UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Rafael Vales Bettker

USO DE ADAPTADORES PARA FACILITAR A DISTRIBUIÇÃO DE GAME ENGINE EM SIMULAÇÃO DISTRIBUÍDA

Santa Maria, RS
2021

Rafael Vales Bettker

USO DE ADAPTADORES PARA FACILITAR A DISTRIBUIÇÃO DE GAME ENGINE EM SIMULAÇÃO DISTRIBUÍDA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação, Área de Concentração em Sistemas Distribuídos, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação**. Defesa realizada por videoconferência.

ORIENTADOR: Prof. Joaquim Vinícius Carvalho Assunção

COORIENTADOR: Prof. Raul Ceretta Nunes

490
Santa Maria, RS
2021

End. Eletr.: rvales@inf.ufsm.br

Rafael Vales Bettker

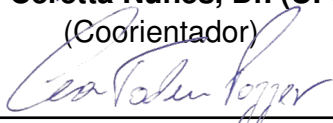
USO DE ADAPTADORES PARA FACILITAR A DISTRIBUIÇÃO DE GAME ENGINE EM SIMULAÇÃO DISTRIBUÍDA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação, Área de Concentração em Sistemas Distribuídos, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação**.

Aprovado em 11 de fevereiro de 2021:

Joaquim Vinícius Carvalho Assunção, Dr. (UFSM)
(Presidente/Orientador)

Raul Ceretta Nunes, Dr. (UFSM)
(Coorientador)



Cesar Tadeu Pozzer, Dr. (UFSM)

Carlos Raniery Paula dos Santos, Dr. (UFSM)

Santa Maria, RS
2021

AGRADECIMENTOS

À Deus, pela saúde e força para superar as dificuldades.

À minha mãe, Fátima Antunes Vales, pelo apoio e suporte durante toda a minha caminhada até aqui, me ajudando a me tornar quem eu sou hoje.

Ao meu pai, Roberto dos Reis Bettker, que sempre me apoiou e torceu por mim, e sei que continua torcendo.

À minha família, pela confiança e apoio nos mais diversos momentos.

Aos meus orientadores, Joaquim Vinícius Carvalho Assunção e Raul Ceretta Nunes, pela orientação e apoio durante o desenvolvimento deste trabalho.

Aos professores Carlos Raniery Paula dos Santos e Cesar Tadeu Pozzer, por terem aceitado participar da banca e disponibilizado seu tempo para ler, avaliar e dar sugestões para esse trabalho.

Ao projeto SIS-ASTROS GMF, pela oportunidade de desenvolver esse trabalho.

Ao professor Raul Ceretta Nunes, pela confiança e oportunidade de crescer profissionalmente.

Ao professor Joaquim Vinícius Carvalho Assunção, pelas atividades de iniciação científica que trouxeram conhecimentos de grande importância para a sequência da minha vida acadêmica.

À Universidade Federal de Santa Maria, ao curso de Ciência da Computação, e a todos professores e colegas que de alguma maneira fizeram parte dessa caminhada.

A dor é passageira, mas a glória é eterna.

(Pearl Harbor)

RESUMO

USO DE ADAPTADORES PARA FACILITAR A DISTRIBUIÇÃO DE GAME ENGINE EM SIMULAÇÃO DISTRIBUÍDA

AUTOR: Rafael Vales Bettker
ORIENTADOR: Joaquim Vinícius Carvalho Assunção
COORIENTADOR: Raul Ceretta Nunes

Os simuladores, assim como os jogos em geral, são construídos com a ajuda de game engines. Estas desempenham um importante papel em prover um conjunto de funcionalidades comuns à todos sistemas de simulação. Porém, ao fazer o projeto de um simulador deve-se ter atenção com a game engine escolhida, pois sua descontinuidade em algum momento futuro pode trazer empecilhos ao sistema, como a falta de suporte, ou estagnação tecnológica e gráfica. Um caminho escolhido para estes casos é a substituição da engine por outra que supra as necessidades. Contudo, fazer esta troca resulta em uma reprogramação completa do simulador, que dependendo do tamanho e da complexidade do sistema, pode demandar de um longo período. No entanto, algumas atividades não podem esperar, necessitando da substituição sem que o sistema pare de funcionar, e de forma transparente a quem o esteja usando. Para buscar formas de fazer esta atualização, é necessário uma metodologia baseada na substituição por partes, a qual tem como etapa base a distribuição da comunicação entre engines. Para isso, este trabalho apresenta o uso de adaptadores para realizar a comunicação de forma transparente entre engines distintas, mantendo a interoperabilidade do sistema. Os adaptadores se mostram ser capazes de realizar a comunicação entre diferentes engines. Contudo, para alcançar o objetivo final de substituição de engine, são necessários estudos em outros pontos da metodologia, como a modularização do simulador ou a separação das amarras de motores de física, por exemplo, a fim de testar sua viabilidade.

Palavras-chave: Sistemas Distribuídos. Simulação. Data Distribution Service. Middleware. Adaptadores.

ABSTRACT

USE OF PLUGINS TO FACILITATE THE DISTRIBUTION OF GAME ENGINE IN DISTRIBUTED SIMULATION

AUTHOR: Rafael Vales Bettker
ADVISOR: Joaquim Vinícius Carvalho Assunção
CO-ADVISOR: Raul Ceretta Nunes

The simulators, as well as games in general, are built upon game engines. These simulators have an important role in providing a set of features common to all simulation systems. However, when designing a simulator, attention should be paid to the chosen game engine, as its future deprecation may bring obstacles to the system, such as the end of support, or technological and graphic stagnation. One way to be chosen for these cases is to replace the engine with one that meets the needs. However, making this replacement results in a complete reprogramming of the simulator, which depending on the size and complexity of the system, can demand a long time. However, some activities cannot wait, requiring replacement without the system stops, and transparently to whoever is using it. To find the best way to make this update, it is necessary a methodology based on the substitution by modules, which has as base stage the distribution of communication between engines. For this, this work presents the use of plugins to make the communication in a transparent way between different engines, maintaining the interoperability of the system. Plugins are shown to be capable of communicating between different engines. However, to reach the final goal of engine replacement, studies are needed in other points of the methodology, such as the modularization of the simulator or the separation of the physics engine ties, for example, to test its viability.

Keywords: Distributed Systems. Simulation. Data Distribution Service. Middleware. Plugins.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API	Interface de Programação de Aplicações
$DCPS$	Data-Centric Publish-Subscribe
DDS	Data Distribution Service
DIS	Distributed Interactive Simulation
DLL	Dynamic-link library
$DLRL$	Data-Local Reconstruction Layer
HLA	High Level Architecture
$HLAPI$	High Level API
IDL	Interface Definition Language
IoT	Internet das Coisas
$LLAPI$	Low Level API
OMG	Object Management Group
QoS	Quality of Service
SO	Sistema Operacional
$UNet$	Unity Networking

SUMÁRIO

1 INTRODUÇÃO

A simulação é uma forma aproximada de representar um processo do mundo real, no mundo virtual. Possui diversos tipos de aplicações e é bastante explorada para o treinamento e aprendizado em diversas áreas (AEBERSOLD, 2016), uma vez que ajuda quem está usando a adquirir conhecimento sem precisar necessariamente executar a tarefa no mundo real, evitando gastos de matéria-prima, mão de obra, ou tempo.

Um simulador pode desempenhar diversos papéis, como o de direção presente nas autoescolas, que funcionam como um método de conhecimento inicial no veículo. Um treinamento inicial neste tipo de simulador ajuda o aluno a obter um domínio básico sobre a direção, além de vivenciar as situações do trânsito de uma cidade. Assim, permite com que o aluno possa ganhar uma confiança adicional antes de iniciar as aulas práticas.

Outro campo em que os simuladores desempenham um importante papel é o militar. O treinamento das tropas militares é uma tarefa que deve ser feita constantemente, visto que devem estar preparadas para quando houver um chamado. Utilizar simuladores para este treinamento é algo amplamente adotado pelos exércitos ao redor do mundo, visto que é uma boa forma de fazer um treinamento com riscos e custos mínimos, já que evita gastos de munições, ou possíveis acidentes que possam ocorrer por falta de técnica do usuário, que no mundo virtual não trás consequências. Além disso, é possível treinar técnicas de combate em terrenos sem a necessidade de deslocar a tropa até o local.

