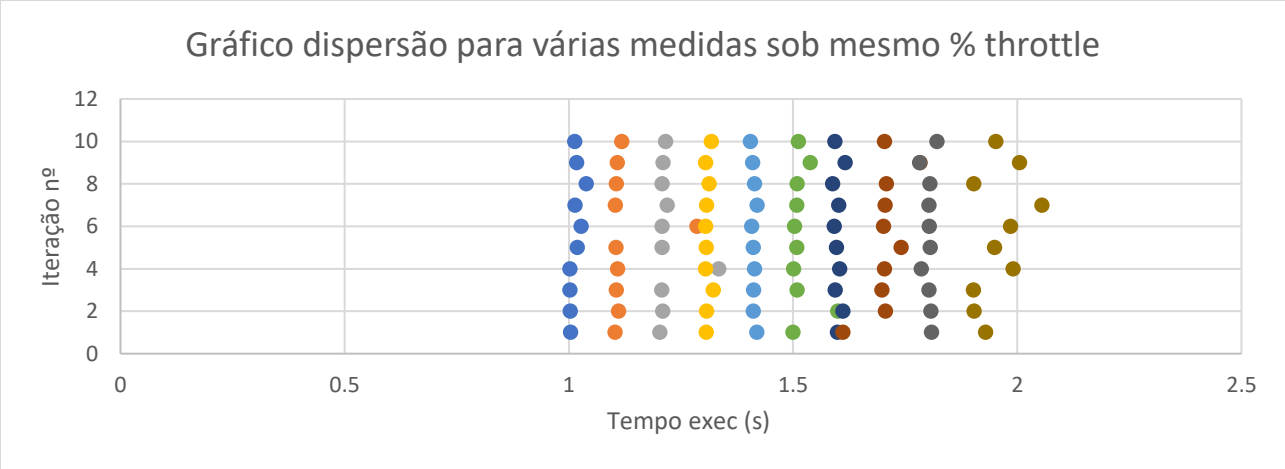


Figura 51 - Gráfico de dispersão dos dados

O gráfico de dispersão acima relaciona várias execuções do teste referido com o tempo de execução.

[14] http://www.statisticshowto.com/probability-and-statistics/regression-analysis/scatter-plot-chart/

Cada cor diz respeito à submissão sob uma percentagem de *throttle.* Cada ponto da mesma cor diz respeito a instâncias de execução sob a mesma percentagem de *throttle.*

Posto isto, a partir deste gráfico conclui-se que, quanto maior a percentagem de *throttle* (e por conseguinte maior tempo de execução), mais os dados obtidos nas execuções são dispersos, como se pode verificar por exemplo nas instâncias que demoraram aproximadamente 2 segundos de execução: houve uma dispersão de dados mais significativa nestas execuções, nomeadamente valores a resultarem entre 1.8 e 2.2 segundos.

No gráfico seguinte podemos verificar a influência da percentagem de *throttle* no desvio padrão.

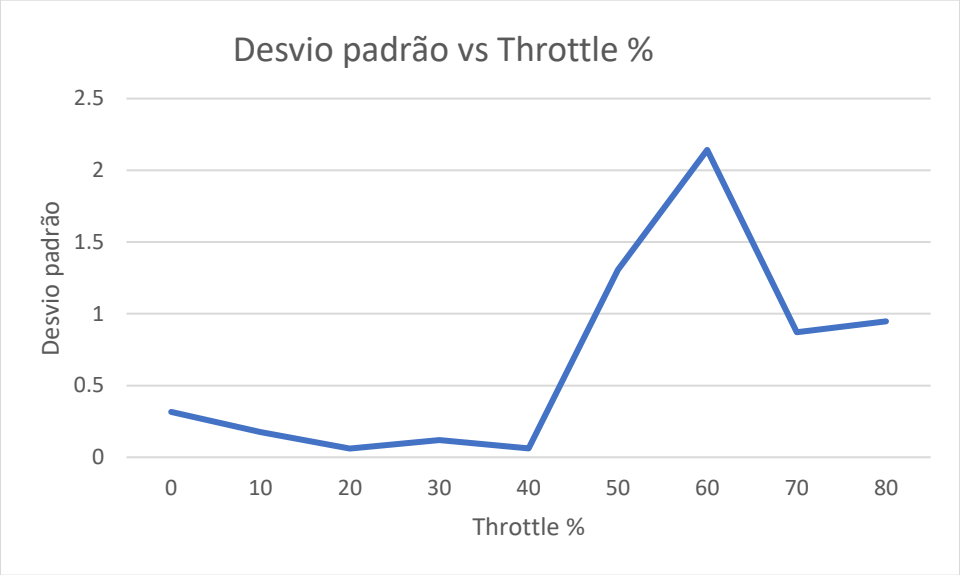


Figura 52 - Desvio padrão vs % throttle

O gráfico acima mostra que a partir dos 40% de *throttle*, o desvio padrão aumenta significativamente. Existem algumas flutuações ao chegar aos 60%, mas isto pode dever-se a vários fatores, como por exemplo, a máquina que está a executar a aplicação de testes estar ocupada com outros processos naquele momento. Como é complicado estudar esta dispersão de dados através de uma visão global, foi criado um gráfico *boxplot* para cada percentagem de *throttle* que foi medida.

