Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Protótipo de *Cloud Gaming* utilizando tecnologias de código aberto

Nathan de Oliveira Nunes

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Alfredo Goldman Vel Lejbman
Cossupervisor: Bruno Lobo Motta

São Paulo
2023

Aos meus queridos familiares e amigos, cujo apoio e confiança foram as engrenagens que impulsionaram meu progresso acadêmico.

Agradecimentos

"A future is not given to you. It is something you must take for yourself."
— Pod 042

Gostaria de expressar minha gratidão ao meu orientador, Alfredo Goldman, pela orientação e apoio contínuo ao longo deste trabalho. Além disso, sou grato por me fornecer acesso à conta da AWS, que possibilitou a realização dos experimentos na nuvem.

Também desejo agradecer ao meu coorientador, Bruno Motta, pelos insights e orientações na área de *cloud gaming*. Suas contribuições foram importantes para o desenvolvimento deste trabalho.

Agradeço a Deus, aos meus amigos e familiares pelo apoio ao longo de toda a minha jornada acadêmica. Suas palavras de incentivo, encorajamento e compreensão foram essenciais nos momentos de desafio.

Por fim, quero expressar minha gratidão a todos os professores, colegas e demais pessoas que contribuíram, direta ou indiretamente, para o meu crescimento pessoal e acadêmico. Cada interação, cada conversa e cada colaboração deixaram uma marca profunda em minha jornada, e sou grato por todas as oportunidades de aprendizado e crescimento que surgiram ao longo do caminho.

Resumo

Nathan de Oliveira Nunes. **Protótipo de *Cloud Gaming* utilizando tecnologias de código aberto**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

Cloud gaming é um modelo de serviço de jogos que possibilita que os usuários possam jogar jogos sem a necessidade de um *hardware* especializado ou de armazená-los em suas máquinas. Devido aos altos custos de se manter um serviço de *cloud gaming*, normalmente apenas grandes empresas conseguem obter sucesso. Apesar do sucesso de alguns serviços, ainda existem diversas lacunas nos serviços atualmente disponíveis como resoluções *ultrawide*, problemas de escalabilidade e disponibilidade em diversos países. Com o objetivo de apontar possíveis soluções para problemas comuns em *cloud gaming*, construímos um protótipo de um serviço de *cloud gaming* em Kubernetes utilizando a nuvem pública da Amazon. Foram utilizadas diversas tecnologias de código aberto para construí-lo, como containerd; Wayland; Pipewire; X11; Sunshine; Moonlight e Kubernetes. Conseguimos uma boa performance no protótipo que conseguiu executar um jogo chamado AstroMenace. Além disso, com base na arquitetura proposta, conseguimos apontar possíveis soluções para problemas que fugiam do escopo do protótipo como salvamento de progresso e escalabilidade.

Palavras-chave: Jogando na nuvem. Kubernetes. Sunshine.

Abstract

Nathan de Oliveira Nunes. **Cloud Gaming prototype using open source technologies**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

Cloud Gaming is gaming service model that allows users to play games without owning dedicated hardware or storing games in their machines. Due to the high costs of maintaining this kind of service, usually only big enterprises can suceed in this market. Although some services managed to achieve success, there are still a lot of gaps in them such as ultrawide resolutions, scalability issues and availability in a large number of countries. To address these issues and search for possible solutions to common problems in cloud gaming, we built a prototype of a cloud gaming service in Kubernetes using AWS public cloud. Many open source technologies were used build this prototype including containerd; Wayland; Pipewire; X11; Sunshine; Moonlight e Kubernetes. We achieved a good performance in the prototype, that managed to execute a game called AstroMenace. Furthermore, based on the proposed architecture, it was possible to point some solutions to problems that are outside of the scope of the prototype such as progress saving and scalability.

Keywords: Cloud Gaming. Kubernetes. Sunshine.

Lista de abreviaturas

URL	Localizador Uniforme de Recursos (*Uniform Resource Locator*)
MIG	(*Multi-Instance-GPU*)
MPS	(*Multi-Process Service*)
TCP	Protocolo de Controle de Transmissão (*Transmission Control Protocol*)
UDP	Protocolo de Datagrama do Usuário (*User Datagram Protocol*)
IP	Protocolo de Internet (*Internet Protocol*)
MIT	Instituto de Tecnologia de Massachusetts (*Massachusetts Institute of Technology*)
NES	(*Nintendo Entertainment System*)
SNES	(*Super Nintendo Entertainment System*)
CPU	Unidade de Processamento Central (*Central Processing Unit*)
GPU	Unidade de Processamento Gráfico (*Graphics Processing Unit*)
HDD	Unidade de Disco Rígido (*Hard Drive Disk*)
SSD	Disco de Estado Sólido (*Solid State Drive*)
TPU	Unidade de Processamento de Tensor (*Tensor Processing Unit*)
FPS	Quadros por Segundo (*Frames per Second*)
PSN	Playstation Network
AWS	*Amazon Web Services*
LTS	*Long Term Support*
EKS	*Elastic Kubernetes Service*
API	*Application Programming Interface*
RAM	Memória de Acesso Aleatório (*Random Access Memory*)
RFB	*Remote Frame Buffer*
VNC	*Virtual Network Computer*
HTTP	*Hypertext Transfer Protocol*
gRPC	*Google Remote Procedure Call*
webRTC	*Web Real Time Communication*
KVM	*Kernel Virtual Machine*
CNCF	*Cloud Native Computing Foundation*
IME	Instituto de Matemática e Estatística
USP	Universidade de São Paulo

Lista de figuras

Lista de tabelas

Lista de programas

Sumário

Capítulo 1

Introdução

1.1 Jogos e tecnologia

Jogar vem se tornando uma atividade cada vez mais comum entre as pessoas. Dados indicam que na última década mais de um bilhão de pessoas passaram a jogar com alguma frequência, como mostra a figura 1.1, e que a tendência no consumo desse tipo de mídia é crescente. Não apenas o número de pessoas que jogam está aumentando, como também está aos poucos se tornando uma atividade mais uniforme em questão de gênero. A figura 1.2 mostra que jogar está se tornando cada vez mais igualitária em termos de gênero, diminuindo consideravelmente a diferença de gosto pela mídia. Outro aspecto importante é a idade média dos jogadores, muitas vezes associada ao público mais jovem, vem crescendo sua adesão em outras faixas etárias conforme a familiaridade com a mídia cresce 1.3.

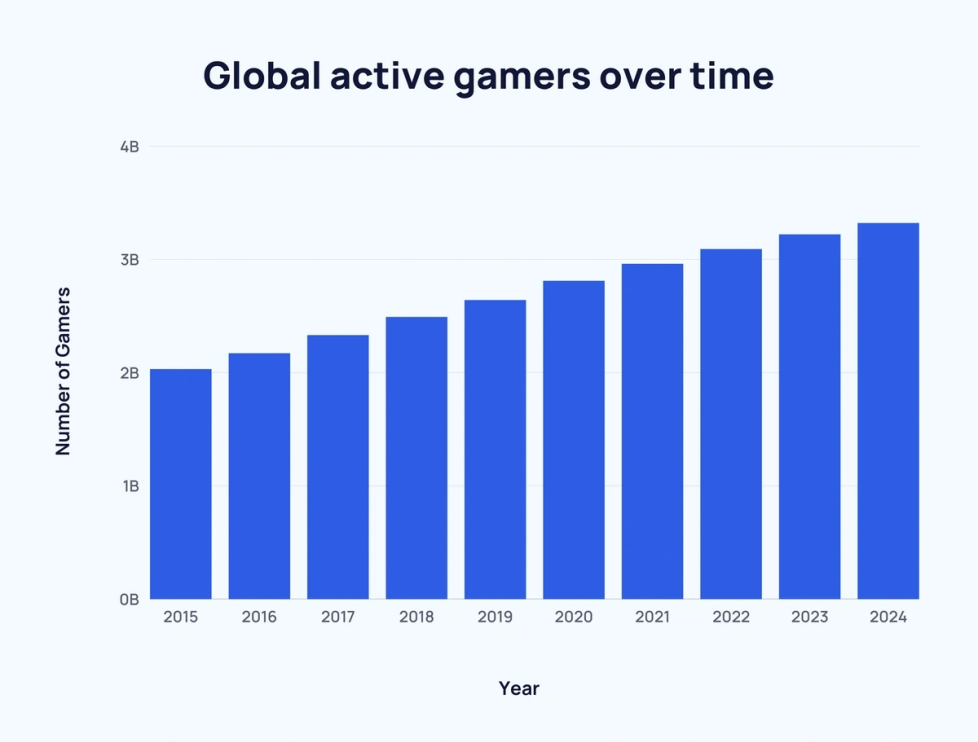

Figura 1.1: *Crescimento do número total de jogadores ao longo dos anos. Fonte: Exploding topics*

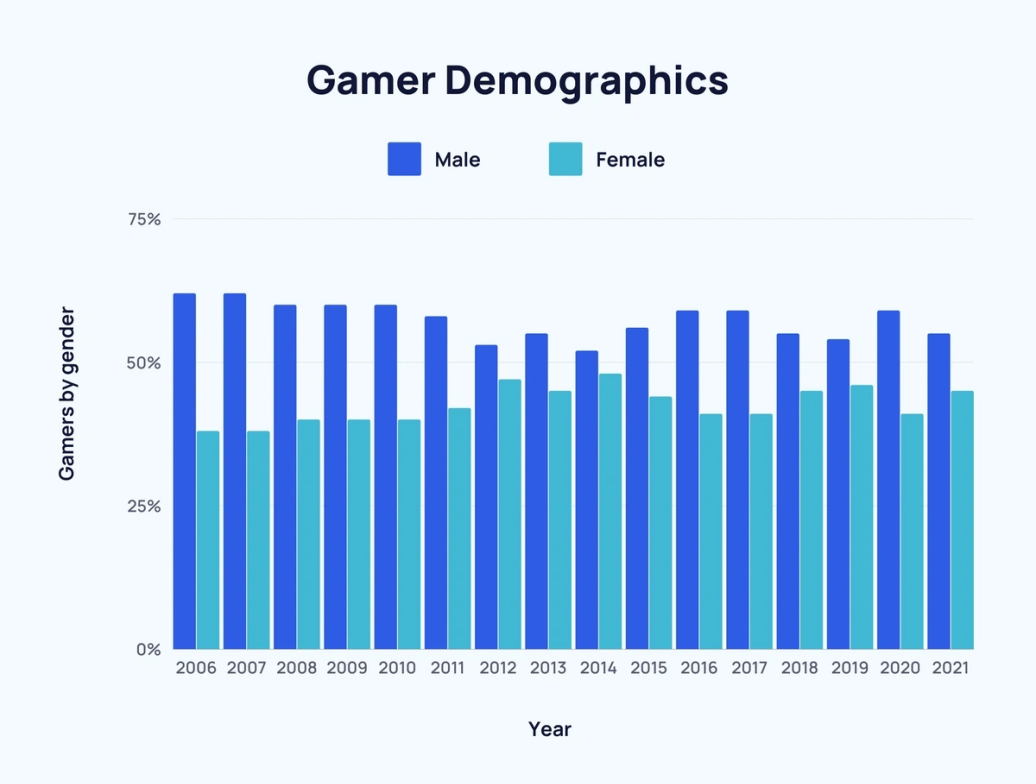

Figura 1.2: *Proporção de gênero ao longo dos anos. Fonte: Exploding topics*

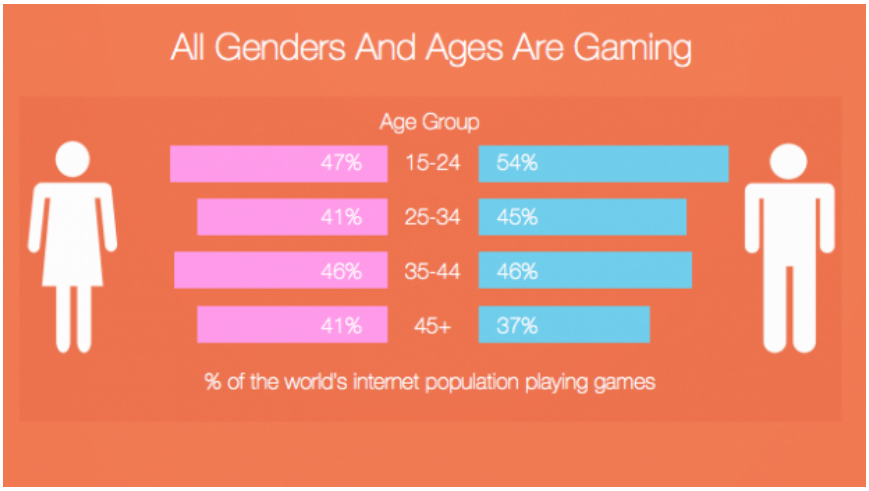

Figura 1.3: *Comparativo da idade dos jogadores. Fonte: Geekwire*

1.1.1 História dos jogos

Ao passo que a tecnologia evoluiu, os jogos também evoluíram. Os primeiros jogos surgiram na década de 1950 e 1960, com jogos como *Bertie the Brain* e *Spacewar!*, e contavam com capacidades de *hardware* bem reduzidas e possuíam poucos recursos. Spacewar! foi desenvolvido no MIT em 1961 para o computador *mainframe* PDP-1. Spacewar! é considerado o primeiro amplamente disponível e influente jogo eletrônico Russell *et al.*, 1962. O jogo permitia que dois jogadores controlassem uma nave e simulassem um combate espacial em um monitor.

O primeiro vídeo game console foi o Magnavox Odyssey lançado em 1972 pela Sanders Associates e a Magnavox. No total foram desenvolvidos 28 jogos para o console, sendo um deles, um jogo de ping pong que se tornou uma forte influência para o famoso Pong de Atari. O console foi descontinuado em 1975, mas rendeu muitos dividendos para os criadores em forma de patentes pelo pioneirismo.

Figura 1.4: *O primeiro videogame console, Magnavox Odyssey. Fonte: Wikipedia*

Space invaders lançou em 1978 e marcou o início da era de ouro dos jogos de arcade que durou cerca de 4 anos. Essa época trouxe muitas novidades e conceitos para os jogos como

narrativas e caracterização de personagens. O avanço da indústria de jogos impressionou, batendo todas a outras mídias de entretenimento em receita Rogers e Larsen, 1984, sendo que considerando apenas os operadores de moedas de arcades, a receita foi de 7,7 bilhões de dólares só nos Estados Unidos, superando a música pop (4 bilhões) e os filmes de Hollywood (3 bilhões).

A segunda geração foi marcada pelo uso de cartuchos e os primeiros vídeo games portáteis. O uso de cartuchos permitiu que um mesmo dispositivo tivesse diversos jogos criados separadamente, diferente dos primeiros arcades e primeiros consoles de consumo. Com a possibilidade de criar jogos separadamente do processo de criação de um console, surgiu a oportunidade de se criar jogos independentes. A Activision foi fundada em 1979 por ex-desenvolvedores da Atari descontentes com as políticas da empresa, e começaram a desenvolver jogos para o Atari 2500 de forma independente, o que marcou o início dos jogos conhecidos como *third party*.

Devido à facilidade de se desenvolver novos jogos utilizando cartuchos, o volume de jogos desenvolvidos começou a aumentar. No entanto, a indústria se sentiu motivada a lançar jogos mais rapidamente e com menor qualidade, produzindo muitos jogos genéricos e sem identidade. Isso fez com que os jogadores perdessem um pouco do interesse, pois os jogos que eram melhor desenvolvidos, custavam mais caro que os genéricos Fleming, 2007. Essa recessão durou de 1983 até 1985, quando as publicadoras japonesas que não foram muito afetadas no japão pelo controle de qualidade dos jogos mais rígido entrou no mercado ocidental com mais força, principalmente a Nintendo com o *Nintendo Enterteiment System* (NES).

Ao mesmo tempo, os computadores pessoais foram se tornando mais acessíveis e poderosos, se tornando uma plataforma viável para os jogos. Séries famosas foram lançadas nessa plataforma durante a década de 1980, incluindo Metal Gear (1987) e Castlevania (1986). Um dos computadores mais importantes para a história dos jogos foi o IBM PC, que trouxe as primeiras placas dedicadas de vídeo e de áudio, que possibilitou que jogos se tornassem mais imersivos e com qualidade técnica superior.

Por volta de 1990, a internet começou a surgir, e com ela vieram os primeiros jogos online e possibilitou que pessoas jogassem juntas mesmo sem estarem fisicamente juntas. MUD, Doom e MIDI Maze foram uns dos primeiros jogos online.

Depois da recessão de 1983, o mercado de video games fora do Japão voltou a crescer. A terceira e quarta gerações de video games foi marcada pelos consoles de 8 bits (NES, SG-1000) e 16 bits (SNES, Mega Drive). Essas gerações trouxeram novidades como adaptadores gráficos dedicados para os consoles, algo até então presente apenas nos computadores, cartuchos de memória (mais conhecidos pelos jogadores pelo nome em inglês, *memory card*) que possibilitaram que os jogadores salvassem seu progresso em um jogo e movessem isso para outro dispositivo compatível ou para backup. Franquias lendárias também surgiram nessa época, como Sonic, Mario, Final Fantasy e Dragon Quest.