Quando um simulador está dividido em mais de um computador, interconectados por uma rede de comunicação, tem-se uma simulação distribuída (FUJIMOTO, 2000). Todas as máquinas podem estar dentro de uma mesma sala, ou até mesmo distribuídas geograficamente. A distribuição do sistema serve para que consiga abranger mais de um agente interagindo no sistema de simulação ao mesmo tempo, ou para distribuir o poder de processamento necessário para executar a tarefa para o qual ele foi desenvolvido.

Um sistema de simulação é construído a partir de diversas ferramentas que são responsáveis pelos componentes do simulador, desde o padrão de troca de mensagens presente nas simulações distribuídas, até o motor de jogo (*game engine*) que proveem diversos recursos. A descontinuidade de uma dessas ferramentas pode trazer alguns empecilhos ao sistema, como a falta de suporte ou a estagnação das tecnologias, impedindo sua evolução.

Quando isso acontece com a *game engine* o problema pode ser ainda maior, uma vez que o simulador, na maioria dos casos, é projetado e construído em cima dela, podendo dificultar sua substituição e demandar um tempo maior até que o sistema completo possa ser substituído. Porém, essa espera pode se tornar inviável para simuladores que estão ativos e em constante funcionamento, tornando necessário buscar meios de realizar a substituição em partes e mantendo a interoperabilidade do sistema.

1.1 JUSTIFICATIVA

Quando um simulador é construído, os desenvolvedores utilizam tecnologias de terceiros para suprir necessidades comuns, buscando aquele que mais se adapta à necessidade do projeto. Uma tecnologia essencial para qualquer simulador é a *game engine*, que se encarrega de suprir diversas funcionalidades básicas a um simulador. Tais funções vão desde a física envolvida no comportamento de objetos cujas ações proveem de um único comando, ou pela comunicação entre acontecimentos gerados pelo back-end e pela parte gráfica, entre outros.

Na maioria das vezes, é comum utilizar *game engines* populares, e já consolidadas no mercado, por serem mais completas em termos de funcionalidades disponíveis e terem uma comunidade maior para prover suporte. Porém, a maioria delas são genéricas para jogos em geral e, em alguns casos, é mais vantajoso buscar uma *engine* voltada ao contexto do projeto, de modo a facilitar a sequência do desenvolvimento.

Para esses casos, é possível dar um exemplo no cenário militar. Há algumas *engines* voltadas à simulação militar, como o Virtual Battlespace 4 (VBS4) (VBS4, 2020) e o Delta3D (DARKEN; MCDOWELL; JOHNSON, 2005). O que diferencia estas de outras, é que elas permitem um desenvolvimento mais facilitado para simulações neste contexto, tendo já recursos comuns à simuladores militares, como modificação completa do ambiente para se adequar à necessidade do treinamento, funcionalidades de gravação da simulação para análise posterior, entre outros. Estas são funções que estão disponíveis nestas *engines* específicas, e que se fossem implementadas em *engines* comuns aumentariam os custos do projeto.

Por serem específicas para um certo nicho, tais *engines* acabam não sendo tão populares para outros fins. A baixa procura por certa *engine* pode fazer com que os esforços para mantê-la atualizada não tragam retorno e, com isso, acaba por ser descontinuada. Esse é um grande problema que deve ser considerado pelo projetista ao escolher a *engine* para um sistema de simulação.

Assim, as *engines* especialistas e outras menos populares se tornam mais vulneráveis a serem descontinuadas, podendo trazer problemas à quem esteja dando suporte a um simulador construído em cima de uma destas. Sendo um simulador construído para ter seu funcionamento a longo prazo, a chance dele passar por isso, caso a *engine* não tenha um futuro promissor, aumenta.

Caso a descontinuação da *engine* em uso aconteça, o simulador pode parar de evoluir, tanto tecnologicamente como graficamente, ou no pior cenário, apresentar problemas que comprometam seu funcionamento, ou ainda, por falta de compatibilidade, impedir que o hardware ou o sistema operacional (SO) seja atualizado. Sem o suporte da *engine*, podem ser necessários custos adicionais para as correções.

Uma vez que a tecnologia foi descontinuada, ou não possui atualizações tecnoló-

gicas e gráficas que se adéquem às necessidades do simulador, é preciso buscar meios de migrar o simulador para outra *engine* que dê as funcionalidades e suporte necessário. Essa migração pode ser bastante complexa, já que na maioria das vezes, todo sistema é construído tomando como base a *game engine*. Então, o processo deve ser bem planejado de modo a não trazer grandes impactos ao simulador em si.

Uma maneira de migrar para uma nova tecnologia é fazer a reprogramação completa do simulador na nova *engine*, para então usá-la no lugar daquela descontinuada. Esse é um processo que, dependendo do tamanho e complexidade do simulador, se torna trabalhoso.

O tempo pode ser outro ponto importante nesse processo. Alguns simuladores podem ser bastante usados e estar em constante funcionamento. Esperar até que a reprogramação completa do sistema seja feita em outra *engine* pode ser inviável. Assim sendo, outras formas devem ser encontradas para a execução do processo.

1.2 PROPOSTA

Considerando o cenário de um simulador distribuído desenvolvido em cima de uma certa tecnologia, quando esta for descontinuada, deve haver uma maneira de migrar para uma nova. Assim, este trabalho propõe uma abordagem para facilitar esta migração, de modo que todo processo seja transparente para quem está usando o simulador.

Uma hipótese considerada é que a lógica de negócio deste simulador pode ser modularizada, ou seja, os diferentes componentes (e.g., que controla os veículos, que faz os cálculos de física etc.) podem ser separados de modo independente. Uma vez orientado à objetos, é natural que os sistemas sejam desenvolvidos desta maneira, então um simulador bem planejado cumprirá com esta condição.

Partindo desta premissa, o objetivo final é fazer o desenvolvimento de partes funcionais em outra *engine*, e então, fazer esta substituição aos poucos a medida que elas vão sendo reconstruídas na nova tecnologia. Para isso, é necessário haver uma comunicação transparente entre ambas *engines*, mantendo a interoperabilidade do sistema.

O principal fator necessário para manter o sistema interoperante é que a comunicação de diferentes componentes seja independente de *game engine*. A abordagem mais adequada para isso é a implementação de adaptadores para intermediar a comunicação, de modo a serem responsáveis por converter as mensagens passadas de uma ponta a outra e garantir que a comunicação flua independente de *engine*. Tal solução se torna mais simples e eficaz uma vez que não é necessário fazer mudanças em componentes já existentes do simulador, apenas a implementação deste novo módulo que deve realizar a integração.

A construção de um adaptador para cada *engine* é necessária pois, na maioria das vezes, cada uma deve possuir seu próprio padrão de comunicação, dificultando uma comunicação direta entre ambas as pontas. Uma solução para isto é utilizar o padrão *Data Distribution Service* (DDS) nos adaptadores como forma de garantir a interoperabilidade do sistema. Este conta com o paradigma *Publish-Subscribe* que permite uma comunicação ponta-a-ponta sem *broker* de mensagens, melhorando sua confiabilidade, além de possuir uma adição simplificada de novos tipos de dados sem a necessidade de modificar aquilo que já existe, se adequando à abordagem modular buscada.

1.3 OBJETIVOS

1.3.1 Geral

Propor o uso de adaptadores como meio para partilhar uma simulação distribuída em diferentes *game engines*, que serão responsáveis pela comunicação transparente entre as pontas, e servirão como base para a aplicação de uma metodologia voltada em rejuvenescer simuladores a partir da troca de *game engine*, com foco em manter o sistema em funcionamento durante todo o processo.

1.3.2 Específicos

- Aprofundar o estudo em diferentes *game engines* a fim de descobrir formas de integrar suas comunicações com DDS;
- Apresentar uma metodologia para a substituição modular de componentes entre *game engines* em um simulador distribuído, mantendo sua interoperabilidade;
- Desenvolver adaptadores que façam a conversão de mensagens trocadas internamente em uma *game engine*, para o padrão DDS;
- Realizar testes com duas *game engines* e seus adaptadores de modo a verificar a interoperabilidade do método;
- Implementar uma Interface de Programação de Aplicações (API) que facilite o uso do DDS no contexto de simuladores;
- Fazer a utilização dos adaptadores em um sistema de simulação para validar e verificar a eficácia da abordagem.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta uma descrição sobre os assuntos e tecnologias exploradas neste trabalho. As seções 2.1 e 2.2 apresentam uma introdução e conceitos básicos sobre simulação e sistemas distribuídos, respectivamente. A simulação distribuída é apresentada na seção 2.3. A seção 2.4 aborda as *game engines*, englobando o Delta3D e Unity. Por último, é apresentado o padrão *Data Distribution Service* na seção 2.5, utilizado pelos adaptadores.