Na figura seguinte pode verificar-se a distribuição e dispersão dos dados para 10% de throttle, por exemplo.

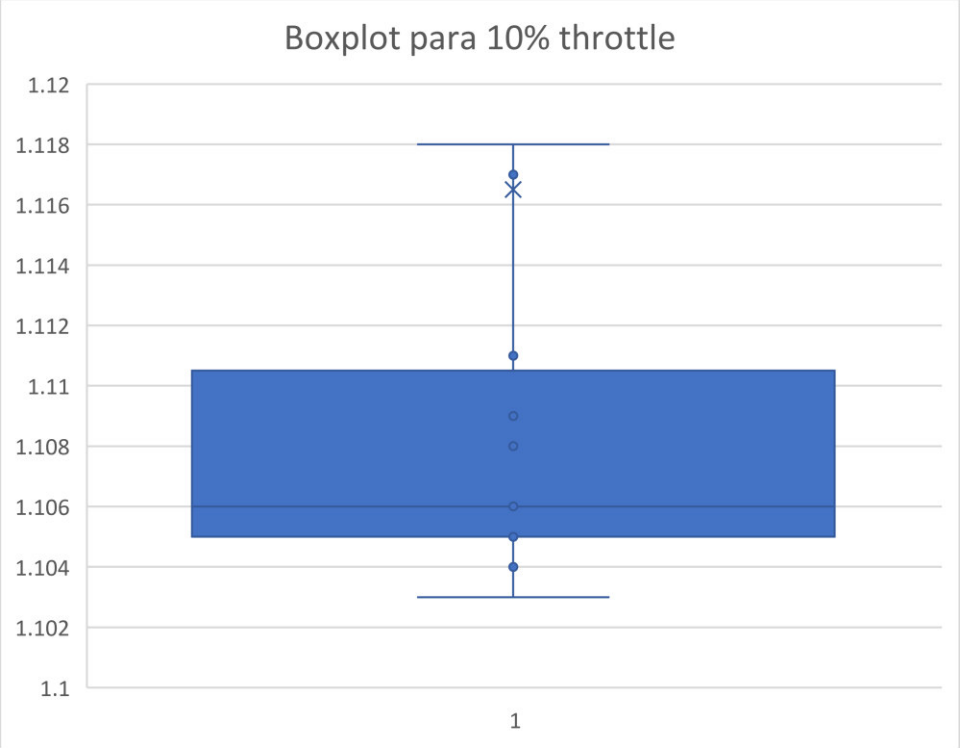


Figura 53 - Boxplot para 10% de throttle

Tendo em conta a granularidade das medidas, conclui-se que os valores obtidos são pouco dispersos para 10% de throttle, resultando em valores entre aproximadamente 1.104 e 1.118 segundos.

No que toca a testes baseados em tamanho de vetor, foram realizados mais dois, um com metade e outro com um quarto das iterações deste último, respetivamente. Os resultados da curva são semelhantes, isto é, continua a observar-se uma relação quadrática entre a percentagem de *throttle* e o tempo de execução.

6.3.2.2. Teste 2

Foram executados mais testes para aferir o comportamento do mecanismo, desta vez não com ordenação de vetores, mas com contagem de tempo como condição de paragem.

Em testes anteriores verificou-se que os relógios e temporizadores não estavam a ser influenciados, mas após nova implementação do mecanismo de *throttle* e subsequente realização deste teste, verifica-se que já não é o caso. Este teste foi realizado com a conclusão de 1 segundo de execução como condição de paragem.