O uso de mídia óptica como CDs começou a ser popularizada na indústria musical por volta de meados de 1980. Conforme a mídia óptica ia se tornando mais barata e com maior capacidade de armazenamento, ela acabou se tornando viável como um meio armazenar jogos ao invés dos cartuchos Alpert, 1992. Em 1994 a Sony lançou seu primeiro console,

usando exclusivamente a mídia óptica, o PlayStation. Também nessa época, impulsionado pelo avanço no hardware, começaram a surgir os primeiros jogos em 3D, assim como os primeiros motores gráficos para jogos 3D, como Unreal Engine e CryEngine, que tornaram o desenvolvimento de jogos 3D mais simples. A Sega e a Nintendo também lançaram seus consoles com suporte a jogos 3D, o Sega Saturn e o Nintendo 64, respectivamente. O Sega Saturn é um console com uma CPU de 32 bits que começou com um bom volume de vendas, mas acabou ficando na sombra do PlayStation. O Nintendo 64 foi um dos primeiros consoles com uma CPU de 64 bits e trouxe muitos títulos de peso.

No começo do milênio, a Microsoft também entrou no mercado de consoles, lançando o Xbox. Na mesma época, a Sony lançou o PlayStation 2 que tornou o console mais vendido de todos os tempos. A Sega lançou seu último console nessa época, o Dreamcast, não conseguindo mais competir com os 3 grandes (Nintendo, Microsoft e Sony) competidores no mercado de consoles. Na segunda metade dessa década, foram lançados novos consoles (Nintendo Wii, PS3 e Xbox 360) que trouxeram um avanço em capacidades visuais em relação à geração passada. Espaço de armazenamento se tornou mais barato, podendo armazenar internamente uma maior quantidade de jogos. A internet se tornou pupular, sendo acessível à grande parte da população mundial, permitindo que jogos online fossem mais audaciosos e conseguissem colocar muitas pessoas jogando juntas ao mesmo tempo.

Muitas tecnologias têm ganhado destaque nos últimos anos. Tracejamento de raios, mais conhecido pelo nome em inglês *Ray Tracing*, é uma técnica para modelagem da luz que já vinha sendo utilizada na iluminação de algumas animações como A Casa Monstro (2006) e Está Chovendo Hamburguer (2009). Devido ao seu alto custo computacional, a adoção de *Ray Tracing* em aplicações de tempo real, como jogos, pensava ser impossível. No entanto, com o lançamento de placas gráficas de consumo, com *Ray Tracing* acelerado por hardware, tornou-se possível o uso dessa técnica de iluminação em jogos. Battlefield V foi o primeiro jogo a utilizar *Ray tracing* em tempo real.

1.1.2 Evolução do Hardware

A medida que a forma da evolução do *hardware*, movida principalmente pela miniaturização do transistor, começa a desacelerar, novas formas de desenvolvimento da tecnologia vem sendo propostas. Melhorias em software e em arquitetura de hardware vem sendo desenvolvidas para continuar com o avanço. A especialização do hardware para determinados tipos de tarefas Leiserson *et al.*, 2020 é um exemplo desses esforços, sendo evidenciado pela GPU (*graphics processing unit*) que é responsável por processar o pipeline gráfico (projeção, renderização, etc.) e é o componente mais fundamental para a execução de jogos modernos. Outro exemplo são as TPUs (*Tensor Processing Unit*) muito usadas para aplicações de inteligência artificial.

A tecnologia vem evoluindo nas últimas décadas, tornando computadores, consoles, celulares e outros dispositivos mais poderosos e acessíveis. A miniaturização do hardware possibilitou inovações como os consoles portáteis e jogar em *smartphones*, no entanto, a tendência de miniaturização do hardware vem decrescendo cada vez mais Shalf, 2020. O fenômeno da tendência de miniaturização do *hardware* é conhecido como *Moore's Law* ou lei de Moore. A lei é, na verdade, uma observação feita pelo cofundador da Intel,

Gordon Moore, que dizia que o número de transistores em um chip iria dobrar a cada dois anos.

1.2 Cloud gaming

1.2.1 Introdução

Cloud gaming ou *Gaming on demand* é um modo de jogar onde o jogo é executado em um serviço enquanto o vídeo do jogo é transmitido pela internet a um cliente que o usuário está usando, sendo este cliente também responsável por enviar as ações do usuário ao serviço. Este modo de jogar estabelece um contraste com o modelo tradicional onde o jogo é executado no computador pessoal do usuário ou console.

1.2.2 História

A primeira tentativa de se implementar um serviço de *cloud gaming* surgiu no começo dos anos 2000, com uma *startup* chamada G-cluster (Game Cluster). O G-cluster foi anunciado na E3 de 2000 e lançado em 2003 e inovou no modelo de negócio. A solução do serviço envolvia rodar os jogos em seus servidores enquanto um pequeno dispositivo atuava como clientedo lado do usuário para enviar os comandos e receber a transmissão de vídeo Ojala e Tyrvainen, 2011. Devido à popularidade dos jogos gratuitos em computadores pessoais e no custo baixo dos computadores para rodar esses jogos, por volta de 2010, a empresa resolveu focar em IPTV.

Outra tentativa foi de uma empresa chamada Infinium Labs. A tentativa veio na forma de um console chamado Phantom, que seria capaz de transmitir jogos por demanda ao custo de uma assinatura mensal. A Infinium Labs chegou a apresentar um protótipo em 2004 na E3, mas a empresa faliu em 2008. A Crytek também chegou a iniciar uma pesquisa na viabilidade de uma plataforma para Crysis, mas resolveu esperar até que a infraestrutura global fosse mais madura gamesindustry.biz, 2009.

OnLive e Gaikai foram duas plataformas de *cloud gaming* que surgiram no começo da década de 2010. OnLive foi um dos primeiros serviços de *cloud gaming* a realmente ter algum impacto Ricker, 2009. Apesar de algumas críticas a respeito da qualidade do vídeo e do atraso entre a entrada do usuário e a ação ser refletida na tela (lag), o serviço ganhou destaque por algumas características como a habilidade de gravar sessões e ter espectadores nelas. Outro ponto destacado na época, foi o fato do serviço permitir que as pessoas testassem os jogos antes de adquiri-los. Sua arquitetura era composta de uma coleção de *data centers*, onde os servidores usavam *chips* proprietários de compressão de vídeo, além de CPUs e GPUs comuns. Jogos mais antigos, que necessitavam de *hardware* menos potente, podiam ser executados concorrentemente no mesmo servidor usando tecnologias de virtualização, enquanto jogos mais pesados precisavam de uma GPU por instância do jogo sendo executado. Problemas administrativos e a falta de ajuste no modelo de negócio, que cobrava por jogo e pelo *streaming*, acabou levando a venda da tecnologia e patentes para a Sony ORLAND, 2015.

O serviço Gaikai também teve um destino similar. O serviço tinha dois modelos de

negócio em paralelo. O primeiro, chamado de *Ad Network*, permitia a transmissão de jogos para o navegador web dos usuários como se fosse um anúncio. O administrador do site era pago quando as pessoas jogassem, a ideia era que as publicadoras de jogos fornecessem uma demo de seus jogos em forma de anúncio e que os usuários comprassem o jogo se tivessem gostado, para continuar jogando. O outro modelo de negócio, chamado de *Open Platform* funcionava com base em parcerias com outras empresas, de forma que os jogos disponíveis, as plataformas onde poderia ser jogado, e o modelo de negócio dos parceiros poderiam ser customizados. A Samsumg, por exemplo, tinha anunciado que teria um serviço de *cloud gaming* fornecido pela Gaikai para suas TVs inteligentes mais poderosas. Em julho de 2012, a Sony comprou a Gaikai por 380 milhões dólares. A compra da Gaikai pela Sony, com a compra da OnLive, levou a Sony a adquirir várias patentes e tecnologias relacionadas a *cloud gaming* Hollister, 2019.

1.2.3 Principais plataformas

As principais plataformas de *cloud gaming* hoje são das grandes empresas de tecnologia. Em maio de 2012, a Nvidia anunciou o seu serviço de *cloud gaming*, Nvidia Grid, que mais tarde foi renomeado como Geforce Now. Ele foi lançado em novembro de 2014 com uma coleção de jogos bastante limitada. Mais tarde, em 2017, a Nvidia permitiu que os jogadores usassem a sua própria biblioteca de jogos adquirida em lojas digitais como a Steam e a Epic Games. No começo de 2020, a plataforma ainda estava em beta com acesso limitado, mas em fevereiro daquele ano foi disponibilizado ao público por meio de uma assinatura mensal. Algumas publicadoras não gostaram desse modelo de negócio, como a Bethesda e a Activision, afirmando que as compras em lojas digitais eram projetadas apenas para computadores pessoais e forçaram a Nvidia a retirar seus jogos da plataforma Mangalindan, 2020. A Nvidia possui uma vantagem em projetar soluções de *cloud gaming* que é a fabricação própria de GPUs, podendo fazer customizações para especializar o uso das GPUs para o serviço.

A Google anunciou o Project Stream em 2018, mas já estava trabalhando nele desde 2016 com o nome de Project Yeti. Em 2019, o Project Stream foi renomeado Stadia, e com isso, a Google criou uma divisão dedicada para criar jogos exclusivos para a plataforma. A arquitetura da plataforma foi feita em cima de Hardware especializado. Um processador Intel com *clock* de 2,7 Ghz e uma GPU AMD baseada na arquitetura VEGA foram utilizados para o hardware da plataforma Nelius, 2019. O serviço rodava em servidores Debian, utilizando Vulkan como API gráfica. A Google fez um controle customizado para a plataforma. A estratégia no uso de um controle próprio invés de terceiros se justifica, pois a Google introduziu um mecanismo que permite conectar o controle direto no roteador para evitar que o tráfego tivesse que sair do controle para o dispositivo cliente do usuário e só depois para os servidores da plataforma, reduzindo então, o atraso da entrada dos comandos (*input lag*). A figura 1.5 mostra uma foto do controle do Stadia.

Apesar do esforço do Google em consolidar a plataforma com várias parcerias e investimentos, e da qualidade técnica bastante impressionante, possibilitando rodar jogos com resolução 4K e 60 FPS, a plataforma não fez o sucesso esperado pela Google. Em fevereiro de 2021, a Google fechou a divisão dedicada para criação de jogos para o Stadia, anunciando que o Stadia iria focar em ser uma plataforma para que empresas terceiras

Figura 1.5: *O controle do Stadia. Fonte: Wikipedia*

publicassem seus jogos na plataforma. Já em setembro de 2022, a Google anunciou que descontinuaria o Stadia, citando a falta de adoção pelos usuários. Muito se discute dos motivos do Stadia não ter obtido sucesso, muitos citando o modelo de negócio adotado pela plataforma. A plataforma cobrava uma assinatura mensal e os usuários precisavam comprar os jogos para a plataforma. Outro ponto bastante citado era um catálogo de jogos pequenos, devido a dois principais fatores. O baixo volume e impacto dos jogos exclusivos produzidos para o Stadia não conseguiram atrair muitos jogadores. A plataforma também exigia que os jogos fossem desenvolvidos para ela, acarretando custos ao se portar um jogo para uma plataforma nova, que não tinha tantos usuários.

Xbox Cloud Gaming é o serviço de *Cloud Gaming* da Microsoft. Seu beta foi lançado em novembro de 2019 e lançado em 15 setembro de 2020. A plataforma tem uma vantagem competitiva em cima de plataformas como o Stadia devida a ampla gama de jogos disponíveis no serviço devido à Microsoft estar a muito tempo no mercado de jogos. Além da vantagem do catálogo de jogos, o Xbox Game Pass Ultimate, serviço por assinatura de jogos, fornece uma ampla gama de jogos que podem ser jogados sem custo adicional, e possui acesso ao Xbox Cloud Gaming. A arquitetura do Xbox Cloud Gaming é fortemente baseada nos consoles da Microsoft, tendo *datacenters* espalhados ao redor do mundo. Inicialmente cada servidor era composto de 4 unidades customizadas do Xbox Series S. Atualmente, cada servidor é composto por 8 Xbox Series X Wild, 2020. O serviço possibilita a transmissão em Full HD até 60 FPS, especialmente pelo fato do serviço ser limitado pelo *hardware* do Xbox.

O Amazon Luna, plataforma de cloud gaming feita pela Amazon, foi lançado como acesso antecipado em outubro de 2020, com um catálogo de apenas 100 jogos. Foi oficialmente lançado em março de 2022 nos Estados Unidos e em março de 2023 no Canada, Reino Unido e Alemanha. Sendo uma plataforma nova, de uma empresa com pouca presença

no mercado de jogos, a plataforma possui poucos jogos disponíveis. A plataforma está disponível em poucos países, apesar do alcance de seus *datacenters* devido ao popular *Amazon Web Services* (AWS).

As várias patentes e tecnologias dos primeiros serviços de cloud gaming, como Gaikai e OnLive, que foram adquiridos pela Sony foram incorporados ao Playstation Plus. A Playstation Plus possui milhões de assinantes, mas a plataforma não é dedicada a *cloud gaming*, ela é, na verdade, um nível premium da Playstation Network (PSN) que possui outras funcionalidades e vantagens além de *cloud gaming*. A plataforma não funciona como um serviço de *cloud gaming* de propósito geral, ela permite que se jogue jogos dos consoles de gerações passadas da Sony por *streaming*, não tendo suporte para os jogos da geração do PS5.

1.3 Principais tecnologias relacionadas

1.3.1 Hardware

Soluções de *cloud gaming* requerem o uso de *hardware* especializado e com muito desempenho, pois rodar jogos em escala para milhões de pessoas requer poder computacional que poucas empresas conseguem oferecer. Os principais componentes de hardware em uma solução de *cloud gaming* são a CPU, GPU e memória RAM. A depender da arquitetura do sistema, espaço de armazenamento pode não ser crucial, considerando que não há necessidade de armazenar o jogo mais do que a quantidade de instâncias em execução e que com o passar dos anos, apesar dos jogos estarem ocupando mais espaço, espaço de armazenamento vem se tornando mais barato e rápido. A GPU em particular tende a ser um dos hardwares mais caros e especializados em uma solução de *cloud gaming* pois normalmente requerem o uso dedicado para o serviço, não podendo usualmente serem reaproveitadas ou terem seu poder computacional particionado.

1.3.2 Protocolos

É necessária uma estratégia eficiente para a comunicação entre usuário e provedor, uma vez que essa comunicação envolve dados complexos de diferentes tipo e que precisam ter um certo nível de sincronia. Os principais dados envolvidos nessa comunicação é o video, áudio e os comandos do jogador. Existem muitos protocolos que transmitem dados dos mais diversos tipos pela internet. Dentre os mais promissores ou já utilizados para fins similares podemos destacar alguns. Um dos mais antigos protocolos para uso de um computador remoto surgiu no começo de 1998 com o protocolo aberto RFB (*remote framebuffer*). RFB suporta diferentes tipos de *encodings* o que permite certa flexibilidade. O protocolo foi utilizado pelos sistemas de VNC (*virtual networking computing*). Os sistemas de VNC foram desenvolvidos no começo dos anos 2000. Esses sistemas funcionam como uma arquitetura cliente e servidor, onde o servidor transmite a tela para o cliente e o cliente interage normalmente com a tela como se fosse um sistema local. Na maioria das implementações de VNC (TigerVNC, TurboVNC, etc.) a tela pode ser virtual, ou seja, não ha a necessidade de ter um monitor conectado do lado do servidor, nesse caso dizemos que o servidor é *"headless"* ATT, 2023.

Apesar da popularidade dos VNCs para uma grande gama de atividades, seu uso não foi projetado para aplicações que requerem processamento gráfico mais intenso ou mesmo com uma baixa latência suficiente para se jogar jogos mais movimentados e que requerem um tempo de resposta mais rápido, como jogos de ritmo ou de tiro. Muitas das implementações de VNC não são aceleradas por GPU, o que gera sérias dificuldades para se assistir vídeos, por exemplo, sem ter a sensação de travamento.

Devido a essas limitações, outras abordagens começaram a ser testadas, principalmente separando as responsabilidades de *encoding* e *decoding* do protocolo de transmissão. A chegada de protocolos como HTTP/2, gRPC e webRTC permitiram que as transmissões se tornasse mais simples e com maior desempenho em relação aos protocolos anteriores. Combinar ferramentas dedicadas a tarefas específicas e protocolos de transmissão eficientes é uma forma de tornar o problema de *cloud gaming* mais modular e tratável.

Outro protocolo que merece atenção é um protocolo criado pela NVidia especialmente para jogos, chamado GameStream, esse protocolo foi criado inicialmente para transmissão utilizando os dispositivos NVidia Shield. O suporte da NVidia ao protocolo foi descontinuado no começo de 2023, mas existem implementações de código aberto do protocolo PURDY, 2023.