2.1 SIMULAÇÃO

Os métodos de simulação possuem um papel importante no estudo de técnicas do mundo real em diversos contextos. Desde jogos de simulação para aproximar uma pessoa comum de certa tarefa, até simuladores militares para treinamento de tropas, estes sistemas ajudam as pessoas a desenvolver um conhecimento maior sobre determinado assunto.

A simulação também é usada como meio para estudar sistemas complexos que métodos analíticos não são suficientes (HEERMANN, 1990). Neste contexto, foca-se no estudo de como o comportamento de um sistema se desenvolve de acordo com certos parâmetros. Um exemplo é a simulação de multidão, onde são criados cenários virtuais para testar o comportamento de um grande número de pessoas se movendo pelo ambiente. Segundo DOD (1998), este é um caso de simulação construtiva, que ao lado de viva e virtual, classificam uma simulação de acordo com suas características:

- **Viva**: formado por pessoas reais utilizando equipamentos reais, como veículos ou armamento.
- **Virtual**: formado por pessoas reais utilizando equipamentos virtuais, simulados no computador.
- **Construtiva**: formado por pessoas simuladas em um ambiente simulado, onde pessoas reais apenas definem os parâmetros de entrada da simulação.

Um simulador pode possuir mais de uma destas classificações em conjunto. Uma simulação viva pode estar sendo executada em determinado local, enquanto que, em tempo real, suas informações podem estarem sendo transmitidas para um simulador virtual que conseguirá reproduzir o exato ambiente no computador para que outras pessoas possam participar da simulação, de forma virtual.

2.2 SISTEMAS DISTRIBUÍDOS

De acordo com (TANENBAUM; STEEN, 2007), sistema distribuído é um conjunto de computadores independentes que se mostram ao usuário como um único sistema consistente. Estes se tornam uma boa opção pois exploram o trabalho em rede, conseguindo ganhos consideráveis a partir da paralelização das tarefas.

Ao lado do desempenho, tais sistemas são construídos buscando possuir uma maior escalabilidade e confiabilidade à aplicação. A escolha de dividir o sistema em várias partes, que trabalham separadamente, torna mais fácil a adição de novos componentes quando necessário. Adicionalmente, a divisão entre várias máquinas ajuda com que o sistema seja tolerante a falhas, já que um problema em uma única máquina geralmente não afetará o sistema como um todo.

Hoje, os sistemas distribuídos são usados em diferentes contextos (STEEN; TANENBAUM, 2016). Dentre eles, há a computação distribuída de alto desempenho, que engloba a computação em nuvem, *cluster* ou *grid*; os sistemas de informação distribuídos, tratando do processamento distribuído de informações; ou os sistemas persuasivos, como sistemas de computação móvel, computação ubíqua, ou rede de sensores.

Dois conceitos de sistemas distribuídos, bastante explorados neste trabalho, são apresentados na sequência. A camada de middleware, descrito na seção 2.2.1, e a questão de um sistema ser interoperável, detalhada na seção 2.2.2.

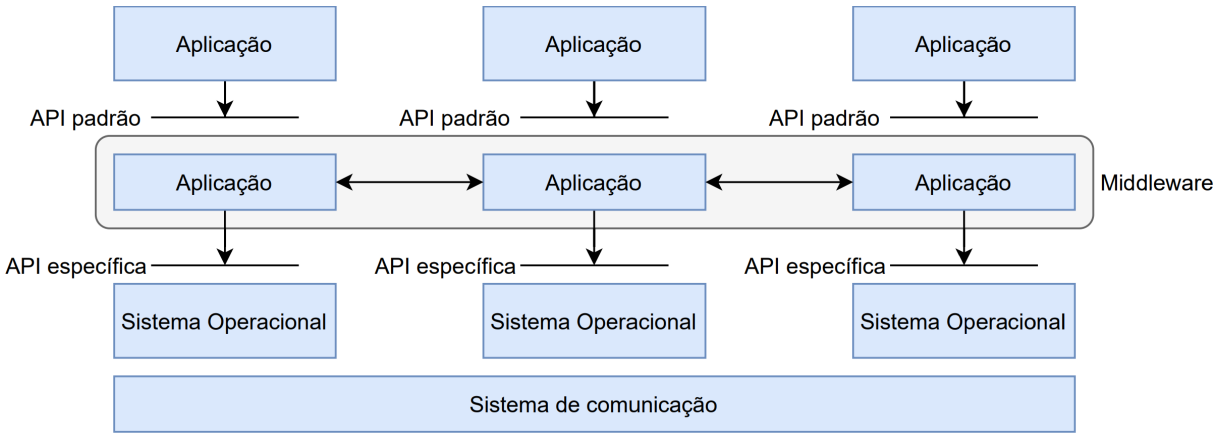
2.2.1 Middleware

Um *middleware* age como uma camada adicional posicionada entre o sistema operacional e a aplicação. Seu propósito é fornecer uma abstração das camadas inferiores, de modo a ocultar a complexidade e tornar mais simples o desenvolvimento de sistemas distribuídos. A Figura 2.1 mostra como o *middleware* se encaixa em um sistema.

Sendo um termo bastante abrangente, o *middleware* pode ter funções que vão desde ajudar o desenvolvimento de novas aplicações, dando ao desenvolvedor acesso à certas funcionalidades, até a otimização de aplicações, onde desempenha um papel de facilitador, provendo mais facilidade à uma portabilidade da aplicação, por exemplo.

Em aplicações distribuídas, é usado para auxiliar no transporte de informações e dados entre programas que utilizam diferentes protocolos, o qual geralmente proporciona essa integração a partir de APIs de alto nível que possibilita uma independência ao dispositivo em questão. Deste modo, desenvolvedores podem focar os esforços no propósito central da aplicação, sem que diferenças de protocolos ou arquiteturas atrapalhem o desenvolvimento.

Figura 2.1 – Organização de um middleware



Fonte: (KRAKOWIAK, 2007)

2.2.2 Interoperabilidade

A interoperabilidade é uma soma de diversas integrações: sistemas, redes, dados etc. Diferentes sistemas podem divergir em questões de plataforma, hardware, ou software, porém, seguindo certos padrões abertos é possível que a comunicação ocorra de forma clara, característica principal de um sistema interoperável (PUDER; RÖMER; PILHOFER, 2011).

Com o avanço dos sistemas distribuídos, cada vez mais há a necessidade de que um sistema possua características de integração. Havendo a distribuição de um sistema em diferentes processos, a troca de informações entre eles é o ponto base para o funcionamento do todo. Os desenvolvedores mantém um esforço contínuo para garantir que sistemas possuam características de integração e reuso de informações.

Existem padrões abertos que buscam homogeneizar a troca de dados entre dois sistemas distribuídos diferentes. Algumas empresas trabalham nisso, como é o caso da *Object Management Group* (OMG), criadora de padrões como o CORBA e DDS, sendo o DDS o padrão usado neste trabalho e que é detalhado mais à frente na seção 2.5.