Tabela 11 - Resultados teste 4

% Throttle	Média(s)	Máximo(s)	Mínimo(s)	Desvio Padrão
0	1.01485	1.039	1.002	0.008627
10	1.1165	1.286	1.103	0.039081
20	1.23985	1.618	1.203	0.091662
30	1.3085	1.322	1.297	0.007011
40	1.4238	1.636	1.402	0.049883
50	1.515	1.6	1.496	0.022764
60	1.6232	2.04	1.588	0.095951
70	1.71395	1.89	1.611	0.049639
80	1.8053	1.836	1.778	0.013752
90	1.9271	2.055	1.815	0.0597

Após a obtenção de dados, procedeu-se ao estudo da curva, representado na figura seguinte:

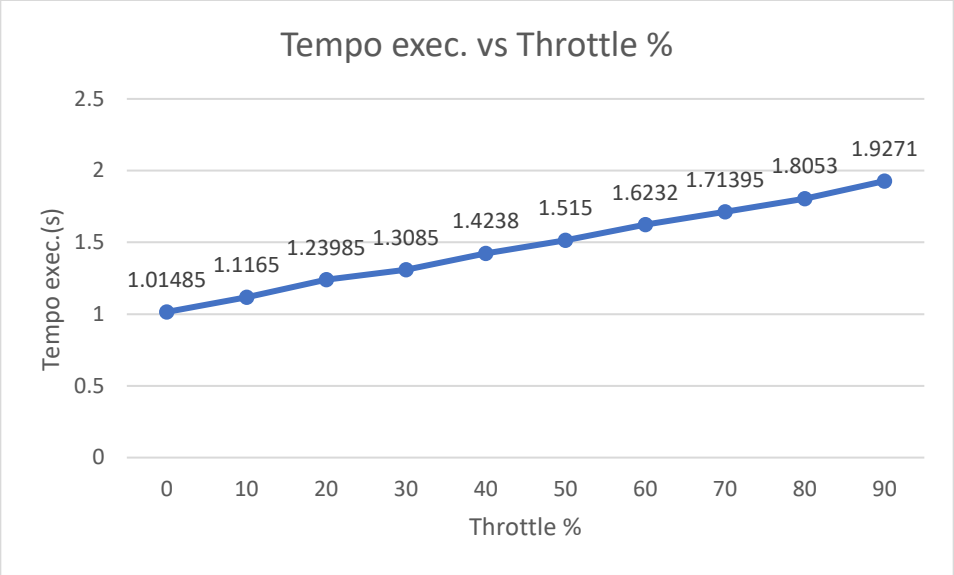


Figura 54 - Gráfico tempo exec. Vs % throttle (tempo exec. como condição paragem)

Curiosamente, neste teste, o incremento do tempo de execução é linear. Isto leva-nos a concluir que o mecanismo de *throttle* tem um efeito mais significativo quanto maior a carga computacional a que é submetido o processador, como se verificou através da figura 43.

Todavia, apesar de os valores do desvio padrão serem mais próximos, através de um gráfico de dispersão podemos aferir que as mesmas conclusões se mantêm: quanto maior o tempo de execução, mais dispersos serão os dados obtidos.

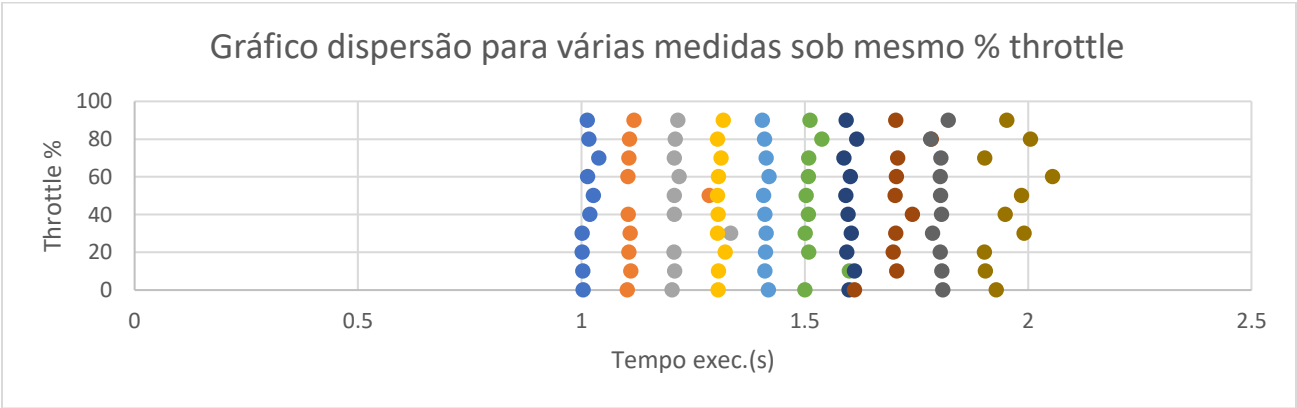


Figura 55 - Gráfico de dispersão (teste 4)

Findo este teste, e após verificar as diferenças no comportamento do mecanismo de *throttle* em relação ao contexto sobre o qual é submetido, foi proposto outro teste que engloba todos os outros: realizar o máximo possível de ordenações de vetores, usando um temporizador como condição de paragem. Desta forma, e analisando estes dados, seria possível obter uma conclusão mais concreta, abrangendo mais possibilidades de execução, uma vez que condições de paragem nunca serão só temporizadores ou ordenações de vetores.