1.3.3 Ambientes

Para tornar uma solução de *cloud gaming* viável é necessário que ela, além de ter uma boa qualidade, consiga ser escalável. Por exemplo, ter a necessidade de um monitor para cada pessoa que esteja usando o serviço elevaria os custos do *datacenter* de forma que o serviço poderia ser inviável. Tecnologias de virtualização como máquinas virtuais vem sendo desenvolvidas desde o século passado e obtendo sucesso numa alta gama de aplicações. Uma máquina virtual permite características desejáveis como isolamento e capacidades mais flexíveis em relação a uma máquina física. Soluções de virtualização como Virtual Box e KVM permitem a passagem de uma GPU para uma máquina virtual mediante uma tecnologia conhecida como *GPU passthrough*, no entanto, essa solução não permite o compartilhamento da GPU com mais de uma máquina virtual, apesar de existirem projetos com avanços nessa frente Yadav e Annappa, 2017.

Contêineres vem surgindo como alternativas viáveis a máquinas virtuais por serem mais leves. Contêineres são capazes de isolar a execução de aplicações ao mesmo tempo que compartilham o *kernel* do *host* e não precisam de recursos dedicados e fixos para eles. Isso permite uma maior flexibilidade na hora de alocar recursos computacionais para os contêineres conforme a necessidade, além da maior facilidade de manutenção. Docker é o principal tipo de contêiner, onde a configuração do contêiner é especificada por meio de um arquivo chamado Dockerfile. Algumas placas da Nvidia possuem a capacidade de serem usados por mais de um contêiner ao mesmo tempo, utilizando diferentes estratégias para esse compartilhamento. Essa demanda por compartilhar GPUs entre diferentes contêineres surgiu principalmente por conta do uso GPUs para treinamentos de modelos de aprendizado de máquina Wu *et al.*, 2023.

Kubernetes é um orquestrador de contêineres de código aberto desenvolvido pelo Google, sendo um dos maiores projetos de código aberto do mundo. Kubernetes funciona

abstraindo o hardware em *"resources"* que podem ser requisitados pelas aplicações de diversas formas. Kubernetes possui diversos componentes que precisam ser instalados nos nós que serão parte do *cluster*, porém os grandes provedores de serviços em nuvem possuem serviços prontos de Kubernetes onde você não precisa administrar as máquinas onde os componentes do Kubernetes roda. Kubernetes se destaca em aplicações de nuvem, pois possui muitos mecanismos que fazem com que características desejáveis sejam alcançadas como resiliência e alta disponibilidade, ao mesmo tempo que é altamente flexível e consegue ser usado para implementar várias arquiteturas diferentes.

1.3.4 Softwares

Alguns softwares possuem um papel bastante fundamental em *cloud gaming*. Dentre estes podemos destacar o FFmpeg, que é um projeto *open source* que consiste num conjunto de programas e bibliotecas que lidam com video e áudio de diversas formas, incluindo para transmissão de dados por meio de diversos protocolos e formatos.

Outros softwares mais diretamente ligados com jogos que são importantes de serem mencionados são o Parsec e o Moonlight, estes softwares foram feitos pensando em *streaming* e Co-op de jogos utilizando a sua própria máquina pessoal para fazer o hosteamento. O parsec é um software proprietário de uso gratuito e o Moonlight e de código aberto e implementa o protocolo GameStream da NVidia.

Capítulo 2

Desafios e objetivos

2.1 Motivação

Devido à relevância do entretenimento e da arte na vida das pessoas, é relevante se discutir formas de se melhorar as experiências das pessoas com o entretenimento. Jogos eletrônicos constituem-se atualmente numa atividade realizada por bilhões de pessoas no mundo, que cada vez mais é independente de gênero ou idade. Dado esse cenário, há uma grande atividade econômica neste setor, sendo que em 2022 foi estimado que o mercado globalmente valia 224,9 bilhões de dólares Research, 2023. O mercado de jogos eletrônicos movimenta bastante dinheiro por conta de diversos aspectos da atividade.

Os custos de se fazer um jogo vem crescendo, assim como o tempo de desenvolvimento. O custo médio de jogos de grandes publicadores é geralmente entre 60 a 80 milhões de dólares Kevuru, 2023. Além do custo, o tempo de produção é bastante extenso, e vem levando cada vez mais tempo Arc, 2023. Essas coisas fazem com que as publicadoras precisem aumentar o preço do jogo de tempos em tempos, e há interesse de que o jogo se mantenha lucrativo durante o período entre os lançamentos.

Além do jogo, é necessário um meio pelo qual se possa jogá-lo. Atualmente, há consoles, computadores, *smartphones*, entre outros. Os componentes principais da interação com o jogo são a tela, o áudio e os comandos do usuário. Mesmo dispositivos que usam recursos especiais, como os óculos de realidade virtual ou aumentada, se utilizam desses componentes. Talvez mais fundamentalmente, do lado do usuário, os sentidos mais importantes são a visão e a audição para perceber a reação do jogo, e os seus comandos para interagir com o jogo. Dispositivos que provêm essas interações são necessários para a interação com o jogo. No entanto, existem outras condições necessárias. Para executar um jogo é necessário processamento, processamento gráfico, armazenamento, memória e conectividade com a internet. As plataformas locais, isto é, computadores, consoles e *smartphones*, permitem que o jogo seja executado com o hardware do usuário. Na maior parte do tempo, o poder computacional dos dispositivos do usuário está ocioso. Além disso, com a necessidade de *hardware* cada vez mais poderoso para jogos, para muitos usuários não se justifica a compra de equipamentos tão caros para que ele possa jogar.

Levando em consideração esse cenário, plataformas de *cloud gaming* surgem como

uma alternativa viável a se jogar com equipamentos locais. As plataformas conseguem otimizar o uso do *hardware*, uma vez que não há necessidade do *hardware* estar ocioso quando o usuário que estava utilizando parar de utilizar.

2.2 Estado da arte

Levando em consideração as principais plataformas de *cloud gaming*, vamos comparar suas características de performance do ponto de vista do usuário e levantar alguns pontos sobre arquitetura da plataforma quando essa informação é conhecida.

2.2.1 Experiência do usuário

Algumas características possuem são bastante impacto no usuário. Resolução é quantidade de *pixels* em relação ao tamanho da tela. Quanto maior for a resolução da transmissão, dada que a resolução da tela do usuário é maior ou igual que a resolução de transmissão, melhor. As resoluções mais comuns hoje em dia são 1280x720p(HD), 1920x1080p(Full HD), 2560x1440p (2K) e 3840x2160p(4K). Essas resoluções possuem *aspect ratio* de 16:9, sendo a mais comum. *Aspect ratio* (razão de aspecto) é a proporção da quantidade de pixels na horizontal em relação à quantidade de pixels na vertical. Nos últimos tempos, resoluções *ultrawide* vem sendo adotadas em monitores para produtividade e jogos. 21:9 e 32:9 (também conhecido como *super ultrawide*) são os *aspect ratios* mais comuns em monitores ultrawide. O uso de telas *ultrawide* afeta a quantidade de pixels, por exemplo, uma resolução Full HD em 21x9 fica 2560x1080. O uso de resoluções maiores aumenta a fidelidade dos detalhes, mas também aumenta a necessidade de mais poder computacional para renderizar nessa resolução. Conforme a pesquisa da Steam de setembro de 2023 2.1, a resolução mais comum é Full HD, mas 1440p tem um crescimento maior.

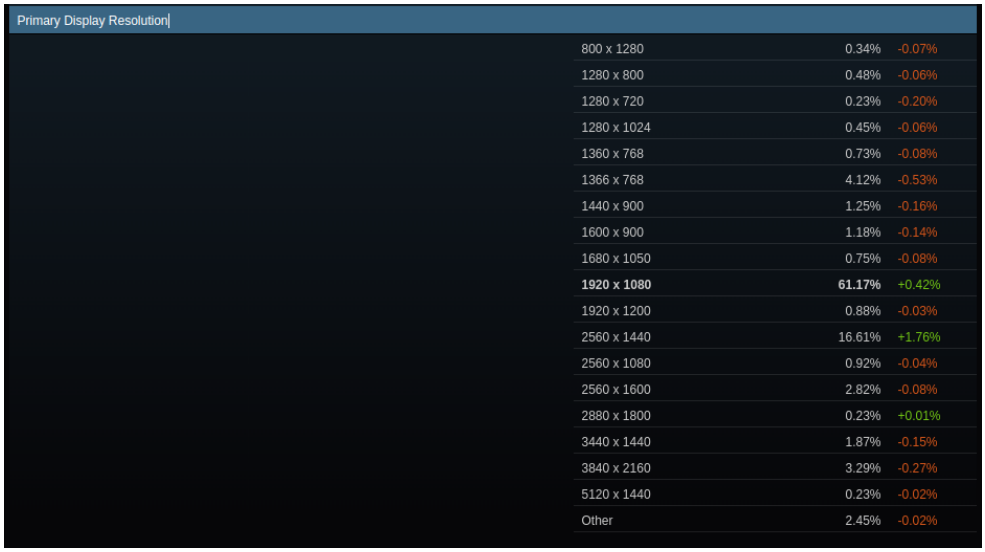

Figura 2.1: *Pesquisa da Steam, resoluções de tela, setembro de 2023. Fonte: https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam*

Quadros por segundo (FPS, em inglês *Frames per Second*) é a quantidade de imagens em sequência em um segundo. Em geral, quanto maior esse número, maior é a sensação

de fluidez. O padrão do cinema é utilizar 24 FPS, que é o mínimo necessário para não se notar quadros individuais. No entanto, jogos costumam se beneficiar de uma taxa mais alta de FPS devido a fatores como fluidez do movimento. É conhecido que uma alta taxa de FPS é necessária para que os olhos não façam esforço para preencher as lacunas de movimento entre os quadros Wang *et al.*, 2023. De modo geral, quanto maior o FPS mais fluido é a sensação, mas também mais computacionalmente exigente. As taxas de FPS em jogos variam bastante e geralmente é um número dinâmico. No entanto, esse número é limitado pela taxa de atualização da tela, medido em Hz, mas tem uma proporção 1:1 em relação à taxa de FPS. Como FPS é um número dinâmico, faz mais sentido mirar as taxas de atualização de telas mais comuns que taxas de FPS arbitrárias. As taxas de atualização mais comuns são 60 hz, 75 hz, 120 hz, 144 hz e 240 hz. Também vale a destacar 30 FPS como um alvo frequente na indústria, pois é um bom compromisso entre fluidez e performance.

As principais plataformas de *cloud gaming* adotaram diferentes abordagens e modelos de negócio para disponibilizar jogos em suas plataformas. Devido ao grande número de jogos feitos por empresas que lançam seus jogos para várias plataformas e não possuem afiliações com nenhuma fornecedora de serviços de *cloud gaming*, as plataformas desse tipo de serviço não podem apenas contar com seus próprios jogos, e muitas vezes as empresas dessas plataformas nem estão muito inseridas no mercado. Dessa forma, é necessário fazer acordos com as publicadoras de jogos para poder disponibilizar um catálogo de jogos mais amplo. Algumas empresas fornecessem jogos gratuitos para os seus usuários, por meio de uma licença ou acordo adquirido com a publicadora, enquanto outras permitem que seus usuários usem os jogos adquiridos em lojas digitais. No entanto, mesmo nesse último modelo, algumas publicadoras não concordam com o uso jogo através do serviço, como foi o caso da Activision e da Bethesda, que solicitaram que a Nvidia que retirasse seus jogos de seu serviço Statt, 2020. A tabela 2.1 mostra alguns dados básicos sobre as principais plataformas em termos do catálogo de jogos e qualidade técnica do serviço.

Plataforma	Resolução Máxima	FPS Máximo	Tamanho do catálogo	Ultrawide	Preço	Inclui jogos	Disponibilidade	Usuários (Milhões)	Requerimento de rede
Xbox Cloud Gaming	Full HD	60	382	Não	17 USD	Sim	Alta	20	10 Mbit/s
Geforce Now	4K	120	1500	Sim	20 USD	Não	Média	25	50 Mbit/s
Amazon Luna	Full HD	60	298	Não	10 USD	Não	Baixa	270 (mil)	10 Mbit/s
Shadow	4K	60	Steam	Sim	50 USD	Não	Baixa	100 (mil)	15 Mbit/s
Playstation Plus	Full HD	60	283	Não	18 USD	Sim	Alta	47.4	15 Mbit/s
Stadia	4K	60	279	Não	10 USD	Alguns	Descontinuado	1.3	35 Mbit/s

Tabela 2.1: *Comparação de plataformas de cloud gaming. Fonte: Própria*

2.2.2 Arquitetura

Devido à diversidade de infraestrutura, patentes e tecnologia, as arquiteturas dos vários serviços de *cloud gaming* são bastante variadas. A Microsoft, por exemplo, optou aproveitar o hardware do Xbox Series X, para montar a infraestrutura do Xbox Cloud Gaming. Ao fazer isso, a Microsoft conseguiu reutilizar a plataforma do Xbox, de forma que milhares de jogos já são compatíveis com o serviço, algo que causou bastante problema para o serviço do Google, o Stadia. Apesar de ser uma plataforma consolidada e otimizada para jogos, o uso do Xbox Series X trouxe algumas desvantagens. Por diversos motivos, as gerações de console tendem a durar entre 6 a 8 anos, com, às vezes, uma melhoria no meio da geração. Ao fixar a arquitetura em console que tem um hardware fixo, você fica preso a uma plataforma, até que você troque ela por outro. Você também fica com um hardware

menos modular em relação à modelos mais tradicionais de arquitetura de servidores, tornando a substituição e melhoria de componentes individuais mais difíceis.

Outras empresas como a Shadow e a Nvidia possuem uma arquitetura mais flexível, mas alocam recursos dedicados para a máquina que usuário está usando. O Shadow é um PC dedicado, então ele está sempre alocado para o usuário, mas o custo da assinatura também é bem mais caro e há um desperdício de recursos para a plataforma. A NVidia é um pouco mais flexível, alocando apenas quando o usuário está usando, mas ela sempre vai alocar o máximo que a assinatura do usuário permite, gerando desperdício a depender do jogo que o usuário esteja jogando EISLER, 2021.

2.3 Lacunas e oportunidades

2.3.1 Desafios relacionados à *cloud gaming*

Um dos principais desafios de *cloud gaming* sempre foi a latência. A latência é definida como o tempo de ida e volta de alguma informação na internet. No caso de *cloud gaming* a informação que precisa trafegar na rede é bidirecional, vídeo e som precisam ir para o usuário enquanto a interação do usuário precisa ir para o *datacenter*. Além do atraso operacional, relacionada ao processamento das informações pelo jogo, *encoding* e *decoding* de vídeo, temos a latência relacionada a transferência das informações pela internet. A velocidade de transmissão de dados pela internet é limitada pela velocidade da luz. Considerando uma rede de fibra óptica de qualidade, quanto mais próximo o *datacenter* estiver do usuário, menor será a latência.

Outro desafio está relacionado as tecnologias envolvidas no processo de execução, renderização e transmissão de dados. Existem muitos *trade-offs* em muitas partes do processo, por exemplo, escolher o nível ideal de compressão de dados, além da qualidade percebida pelo usuário, outro fator que aparece nesse cenário é você gastar mais tempo para comprimir um dado ou enviar um dado maior para o usuário. Outra pergunta que surge é como otimizar o seu uso de hardware. Um exemplo disso é o uso da GPU, onde existem sérias dificuldades em compartilhar seu uso com outras aplicações simultaneamente Yadav e Annappa, 2017.

O catálogo de jogos disponível na plataforma é um dos fatores mais importantes, para isso é necessário ter uma plataforma que não exige esforço para receber novos jogos, ou seja, não é necessário modificar o jogo para que ele rode em uma nova plataforma. Com isso, é necessário estabelecer um bom modelo de negócio de forma que as publicadoras de jogos permitam a disponibilização dos jogos delas através da plataforma, ao mesmo tempo que o preço seja atrativo aos usuários e rentável para a provedora do serviço.

2.3.2 Melhorias arquiteturais

Uma arquitetura e infraestrutura melhores podem ajudar a diminuir custos a curto, médio e longo prazo. A eficiência de uma arquitetura permite que você consiga extrair mais potencial da infraestrutura. Um exemplo disso é o reuso de infraestrutura. Quando há baixa demanda pelo serviço, você consegue utilizar a infraestrutura excedente para

outras tarefas que não requerem uso contínuo, podendo ser realocadas para o serviço conforme necessário. O uso de GPUs é bastante utilizado para treinamento de modelos de aprendizado de máquina. Tais modelos conseguem ser treinados por muitas GPUs ao mesmo tempo, além de poderem ter seu treinamento pausado para serem treinados depois. Além disso, o treinamento de modelos não é uma tarefa contínua. O modelo é treinado para que após o treinamento, o modelo possa ser utilizado para inferência, que costuma ser uma tarefa menos computacionalmente intensiva a depender da escala e da tarefa. Isso demonstra o uso do excedente de GPUs para outras tarefas, mas isso é apenas possível se essa realocação de recursos puder ser feita de forma fácil, rápida e barata.