2.3 SIMULAÇÃO DISTRIBUÍDA

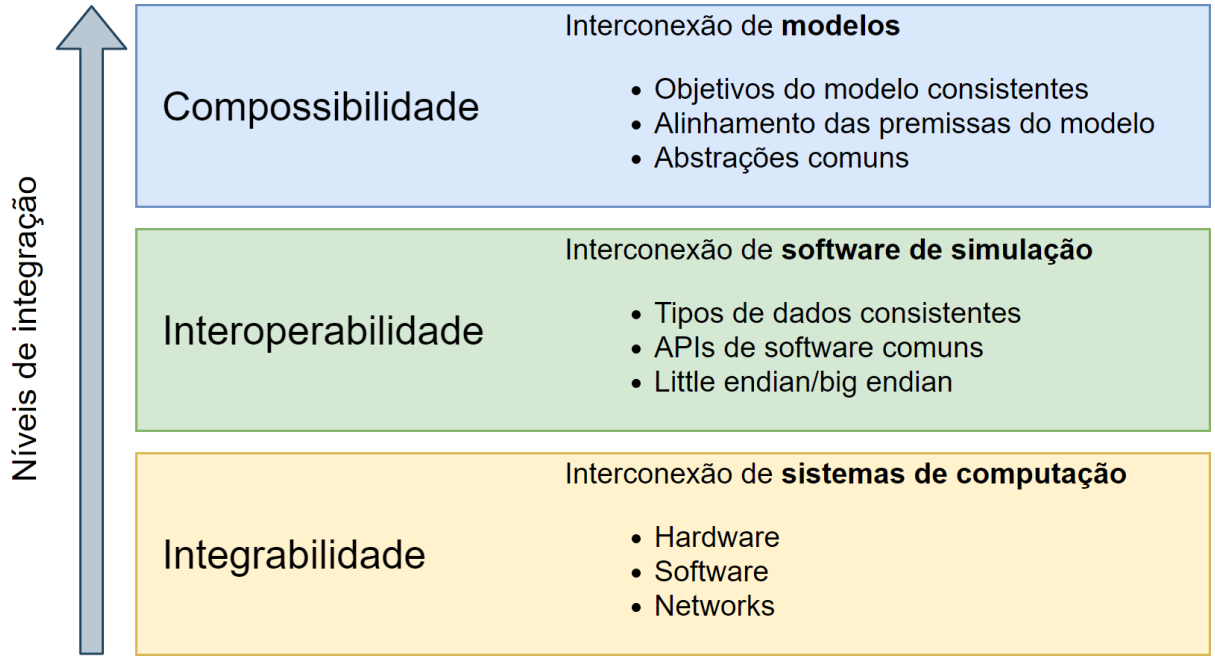
A simulação distribuída e paralela são tópicos que crescem desde meados dos anos 80. Enquanto a simulação paralela está concentrada em dividir o poder computacional em vários processadores, a simulação distribuída abrange um escopo maior. Podendo executar em diferentes máquinas interligadas por uma rede local, distribuídas em pontos geográficos distintos e se comunicando pela Internet, ou conectadas com o ambiente por meio de sensores, há diversos argumentos (TAYLOR, 2019) para o uso desta tecnologia:

- **Tempo de execução**: a divisão de uma simulação entre múltiplos computadores, explorando a paralelização, torna ela mais rápida de ser executada;
- **Compatibilidade de modelos e reuso**: ao reusar a simulação como um subcomponente de alguma aplicação, ou com outros modelos de simulação, pode ocorrer incompatibilidade de pacotes, linguagem etc. Tratar ela como uma simulação distribuída pode ser menos trabalhoso, uma vez que permite que a atenção seja voltada apenas à comunicação;
- **Manutenção**: um sistema de simulação dividido entre vários componentes se torna mais fácil de ser atualizado, uma vez que eles podem ser gerenciados independentemente;
- **Privacidade**: alguns tipos de simuladores podem conter informações confidenciais em seu código, como os militares. Um simulador distribuído faz com que estas informações fiquem restritas apenas para quem está desenvolvendo o componente onde serão usadas, sem que desenvolvedores de outros componentes tenham acesso as mesmas;
- **Integridade dos dados**: se uma simulação acessa certo banco de dados, e uma nova simulação é criada, dependendo do caso pode haver a necessidade de duplicar os dados para ser acessados por ambas, além de ser necessário manter a integridade dos mesmos. Um modelo distribuído pode ter módulos para evitar este problema;
- **Simulação híbrida**: facilita a integração de simuladores a partir da ajuda de pacotes comerciais.

Segundo TAYLOR (2019), há três modos de sistemas de simulação distribuída. O primeiro é quando há a necessidade de aumentar o desempenho de uma simulação, transformando um modelo único em vários submodelos buscando separar o poder computacional em diferentes processos, se comunicando entre si pela rede. Outro caso é quando se deseja ter mais de um modelo na mesma simulação, conectando-os por uma rede de comunicação para que consigam interagir de maneira conjunta no mesmo ambiente. O último modo é quando se deseja executar experimentos na simulação, e o uso de um gerenciador que envia comandos para múltiplos processos ao mesmo tempo se torna melhor que executar várias vezes o mesmo, sequencialmente.

Havendo vários modelos, é necessário que estes estejam integrados para trabalhar como um único sistema. A integração na simulação distribuída ocorre por meio de mensagens de eventos que são trocadas entre as diferentes simulações. Para além da troca de mensagens, a Figura 2.2 mostra as três dimensões propostas por PAGE; BRIGGS; TUFAROLO (2004) para a integração de um sistema de simulação: interconexão entre os sistemas computacionais, software de simulação, e modelos.

Figura 2.2 – Dimensões de integração em simulação distribuída



Fonte: (TAYLOR, 2019)

A integrabilidade foca no contexto de baixo-nível, como *hardware*, *firmware*, ou a rede e seus protocolos, permitindo uma comunicação entre os sistemas computacionais. A comunicação entre os softwares de simulação fica no domínio da interoperabilidade, onde deve haver padrões de tipos de dados e APIs bem definidas que permitam diferentes simulações se comunicarem. O último domínio é a compossibilidade, onde os objetivos e alinhamento dos modelos devem ser comum. Todos estes são independentes, ou seja, dois sistemas podem possuir interoperabilidade entre si, mas não integrabilidade ou compossibilidade.

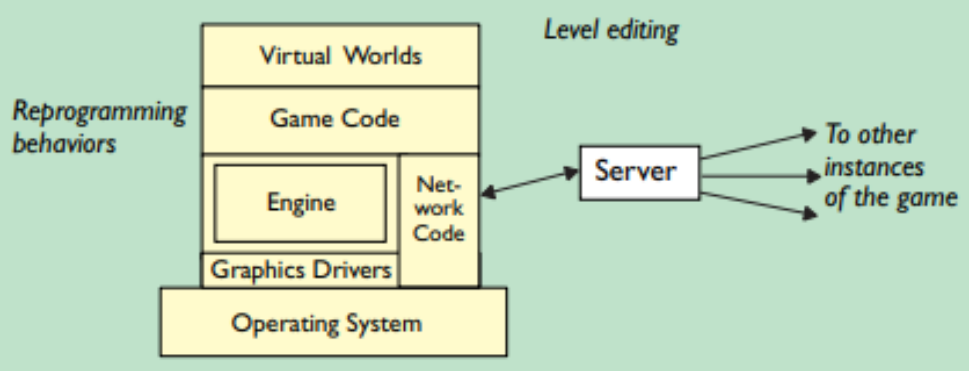
2.4 GAME ENGINE

Os jogos possuem diversas funcionalidades que são comuns à todos eles. Sempre haverá uma entrada, seja por um teclado ou controle, por onde o jogador deve interagir com o jogo, um sistema para simular a física, ou mecanismos de rede necessários para jogos multijogador. Todas estas funções comuns podem ser abstraídas aos desenvolvedores, poupando um esforço desnecessário. Este é o trabalho de uma *game engine*.

Uma *game engine* é um *software* ou conjunto de bibliotecas que implementa diversas funcionalidades comuns aos jogos – ou qualquer aplicação gráfica que funcione em tempo real – e ajuda a simplificar o trabalho de um desenvolvedor. Em geral, a *engine* fornece de forma facilitada a renderização, simulação de física, suporte para animação,

áudio, módulos de inteligência artificial, rede, linguagens de *script*, grafos de cena etc (GREGORY, 2018). Algumas ainda facilitam a portabilidade da aplicação para várias plataformas. A Figura 2.3 apresenta a estruturação de uma *engine*.

Figura 2.3 – Estruturação modular de uma *game engine*



Fonte: (LEWIS; JACOBSON, 2002)

Na camada superior ao sistema operacional são executados os *drivers* gráficos e a *engine*, que representam a parte abstraída ao desenvolvedor. Na sequência, está o código programado do jogo, que utiliza as funcionalidades da *engine*, além da rede. A camada mais superior é o mundo virtual, criado a partir do código do jogo e apresentado ao usuário da aplicação.