Após implementação deste teste, verificou-se que independentemente da percentagem de *throttle*, o número de ordenações e os *timestamp* em que era efetuada uma ordenação eram sempre os mesmos.

Curiosamente, este é um resultado positivo porque indica que o sistema está a ser influenciado como um todo pelo mecanismo de *throttle*. Em suma, devido à influência do mecanismo de *throttle* nos temporizadores, se for realizado um pedido à máquina de testes com temporizadores como condição de paragem de por exemplo 10 segundos, no contexto da mesma irão passar 10 segundos ainda que sejam 20 segundos em tempo real.

Nesses 10 segundos o número de ordenações que serão realizadas será virtualmente igual, independentemente da percentagem de *throttle*. Apesar de com o mecanismo de *throttle* ativo, o tempo real no *guest* passar mais devagar, a capacidade computacional e velocidade de ordenação de vetores também diminui e mantém-se em função da percentagem de *throttle* porque o seu comportamento é determinístico.

A arquitetura da aplicação de testes, na sua forma atual, faz com que a máquina *host* conte o tempo desde o pedido até à receção da resposta por parte da máquina de testes. Tendo este facto em conta, asserta-se que não é possível prosseguir com esta abordagem para testar sobre estas condições. Seriam necessárias alterações estruturais significativas da aplicação de testes para suportar comunicação entre o *host* e o *guest,* entre iterações no decorrer da ordenação, para ser possível medir o tempo real entre estas iterações, o que irá ficar para trabalho futuro.

Capítulo 7. Conclusões

Neste capítulo são apresentadas as conclusões relativas ao trabalho efetuado. Em primeiro lugar é apresentado um resumo das soluções, seguido dos objetivos atingidos e finalmente, é apresentada uma descrição das limitações encontradas no decorrer do projeto. Adicionalmente, é feita uma apreciação pessoal tendo em atenção vertentes como o planeamento e as soluções propostas.

7.1.Objetivos atingidos

Em seguida, estão enumerados os objetivos iniciais do projeto:

- Criação de uma biblioteca que permite utilizar e integrar o QEMU noutros módulos da plataforma KhronoSim;
- Obter uma versão alterada do QEMU que permita a redução da velocidade de execução de sistemas com arquitetura ARM;
- Entender a influência que este mecanismo de redução de velocidade tem nos relógios do QEMU;
- Conseguir controlar *cores* de uma máquina virtual.

Posto isto, os seguintes objetivos foram atingidos:

- Foi implementada uma biblioteca que permite facilitar a utilização do QEMU, bem como, fornecer uma interface de manipulação de várias instâncias.
- Foi implementado um mecanismo que permite baixar a velocidade de processamento do QEMU, e ajustar a sua implementação para influenciar o sistema como um todo. Foi também extensivamente testada esta funcionalidade, permitindo obter resultados concretos acerca da mesma.
- Foi desenvolvido um método para controlo de *cores* de uma máquina virtual em execução que permite obter um maior controlo dos ciclos de teste como se pode verificar no Anexo 1.

Adicionalmente, foram escritos dois artigos científicos, que descrevem os contributos e especificidades do mecanismo de *throttle*, sendo que um destes artigos foi apresentado na conferência ICPS 2018 [10] e o outro na conferência INForum 2018 [11].

7.2.Resumo das soluções

No caso da biblioteca, esta foi implementada contendo todos os requisitos enumerados pelo cliente. Esta biblioteca é modular, extensível e concisa. Através dela, é possível manipular instâncias de QEMU individualmente, seja ligar/desligar máquinas ou o envio de comandos via *tcp* sockets. Adicionalmente é possível alterar algumas definições via ficheiro de configurações e os *logs* relativos a grande parte das operações possíveis são efetuados automaticamente. Foram implementados os devidos testes, nas funcionalidades pertinentes, tais como criação de instâncias, parsing de comandos, etc.

Em relação ao mecanismo de *throttle,* o código fonte do QEMU foi alterado para suportar esta funcionalidade de maneira o mais correta possível. De forma semelhante, foi implementada uma forma de invocar esta funcionalidade via comando monitor, e, por extensão, através da biblioteca. Adicionalmente foi criada uma aplicação de testes com arquitetura cliente-servidor que permitiu aferir os efeitos do mecanismo implementado via bateria de testes submetidos ao mesmo. Esta aplicação foi crucial, uma vez que permitiu atingir resultados numa aproximação prática, o que irá facilitar a integração deste mecanismo no *KhronoSim*.

A curva do tempo de execução do mecanismo de *throttle* é linear quando a condição de paragem é um temporizador e a carga computacional é virtualmente nula. Quando existe ordenação de vetores com um algoritmo de complexidade $n * \log n$, a curva obtida é quadrática.