Jogos ocupam bastante espaço em disco e memória. Apesar de memória e disco estar se tornando mais barato, o consumo desses recursos vem crescendo. Não apenas os jogos estão ocupando cada vez mais espaço devido a texturas de alta resolução e *assets* mais realistas, HDDs já não são mais recomendados pelas produtoras de alguns jogos, devido à limitação da taxa de leitura de HDDs. SSDs podem ser cerca de 100 vezes mais rápidos que HDDs, mas costumam custar mais caro e ter um número limitado da quantidade de escritas possíveis. Por estes motivos, é importante utilizar de forma eficiente o espaço disponível.

2.4 Objetivos

Este projeto visa esclarecer possíveis soluções para problemas comuns em serviços de *cloud gaming*. O projeto terá um foco em se utilizar de tecnologias de código aberto, deixando claro quando a tecnologia sugerida ou mencionada não for aberta.

2.4.1 Escopo

O escopo do projeto está centrado no desenvolvimento de um protótipo, que cobrirá a maioria do que será discutido nesta monografia. O protótipo não visa ser uma solução *end-to-end*, completa e polida para o uso de usuários não técnicos. Os conceitos e soluções apresentadas podem ser parte de uma solução final. A apresentação de alguns conceitos, e usos de casos em outras áreas, podem ser utilizados para discutir possíveis soluções técnicas que podem facilmente serem utilizados para a implementação de funções adicionais.

Não há a intenção de se fazer medições de performance do protótipo, pois é um protótipo para averiguar as soluções técnicas. As avaliações de performance serão feitas de forma qualitativa e sucintamente apenas para ilustrar a capacidade da solução. A racionalidade para isso é priorizar a discussão do maior número possível de soluções e problemas ao invés de focar em um único aspecto. Como o protótipo não foi otimizado para performance e sim para testes e extensibilidade, não faz muito sentido tirar medições precisas.

2.4.2 Protótipo

O objetivo do protótipo é ilustrar como alguns conceitos podem ser implementados na prática. O protótipo pretende ser facilmente reprodutível. Para tal, será possível replicar a infraestrutura e funcionalidade, quase totalmente de forma automatizada. Para tornar a

infraestrutura reprodutível e poder realizar os testes com GPUs de *datacenter*, foi escolhido utilizar a infraestrutura da AWS.

Capacidades

Algumas capacidades que serão discutidas a partir do protótipo são:

- Resoluções *Ultrawide*
- Resolução flexível
- Salvamento de progresso
- Modularidade da solução
- Tempo de carregamento de jogo
- Reuso do hardware
- Disponibilidade (geográfica)
- Baixa latência
- Qualidade da transmissão

Capítulo 3

Metodologia e Resultados

3.1 Arquitetura

Para satisfazer os requerimentos do protótipo foram tomadas algumas decisões. O processo de construção e design do protótipo foi feito a partir de uma abordagem *bottom-up*, ou seja, foi investigado e projetado como os detalhes de implementação seriam resolvidos, antes de generalizar a solução. Um dos motivos para a adoção dessa abordagem foi por ser um sistema complexo, com muitas partes que interagem entre si e pela abordagem experimental do projeto. Devido a essa complexidade, definir detalhes mais gerais da arquitetura poderia limitar as opções.

Será descrito a arquitetura seguindo o mesmo processo adotado no design e implementação do protótipo. Para não sobrecarregar o leitor, descrições técnicas das tecnologias e ferramentas utilizadas, cujo funcionamento e detalhe não são necessários para a compreensão, serão feitas no apêndice A.

A figura 3.1 mostra uma visão geral do protótipo, ao nível apenas do Kubernetes. Essa arquitetura permite também a discussão de uma série de estratégias que são viáveis e simples de serem implementadas para a resolução de problemas comuns em *cloud gaming*.

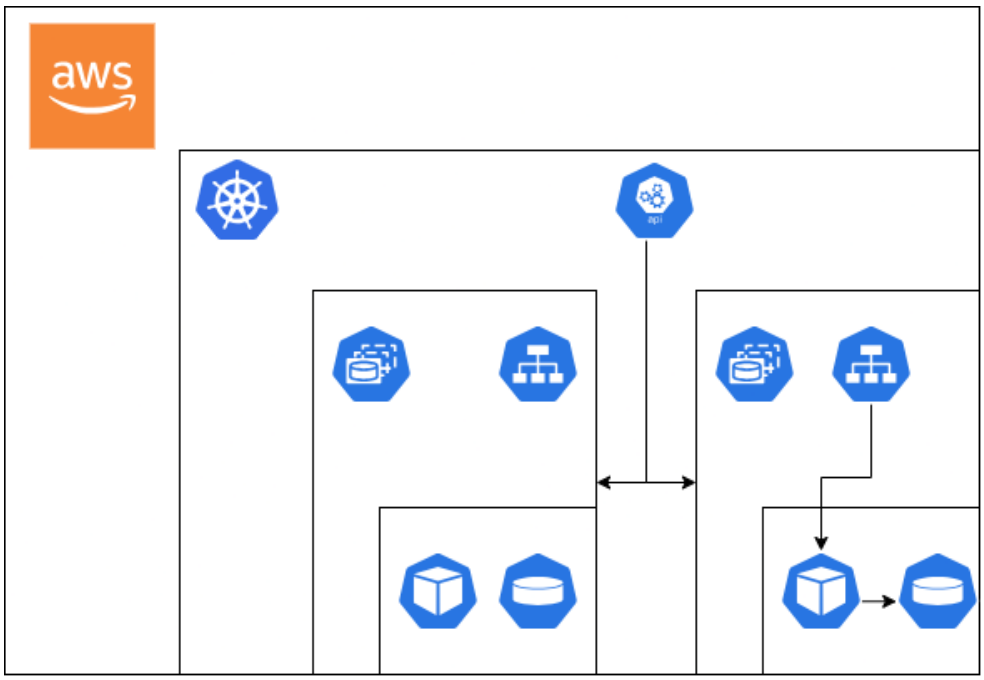

Figura 3.1: *Arquitetura de protótipo ao nível de recursos do Kubernetes. Fonte: Própria*

3.2 Aplicação

A aplicação foi construída tendo como base principal dois programas, o Sunshine e o Moonlight. O Moonlight faz o papel de cliente e o Sunshine o papel de servidor em uma arquitetura cliente-servidor. O Moonlight surgiu primeiro e tinha uma ferramenta embutida para fazer o papel de servidor chamada de *Internet Hosting Tool*. O Moonlight foi desenvolvido inicialmente em um hackaton, uma espécie de maratona de desenvolvimento de software e projetos relacionados a tecnologia, em 2013. Os criadores resolveram continuar o desenvolvimento após o hackaton. Para aumentar a modularidade da solução, e melhorar a parte do servidor, foi criado o Sunshine em 2020. Ambos os programas implementam o protocolo GameStream da Nvidia que foi feito para a transmissão de jogos em uma rede local.

A forma usual de se utilizar esses dois programas consiste em rodar o Sunshine na máquina que rodará os jogos e o Moonlight na máquina que visualizará e interagirá com o jogo. Existem vários clientes do Moonlight, tendo clientes para Linux, Android, MacOS, IOS, Windows e *ports* feitos pela comunidade. O cliente é responsável por receber o vídeo e o áudio do servidor, além de receber os comandos do jogador e enviar para o servidor. O cliente do Moonlight possui uma interface gráfica para o usuário se conectar a um servidor e escolher o jogo disponível. O servidor é responsável por rodar o jogo, enviar o vídeo e áudio gerados pelo jogo e processar os comandos do jogador enviados pelo cliente.

Como mencionado anteriormente, as duas aplicações se comunicam por uma implementação do protocolo GameStream da Nvidia. A Nvidia descontinuou a implementação proprietária deste protocolo no começo de 2023, mas isso não afeta a implementação aberta feita pelos criadores. Vale destacar o uso da biblioteca FFmpeg, muito utilizada em diversas aplicações que lidam com vídeo e áudio, e que também é utilizada pelas aplicações

para realizar *encoding* e *decoding* de vídeo e áudio eficientemente. O uso do Moonlight e Sunshine na construção do protótipo é fundamental, pois permite que não seja necessário reimplementar todo o tratamento de vídeo, áudio e controles do jogador, além da definição de um protocolo.

3.2.1 Plataforma de instalação

Apesar das duas aplicações lidarem perfeitamente com o *streaming* de jogos, elas, por si só, não formam uma plataforma de *cloud gaming*. Isso se deve ao fato de que elas cobrem o uso de uma única pessoa usando. É preciso uma instância do Sunshine e uma instância do Moonlight para uma pessoa jogar. Em uma solução de *cloud gaming* é necessário milhares de instâncias do Sunshine rodando do lado servidor, enquanto cada usuário roda a sua instância do Moonlight.

Para que o serviço possa executar milhares de instâncias do Sunshine, algumas condições são necessárias. É preciso que cada instância tenha isolamento entre as outras, pois como cada instância do Sunshine executará um jogo para um único usuário, é importante que os comandos do usuário não conflitem com outras instâncias do Sunshine. Além disso, o isolamento das instâncias permite que diferentes recursos sejam associados com uma instância. Isso permite, por exemplo, que um usuário que esteja jogando um jogo mais leve, uso apenas os recursos necessários, sem limitar os recursos de um usuário que esteja jogando um jogo mais pesado e nem desperdiçar recursos que poderiam estar sendo usados para outras coisas.

Para fazer esse isolamento da instância utilizamos contêineres, que funcionam como máquinas virtuais leves. Contêineres proveem isolamento de dados, permitem um identificador de rede único, isolamento dos programas e da memória. Apesar do isolamento, eles oferecem vantagens como compartilhar o *kernel* do sistema operacional da máquina onde estão rodando, possuindo acesso rápido e eficiente ao *hardware*. Contêineres são executados por um motor como Docker ou Containerd. A especificação de um contêiner é geralmente feita com um arquivo chamado Dockerfile, mesmo para os motores que não sejam o Docker. Para mais informações sobre contêineres ver apêndice A.1.

Foram feitos testes a respeito da conteinerização do Sunshine em ambiente local, para garantir que o *overhead* introduzido pela conteinerização não iria atrapalhar o desempenho da solução. Usualmente o uso mais comum do Sunshine é numa máquina normal que possui acesso a uma GPU, possui uma placa de som e está conectado a um monitor. Para a solução ser escalável, é necessário remover essas dependências que não são necessárias do lado do servidor, apenas do cliente.

3.2.2 Vídeo

Contêineres são normalmente baseados em Linux, então precisamos trabalhar com a suíte de tecnologias e ferramentas nativas do Linux para resolver esses problemas. Para remover o requerimento do monitor, precisamos fazer com que o jogo e o Sunshine operem de forma *headless*. No Linux, *Display servers* são responsáveis pela visualização das aplicações que estão rodando. Existem dois principais tipos de *Display servers*, os que implementam o protocolo Wayland e os que implementam o protocolo X11. Devido ao

Wayland ser o protocolo mais novo, resolver questões de segurança e o X11 estar em modo de manutenção, sendo gradualmente trocado pelo Wayland na maioria dos ambientes Linux, ele foi o escolhido para fazer parte da implementação inicial.

O Sway é um compositor que fala o protocolo Wayland e ele tem suporte embutido para operar de forma *headless*, ou seja, sem a necessidade de um monitor. Ao rodar um jogo dentro do Sway que está rodando de forma headless, o jogo consegue renderizar em um *framebuffer* virtual normalmente e ao utilizar o Sunshine ele consegue gravar essa tela virtual, fazer o *encoding* utilizando o FFmpeg e enviar para o Moonlight pelo protocolo GameStream.

3.2.3 Áudio

No Linux, existem diversas *stacks* de áudio que servem para diferentes propósitos. O Alsa é o módulo do *Kernel* responsável pelas implementações de baixo nível. No espaço de usuário temos o PulseAudio, o Jack e o PipeWire. Escolhemos utilizar o PipeWire para lidar com a questão do áudio dentro do contêiner, principalmente pelo fato dele normalmente estar mais integrado com o ecossistema Wayland, ter mais flexibilidade em manipular fluxos de áudio que o PulseAudio e ser mais simples de operar que o Jack.

Como não há a necessidade de reprodução de som do lado do servidor e usualmente não serem equipados com placas de som, é necessário tratar para que o som do jogo seja transmitido. Para isso é necessário que dois ajustes sejam feitos. O primeiro é feito ao se iniciar o Pipewire no contêiner. Foi configurado uma *sink* virtual nas configurações do Pipewire. *Sinks* permitem que programas escrevam bytes de áudio nelas. O jogo ao executar manda o áudio para qualquer *sink* que tiver disponível, no caso a nossa é a única. O segundo ajuste é nas configurações do Sunshine, onde é definido de qual *sink* ele lerá o áudio. Ajustamos essa configuração para a nossa *sink* criada anteriormente.

3.2.4 Entrada

3.3 Configuração do contêiner

Definamos mais precisamente a configuração feita para montar o protótipo localmente. No repositório https://github.com/naythanN/TCC, encontramos essas configurações e uma instrução simples de como utilizar. Como os arquivos são poucos e curtos, mostraremos as configurações aqui na íntegra, pare que seja explicado os conceitos e funcionamento de forma mais detalhada, assim como discutir os possíveis problemas.

3.3.1 Dockerfile

Como foi mencionado anteriormente, o Dockerfile contém toda a informação para montar uma imagem.

Programa 3.1 Dockerfile para execução local

```
1  # syntax=docker/dockerfile:1.4
2  # artifacts: true
3  # platforms: linux/amd64,linux/arm64/v8
4  # platforms_pr: linux/amd64
5  # no-cache-filters: sunshine-base,artifacts,sunshine
6
7  FROM lizardbyte/sunshine:8ff2022-ubuntu-22.04
8  ENV DEBIAN_FRONTEND=noninteractive
9
10 USER root
11 RUN # Install dependencies \
12    apt-get update && apt-get install -y astromenace x11-apps mesa-utils
          libxrender1 libwayland-server0 libvulkan1 libxrender-dev libdrm-amdgpu1
           \
13    libva-drm2 libva-x11-2 va-driver-all libgbm1 libgles2-mesa libegl1 libgl1-
          mesa-dri libvulkan1 libvdpau1 libnuma1 \
14    libavahi-client3 libgles2-mesa mesa-vulkan-drivers mesa-vdpau-drivers
          libwayland-egl1 libgles2-mesa-dev mesa-utils-extra libappindicator3-1
          libopus0 libmfx1 vainfo apt-file libboost-chrono1.74.0 udev \
15    libcap2-bin libxres-dev libxres1 libsdl2-2.0-0 sway xwayland libgles2
          pciutils kmod libwayland-egl1 mesa-va-drivers \
16    pipewire pipewire-pulse wget sudo make pulseaudio pipewire-audio-client-
          libraries libspa-0.2-dev libpipewire-0.3-dev gstreamer1.0-pipewire dbus
          -x11 bash-completion wireplumber alsa-utils curl gnupg2\
17    libcudart11.0 nvidia-cuda-dev nvidia-profiler libnvidia-fbc1-535 libdrm-
          nouveau2 libdrm2 && \
18    # Create some directories and set permissions \
19    mkdir -p /run/dbus && mkdir -p /dev/snd && mkdir -p /run/user/1000 &&
          chown lizard /run/user/1000
20
21 # Install Steam
22 RUN mkdir -p /src && \
23    cd /src && \
24    wget http://repo.steampowered.com/steam/archive/precise/steam_latest.tar.
          gz && \
25    tar xzvf steam_latest.tar.gz && \
26    cd /src/steam-launcher && \
27    make install && echo "========== Setup sudoers ==========" && \
28    echo "lizard ALL=(ALL:ALL) NOPASSWD: ALL" >> /etc/sudoers && \
29    echo "========== Install Steam =========="
30
31 # set special permissions
32 RUN echo 'root:123' | chpasswd && setcap cap_sys_admin+p $(readlink -f $(
       which sunshine))
33
34 # Set the user to run the container
35 USER lizard
36
37 # Set environment variables
38
39 ENV WLR_BACKENDS=headless
40 ENV DISABLE_RTKIT=y
```

cont ⟶

⟶ *cont*

```
41  ENV PIPEWIRE_RUNTIME_DIR=/run/user/1000
42  ENV PULSE_RUNTIME_DIR=/run/user/1000
43  ENV WLR_LIBINPUT_NO_DEVICES=1
44  ENV WAYLAND_DISPLAY=wayland-1
45  ENV XDG_RUNTIME_DIR=/run/user/1000
46  ENV XDG_SESSION_TYPE=wayland
47  ENV DISPLAY=:0
48
49  COPY setup.sh $HOME/setup.sh
50  COPY sunshine/ /home/lizard/.config/sunshine
51
52  ENTRYPOINT /home/lizard/setup.sh
```

A primeira escolha a ser feita em um Dockerfile, é a escolha da imagem base. Existem vários critérios para a escolha da imagem base, como tamanho, quantidade de dependências pré-instaladas e sistema operacional da imagem base. Escolhemos utilizar a imagem base disponibiliza pelo projeto do Sunshine. Essa imagem, apesar de ser disponibilizada pelo Sunshine, não é recomendado usar ela do jeito que ela vem, conforme a própria documentação. A imagem base do Sunshine contém o binário da aplicação e as suas dependências. Por sua vez, a imagem base do Sunshine é baseada na imagem oficial Ubuntu 22.04 LTS, distribuída pela Canonical. Outras opções viáveis eram imagens que já continham dependências de *drivers* gráficos e bibliotecas relacionadas. Foi escolhido esta imagem, pois não seria preciso encontrar e instalar as dependências do binário manualmente, além de termos mais liberdade para definir e customizar as dependências gráficas.