A rede pode ser apresentada como uma camada independente – como mostrada na arquitetura acima –, o que geralmente ocorre nas engines mais antigas; ou juntamente dentro da *engine*, abstraída ao desenvolvedor. O Delta3D, apresentado na seção 2.4.1, e o Unity, mostrado na seção 2.4.2, são exemplos de *game engines* que serão abordadas neste trabalho.

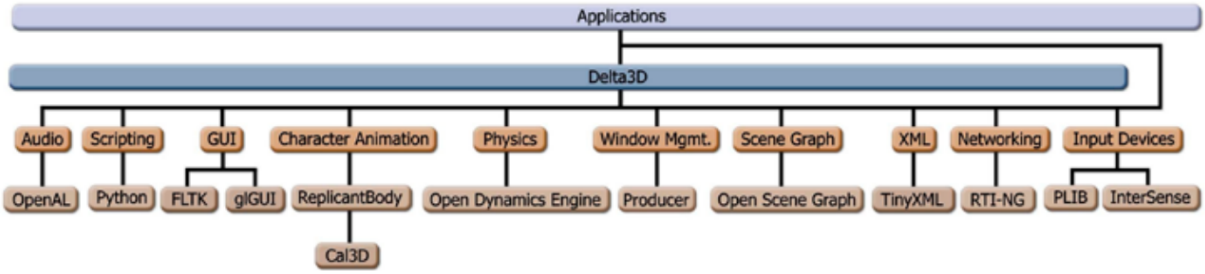
2.4.1 Delta3D

O Delta3D é uma *game engine* de código aberto desenvolvido pelo Instituto MOVES da *Naval Postgraduate School* com foco no treinamento militar. Nos primeiros anos a partir do seu lançamento, em 2004, foi usado em diversas aplicações de empresas privadas e governamentais, atraídas principalmente pela isenção de taxas de uso e por disponibilizar métodos de criação de sistemas de treinamento de maneira rápida e sofisticada para a época.

A *engine* foi construída com base em um conjunto de bibliotecas (Figura 2.4) de código aberto, de modo a dar oportunidade aos desenvolvedores de fazer modificações que achem necessárias para suprir as necessidades de suas aplicações. Além disso, todo código do Delta3D foi construído de maneira modular – há uma biblioteca responsável

pela física, IA, animação, entre outros –, para que quando alguma destas ficasse ultrapassada, pudesse ser substituída sem precisar de mudanças nas demais (MCDOWELL et al., 2006a).

Figura 2.4 – Conjunto de bibliotecas de código aberto que compõem o Delta3D



Fonte: (MCDOWELL et al., 2006b)

Por possuir uma base em um conjunto de bibliotecas independentes, problemas de compatibilidade começaram a surgir com o tempo. Algumas bibliotecas deixaram de ser atualizadas, enquanto outras seguiram ativas, resultando em requerimentos de sistema diferentes para instalação de cada uma, tornando necessário o uso de sistemas de controle de versão de *software*. Este e outros motivos fizeram com que o Delta3D parasse de receber atualizações no ano de 2015.

O principal ponto do Delta3D a ser explorado neste trabalho, é a sua rede de comunicação, a qual os adaptadores devem se conectar para realizar a troca de mensagens entre *engines*. Toda aplicação construída com Delta3D se comunica via mensagens que são enviadas e recebidas por meio de um *Game Manager*. É possível fazer o envio interno, para todos os outros componentes que estão executando no mesmo processo, ou pela rede, para componentes de outros processos. Todo componente possui uma *callback* que é chamada quando uma nova mensagem é recebida. As mensagens trocadas devem ser derivadas do módulo de mensagens do Delta3D, ou seja, suas estruturas devem ser declaradas previamente.

Além disso, existem módulos que suportam o protocolo *Distributed Interactive Simulation* (DIS) – usado pelo Departamento de Defesa dos Estados Unidos e outras instituições – e o padrão *High Level Architecture* (HLA), possibilitando uma conexão destes com a rede interna do *Game Manager*.

2.4.2 Unity

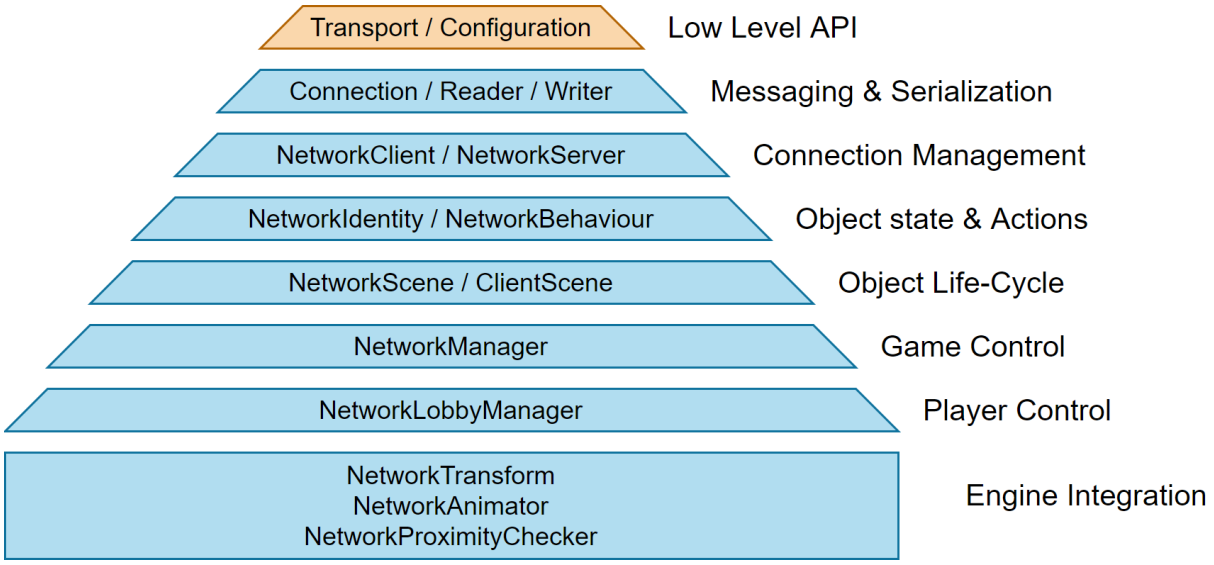
A *engine* mais popular no mercado atualmente é o Unity. Além de possuir todas funcionalidades de um motor de jogo, também trás pontos positivos em relação à facilidade de uso, compatibilidade para várias plataformas, e uma grande comunidade para auxiliar

com tutoriais e suporte quando necessário. Hoje, inúmeros dos jogos dos mais populares são feitos nesta *engine*.

O módulo de rede do Unity é chamado de *Unity Networking* (UNet), e é dividido em duas APIs: *High Level API* (HLAPI) e *Low Level API* (LLAPI). O HLAPI é uma forma de disponibilizar ao desenvolvedor todos recursos necessários para a construção de um jogo multijogador, utilizando a arquitetura cliente-servidor (UNITY TECHNOLOGIES, 2020).

A Figura 2.5 apresenta as diversas camadas do HLAPI, que podem ser utilizadas para facilitar o uso dos respectivos serviços. A utilização é flexível, certas camadas podem ser usadas enquanto outras não. O LLAPI é uma parte do HLAPI, a camada mais inferior, que concede uma total liberdade àqueles que querem explorar as vantagens da programação baixo nível.

Figura 2.5 – Unity Networking High Level API



Fonte: (UNITY TECHNOLOGIES, 2020)

Uma vez que o UNet é uma rede interna ao Unity, seu uso torna a comunicação restrita apenas a outros clientes Unity, tendo que buscar métodos alternativos para haver uma integração com aplicações externas. Recentemente, foi anunciado que o UNet será descontinuado e substituído por uma nova camada de rede, que focará no desemprenho, escalabilidade, e segurança.