O método de controlo de *cores* ficou concluído e verificou-se funcional na versão de *kernel 4.7.8* compilado para a plataforma *Sabrelite.*

Como esperado, tanto a biblioteca como o mecanismo de *throttle* e o método de controlo de *cores*, atingem o seu objetivo sem erros ou problemas a reportar.

7.3.Limitações e trabalho futuro

Em termos de limitações, existiu um extenso reportório. A limitação mais impactante foi a falta de documentação do QEMU, bem como a extensão do código.

Os tempos de compilação do QEMU e de diversos kernels testados (especialmente o *Sabrelite)* também foram uma limitação, uma vez que baixaram o nível de produtividade diária.

Para trabalho futuro, destaca-se a continuação dos testes do mecanismo de *throttle,* que ainda necessita de mais testes para se poder chegar a uma conclusão mais abrangente, bem como, o estudo da influência do mecanismo de *throttle* para sistemas *multi-core.*

Adicionalmente, terá lugar a implementação da versão alterada do QEMU e da biblioteca no projeto *KhronoSim*.

7.4.Apreciação final

7.4.1. Do estudante

Achei este projeto extremamente aliciante. Desde que comecei a ganhar alguma noção da área da Informática, tive imenso interesse em emuladores, mais especificamente aqueles dedicados a consolas.

Como tal, quando me foi proposto este projeto, de imediato fiquei motivado para participar no mesmo. Essencialmente, o projeto manteve-se interessante ao longo da sua duração permitindo-me melhorar os meus conhecimentos de programação de baixo nível e também de Engenharia de Software no design da biblioteca. Além disso, entendi a estrutura e funcionamento de um emulador, as diversas camadas que o compõe e a necessidade de criar uma boa documentação aquando o desenvolvimento de uma solução.

Em relação ao orientador e ao supervisor, estes mostraram-se sempre disponíveis para responder a eventuais questões, mantendo contacto constante ao longo do projeto e propondo sempre novos desafios, o que fez com que o projeto se mantivesse interessante.

7.4.2. Do planeamento

O planeamento, realizado com o apoio do orientador, está bem estruturado, conciso e com objetivos bem definidos. Os prazos e durações eram realistas e provaram ser precisos ao longo de todo projeto.

Como tal, conclui-se que o planeamento proposto foi crucial para o desenvolvimento saudável do projeto e foi respeitado dentro dos possíveis.

7.4.3. Da solução

Penso que no geral, a solução obtida foi satisfatória. Esta conclusão parte da minha opinião pessoal, bem como da opinião do orientador e do supervisor. Obviamente, trabalho futuro é imperativo para dar respostas a problemas que possam surgir, e como tal, dar continuidade ao sucesso do projeto.

Esta solução permite placar os problemas iniciais, bem como, abrir portas para soluções futuras para problemas semelhantes e implementação de novas funcionalidades no QEMU.

Conclui-se que, caso existisse uma solução deste tipo à *priori*, o desenvolvimento deste projeto iria decorrer mais suavemente.

Capítulo 8. Bibliografia

[1] Alexandrescu, A. (2001). Modern C++ design: generic programming and design patterns applied. *Addison-Wesley*.

[2] Balaji, S., & Murugaiyan, M. S. (2012). Waterfall vs. V-Model vs. Agile: A comparative study on SDLC. *International Journal of Information Technology and Business Management*, *2*(1), 26-30.

[3] Critical Software, Khronosim - System for Simulation and Test of Complex Cystems, Portuguese National Project POCI-01-0247-FEDER-017611

[4] Edelson, D., & Pohl, I. (1992). *Smart pointers: They're smart, but they're not pointers*. University of California, Santa Cruz, Computer Research Laboratory.

[5] Eeles, P. (2005). Capturing architectural requirements. *IBM Rational developer works*.

[6] Jones, M. T. (2011) Platform emulation with Bochs. http://www.ibm.com/developerworks/library/l-bochs/. Last accessed 04/2018.

[7] Lee, J., Bagheri, B., & Kao, H. A. (2015). A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, *3*, 18-23.

[8] Lira Nunez, David & Borsato, Milton. (2015). Panorama atual dos sistemas ciber-físicos no contexto da manufatura.

[9] McGregor, I. (2002, December). The relationship between simulation and emulation. In *Simulation Conference, 2002. Proceedings of the Winter* (Vol. 2, pp. 1683-1688). IEEE.

[10] Oliveira, P. R., Meireles, M., Maia, C., Pinho, L. M., Gouveia, G., & Esteves, J. (2018, May). Emulation-in-the-loop for simulation and testing of real-time critical CPS. In *2018 IEEE Industrial Cyber-Physical Systems (ICPS)* (pp. 258-263). IEEE.