Uma das principais funções de contêineres é conseguir isolar dependências da aplicação em relação à máquina. Embora há algumas exceções, como veremos a respeito de alguns *drivers* da placa de vídeo e dependências que ajudam a injetar dispositivos para dentro do contêiner, a maioria das dependências podem ser instaladas no contêiner. A instrução RUN na linha 11 instala as nossas dependências.

Algumas configurações de ambiente são necessárias para o funcionamento da aplicação e dos outros componentes para que o protótipo funcione. É necessário configurar algumas variáveis de ambiente para que as aplicações envolvidas funcionem corretamente. Algumas delas, como as das linhas 39 e 46, são necessárias para que as aplicações saibam que estão rodando em uma sessão Wayland. Outras como na linha 40 é necessário para o funcionamento correto do áudio no Pipewire.

Script de inicialização

A listagem 3.3.1 mostra o *script* de inicialização. Ele define quais componentes vão ser inicializados e a sua ordem de subida.

O primeiro passo na execução local é garantir em qual lugar está o *Render Device*. Os *Render Devices* ficam em caminho da forma /dev/dri/renderD*. O Sunshine sempre utiliza o *Render Device* na posição /dev/dri/renderD128. Em casos de sistemas com mais de uma GPU, é necessário fazer alguns ajustes. Como a máquina de teste possuía duas

GPUs, e seria ideal usar a GPU mais potente para o teste. Foi feito uma pequena adaptação utilizando links simbólicos como mostra o bloco de comandos da linha 5 do *script*.

Outro passo necessário é inicializar o udevd e o Dbus (linha 10) que são serviços de comunicação entre aplicações e/ou dispositivos. Esses serviços são normalmente inicializados pelo Systemd em um *desktop* comum, mas em contêineres geralmente não é utilizado o Systemd, pois não costuma ser necessário e torna a inicialização do contêiner mais pesada devido ao alto número de serviços que ele traz.

Após inicializar o udevd e dbus, inicializamos o Sway (linha 19). É importante inicializar o Sway antes do Áudio, pois ele vai inicializar os principais recursos do Wayland que também vão ser utilizados pelo Pipewire. Depois do Sway ser inicializado, precisamos de 3 componentes do Pipewire: pipewire, pipewire-pulse e wireplumber (linha 23). Depois do Pipewire subir é necessário configurar uma *Sink* com um nome específico que o Sunshine vai utilizar para ler o áudio escrito pelo jogo nessa *Sink*. Essa configuração da *Sink* é feita na linha 31.

Em seguida, iniciamos o Sunshine passando um arquivo de configuração (linha 35). No nosso protótipo só foi necessário configurar a *Sink* que o Sunshine utilizaria: `audio_sink = sink-sunshine-stereo`. Depois que temos tudo isso rodando dentro do contêiner, podemos iniciar o jogo na linha 39.

Programa 3.2 *Script* de inicialização para execução local em Wayland usando placas de vídeo AMD

```
1  #!/bin/bash
2
3  echo "Setting up init"
4
5  echo "setting correct video card"
6  #sudo rm /dev/dri/renderD128
7  #sudo ln -s /dev/dri/renderD129 /dev/dri/renderD128
8  sudo chmod 777 -R /dev/dri
9
10 echo "initting udevd"
11 XDG_RUNTIME_DIR=/tmp sudo /lib/systemd/systemd-udevd &
12 sleep 5
13
14 echo "starting dbus"
15 XDG_RUNTIME_DIR=/tmp sudo dbus-daemon --system &
16 XDG_RUNTIME_DIR=/run/user/1000 dbus-daemon --session &
17 sleep 5
18
19 echo "starting sway"
20 XDG_RUNTIME_DIR=/run/user/1000 sway &
21 sleep 5
22
23 echo "starting pipewire"
24 XDG_RUNTIME_DIR=/run/user/1000 pipewire &
25 sleep 5
26 XDG_RUNTIME_DIR=/run/user/1000 pipewire-pulse &
```

cont ⟶

⟶ *cont*

```
27  sleep 5
28  XDG_RUNTIME_DIR=/run/user/1000 wireplumber &
29
30  sleep 5
31  pw-loopback -m '[ FL FR]' --capture-props='media.class=Audio/Sink node.name=
        sink-sunshine-stereo' &
32
33  # echo "starting sunshine"
34  sleep 5
35  XDG_RUNTIME_DIR=/run/user/1000 sunshine /home/lizard/.config/sunshine/
        sunshine.conf &
36
37  # echo "starting game"
38  sleep 5
39  XDG_RUNTIME_DIR=/run/user/1000 /usr/games/AstroMenace
```

O Dockerfile sozinho não constitui uma imagem ou contêiner. Precisamos utilizar o Dockerfile para construir (*build*) uma imagem e a partir da imagem produzir um contêiner. Para realizar o *build* do contêiner utilizamos o comando: `docker build -t example ./` na pasta onde o Dockerfile se encontra. Dessa forma é obtida uma imagem que pode ser utilizada para criar um contêiner. O seguinte comando pode ser utilizado para criar um contêiner a partir da imagem.

```
1  docker run -it --device=/dev/dri/renderD129 --privileged --cap-add SYS_ADMIN \

2    -p 47984-47990:47984-47990/tcp \
3    -p 48010:48010 \
4    -p 47998-48000:47998-48000/udp \
5    -v /home/nijima/sunshine:/home/lizard/.config/sunshine \
6    -v /sys/fs/cgroup:/sys/fs/cgroup:ro \
7    -e PUID=1000 -e PGID=100 example
```

Para que o protótipo funcione corretamente, é necessário especificar as portas que devem ser abertas no contêiner e qual o mapeamento delas externamente. Aqui foi mapeado as mesmas portas, tanto externamente quanto internamente. Além disso, é necessário especificar o volume onde se encontra a configuração do Sunshine. Para que os comandos de entrada como mouse, teclado e controle funcionem é necessário permissões especiais como o mapeamento do cgroup e o SYS_ADMIN.

3.3.2 Execução local

Como mencionado nos objetivos do projeto, foi decidido não fazer análise de desempenho de forma exaustiva, apenas uma análise qualitativa.

Nesta fase, utilizamos o jogo de teste AstroMenace. AstroMenace é um jogo onde você controla uma nave e precisa destruir asteroides, desviar deles e destruir outras naves. Escolhemos ele por ser um jogo simples, de código aberto, já disponível nos repositórios do Ubuntu. Este jogo foi escolhido, pois atende algumas características. Ele é um jogo considerado leve, mas possui gráficos 3D. Dessa forma, torna-se inviável renderizar o

jogo sem aceleração da placa de vídeo. Outra característica importante é o fato dele ser um jogo de movimentação rápida, isso faz ele ser bastante sensível a atrasos e falta de fluidez. Isso torna o nosso teste mais eficaz enquanto o jogo vai punir o protótipo por qualquer ineficiência, ao mesmo tempo que pode ser rodado em placas gráficas integradas ou discretas de baixo desempenho.

Para realizar o teste rodamos o jogo normalmente e pelo contêiner separadamente e medimos o uso de CPU (a GPU é integrada). A máquina utilizada para o teste foi um Notebook da Acer, Ryzen 3500U, 8 GB de RAM e SSD. O sistema operacional utilizado para rodar o teste foi o NixOS.

Ao rodar o jogo normalmente, fora do protótipo, vimos o uso de CPU subir até 40%. Ao colocar o jogo para funcionar dentro do protótipo. O uso de CPU foi a 55%. O aumento do uso de CPU é esperado devido a alguns fatores. O mais relevante é o processamento extra. O uso do Sunshine e do Moonlight implica o *encoding* da transmissão do jogo pelo Sunshine, envio para o Moonlight e o *decoding* pelo Moonlight. Esse processamento extra não depende do quão pesado é o jogo em termos de texturas, memória e outras capacidades gráficas. O uso de processamento para *encoding* e *decoding* depende da resolução da transmissão, FPS e a qualidade do encoding utilizado.

Normalmente, em processamento de vídeo em geral, não há *encoding* sem perda (*lossless*). Isso se deve ao fato que o tempo de processamento seria muito grande, ou seria tão pesado quanto o própio jogo. Em relação ao tempo gasto no *encoding* e *decoding*, a depender das configurações utilizadas na biblioteca FFmpeg, existem alguns *trade-offs*. É possível utilizar um encoder que seja mais rápido, mas sua decodificação leva mais tempo e vice e versa. No contexto de *cloud gaming*, faz sentido que o processamento pesado ocorra do lado do servidor por ser uma máquina mais potente. Um *trade-off* mais interessante de ser pensado é em relação ao uso de um *encoding* com maior grau de fidelidade em relação ao tempo de processamento. Essa discussão é interessante, pois influencia duas das importantes características do sistema, a qualidade visual e o atraso das informações.

Tanto a qualidade visual quanto o atraso das informações podem impactar a experiência do usuário. O uso de um *encoding* com maior fidelidade, traz imagens mais nítidas e com menos artefatos. No entanto, exige um tempo de processamento maior ou máquinas mais potentes, que influencia respectivamente o atraso ou o custo de manter o serviço.

Ao comparar o jogo rodando localmente com o rodado dentro do contêiner, não foi notado diferenças consideráveis em termos de qualidade visual, latência e fluidez. O som também foi bastante fluído, apesar de ser possível notar alguns artefatos no som ocasionalmente.

Apesar dos comandos do mouse, do teclado e do controle fluírem pelos mesmos canais, os comandos de mouse não estavam funcionando no protótipo. Por serem semelhantes, é muito provável ser algum tipo de detalhe na configuração da imagem ou no *script* de inicialização, não sendo uma limitação da arquitetura escolhida ou das aplicações envolvidas.

3.4 Infraestrutura

Dado que a solução em contêiner funcionou bem localmente, num cenário um para um. É necessário procurar formas de escalar a solução para ser possível tirar proveito dos benefícios de se rodar em larga escala. Para testarmos a solução em um ambiente de nuvem real e com os recursos apropriados para uma solução do tipo, escolhemos utilizar o serviço de nuvem pública da AWS.

3.4.1 Kubernetes

Kubernetes é o maior software de código aberto para orquestração de contêineres do mundo. Ele é composto por várias camadas de abstrações que facilitam a implantação de diversos tipos de aplicações em larga escala e com alta resiliência.

Devido a sua popularidade, as grandes provedoras de serviços de nuvem disponibilizam serviços baseados em Kubernetes. A Amazon disponibiliza o EKS (*Elastic Kubernetes Service*), como um serviço de kubernetes gerenciado pela Amazon. Como o Kubernetes possui muitos componentes e partes, um serviço gerenciado ajuda a abstrair certas complexidades, como, por exemplo, a falta da necessidade de configuração do kubelet e do *control plane*. Para mais detalhes sobre Kubernetes e seus componentes, ver o apêndice A.1.

Escolhemos utilizar Kubernetes, pois é uma plataforma aberta, amplamente utilizada e bastante poderosa. Se necessário, poderíamos utilizar o serviço de nuvem pública do Google ou da Microsoft mudando poucas coisas, pois ambos fornecessem serviços de Kubernetes. Utilizamos a ferramenta eksctl para ajudar no provisionamento do *cluster*.

Por ser uma plataforma aberta, Kubernetes permite que vários *plugins* sejam feitos. Kubernetes possui uma API poderosa que permite que vários recursos sejam controlados programaticamente.

3.4.2 Nvidia

GPUs são necessárias para rodar jogos. Escolhemos utilizar as placas de vídeo da Nvidia devido a uma série de tecnologias que permitem o compartilhamento de uma GPU entre diversos contêineres e um ecossistema mais maduro para Kubernetes e contêineres do que a AMD Nvidia, 2023a. Existem mais detalhes sobre como funcionam essas tecnologias de compartilhamento, outros componentes externos desenvolvidos pela Nvidia e outros projetos que interagem com as placas, especialmente componentes projetados para rodarem em Kubernetes como o GPU Operator e o Nos. Estes detalhes podem ser vistos no apêndice A.2.

3.5 Configuração do protótipo em nuvem

Dado o contexto básico dos componentes básicos apresentados e a informação suplementar disponível no apêndice A.1. Vamos mostrar o passo a passo para subir o protótipo de forma funcional no EKS da Amazon. Todos os arquivos de definição

e *scripts* de inicialização para essa parte do protótipo também estão no repositório https://github.com/naythanN/TCC.

3.5.1 Configuração do *cluster*

Para configurarmos o *cluster* na AWS utilizamos o eksctl. A listagem 3.5.1 mostra o arquivo de definição do *cluster*.

Programa 3.3 Eksctl

```
1   apiVersion: eksctl.io/v1alpha5
2   kind: ClusterConfig
3   metadata:
4     name: gpu
5     region: sa-east-1
6     version: "1.26"
7   iam:
8     withOIDC: true
9   addons:
10    - name: vpc-cni
11    - name: coredns
12    - name: kube-proxy
13  nodeGroups:
14    - name: workers
15       instanceType: g5.xlarge
16       ami: ami-01d79101d667eadcc # sa-east-1
17       amiFamily: Ubuntu2004
18       minSize: 1
19       desiredCapacity: 1
20       maxSize: 1
21       volumeSize: 50
22       overrideBootstrapCommand: |
23         #!/bin/bash
24         source /var/lib/cloud/scripts/eksctl/bootstrap.helper.sh
25         /etc/eks/bootstrap.sh ${CLUSTER_NAME} --kubelet-extra-args "--node-
               labels=${NODE_LABELS}"
26       ssh:
27         allow: true
28         publicKeyPath: ~/.ssh/id_rsa.pub
```

Nesse arquivo é possível configurar algumas opções gerais do *cluster*. Nesse arquivo definimos a versão do Kubernetes utilizadas, aqui utilizamos a versão 1.26 que é uma das mais recentes e tem compatibilidade com as opções que precisamos. Aqui também é definido quais *addons* serão adicionados ao *cluster* em tempo de provisionamento, pois também é possível adicioná-los mais tarde. A principal configuração presente no arquivo é sobre os nós. Um nó em Kubernetes é basicamente uma máquina física ou virtual onde é instalado alguns componentes como o Kubelet, necessários para a criação dos outros objetos do Kubernetes.

Na seção do arquivo YAML iniciada na linha 13 é onde é definido os nós. Utilizamos um conjunto de nós, que possui uma máquina (linha 19), com espaço de armazenamento

de 50 Gb (linha 21), do tipo g5.xlarge (15) e utilizando como sistema operacional o Ubuntu 20.04 LTS (linha 17).

Essas escolhas se justificam pelos seguintes motivos. Utilizamos apenas um conjunto de nós e um único nó para economizar recursos. Para os fins deste protótipo, conseguiríamos subir até 48 *Stateful-sets* rodando o protótipo, o que é mais que o suficiente para teste, pois permite enxergar todos os cenários usuais. Escolhemos uma máquina do g5.xlarge, pois é a mais nova da família g, e a menor instância possível em termos de capacidade. Essa instância possui 4 cores de CPU (virtualizados), 16 GiB de memória RAM, 250GB de armazenamento e uma GPU A10G da Nvidia com 24 GiB de memória. Escolhemos o Ubuntu 20.04 LTS como sistema operacional por recomendação da documentação da Nvidia para o uso do *GPU operator* Nvidia, 2023b.

Configuração dos nós

A configuração dos nós foi semi-automatizada, mas é certamente possível automatizar completamente a sua criação por meio de configurações do kubernetes, utilizando o kubectl nos *scripts* ou utilizando ferramentas externas como Open Tofu (Terraform) e o CloudFormation da AWS. As partes não automatizadas são descritas como operações em comentários.

Para a criação do *cluster* em si na AWS com o tipo de nó especificado no arquivo 3.5.1, utilizamos o comando da linha 1 do arquivo 3.5.1. O próximo comando do *script* (linha 2) foi uma necessidade devido à falta de uma função num dos componentes do GPU Operator da Nvidia. Esse comando instala os *drivers* da Nvidia diretamente no nó. Idealmente, isso não seria necessário, pois o *driver* já está incluso num dos *Daemonsets* que compõe o GPU Operator. No entanto, ao utilizar o *GPU operator* foi notado que os *render devices* não eram montados dentro dos *Pods*. Não foi encontrado o motivo exato disso. Utilizamos a ferramenta smarter-device-manager para tentar montar os *devices* necessários, mas não funcionou. Por experimentação, ao instalar o *driver* diretamente no nó e desativar o driver que era implantado como um *DaemonSet* pelo GPU Operator, conseguimos acessar os *render devices* dentro do *Pod*. Pelo que foi observado, não existe uma desvantagem em instalar esse *driver* no node, pois não há nenhum compromisso em termos de escalabilidade, performance e resiliência.