2.5 DATA DISTRIBUTION SERVICE

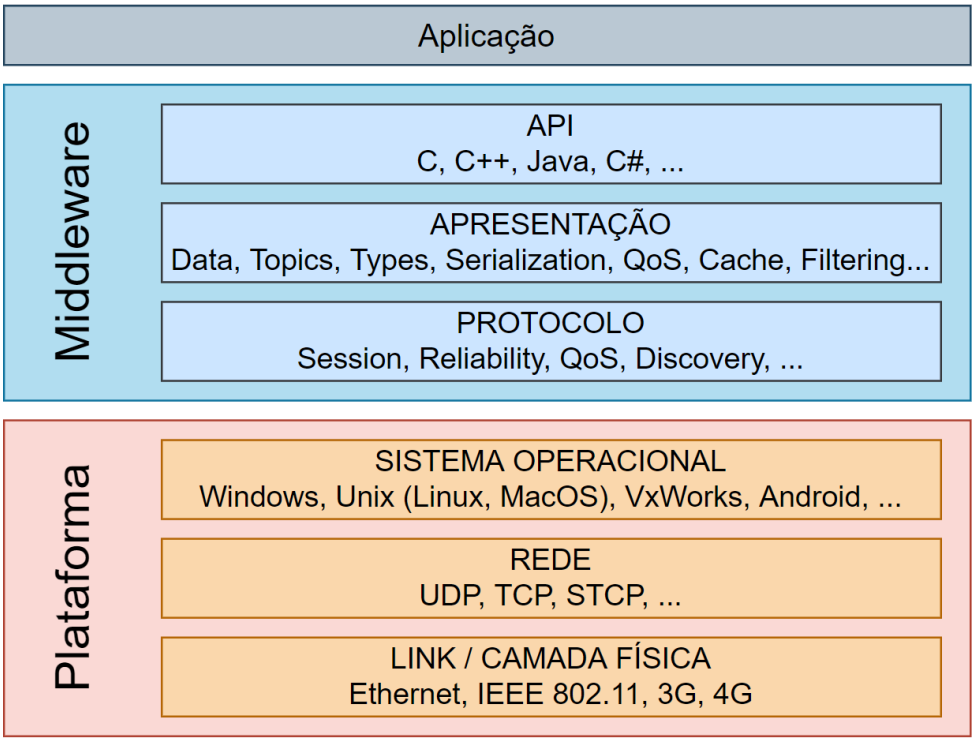
O *Data Distribution Service* é um padrão desenvolvido pelo *Object Management Group* para conectividade e integração de aplicações distribuídas. Sendo baseado no padrão *Publish-Subscribe*, suporta comunicações um-para-um, um-para-muitos, muitos-

para-um, muitos-para-muitos (SCHLESSELMAN; PARDO-CASTELLOTE; FARABAUGH, 2004), que facilita seu uso nos mais diversos tipos de aplicações.

O padrão DDS tem por objetivo fazer a troca de dados em tempo real, de maneira confiável e otimizada para prover um alto desempenho. Possui uma arquitetura altamente escalável, o tornando bastante utilizado em aplicações de Internet das Coisas (IoT). Também é interoperável, pois possibilita uma fácil integração por conta da sua arquitetura centrada em dados, (OBJECT MANAGEMENT GROUP, 2020b).

A Figura 2.6 mostra como o DDS, assumindo o papel de *middleware*, abstrai os detalhes de comunicação de baixo nível. A partir de uma API (OBJECT MANAGEMENT GROUP, 2020a), a aplicação tem acesso às funcionalidades necessárias para realizar a troca de informações, sendo compatível entre diferentes sistemas operacionais, linguagens de programação, ou arquiteturas de processadores. O DDS utiliza o padrão de troca de mensagens *Publish-Subscribe*, detalhado na seção 2.5.1, enquanto que a seção 2.5.2 sua arquitetura de funcionamento.

Figura 2.6 – Camadas de comunicação



Fonte: (OBJECT MANAGEMENT GROUP, 2020b)

2.5.1 Publish-Subscribe

O padrão *Publish-Subscribe* é tipicamente usado em aplicações distribuídas em que um lado (publicador) envia dados, de forma indireta, para um ou mais assinantes. A comu-

nicação ocorre sem que um saiba da existência do outro, tendo em comum apenas o tipo de dado que está sendo transportado. É um paradigma que dá uma grande escalabilidade, e uma topologia de rede mais dinâmica ao sistema.

A interação básica ocorre por meio do publicador, assinante, e dos eventos. Um evento está ligado a um certo tipo de dado. Uma aplicação pode se inscrever para receber as informações de um dado evento, de modo a ser notificada quando houver algo novo. Do outro lado, a aplicação pode publicar uma informação de certo evento, que será entregue aos assinantes correspondentes.

A troca de informações é dissociada em três dimensões: espaço, já que uma das partes não precisa conhecer a outra; tempo, pois um assinante pode receber um dado publicado antes mesmo dele ter se conectado; e sincronização, pelo envio e recebimento dos dados não ocorrerem no fluxo principal da aplicação (EUGSTER et al., 2003).

2.5.2 Arquitetura

O padrão DDS é especificado como um modelo independente de plataforma ou linguagem de programação, possibilitando seu uso nos mais diferentes cenários. Assim como qualquer aplicação distribuída, o DDS é composto por diferentes processos, executando em componentes ou máquinas diferentes do sistema, também chamados de participantes. A interface *Data-Centric Publish-Subscribe* (DCPS) permite uma separação independente entre o lado que publica informações, àquele que se inscreve para recebe-las, possibilitando ao participante efetuar apenas uma tarefa, ou ambas.

A especificação descreve dois níveis de interfaces. O DCPS é o nível inferior, responsável pela troca de informações entre os diferentes componentes do sistema. O nível superior e opcional, chamado *Data-Local Reconstruction Layer* (DLRL), permite uma integração facilitada à camada de aplicação (PARDO-CASTELLOTE, 2003).

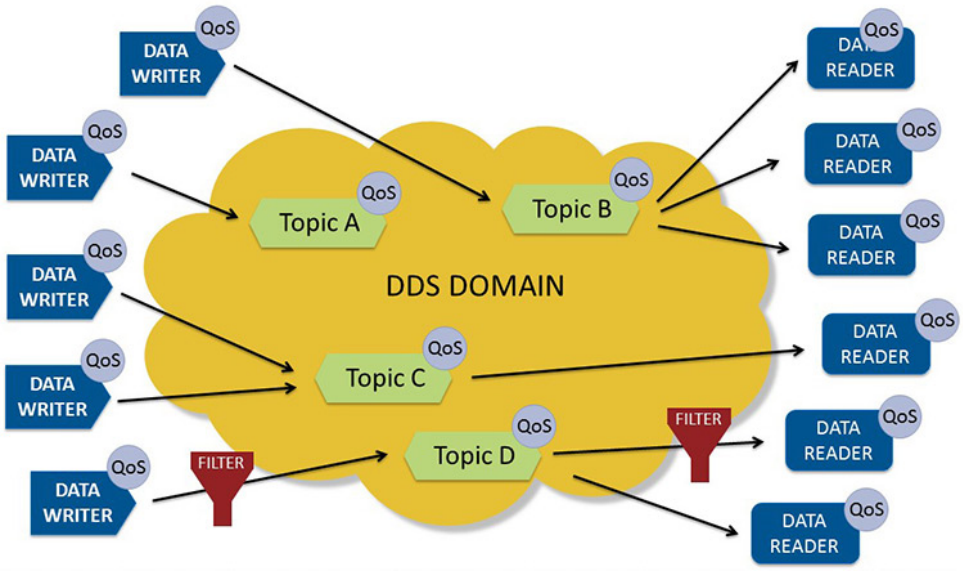
Sua implementação utiliza uma interface tipada, o tornando mais simples, seguro e eficiente (OBJECT MANAGEMENT GROUP, 2020a). Os dados são representados em um arquivo independente de linguagem de programação, o OMG *Interface Definition Language* (IDL), e então mapeados para a linguagem da aplicação. Os tipos de dados disponíveis vão desde os primitivos, como *byte* ou *float*, até sequências e estruturas. Além dos dados, que devem ser comum à todos participantes, cada um deve criar as seguintes entidades para a comunicação:

- ***Domain***: É a base de toda aplicação DDS. Provém acesso ao espaço de dados global, por onde as informações são trocadas pelos participantes;
- ***Topic***: Corresponde a um tipo de dado. É o objeto usado pelos participantes do domínio para troca de informações do tipo de dado correspondente;

- ***DataWriter***: Objeto usado para publicar informações em um tópico;
- ***DataReader***: Objeto usado para receber informações de um tópico de interesse;
- ***Publisher***: Participante do domínio que publica informações;
- ***Subscriber***: Participante do domínio que recebe informações, a partir dos tópicos inscritos, e disponibiliza para a aplicação.