[11] Oliveira, P. R., Maia, C., Pinho, L. M. (2018, September). Exploiting a Throttle Mechanism for QEMU. In *INForum 2018*.

[12] Pervez, S., & Abosaq, N. (2016, April). Gasim Alandjani, IOT Services Impact as a Driving Force on Future Technologies by Addressing Missing Dots. In *16th International Conference on Applied Computer Science (ACS'16), Istanbul, Turkey* (pp. 15-17).

[13] Phillips, G. (2010). Simplicity Betrayed. *Commun. ACM*, *53*(6), 52-58.

[14] QEMU. Qemu project documentation. https://qemu.weilnetz.de/doc/qemu-doc.html. Last accessed 07/2018.

[15] QEMU. Qemu features. https://wiki.qemu.org/Features. Last accessed 07/2018.

[16] QEMU. Qemu architecture. https://wiki.qemu.org/Documentation/Architecture. Last accessed 07/2018.

[17] Rosenthal, D. S. (2015). Emulation & virtualization as preservation strategies. *Andrew W. Mellon Foundation*.

[18] The Bochs Project. bochs: The open source ia-32 emulation project. http://bochs.sourceforge.net/. Last accessed 04/2018.

[19] Vicente, Paulo. (2005). O uso de simulação como metodologia de pesquisa em ciências sociais. *Cadernos EBAPE.BR*, *3*(1), 01-09. https://dx.doi.org/10.1590/S1679-39512005000100008

[20] Wolf, W. H. (2009). Cyber-physical systems. IEEE Computer, 42(3), 88-89.

[21] Rosenthal, D. S. (2015). Emulation & virtualization as preservation strategies. *Andrew W. Mellon Foundation*.

Anexos

Anexo 1. Controlo de *cores* virtuais

Neste anexo vai ser abordado o processo necessário para controlar *cores* virtuais numa *board Sabrelite*, emulada pelo QEMU, bem como os testes efetuados. Esta é uma plataforma de baixo custo, utilizada para desenvolvimento. É dotada do processador i.MX 6Quad, que como o nome indica, contém 4 *cores*.

O controlo dos *cores* deste processador, nesta plataforma em específico, é importante para o projeto na medida em que irá permitir um controlo mais acentuado do fluxo dos testes.

Compilação

A compilação da plataforma *Sabrelite* é um processo escassamente documentado. Para obter um sistema completo, é preciso compilar duas partes distintas, nomeadamente o *kernel* e a estrutura de pastas do sistema operativo. O *kernel* escolhido é o Linux 4.7.5, sendo que é uma das versões suportadas pela plataforma *Sabrelite.* A estrutura do sistema operativo é a outra parte do sistema que é crucial. Caso esta estrutura esteja em falta, o *kernel* não consegue montar as pastas necessárias ao funcionamento do sistema operativo. É necessário compilar esta estrutura numa imagem que é depois associada ao *kernel* para se obter um sistema compatível com a plataforma *Sabrelite.* Adicionalmente, a esta estrutura, é necessário escrever um pequeno *script* que corre o primeiro processo do sistema que, regra geral, é uma instância da *Shell.*

Para compilar esta imagem, recorreu-se ao *Busybox,* versão 1.28.4. Este utilitário serve para compilar outros utilitários pertencentes ao *UNIX* num executável de tamanho reduzido.

Ao compilar o *kernel,* foi necessário especificar que módulos cuja exportação era necessária, mais especificamente os ficheiros .img que contém o código que diz respeito ao sistema operativo e o *rootfs* que serve para manter algumas estruturas de dados que dizem respeito ao sistema operativo. Adicionalmente, foi necessário alterar alguns parâmetros acerca da compilação do *kernel,* mais especificamente a ativação do *Symmetrical Multi-Processing (*SMP), para ativar as capacidades *multi-core* do sistema operativo*,* uma vez que a ativação das mesmas é crucial para prosseguir o estudo.

O *kernel* foi compilado com recurso ao *Gnu Compiler Collection* (gcc) versão 8.2. Adicionalmente, como a plataforma alvo é dotada de uma arquitetura ARM foi necessária a instalação de uma *toolchain* extra, nomeadamente, a *GNU ARM Toolchain.*