Helm é como se fosse um gerenciador de pacotes para Kubernetes. O Helm permite que diversos objetos do Kubernetes sejam implantados de forma automática e que esses objetos sejam configuráveis. Um arquivo do Helm é chamado de *Chart*. Utilizamos vários *Helm Charts* para implantar diversas ferramentas e funcionalidades necessárias para concretizar o funcionamento do protótipo. O comando `helm repo add` é utilizado para adicionar um Chart remoto no ambiente local. Complementarmente, é utilizado o comando `helm repo update` para atualizar os repositórios locais com os remotos.

O primeiro *Helm Chart* que precisa ser instalado é o do GPU Operator da Nvidia. Conforme mencionado anteriormente, o GPU Operator é um componente fundamental para se utilizar GPUs da Nvidia em um ambiente Kubernetes. Ao instalar o GPU Operator, utilizamos os componentes padrões para implantação, mas desabilitamos os *drivers* porque já instalamos ele no nó. O comando para a instalação do GPU Operator se encontra na linha 7. Note que desabilitamos o migManager, pois iremos utilizar o MPS.

Antes de seguir para o próximo passo, é necessário esperar que todos os componentes do GPU Operator estejam com condição *FINISHED* ou *RUNNING*. Como mostrado no apêndice A.2, dois componentes da Nebuly serão importantes. Vamos utilizar o *fork* do Nvidia Device Plugin da Nebuly, que vai nos permitir utilizar o MPS dentro do Kubernetes. Outro compomente que será utilizado é o Nos, que vai permitir o provisionamento automático de *slices* de GPU Nebuly, 2023. Para utilizar o *fork* do Nvidia Device Plugin da Nebuly, precisamos desativar o da Nvidia, removendo o *Daemonset* responsável pela sua implantação. A linha 16 mostra o comando para iniciar a edição. Após rodar esse comando, é necessário encontrar a linha equivalente à linha 17 que vai estar marcada como *true* por padrão e deve ser mudada para *false*.

Após essa operação, precisamos dar uma *label* para o nó como na linha 22, para que o Nos reconheça esse nó como um nó que ele deve operar. Em seguida, utilizamos os *Helm Charts* disponíveis para instalar o *Device Plugin* e o Nos nas linhas 25 e 31.

Para conseguir testarmos mais de uma instância do protótipo é necessário tem algum mecanismo para direcionar o tráfego. Um modo simples de fazer isso é utilizando um balanceador de carga, apesar de não ser a solução ideal, pois ele não garante que mais de um jogador não vá parar no mesmo *StatefulSet*. Uma solução ideal seria utilizar algum tipo de *gateway* para o direcionamento do tráfego. No entanto, um balanceador de carga com *ip-stickness* é suficiente para conseguirmos testar. Já que estamos no ambiente da AWS, utilizamos o AWS *Network Load Balancer* (NLB). As linhas 37, 41 e 53 mostram os passos necessários para a sua criação.

Programa 3.4 Script para configuração dos nós

```
1  eksctl create cluster -f cluster_config.yaml
2  # sudo apt install nvidia-driver-535 run this on node for nvidia, create the /
       dev/dri render devices
3  helm repo add jetstack https://charts.jetstack.io
4  helm repo add nvidia https://helm.ngc.nvidia.com/nvidia \
5     && helm repo update
6
7  helm install --wait --generate-name \
8      -n gpu-operator --create-namespace \
9      nvidia/gpu-operator --version v23.6.0 \
10     --set driver.enabled=false \
11     --set migManager.enabled=false \
12     --set mig.strategy=mixed \
13     --set toolkit.enabled=true
14
15 # Remove default nvidia device driver,
16 # k edit daemonset.apps/nvidia-device-plugin-daemonset -n gpu-operator
17 # nvidia.com/gpu.deploy.device-plugin=false
18
19
20 kubectl apply -f https://github.com/cert-manager/cert-manager/releases/
       download/v1.12.0/cert-manager.yaml
21
22 kubectl label nodes $(kubectl get nodes | cut -f1 -d " " | tail -1) "nos.
```

cont ⟶

```
    ⟶ cont
        nebuly.com/gpu-partitioning=mps"
23  kubectl label nodes $(kubectl get nodes | cut -f1 -d " " | tail -1) "smarter-
        device-manager=enabled"
24
25  helm install oci://ghcr.io/nebuly-ai/helm-charts/nvidia-device-plugin \
26    --version 0.13.0 \
27    --generate-name \
28    -n nebuly-nvidia \
29    --create-namespace
30
31  helm install oci://ghcr.io/nebuly-ai/helm-charts/nos \
32    --version 0.1.2 \
33    --namespace nebuly-nos \
34    --generate-name \
35    --create-namespace
36
37  aws iam create-policy \
38      --policy-name AWSLoadBalancerControllerIAMPolicy \
39      --policy-document file://setup_files/iam_policy.json
40
41  eksctl create iamserviceaccount \
42    --cluster=gpu \
43    --namespace=kube-system \
44    --name=aws-load-balancer-controller \
45    --role-name AmazonEKSLoadBalancerControllerRole \
46    --attach-policy-arn=arn:aws:iam::380285632927:policy/
          AWSLoadBalancerControllerIAMPolicy \
47    --approve
48
49  helm repo add eks https://aws.github.io/eks-charts
50
51  helm repo update eks
52
53  helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
54    -n kube-system \
55    --set clusterName=gpu \
56    --set serviceAccount.create=false \
57    --set serviceAccount.name=aws-load-balancer-controller
```

Implantação do protótipo

Com os recursos computacionais e de rede configurados, podemos fazer a implantação do protótipo. Foi escolhido utilizar *StatefulSets* para realizar a implantação. *StatefulSets* permitem que cada replica de um *StatefulSet* tenha um identificador de rede único, e esteja associado com recursos persistentes. A listagem 3.5.1 mostra a definição do *StatefulSet*. *StatefulSets* permitem que muitos mecanismos necessários sejam implementados apesar de fugirem do escopo deste protótipo. No próximo capítulo serão discutidas algumas características e como poderiam ser implementados.

Para testar o uso do MPS, definimos que teríamos duas réplicas desse StatetulSet (linha 10), cada um com um identificador único. Para que a aplicação seja capaz de acessar

recursos como a GPU, é necessário permitir o uso do IPC do *host* (linha 17). Na linha 23 especificamos a imagem a ser utilizada para rodar o contêiner. A imagem do contêiner é lida do Docker Hub. Para subir a imagem para o Docker Hub executamos o comando `docker push naythan/nijima:latest`.

Além do IPC do *host*, é necessário algumas permissões relacionadas ao uso da simulação de dispositivos de entrada. Então adicionamos essas capacidades na linha 36. O Sunshine utiliza várias portas para fazer a comunicação, então é necessário especificar essas portas na definição do *StatefulSet*.

Programa 3.5 StatefulSet

```
1  apiVersion: apps/v1
2  kind: StatefulSet
3  metadata:
4    name: stateful-nijima
5  spec:
6    selector:
7      matchLabels:
8        app: nijima # has to match .spec.template.metadata.labels
9    serviceName: "service-nijima"
10   replicas: 2
11   minReadySeconds: 10
12   template:
13     metadata:
14       labels:
15         app: nijima # has to match .spec.selector.matchLabels
16     spec:
17       hostIPC: true
18       securityContext:
19         runAsUser: 1000
20       terminationGracePeriodSeconds: 10
21       containers:
22       - name: makoto
23         image: "naythan/nijima:test"
24         imagePullPolicy: Always
25         livenessProbe:
26            exec:
27              command:
28              - touch
29              - /tmp/healthy
30         readinessProbe:
31           exec:
32             command:
33             - touch
34             - /tmp/healthy
35         securityContext:
36           privileged: true
37           allowPrivilegeEscalation: true
38           capabilities:
39             add: ["NET_ADMIN", "SYS_TIME", "SYS_ADMIN"]
40         ports:
41           - containerPort: 47984
```

cont ⟶

⟶ *cont*

```
42            protocol: TCP
43          - containerPort: 47985
44            protocol: TCP
45          - containerPort: 47986
46            protocol: TCP
47          - containerPort: 47987
48            protocol: TCP
49          - containerPort: 47988
50            protocol: TCP
51          - containerPort: 47989
52            protocol: TCP
53          - containerPort: 47990
54            protocol: TCP
55          - containerPort: 48010
56            protocol: TCP
57          - containerPort: 47998
58            protocol: UDP
59          - containerPort: 47999
60            protocol: UDP
61          - containerPort: 48000
62            protocol: UDP
63          - containerPort: 50000
64            protocol: TCP
65          - containerPort: 60000
66            protocol: TCP
67        resources:
68          limits:
69            nvidia.com/gpu-4gb: "1"
```

Apesar de termos especificado as portas no contêiner, não é possível identificarmos essas portas e nem o IP dos *StatefulSets* pois estes IPs são IPs da rede interna do Kubernetes. Para conseguirmos acessar o Sunshine rodando no Kubernetes, precisamos do *Service* 3.5.1. Escolhemos um *Service* do tipo *Load Balancer* (linha 16) pela simplicidade de utilizar no escopo do projeto. Por estarmos usando um balanceador de carga da AWS, precisamos adicionar algumas anotações. Algumas que valem destacar é a da linha 7 que específica que devemos direcionar o tráfego ao nível de IP, e a da linha 14 para conseguirmos acessar pela internet pública. Para os propósitos do nosso teste, além da especificação das portas, o *Session Affinity* e a configuração mais importante. Definimos nosso *Session Affinity* como clientIP na linha 19. Essa configuração nos permite que depois que a conexão com um cliente seja feita, ele continue conectado ao mesmo *StatefulSet*.

Programa 3.6 Service

```
1   apiVersion: v1
2   kind: Service
3   metadata:
4     name: service-nijima
5     annotations:
6       service.beta.kubernetes.io/aws-load-balancer-type: "external"
```

cont ⟶

⟶ *cont*

```
7       service.beta.kubernetes.io/aws-load-balancer-nlb-target-type: "ip"
8       service.beta.kubernetes.io/aws-load-balancer-healthcheck-port: "50000"
9       service.beta.kubernetes.io/aws-load-balancer-healthcheck-protocol: http
10      service.beta.kubernetes.io/aws-load-balancer-healthcheck-healthy-threshold:
            "10"
11      service.beta.kubernetes.io/aws-load-balancer-healthcheck-unhealthy-
            threshold: "10"
12      service.beta.kubernetes.io/aws-load-balancer-healthcheck-timeout: "30"
13      service.beta.kubernetes.io/aws-load-balancer-healthcheck-interval: "300"
14      service.beta.kubernetes.io/aws-load-balancer-scheme: internet-facing
15    spec:
16      type: LoadBalancer
17      selector:
18        app: nijima
19      sessionAffinity: ClientIP
20      sessionAffinityConfig:
21        clientIP:
22          timeoutSeconds: 10000
23      ports:
24        - name: port1
25          port: 47984
26          protocol: TCP
27        - name: port2
28          port: 47985
29          protocol: TCP
30        - name: port3
31          port: 47986
32          protocol: TCP
33        - name: port4
34          port: 47987
35          protocol: TCP
36        - name: port5
37          port: 47988
38          protocol: TCP
39        - name: port6
40          port: 47989
41          protocol: TCP
42        - name: port7
43          port: 47990
44          protocol: TCP
45        - name: port8
46          protocol: TCP
47          port: 48010
48        - name: port9
49          port: 47998
50          protocol: UDP
51        - name: port10
52          port: 47999
53          protocol: UDP
54        - name: port11
55          port: 48000
56          protocol: UDP
```

cont ⟶

```
    ⟶ cont
57      - name: port12
58        port: 50000
59        protocol: TCP
60      - name: port13
61        port: 60000
62        protocol: TCP
```

Além de configurar o *Service* com *SessionAffinitty*, é necessário configurar o balanceador de carga com *IP stickness* de forma que a conexão de cada usuário será sempre com os mesmos componentes, mantendo a estabilidade da conexão e por consequência mantendo a jogatina estável. O *script* 3.5.1 aplica o *IP stickness* em todos as portas do *Service*.

Programa 3.7 Script para a configuração do balanceador de carga

```
1   #!/bin/bash
2
3   # Recupera uma lista de ARNs
4   target_group_arns=$(aws elbv2 describe-target-groups --query 'TargetGroups[].
        TargetGroupArn' | grep k8s | grep -oP "[A-z0-9-:/_]*")
5
6   # Itera nos ARN
7   while IFS= read -r target_group_arn; do
8     # Habilita o ip stickness
9     aws elbv2 modify-target-group-attributes --target-group-arn "
          $target_group_arn" --attributes Key=stickiness.enabled,Value=true Key=
          preserve_client_ip.enabled,Value=true
10  done <<< "$target_group_arns"
```

3.5.2 Execução em nuvem

Testamos duas instâncias do protótipo na infraestrutura configurada. Foi notado um problema com o uso de um *Display Server* Wayland nas placas da Nvidia. Existem dois principais *drivers* para placas de vídeo da Nvidia. Um deles é de código aberto e está disponível em placas de consumo, conhecido como *driver* nouveau. O outro *driver* é proprietário e existe tanto para as placas de consumo quanto as feitas para uso profissional e de *datacenter*. Como estávamos usando uma placa de vídeo para *datacenter* estávamos limitados a utilizar o *driver* proprietário.

Em geral, os *drivers* da Nvidia não funcionam muito bem com o Wayland. Existe uma limitação fundamental no *driver* que inviabiliza o uso do *driver* para os nossos propósitos. Essa limitação se dá atualmente pela falta de uma instrução no *driver* que era necessário para o Sway e, portanto, para o protótipo. Descobrimos esse fato olhando as discussões no Github da Nvidia GNIF, 2022. Devido a esse problema, tivemos que testar o protótipo utilizando o X11.

O Dockerfile 3.8 mostra as mudanças realizadas para o uso da imagem no Kubernetes com placas da Nvidia. As mudanças no Dockerfile foram leves. Se resumem a instalar as

dependências dos *drivers* da Nvidia e do X11. Também foram removidas as variáveis de ambiente que definiam alguns comportamentos para sessões Wayland.

Programa 3.8 Dockerfile para execução em nuvem com placas Nvidia

```
1  # syntax=docker/dockerfile:1.4
2  # artifacts: true
3  # platforms: linux/amd64,linux/arm64/v8
4  # platforms_pr: linux/amd64
5  # no-cache-filters: sunshine-base,artifacts,sunshine
6
7  FROM lizardbyte/sunshine:8ff2022-ubuntu-22.04
8  ENV DEBIAN_FRONTEND=noninteractive
9  USER root
10 RUN apt-get update && apt-get install -y astromenace x11-apps mesa-utils libxrender1
       libvulkan1 libxrender-dev \
11     libva-drm2 libva-x11-2 va-driver-all libgbm1 libgles2-mesa libegl1 libgl1-mesa-dri
           libvulkan1 libvdpau1 libnuma1 \
12     libavahi-client3 libgles2-mesa mesa-vulkan-drivers mesa-vdpau-drivers libgles2-
           mesa-dev mesa-utils-extra libappindicator3-1 libopus0 libmfx1 vainfo apt-file
           libboost-chrono1.74.0 udev \
13     libcap2-bin libxres-dev libxres1 libsdl2-2.0-0 libgles2 pciutils kmod mesa-va-drivers
           \
14     pipewire pipewire-pulse wget sudo make pulseaudio pipewire-audio-client-libraries
           libspa-0.2-dev libpipewire-0.3-dev gstreamer1.0-pipewire dbus-x11 bash-
           completion && apt-file update
15
16 RUN mkdir -p /src && \
17     cd /src && \
18     wget http://repo.steampowered.com/steam/archive/precise/steam_latest.tar.gz && \
19     tar xzvf steam_latest.tar.gz && \
20     cd /src/steam-launcher && \
21     make install && echo "========== Setup sudoers ==========" && \
22     echo "lizard ALL=(ALL:ALL) NOPASSWD: ALL" >> /etc/sudoers && \
23     echo "========== Install Steam =========="
24
25 RUN echo 'root:123' | chpasswd
26 RUN setcap cap_sys_admin+p $(readlink -f $(which sunshine))
27 RUN mkdir -p /run/dbus
28 RUN mkdir -p /dev/snd
29 RUN mkdir -p /run/user/1000 && chown lizard /run/user/1000
30
31 RUN apt-get install -y wireplumber alsa-utils curl gnupg2 libcudart11.0 nvidia-cuda-dev
       nvidia-profiler
32
33 RUN distribution=$(. /etc/os-release;echo $ID$VERSION_ID) &&\
34     echo "$distribution" &&\
35     curl -s -L https://nvidia.github.io/libnvidia-container/gpgkey | apt-key add - &&\
```

cont ⟶

```
    ⟶ cont
36      curl -s -L https://nvidia.github.io/libnvidia-container/$distribution/libnvidia-
            container.list | tee /etc/apt/sources.list.d/libnvidia-container.list &&\
37      apt-get update && apt-get install -y nvidia-container-toolkit
38
39  RUN apt-get install -y libdrm2 xserver-xorg-core xserver-xorg-video-nvidia-535
        libnvidia-fbc1-535 @\label{docker:nvidia}@ libnvidia-gl-535 nvidia-utils-535 xcvt
        xfonts-base x11-apps \
40      x11-session-utils x11-utils x11-xfs-utils \
41      x11-xserver-utils xauth x11-common \
42      xz-utils unzip avahi-utils dbus \
43      mesa-utils mesa-utils-extra \
44      vulkan-tools \
45      libgl1-mesa-glx libgl1-mesa-dri libglu1-mesa \
46      xserver-xorg-video-all \
47      xserver-xorg-input-libinput \
48      jwm libxft2 libxext6 breeze-cursor-theme libnvidia-decode-535 libnvidia-encode-535
            vim
49
50  USER lizard
51
52  ENV DISABLE_RTKIT=y
53  ENV PIPEWIRE_RUNTIME_DIR=/run/user/1000
54  ENV PULSE_RUNTIME_DIR=/run/user/1000
55  ENV XDG_SESSION_TYPE=x11
56  ENV DISPLAY=:1
57
58  COPY setup.sh $HOME/setup.sh
59  COPY sunshine/ /home/lizard/.config/sunshine
60
61  ENTRYPOINT /home/lizard/setup.sh
```

Além das mudanças no Dockerfile, tivemos que mudar o *script* de inicialização. As mudanças foram poucas, mantendo a mesma instrução. Tivemos que utilizar o comando da linha 6 para que a configuração do Xorg (a principal implementação do protocolo X11) fosse feita para a placa da Nvidia que estávamos utilizando. Foram removidos os comandos que estavam associados ao início da sessão Wayland (Sway) e adicionado o comando da sessão X11 na linha 16.