As informações enviadas por um participante cruzam pelo espaço de dados global, o qual é único para cada domínio DDS, e chegam aos interessados. Há a possibilidade de usar filtros para selecionar apenas informações do interesse dentro de certo tópico. A Figura 2.7 ilustra o processo de comunicação.

Figura 2.7 – Espaço de dados global do DDS



Fonte: (OBJECT MANAGEMENT GROUP, 2020b)

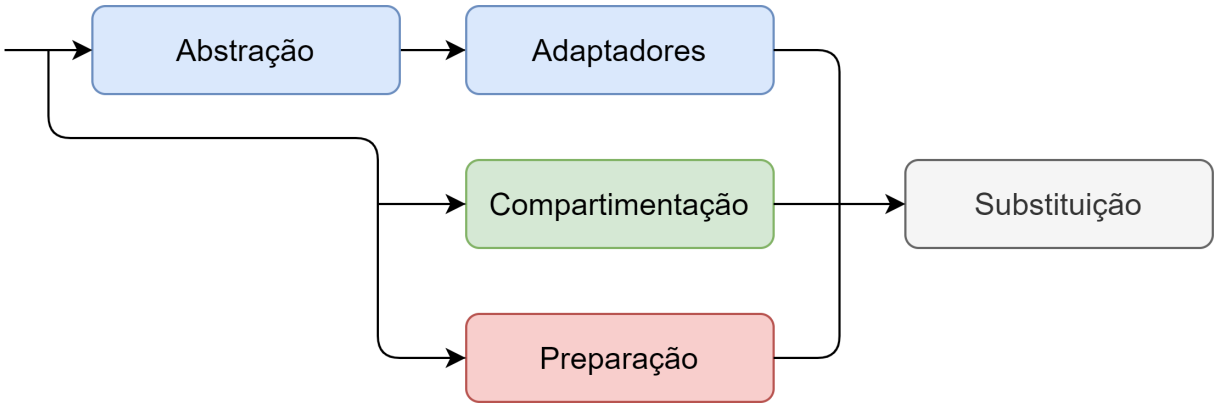
Adicionalmente, o uso de políticas de *Quality of Service* (QoS) permitem controlar o comportamento do serviço, oferecendo propriedades quanto à disponibilidade, entrega, tempo de vida, e uso dos dados (CORSARO, 2014). O DDS realiza a comparação e apenas entrega aos interessados os dados que coincidem com a dada política QoS, evitando que sejam entregues informações desnecessárias e aumentando a eficiência da comunicação.

3 METODOLOGIA PARA USO DE ADAPTADORES

A abordagem deste trabalho apresenta o uso de adaptadores para facilitar a substituição de *game engine* em simuladores distribuídos. Sua função é tratar da comunicação entre *engines* e garantir a interoperabilidade do sistema durante a substituição da tecnologia. Estes adaptadores usam o padrão DDS para conectar a *engine* corrente do simulador e a nova escolhida para ser usada no lugar.

Porém, para que o adaptador desempenhe o papel para o qual foi projetado, de rejuvenescimento do simulador, são precisos alguns passos adicionais. Neste capítulo é brevemente descrito uma proposta de metodologia para a substituição de *game engine* do simulador com o uso dos adaptadores desenvolvidos neste trabalho. A Figura 3.1 mostra as etapas do processo.

Figura 3.1 – Etapas da metodologia



Fonte: Autor

Há três linhas para serem seguidas até ser possível realizar de fato a substituição: adaptadores, compartimentação e preparação. Objetivos deste trabalho, a construção dos adaptadores procede a implementação da camada de abstração ao DDS. A etapa de compartimentação, necessária para dividir o simulador em partes para posterior substituição gradual. Também, a preparação do novo ambiente do simulador, o que inclui terrenos, viaturas etc. Finalmente, a etapa de substituição é feita após as etapas anteriores serem concluídas.

Este trabalho busca se aprofundar nos adaptadores, o qual tem seu desenvolvimento detalhado nos capítulos seguintes. O restante das etapas são, na sequência, brevemente apresentadas e devem serem estudadas em detalhes para a completude da metodologia.

API: Uma Camada Superior de Abstração

O DDS permite políticas de QoS para quase todos objetos, seja para a mensagem, para o *Publisher*, para o *DataWriter* etc. Isto, somado a outros fatores, fazem com que uma simples mensagem demande de vários comandos para ser criada e enviada, podendo ser abstraído à quem esteja desenvolvendo o adaptador.

Uma vez que o adaptador sempre fará a mesma tarefa, de converter mensagens de um tipo para outro, pode haver uma camada superior de abstração ao DDS. Esta camada, API, deve prover métodos que simplifiquem o trabalho de programação, reaproveitando códigos otimizados e reduzindo o esforço de programação. Esta é uma etapa que é feita com o objetivo de manter os adaptadores com um visual mais limpo, mas não é obrigatória para a aplicação da metodologia.

Construção de Adaptadores

O trabalho de uma *game engine* é abstrair ao desenvolvedor as funcionalidades de mais baixo nível, incluindo a comunicação. A partir disso, a maioria delas implementa módulos de comunicação que são usados para a troca de dados na aplicação. Estes módulos, na maioria das vezes, são específicos da *engine* e apenas trocam informações dentro dela mesma, dificultando uma comunicação com aplicações externas. A abordagem proposta visa uma comunicação transparente entre *engines* a partir da construção dos adaptadores que servirão como ponte na comunicação.

O uso de adaptadores para fazer este trabalho de ponte torna simples todo o processo de substituição, isto porque evita com que tenham que ser feitas mudanças no código fonte do simulador, podendo deixar o processo muito mais complexo, além de ocasionalmente gerar a adição de *bugs* e comprometer o funcionamento geral do sistema. A partir do uso de um adaptador, o método *plug and play* pode ser adotado, concentrando os trabalhos na modelagem e implementação apenas no módulo de simulação a ser migrado e no seu respectivo adaptador.

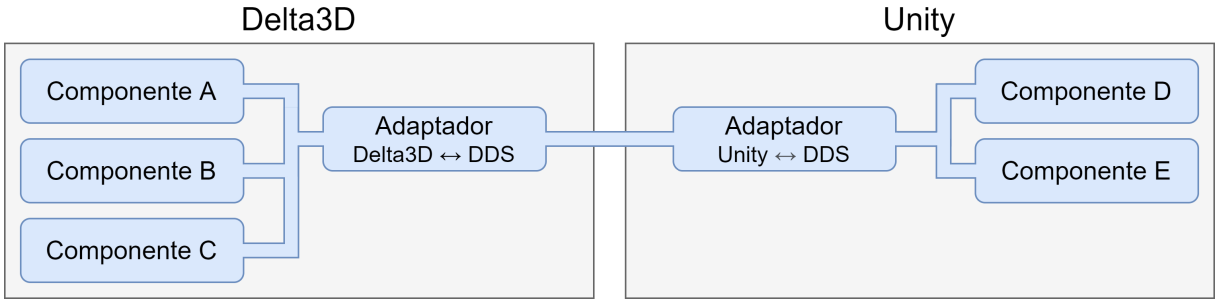
Apesar de, momentaneamente durante o processo de substituição, o simulador estar sendo executado em duas *engines*, ele é um sistema único que deve funcionar de maneira interoperante. Então, todas as informações trocadas na rede interna de uma *engine* fazem parte da simulação e devem circular pela rede interna da outra também.

Isto faz com que o adaptador tenha que ouvir todas as informações trocadas na sua *engine*, converter para o padrão DDS, e enviar para o adaptador da outra ponta. As informações recebidas pelo outro adaptador serão convertidas para o seu padrão interno, e enviadas para a rede da *engine* distribuir aos componentes que dependem daquela informação.

O adaptador então deve possuir duas funções: converter todas as informações da rede interna de sua *engine*, para o padrão DDS, e enviar para o adaptador do outro lado; e converter as mensagens recebidas por DDS, do outro adaptador, para o padrão de comunicação da sua *engine*, e então enviá-las para a sua rede interna.

Simuladores que possuem seus próprios tipos de dados devem reimplementá-los usando o IDL, de modo a possuir um tipo correspondente no padrão DDS para facilitar o processo de conversão e desconversão. Assim, quando um adaptador receber uma mensagem da rede interna de sua *engine*, ele apenas deve copiar os campos da mensagem recebida para uma nova do padrão DDS, sendo um processo simples. O mesmo vale ao receber mensagens do outro adaptador. A Figura 3.2 mostra os adaptadores conectados à rede de cada *engine*, repassando as informações à outra ponta.