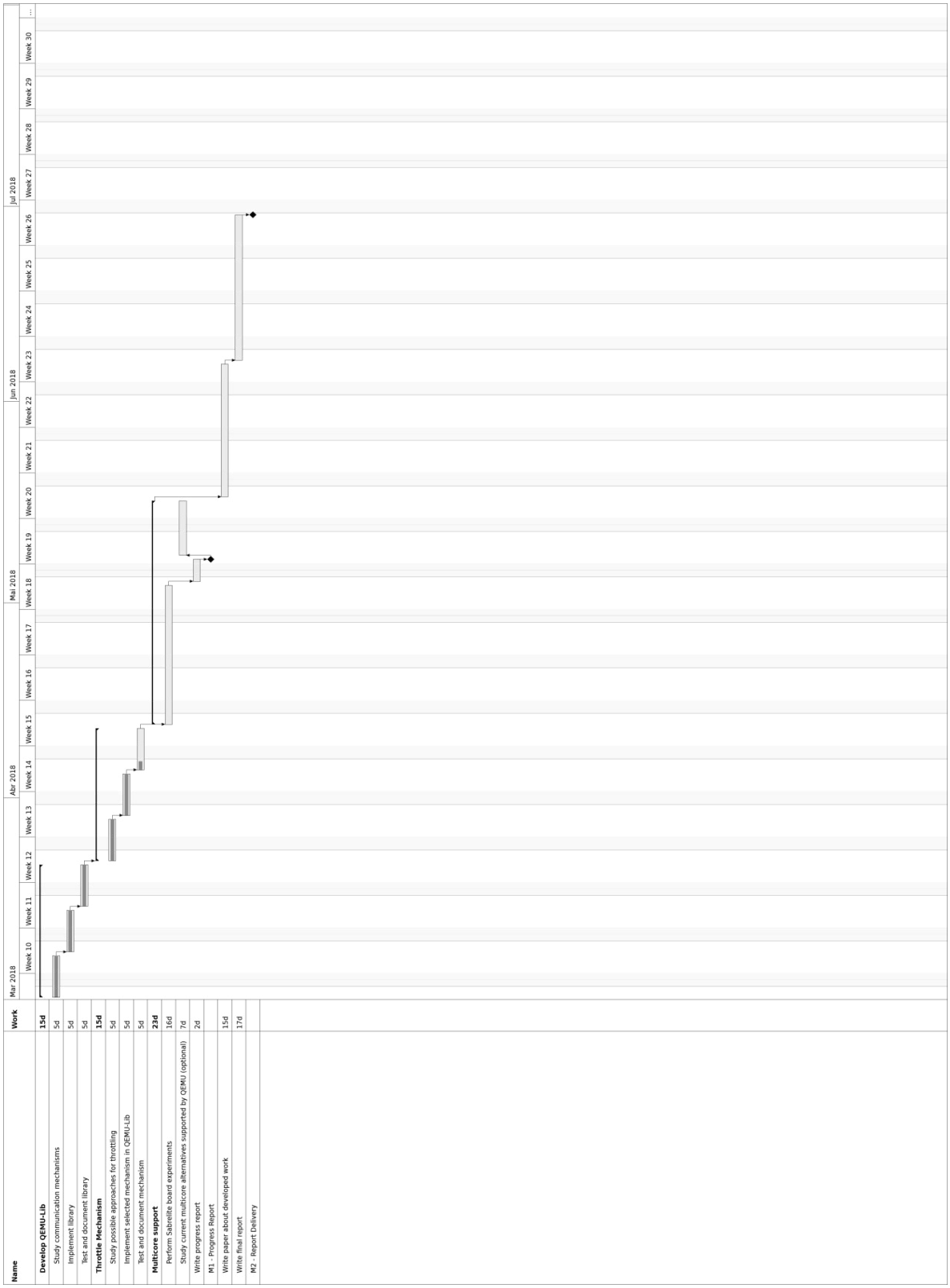
Testes e experiências

O objetivo desta parte do projeto é permitir o controlo de *cores* de uma máquina enquanto esta está em execução. Em sistemas *Unix* mais recentes (como é o caso da versão alvo), é possível ter acesso a esta funcionalidade. Cada *core* do processador tem um ficheiro a ele associado que indica se este está ativo ou não, refletindo o estado no *hardware* que é neste caso emulado. Estes ficheiros estão localizados no caminho $/sys/devices/system/cpu/$ sendo que neste nível encontra-se uma pasta para cada *core* existente. Dentro de cada uma dessas pastas existe um ficheiro denominado por $online$ que apenas tem o valor booleano relativo ao seu estado, sendo que basta alterar esse ficheiro para alterar o estado do *core.*

Posto isto, ao tentar tirar partido desta funcionalidade na plataforma *Sabrelite,* verificamos que apenas um dos *cores* estava disponível em detrimento de 4 suportados pela plataforma. Isto deve-se ao facto do QEMU, por defeito, inicializar a máquina com apenas um *core,* sendo que enviando um parâmetro extra (-smp maxcpus=4, cpus=4) é possível resolver o problema. O parâmetro diz ao QEMU para tirar partido do *Symmetrical Multi-Processing,* funcionalidade que ativamos de igual modo no *kernel,* caso contrário voltaríamos a ter resultados negativos.