Programa 3.9 *Script* de inicialização para execução na AWS em X11 usando placas de vídeo NVidia

```
1  #!/bin/bash
2
3  echo "Setting up init"
4
5  echo "setting correct video card"
```

cont ⟶

⟶ *cont*

```
6   sudo nvidia-xconfig
7   echo "initting udevd"
8   XDG_RUNTIME_DIR=/tmp sudo /lib/systemd/systemd-udevd &
9   sleep 5
10
11  echo "starting dbus"
12  XDG_RUNTIME_DIR=/tmp sudo dbus-daemon --system &
13  XDG_RUNTIME_DIR=/run/user/1000 dbus-daemon --session &
14
15  echo "starting xorg"
16  sudo XDG_RUNTIME_DIR=/tmp/.X11-unix Xorg -ac -noreset +extension GLX +
        extension RANDR +extension RENDER vt1 "${DISPLAY}" &
17
18  sleep 5
19
20  echo "starting pipewire"
21  XDG_RUNTIME_DIR=/run/user/1000 pipewire &
22  sleep 5
23  XDG_RUNTIME_DIR=/run/user/1000 pipewire-pulse &
24  sleep 5
25  XDG_RUNTIME_DIR=/run/user/1000 wireplumber &
26
27  sleep 5
28  pw-loopback -m '[ FL FR]' --capture-props='media.class=Audio/Sink node.name=
        sink-sunshine-stereo' &
29
30  echo "starting sunshine"
31  sleep 5
32  XDG_RUNTIME_DIR=/run/user/1000 sunshine /home/lizard/.config/sunshine/
        sunshine.conf &
33
34  echo "start pod healthcheck"
35  sleep 5
36  python3 -m http.server 50000 &
37
38  # echo "starting game"
39  sleep 5
40  XDG_RUNTIME_DIR=/run/user/1000 /usr/games/AstroMenace
```

Com a mudança do Wayland para o X11 conseguimos rodar o protótipo na nuvem. No entanto, tivemos um efeito colateral dessa mudança. O Pipewire foi projetado para funcionar em sistemas Wayland. Embora algum nível de compatibilidade exista com o X11, não conseguimos fazer funcionar facilmente dentro do contêiner. Dessa forma testamos sem o áudio nessa parte.

A ausência de áudio nessa parte do protótipo não traz muito impacto por diversos motivos. Dado um sistema Wayland, mostramos que é possível fazer o áudio funcionar. Áudio no Pipewire costuma ser bastante flexível com diversos recursos para utilizar *Sinks* e *Sources* virtuais.

Outro motivo pelo qual a ausência do áudio não tem impacto nessa parte do teste é o

fato que a parte de áudio é muito mais leve que a parte gráfica, permitindo que estressemos a abordagem utilizada. Apenas a parte gráfica é processada na placa de vídeo, de forma que a abordagem utilizada não impacta no áudio. Certamente seria possível nesse cenário fazer o áudio voltar a funcionar utilizando PulseAudio ou Pipewire.

Para testar o funcionamento do protótipo iniciamos duas instâncias e conectamos a instâncias diferentes do mesmo jogo utilizando dispositivos diferentes, em redes diferentes. Nos conectamos à primeira instância utilizando o mesmo Notebook que utilizamos nos testes locais. O resultado foi bastante semelhante ao teste local. Por termos iniciado o nosso *cluster* de Kubernetes em uma região AWS geograficamente próxima do local do teste, não sentimos efeito de latência perceptível neste jogo. Conseguimos controlar e disparar os tiros da nave sem problemas utilizando um controle de Xbox, e a qualidade gráfica foi bastante semelhante ao local.

Após testarmos com uma única instância conectada, conectamos outra instância a partir de um celular Android rodando o Moonlight. O jogo conectou com sucesso, porém o protótipo não foi configurado para funcionar com entrada via toque de tela, pois fugiria do escopo. Após conectarmos com o celular usando a rede 4G, voltamos a instância conectada no Moonlight do notebook e testamos novamente. Não percebemos nenhuma diferença visual ao jogarmos, apesar de estar com duas instâncias rodando na mesma GPU.

Capítulo 4

Conclusão

4.1 Possibilidades

Existem diversos problemas e desafios para se fazer um serviço de *cloud gaming* funcional. Apesar do protótipo construído funcionar e conseguir solucionar a parte mais difícil do problema, existem outras questões que não foram abordadas no protótipo por fugirem do escopo. Iremos discutir algumas soluções para problemas comuns, mas que não foram implementadas por fugirem do escopo do projeto e já serem soluções conhecidas para outros problemas.

4.1.1 Plataforma

No protótipo, fizemos os testes em nuvem de forma manual. Para nos conectarmos precisamos descobrir o IP do balanceador de carga (`kubectl get nodes`) e colocar esse IP no Moonlight para podermos fechar a conexão. Em um serviço real de *cloud gaming* isso não poderia acontecer por diversos motivos. Não seria possível descobrir o IP de componentes da arquitetura, pois os usuários não teriam esse acesso e não deveriam precisar entender como funciona o serviço. A parte de colocar o IP manualmente não seria uma falha de segurança e nem algo que necessitaria de entendimento da parte do usuário, mas também é possível evitar. A figura 4.1 mostra como poderia ser uma arquitetura completa de um serviço de *cloud gaming* baseado em Kubernetes.

A ideia principal da arquitetura seria a criação dinâmica dos objetos do Kubernetes utilizando a sua API. O usuário se conectaria a uma aplicação web por meio de um cliente customizado. Ele logaria na sua conta trazendo suas informações e escolheria qual jogo ele gostaria de jogar. Isso abriria um *pop-up* que abriria o Moonlight já com o IP necessário para estabelecer a conexão com o contêiner que rodaria o jogo dele.

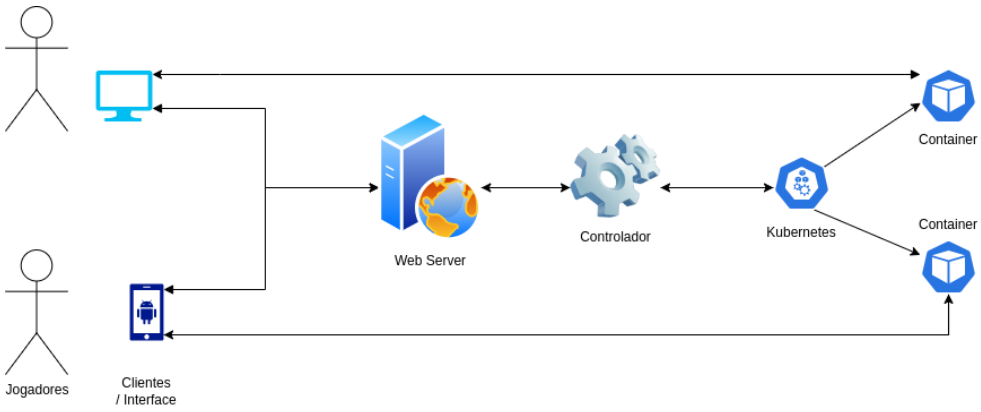

Figura 4.1: *Possível arquitetura de alto nível de uma possível solução completa de cloud gaming. Fonte: Própria*

4.1.2 Funcionalidades

No protótipo construído, instalamos o jogo diretamente na imagem, pois estava disponível nos repositórios do Ubuntu. Normalmente não será possível fazer desta forma e não poderíamos instalar todos os jogos em única imagem. Como uma pessoa só joga um jogo de cada vez, o ideal seria que cada imagem tivesse apenas um jogo para otimizar o uso de espaço e tempo de carregamento e construção da imagem. A maneira mais simples seria copiar o jogo para dentro da imagem durante o processo de construção dela.

Outro ponto a se levar em consideração seria o gerenciamento de sessão. O usuário precisa ser capaz de voltar a jogar o jogo dele de onde ele parou. Para tal, existem diversas opções. Uma opção mais simples seria aproveitar o uso dos sistemas de sincronização existentes como o da Steam. Dessa forma, apenas seria instanciado o jogo e feita a sincronização após o início. Quando essa opção não está disponível, é possível adotar outras abordagens. Como adotamos o uso de StatefulSets, as instâncias têm um identificador único de rede. Dessa maneira, é possível utilizar volumes persistentes e mapear eles nos StatefulSets. Seria armazenado apenas a parte variável do jogo, usualmente o progresso do jogador, para otimizar o uso do espaço. Outra abordagem que pode ser utilizada é se aproveitar da flexibilidade dos contêineres. Seria possível simplesmente pausar o estado do contêiner e guardar o estado dele para posterior execução. Isso traz as vantagens de poder iniciar o jogo mais rapidamente, se recuperar de perdas de conexão mais facilmente e poder pausar o jogo a qualquer momento. No entanto, essa abordagem consome muito mais memória de armazenamento, pois todo o contexto de execução do jogo será salvo ao invés de apenas os arquivos de progresso, além de não permitir a portabilidade do progresso do jogador para outras plataformas.

4.2 Discussão

O protótipo desenvolvido conseguiu atingir o objetivo idealizado para este projeto. Foi possível jogar na nuvem com uma baixa latência e qualidade gráfica usuais, com mais de uma instância rodando na mesma GPU. A arquitetura usada no protótipo é altamente flexível, pois está em cima do Kubernetes que é considerado o sistema operacional da

nuvem. Além disso, foi proposto ideias de como expandir o protótipo em um serviço completo, levantando pontos que ainda precisam ser tratados.

4.2.1 Limitações

Existem algumas limitações do protótipo. Apesar do uso do áudio com o Pipewire em ambiente local ter funcionado corretamente, devido às limitações das placas da Nvidia em ambientes Wayland não conseguimos utilizá-lo na nuvem sem fazer adaptações. Conforme as tecnologias fazem a transição do X para o Wayland, a tendência é que o suporte aumente. Isso, porém, não invalida a demonstração do funcionamento, pois apesar de ser local, o teste local também foi conteinerizado. Foi tentado utilizar placas da AMD na nuvem, para demonstrar o áudio, porém as máquinas não subiram por motivos desconhecidos.

Outra limitação foi o uso de balanceadores de carga, uma tecnologia poderosa para resiliência e escalabilidade de sistemas, mas que não é a ferramenta mais adequada para direcionar tráfego a instâncias específicas. Depois da finalização do protótipo, foi lançado a primeira versão do *Gateway API* para o Kubernetes. Essa ferramenta poderia ser mais adequada para o roteamento de tráfego para objetos específicos, mas ainda precisaria ser analisada. A própria AWS possui serviços de gerenciamento de tráfego como o *API Gateway*, que não está relacionado com o Kubernetes apesar do nome semelhante.

A natureza dos testes realizados foram apenas qualitativos. Não foram feitos testes de estresse no sistema e nem medição quantitativa de desempenho. Existem vários testes que precisariam ser feitos para uma análise de desempenho mais rigorosa. O primeiro é de escalabilidade. Utilizamos apenas duas instâncias em uma GPU que aguenta muito mais, pois devido às limitações do uso de um balanceador de carga seria difícil conectar novas instâncias a partir do mesmo lugar. Dessa forma surgem várias perguntas relacionadas ao uso de várias instâncias.

- Conforme a quantidade de instâncias aumenta, é possível notar degradação de desempenho?
- Esse aumento é linear?
- Em que momento podemos perceber que a GPU está saturada e o desempenho degrada?
- É com 100% do uso de memória ou começa a travar antes?
- O que acontece com jogos mais exigentes graficamente?
- O desempenho é consistente ou oscila?

Além disso, é importante medir as taxas de FPS de diferentes jogos, com diversas quantidades de instância em paralelo para conseguir responder esse tipo de pergunta. Existem também outras perguntas importantes não relacionadas ao desempenho do jogo, mas que pode ser um gargalo no sistema ou atrapalhar a experiência do usuário.

- É fácil escolher um jogo para jogar?
- Os jogos iniciam rápido?

- Existe risco de perda de dados caso aconteça algum travamento no jogo ou perda de conexão do usuário?
- É mais agradável jogar na nuvem ou local?
- É fácil adicionar novos jogos no sistema e fazer melhorias no sistema sem causar interrupções?

Por conta do protótipo ser mais focado em um teste fim a fim, essas perguntas precisariam ser endereçadas em uma segunda versão do protótipo.

Apêndice A

Tecnologias

Será dada uma descrição breve de algumas tecnologias mais específicas utilizadas neste projeto para ajudar na compreensão sem sobrecarregar o texto.

A.1 Kubernetes

Kubernetes é o mais popular orquestrador de contêineres do mundo. Foi solto pelo Google em janeiro de 2015 como um projeto de código aberto. O Google trabalhou com a *Linux Foundation* para criar a CNCF (*Cloud Native Computing Foundation*) e ofereceu o Kubernetes para ser o projeto central da organização.

Kubernetes é bastante complexo e dominar o seu uso requer o conhecimento de diversos conceitos em redes, Linux e DevOps. A base do Kubernetes está centrada em contêineres. Um contêiner é semelhante a uma máquina virtual. A principal diferença entre uma máquina virtual e um contêiner é o fato que o contêiner não simula os componentes físicos do computador, ele utiliza os componentes físicos da máquina hospedeira e opera apenas no nível do sistema operacional. Dessa forma, contêineres conseguem ser muito mais leves que máquinas virtuais e conseguem isolar completamente quaisquer dependências que uma aplicação possa ter.

O tipo de contêiner mais popular atualmente são contêineres Docker. Um contêiner Docker é criado a partir de uma imagem Docker. Uma imagem Docker é criada a partir de um arquivo chamado Dockerfike. No Dockerfile é necessário especificar obrigatoriamente uma imagem base (pode ser um sistema operacional Linux ou outra imagem Docker) e um comando para rodar ao iniciar o contêiner. Construímos o contêiner rodando o comando Docker build. E rodamos a imagem usando Docker run. Devido à popularidade do Docker, outros motores de contêineres como o Podman e o containerd utilizam imagens construídas no padrão Docker para execução.

Contêineres por padrão são efêmeros, o que significa que eles não guardam nenhum estado depois que eles terminam a sua execução. Para conseguir persistência é necessário montar um volume persistente. Apesar de volumes persistentes conseguirem guardar estado de arquivos, essa abordagem não é suficiente para garantir resiliência e tolerância a

falhas, além da natureza imperativa de se definir volumes e abrir portas usando *flags* no comando de rodar o contêiner.

A.1.1 Recursos

Em Kubernetes existem vários níveis de abstração em cima de contêineres. O recurso mais básico do Kubernetes é o *Pod*. O *Pod* é a unidade escalonável mais simples do Kubernetes. A grande maioria dos recursos do Kubernetes são definidos em arquivos Yaml. Para definir um *Pod* é necessário apontar quais imagens serão instanciadas em contêineres dentro do *Pod*. Um *Pod* suporta executar vários contêineres ao mesmo tempo. As boas práticas de Kubernetes recomendam que o número de contêineres em um *Pod* seja o menor possível. Na definição do *Pod* é possível especificar as portas de rede que ele vai utilizar assim como os volumes e volumes persistentes.

Assim como contêineres, *Pods* também são efêmeros. Depois que eles terminam a execução ou morrem, é necessário criar outro para substituir. Existem recursos no Kubernetes que são capazes de criar *Pods* automaticamente. Alguns deles são os *Deployments* e os *StatefulSets*. Os *Deployments* são capazes de definir um conjunto de réplicas de *Pods*. Sempre que o número de *Pods* for diferente do número de réplicas especificado, o *Deployment* vai escalonar para manter o número de *Pods* ideal. Os *StatefulSets* são semelhantes aos *Deployments* com a diferença que cada réplica tem um identificador único e mantém o estado de recursos atrelados a ele como volumes.