Após se verificar que o QEMU inicializava a máquina virtual com os 4 *cores* virtuais, o controlo dos mesmos tornou-se possível, facilmente verificável com a desativação de um ou mais, e utilização do executável top que mostra quais os *cores* com estado ativo.

Anexo 2. Planeamento completo e gráfico de Gantt



WBS	Name	Start	Finish	Work	Duration	Slack	Cost	Assigned to	% Complete
1	**Develop QEMU-Lib**	**Mar 1**	**Mar 21**	**15d**	**15d**	**72d**	**0**		0
1.1	Study communication mechanisms	Mar 1	Mar 7	5d	5d		0		100
1.2	Implement library	Mar 8	Mar 14	5d	5d		0		100
1.3	Test and document library	Mar 15	Mar 21	5d	5d		0		100
2	**Throttle Mechanism**	**Mar 22**	**Abr 11**	**15d**	**15d**	**57d**	**0**		0
2.1	Study possible approaches for throttling	Mar 22	Mar 28	5d	5d		0		100
2.2	Implement selected mechanism in QEMU-Lib	Mar 29	Abr 4	5d	5d		0		100
2.3	Test and document mechanism	Abr 5	Abr 11	5d	5d		0		20
3	**Multicore support**	**Abr 12**	**Mai 16**	**23d**	**25d**		**0**		0
3.1	Perform Sabrelite board experiments	Abr 12	Mai 3	16d	16d		0		0
3.2	Study current multicore alternatives supported by QEMU (optional)	Mai 8	Mai 16	7d	7d		0		0
4	Write progress report	Mai 4	Mai 7	2d	2d		0		0
5	M1 - Progress Report	Mai 7	Mai 7	N/A	N/A		0		0
6	Write paper about developed work	Mai 17	Jun 6	15d	15d		0		0
7	Write final report	Jun 7	Jun 29	17d	17d		0		0
8	M2 - Report Delivery	Jun 29	Jun 29	N/A	N/A		0		0

Anexo 3. Perguntas do livro Engineering Reasoning

Q: What is the purpose of this design?

R: Providenciar respostas ao problema.

Q: What are the market opportunities or mission requirements?

R: Diminuição de custos de testes destrutivos ou que necessitam de um tempo significativo para ser preparados.

Q: Who defines the market opportunities or mission requirements?

R: A própria natureza dos sistemas ciber-físicos.

Q: How does the costumer define value?

R: Valor será uma solução que permita atingir os objetivos do projeto, ou clarificar o caminho para atingir os objetivos do projeto.

Q: Is a new design or technology required?

R: Sim, é necessária uma nova abordagem ao problema.

Q: Can an existing design be adapted?

R: Não, até porque não existe.

Q: What environmental or operating conditions are assumed?

R: Testes em malha fechada.

Q: What happens if we change or discard an assumption?

R: Poderá ser necessário descartar implementações e refaze-las.

Q: What information do we lack? How can we get it?

R: Documentação relativa ao QEMU. Podemos estudar o código-fonte ou navegar nas *mailing lists.*

Q: What experiments should be conducted?

R: Experiências de influência dos mecanismos implementados numa perspetiva de simulação e posterior documentação de resultados.

Q: Have we considered all relevant sources?

R: Sim, as fontes relevantes foram consideradas.

Q: What legacy solutions, shortcomings or problems should be studied and evaluated?

R: Estudos do paradigma multi-core, e acompanhamento do desenvolvimento do QEMU.

Q: Is the available information sufficient? Do we need more data? What is the best way to collect it?

R: Não é suficiente, a documentação é escassa e a que existe é confusa. Uma boa maneira de obter informação é via *trial and error.*

Q: What concepts or theories are applicable to this problem?

R: Teorias sobre emulação, testes em malha fechada etc.

Q: Are there competing models?

R: Não, não existe competição pelo menos a título público e conhecido.

Q: What available technologies or theories are appropriate?

R: O QEMU pela sua versatilidade.

Q: What is the set of viable solutions?

R: Nenhuma.

Q: What are some important implications of the data we've gathered?

R: A dependência na tecnologia e a falta de documentação na mesma pode implicar um trabalho demorado e com uma conclusão complicada.

Q: What are the most important market implications of the technology?

R: Pode baixar muito os custos de testes e permitir acompanhar os ciclos de desenvolvimento cada vez mais curtos.

Q: Is there a path for future design evolution and upgrade?

R: Sem dúvida, pode ser desenvolvida uma feature que permita evolução da mesma.

Q: What are the most important implications of product failure?

R: Pode ser impossível ou não viável efetuar testes emulados de um sistema ciber-físico.

Q: What design features, if changed, profoundly affect other design features?

R: Código fonte interno do QEMU, *patches*, etc.