Outro tipo de recurso bastante utilizado são os *Services*. Devido à natureza móvel de vários objetos do Kubernetes, os IPs internos costumam mudar com frequência. Para conectar componentes entre si dentro do Kubernetes e para acessá-los de fora do *cluster* é necessário utilizar um *Service*. Existem dois tipos principais de *Service*, internos e externos. Um *Service* externo vai externalizar um método para acessar um componente de fora do *cluster*, a depender de suas configurações específicas, *addons* e *plugins* do *cluster*. Já um interno é utilizado para conectar componentes dentro do *cluster*. Ao invés de colocar um endereço de um componente que você gostaria de conectar, você utiliza o nome do *Service*.

Para as demais informações, consultar a documentação oficial do Kubernetes em *https://kubernetes.io/docs/*

A.2 NVidia

A Nvidia é líder no segmento de fabricação de placas de vídeo de consumo e *datacenter*. Embora a AMD venha crescendo no segmento nos últimos anos, as tecnologias que a Nvidia disponibiliza são mais robustas e possuem mais impacto em diversas áreas como jogos e inteligência artificial. Escolhemos utilizar a Nvidia como placa principal de teste na nuvem devido ao seu suporte para Kubernetes, engajamento da comunidade e alguns outros recursos.

A.2.1 Compartilhamento de recursos

Existem três principais formas de se compartilhar recursos entre diversos contêineres com GPUs da Nvidia. Todas as abordagens listadas aqui funcionam tanto no contexto de Kubernetes quanto localmente dado a disponibilidade de uma máquina com uma GPU suportada. Vamos adotar a terminologia utilizada em Kubernetes, que é o foco do estudo. O Kubernetes adota a terminologia de *resources*. É comum especificar os requisitos e limites de memória e CPU que um contêiner deve e pode utilizar. Com a GPU é semelhante, mas varia conforme a forma de compartilhar os recursos. A GPU tem dois recursos principais, os núcleos de processamento e a memória de vídeo.

Time-Slicing

A primeira forma de compartilhar é o *time-slicing* Nvidia, 2022. Esse mecanismo compartilha a GPU inteira em intervalos de tempo. O tempo que cada contêiner tem para rodar na GPU é dividido igualmente. Essa forma compartilha tanto a memória quanto o processamento da GPU. Conforme aumenta a quantidade de contêineres que solicitam a mesma GPU, alguns problemas começam a surgir. O principal problema é a troca de contexto excessiva na GPU. Cada vez que um contêiner vai executar a instruções na placa de vídeo, todo o processamento anterior precisa ser pausado, colocado em uma memória externa, inserido o contexto do novo contêiner e executado. Apesar da semelhança com o escalonamento de processos de um sistema operacional, essa operação se torna mais custosa por alguns motivos. Um processador moderno possui vários núcleos e pode executar tarefas em paralelo. Além disso, a memória principal utilizada não fica dentro do processador, apenas as memórias caches, então a troca de contexto não é tão custosa. Outro ponto a ser considerado é a possibilidade de definir prioridades ao nível de processo, de forma que alguns processos sejam dados prioridade de execução. Apesar da GPU ter vários núcleos e ser altamente paralelizável, a técnica de *time-slicing* não permite que isso seja aproveitado. Outro problema dessa abordagem é que um contêiner, pode não saturar completamente uma GPU, apesar de ter o uso dedicado pelo tempo que executar, não aproveitando os recursos disponíveis.

A técnica de *time-slicing* é particularmente ruim no contexto de jogos. A troca de contexto pode causar latência e travamentos indesejados, pois são aplicações que executam em tempo real e não foram preparadas para terem trocas de contexto frequentes. Além disso, diferentes jogos requerem uma quantidade de recursos muito diferentes um do outro, de forma que o uso de recursos em uma estratégia de *time-slicing* é mal utilizado.

MIG

Uma técnica mais eficaz que o *time-slicing* para a divisão de recursos em uma GPU é o MIG (Multi Instance GPU). O MIG funciona criando uma divisão física dos recursos, o que permite o isolamento completo dos processos rodando na GPU e os seus recursos de processamento e memória. O MIG está disponível apenas para as GPUs focadas em *datacenters* a partir da arquitetura Ampere.

A figura A.1 mostra as possíveis configurações de divisão que podem ser atingidas com uma GPU Nvidia A100. Isso permite que usando uma única GPU, seja possível rodar um

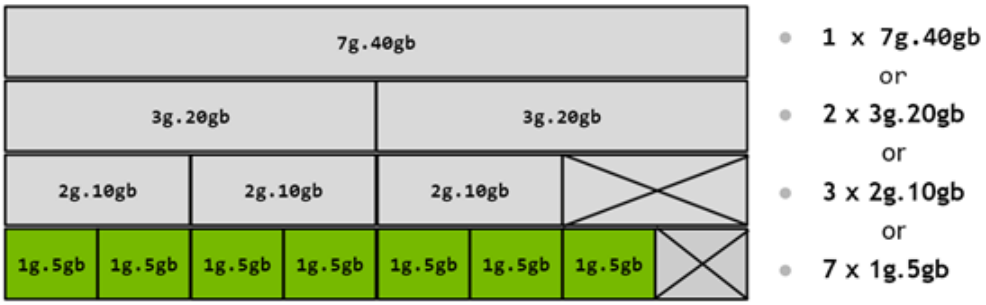

Figura A.1: *Maneiras diferentes de dividir uma GPU A100 utilizando a tecnologia MIG. Fonte: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html*

contêiner com uma unidade de computação e 5 Gb de memória, enquanto outro contêiner utiliza 2 unidades de computação e 10 Gb de memória, da mesma GPU.

Apesar dos benefícios do uso do MIG, ele traz algumas desvantagens. Algumas divisões podem gerar desperdício ao não utilizar toda a memória disponível. Qualquer esquema de partição que inclua 2 unidades de computação ou menos, irá gerar algum desperdício. Embora o MIG aborde o problema de compartilhar recursos computacionais com outros contêineres, ele traz uma granularidade bastante baixa, podendo compartilhar recursos com apenas 7 contêineres. A divisão mais granular vêm com 5 Gb de memória de vídeo, que pode ser mais que o suficiente para jogos mais leves, também gerando desperdícios. Outro problema grave é a falta de suporte para APIs gráficas, que pode comprometer fortemente a performance do jogo.

MPS

Utilizar o MPS (*Multi-Process Service*) é outra maneira de compartilhar recursos entre diversos contêineres. Essa tecnologia está disponível desde a arquitetura Volta da Nvidia, sendo mais antigo que o MIG. Ele funciona executando processos paralelamente na GPU com um nível mediano de isolamento entre os processos, diferente do MIG que possuía isolamento completo dos processos. A granularidade permitida pelo MPS é muito maior que a do MIG, permitindo que até 48 contêineres possam ser executados na mesma GPU. É possível estabelecer limites de recursos para cada contêiner, especificando a quantidade de memória que cada contêiner pode utilizar. Isso permite bastante flexibilidade em aproveitar os recursos da GPU.

Para os nossos requisitos, o MPS é de longe a melhor opção para o compartilhamento de recursos com diversos contêineres. Apesar do isolamento mais fraco que o MIG, isso não interfere na execução de processos, apenas no tratamento de erros mais críticos, que nos experimentos realizados não pôde ser observado. O MPS é o mais granular das 3 opções, tem suporte a APIs gráficas e está disponível em mais GPUs que o MIG.

A.2.2 Kubernetes

A Nvidia possui tecnologias específicas para trabalhar com Kubernetes. A principal delas é o GPU Operator e seus componentes. O GPU Operator é um conjunto de *DaemonSets* que são instalados em cada um dos nós de um *cluster*. *DaemonSets* são semelhantes a recursos como *Deployments* e *StatefulSets* mas criam os *Pods* ao nível do nó ao invés de ser apenas uma abstração. Os *DaemonSets* presentes no GPU Operator implementam diversas funcionalidades. O Nvidia Device Plugin permite que seja especificado uma GPU como *Resource* assim como é feito com *Resources* tradicionais como memória. Além disso, um dos *DaemonSets* instala o Nvidia Container Toolkit, responsável por implementar o uso da GPU em contêineres. Outro *DaemonSet* é responsável por instalar os *drivers* da placa específica que está sendo utilizada. Para utilizar a tecnologia MIG de compartilhamento de GPU entre contêineres, é necessário habilitar o *DaemonSet* que instala o MigManager.

Além do GPU Operator, existem outros componentes no Kubernetes que lidam com GPUs. Devido a uma limitação do Nvidia Device Plugin do GPU Operator, a Nebuly criou um *fork* dele. A limitação consiste no fato que só é possível utilizar o MIG através do Kubernetes. Utilizando o *fork* da Nebuly, podemos também utilizar o MPS, otimizando o uso de recursos.

A Nebuly também criou o Nos. Ele serve para o provisionamento automático de recursos de GPU. No uso normal do GPU Operator você precisa configurar manualmente os *slices* de GPU que você vai utilizar. Com o Nos, ele consegue utilizar as APIs da Nvidia e do Kubernetes para fazer o provisionamento automático dos *slices* no caso do MIG e da quantidade de memória disponível para aquele *Pod* no caso do MPS. Ele faz esse cálculo baseado na quantidade de recursos pedidos na definição do *Pod* ou na definição de um recurso que cria *Pods* como *StatefulSets*.

Referências

[Alpert 1992] Mark Alpert. *CD-ROM: THE NEXT PC REVOLUTION CD-ROMs – compact disks for personal computers – are finally coming on strong. Now you can use them to choose a hotel, track down a patent, or teach your kids to read.* 1992. URL: https://web.archive.org/web/20220808165741/https://archive.fortune.com/magazines/fortune/fortune_archive/1992/06/29/76592/index.htm (acesso em 02/12/2023) (citado na pg. 4).

[Arc 2023] Arc. *How Long Does it Take to Make a Video Game?* 2023. URL: https://arc.academy/how-long-does-it-take-to-make-a-video-game/ (acesso em 03/12/2023) (citado na pg. 13).

[ATT 2023] ATT. *VNC Frequently Asked Questions (FAQ).* 2023. URL: https://web.archive.org/web/20000815062637/http://www.uk.research.att.com/vnc/faq.html (acesso em 03/12/2023) (citado na pg. 9).

[EISLER 2021] PHIL EISLER. *GeForce NOW Gets New Priority Memberships and More.* 2021. URL: https://blogs.nvidia.com/blog/geforce-now-priority/ (acesso em 03/12/2023) (citado na pg. 16).

[Fleming 2007] Jeffrey Fleming. *The History Of Activision.* 2007. URL: https://web.archive.org/web/20161220122651/http://www.gamasutra.com/view/feature/1537/the_history_of_activision.php?print=1 (acesso em 02/12/2023) (citado na pg. 4).

[gamesindustry.biz 2009] gamesindustry.biz. *Cevat Yerli discusses the next round of home consoles and staying ahead of the hardware manufacturers with CryEngine 3.* 2009. URL: https://www.gamesindustry.biz/crysis-core (acesso em 27/09/2023) (citado na pg. 6).

[gnif 2022] gnif. *True DMABUF support.* 2022. URL: https://github.com/NVIDIA/open-gpu-kernel-modules/discussions/243 (acesso em 03/12/2023) (citado na pg. 36).

[Hollister 2019] Sean Hollister. *How Sony bought, and squandered, the future of gaming.* 2019. URL: https://www.theverge.com/2019/12/5/20993828/sony-playstation-now-cloud-gaming-gaikai-onlive-google-stadia-25th-anniversary (acesso em 02/12/2023) (citado na pg. 7).

[Kevuru 2023] Kevuru. *AAA Games - Everything You Need to Know About the Budget*. 2023. URL: https://kevurugames.com/blog/why-are-aaa-games-so-expensive-everything-you-need-to-know-about-the-budget/ (acesso em 03/12/2023) (citado na pg. 13).

[Leiserson *et al.* 2020] Charles E Leiserson *et al.* "There's plenty of room at the top: what will drive computer performance after moore's law?" *Science* 368.6495 (2020), eaam9744 (citado na pg. 5).

[Mangalindan 2020] JP Mangalindan. *Cloud gaming's history of false starts and promising reboots*. 2020. URL: https://www.polygon.com/features/2020/10/15/21499273/cloud-gaming-history-onlive-stadia-google (acesso em 02/12/2023) (citado na pg. 7).

[Nebuly 2023] Nebuly. *Overview*. 2023. URL: https://github.com/nebuly-ai/nos/blob/main/docs/en/docs/dynamic-gpu-partitioning/overview.md (acesso em 03/12/2023) (citado na pg. 31).

[Nelius 2019] Joanna Nelius. *Here's how Stadia's input lag compares to native PC gaming*. 2019. URL: https://www.pcgamer.com/google-stadias-specs-and-latency-revealed/ (acesso em 02/12/2023) (citado na pg. 7).

[Nvidia 2022] Nvidia. *Time-Slicing GPUs in Kubernetes*. 2022. URL: https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/23.9.0/gpu-sharing.html (acesso em 03/12/2023) (citado na pg. 47).

[Nvidia 2023a] Nvidia. *Docs*. 2023. URL: https://docs.nvidia.com/deploy/mps/index.html (acesso em 03/12/2023) (citado na pg. 28).

[Nvidia 2023b] Nvidia. *Getting Started*. 2023. URL: https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/getting-started.html (acesso em 03/12/2023) (citado na pg. 30).

[Ojala e Tyrvainen 2011] Arto Ojala e Pasi Tyrvainen. "Developing cloud business models: a case study on cloud gaming". *IEEE software* 28.4 (2011), pp. 42–47 (citado na pg. 6).

[ORLAND 2015] KYLE ORLAND. *OnLive shuts down streaming games service, sells patents to Sony*. 2015. URL: https://arstechnica.com/gaming/2015/04/onlive-shuts-down-streaming-games-service-sells-patents-to-sony-embargoed-7pm-eastern/ (acesso em 29/09/2023) (citado na pg. 6).

[PURDY 2023] KEVIN PURDY. *Nvidia's GameStream is dead. Sunshine and Moonlight are great replacements*. 2023. URL: https://arstechnica.com/gaming/2023/04/nvidias-gamestream-is-dead-sunshine-and-moonlight-are-better-replacements/ (acesso em 03/12/2023) (citado na pg. 10).

[Research 2023] Precedence Research. *Video Game Market (By Type: Online, Offline; By Platform: Computer, Console, Mobile; By Business Model: Free-to-play, Pay-to-play, Play-to-earn) - Global Industry Analysis, Size, Share, Growth, Trends, Regional Outlook, and Forecast 2023-2032*. 2023. URL: https://www.precedenceresearch.com/video-game-market#:~:text=The%20global%20video%20game%20market,forecast%20period%202023%20to%202032.&text=Key%20Takeaways%3A,of%20around%2042%25%20in%202022. (acesso em 03/12/2023) (citado na pg. 13).

[Ricker 2009] Thomas Ricker. *OnLive killed the game console star?* 2009. URL: https://www.engadget.com/2009-03-24-onlive-killed-the-game-console-star.html (acesso em 29/09/2023) (citado na pg. 6).

[Rogers e Larsen 1984] Everett M Rogers e Judith K Larsen. *Silicon Valley fever: Growth of high-technology culture*. Vol. 112. Basic books New York, 1984 (citado na pg. 4).

[Russell *et al.* 1962] Steve Russell, Martin Graetz e Wayne Witaenem. "Spacewar". *Computer software* (1962) (citado na pg. 3).

[Shalf 2020] John Shalf. "The future of computing beyond moore's law". *Philosophical Transactions of the Royal Society A* 378.2166 (2020), p. 20190061 (citado na pg. 5).

[Statt 2020] Nick Statt. *Bethesda follows Activision in pulling games from Nvidia's GeForce Now*. 2020. URL: https://www.theverge.com/2020/2/21/21147638/nvidia-geforce-now-bethesda-pulling-games-activision-blizzard-cloud-gaming (acesso em 03/12/2023) (citado na pg. 15).

[Wang *et al.* 2023] Jialin Wang *et al.* "Effect of frame rate on user experience, performance, and simulator sickness in virtual reality". *IEEE Transactions on Visualization and Computer Graphics* 29.5 (2023), pp. 2478–2488 (citado na pg. 15).

[Wild 2020] Mike Wild. *Project xCloud Server Blade Analysis*. 2020. URL: https://fragwire.com/project-xcloud-server-blade-analysis/ (acesso em 02/12/2023) (citado na pg. 8).

[Wu *et al.* 2023] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu e Xin Jin. "Transparent {gpu} sharing in container clouds for deep learning workloads". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 2023, pp. 69–85 (citado na pg. 10).

[Yadav e Annappa 2017] Himanshu Yadav e B Annappa. "Adaptive gpu resource scheduling on virtualized servers in cloud gaming". In: *2017 Conference on Information and Communication Technology (CICT)*. 2017, pp. 1–6. DOI: 10.1109/INFOCOMTECH.2017.8340641 (citado nas pgs. 10, 16).

Índice remissivo