Figura 3.2 – Adaptadores servindo de ponte entre ambas *engines*



Fonte: Autor

Compartimentação dos Componentes

Para a substituição ocorrer de forma gradual, o simulador deve ter a possibilidade de ser separado em partes (módulos), que serão substituídas uma a uma. Então, caso ele não esteja explicitamente modularizado, é necessário avaliar como é possível compartimentar o simulador em módulos funcionais independentes de forma que seja possível reimplementar o módulo numa outra *engine*, desativar a sua funcionalidade na *engine* corrente e reativar na nova *engine*, sem que a funcionalidade do simulador seja prejudicada. Aqueles componentes que possuam alguma dependência com outro, devem ser reprogramados e substituídos juntos.

Preparação dos modelos 3D para o mundo virtual

Há diversos elementos gráficos que compõem um ambiente de simulação virtual. Dentre eles, estão os terrenos, construções, personagens, veículos etc. Todos estes elementos devem estar presentes no mundo virtual da nova *engine* também, de forma a manter a consistência na representação visual em ambos os lados.

Em casos específicos, onde os modelos 3D destes elementos foram construídos em um formato amarrado à *engine* antiga, estes modelos deverão ser reconstruídos para a nova, sendo um esforço inevitável. Por outro lado, caso os modelos estejam implementados em um padrão que a nova *engine* também reconheça, o trabalho é poupado e eles podem ser reaproveitados.

Substituição Gradual dos Módulos

Após a preparação do mundo virtual na nova *engine*, juntamente com os adaptadores construídos e conectados em ambas as pontas, é iniciada a reprogramação do simulador na nova tecnologia. Deve ser estudada a ordem com os quais estes serão substituídos. Por exemplo, funcionalidades que estejam com problemas ou que necessitem de novas tecnologias terão prioridade de substituição maior. O processo é reprogramar o código na nova *engine* e prover suas funções por meio da troca de mensagens, entre os adaptadores, de forma remota ao restante do simulador.

Logo depois da reprogramação, deve ser feito uma validação daquele componente antes de fazer a substituição definitiva. As funcionalidades providas devem ser as mesmas, evitando a alteração de tipos ou campos das informações trocadas, podendo ocasionar uma incompatibilidade com o restante já programado na *engine* corrente. Após o componente ter sido validado, é possível fazer a desativação do mesmo, e partir para a reprogramação do próximo.

Opcionalmente, a substituição concede a oportunidade de fazer melhorias aos componentes. Caso a necessidade de troca de *engine* tenha surgido a partir da falta de atualizações, esta pode estar com uma tecnologia muito antiga quando considerada a época atual. Isto significa que ao reprogramar cada componente na *engine* nova, é possível fazer melhorias técnicas ou inserção de novas funcionalidades para o simulador, melhorando aquele que já existe.

A substituição deve ocorrer de forma gradual até que todas as funcionalidades tenham sido reconstruídas e desativadas da *engine* antiga. Após este resultado, não é mais necessário o uso dos adaptadores, os quais podem ser removidos, resultando no simulador completamente reconstruído na nova tecnologia.

4 ADAPTADORES

O uso de adaptadores facilita a abordagem por partes. Sendo o centralizador de toda a comunicação e o encarregado de distribuir as mensagens aos diversos componentes do simulador, o adaptador é responsável pelo trabalho de manter o sistema interoperante enquanto há duas *engines* em funcionamento durante todo o processo de substituição.

Neste capítulo, o modelo de comunicação é apresentado na seção 4.1, enquanto que a seção 4.2 mostra desafios de compatibilidade que podem surgir com diferentes sistemas e *engines*. Por fim, a seção 4.3 comenta sobre os componentes do simulador e sua preparação durante o processo de substituição.

4.1 COMUNICAÇÃO

Uma possibilidade de comunicação para a abordagem a ser pensada é a modificação do módulo de comunicação do simulador, inserindo nele meios de fazer o envio das mensagens à outra *engine*. Esse caminho tornaria o trabalho mais complexo, sendo necessário inserir código adicional ao módulo e podendo comprometer aquilo que já existe. Por esse motivo, adotou-se o método de adaptadores para realizar a comunicação de maneira mais simples, baseado em *plug and play*.

Para tratar desta comunicação foi escolhido para uso o padrão *Data Distribution Service*. Quando comparado o DDS ao HLA – outro padrão, popular entre simuladores militares – existem algumas semelhanças, como a arquitetura *Publish-Subscribe*, a atualizações de objetos por instância, e o padrão de API para portabilidade. Contudo, o DDS possui vantagens sobre o HLA em termos de políticas QoS e na distribuição de dados. Além disso, possui um modelo de dados fortemente tipado, que facilita a leitura e escrita e torna mais fácil a manipulação dos dados, enquanto o HLA deixa a tarefa de *marshalling* para o desenvolvedor (JOSHI; GERARDO-PARDO; CASTELLOTE, 2006).

O DDS é voltado à tempo real e tender a influenciar pouco na lógica do jogo por possuir baixa latência. Além disso, ele trás outros benefícios, como a arquitetura *Publish-Subscribe*, mencionada anteriormente, que é flexível e de fácil escalabilidade, políticas QoS diversificadas, além de ser eficiente (RF WIRELESS WORLD, 2012). Outro ponto importante é que suas principais implementações possuem licença livre, isentando o pagamento de taxas pelo seu uso.

Por outro lado, possui desvantagens ao consumir mais largura de banda que outros protocolos, ter suas políticas QoS restritas apenas aos ambientes DDS, ou não ter interface para *web services*. Estas são desvantagens que não trazem prejuízos à abordagem,

tornando o DDS uma boa opção de uso. Contudo, o foco do trabalho não está no uso do DDS, mas sim na construção dos adaptadores, que podem ser abordados também com outros padrões de comunicação.

4.2 COMPATIBILIDADE

Um dos motivos para o problema apresentado neste trabalho é a descontinuação da *game engine*, causada pela falta de atualizações. Considerando que este tenha sido o fator determinante para a substituição, é provável que o sistema esteja ultrapassado e problemas de compatibilidade podem ser enfrentados durante a substituição. Um exemplo é um sistema operacional antigo que limita a atualização de bibliotecas em um dado tempo, impedindo que versões mais recentes sejam instaladas.

A única biblioteca necessária para uso nos adaptadores, e que deve ser instalada no sistema do simulador corrente, é o DDS. Este, possui várias dependências que precisam estar atualizadas para que tudo funcione corretamente. Caso o sistema operacional seja antigo, é possível que as versões de suas dependências estejam abaixo do requerido para o funcionamento. Então, é necessário fazer o uso de versões mais antigas do DDS, que aceitem dependências nas versões presentes no sistema do simulador corrente. Em contrapartida, funcionalidades adicionadas nas versões mais recentes do DDS não poderão ser aproveitadas.

Além do problema de compatibilidade de sistemas ou bibliotecas, outro empecilho pode ser a linguagem. Há diversas *game engines* no mercado, com diferentes opções de linguagens de programação. Pode-se citar o Unity que trabalha com C#, o Delta3D que usa C++, ou outras que possuem mais de uma linguagem, como é o caso da Unreal Engine, que é compatível com C# e C++, entre outras. Com isso, pode ser preciso a reconstrução do simulador em uma outra linguagem de programação, necessitando de cuidados extras tanto para a comunicação, como para a reimplementação dos componentes.

A abordagem apresentada possibilita isto; o DDS é um facilitador, uma vez que possui implementações em diversas linguagens que são compatíveis entre si, permitindo que um adaptador seja escrito em C++ e outro em Java, por exemplo.

Caso o simulador seja reconstruído em uma linguagem diferente da original, é necessário que os componentes possam ser recriados na nova linguagem sem trazer prejuízos ao simulador. Bibliotecas usadas na versão original devem ser substituídas por outras equivalentes ou recriadas, visando manter a fidelidade e obedecer aos requisitos existentes no simulador.

4.3 COMPONENTES

A separação dos componentes do simulador necessita ser feita com atenção. Algumas partes podem estar explicitamente separadas, tais como classes próprias do simulador que não tem relação com a *engine*, criadas para prover funções necessárias à simulação. Tomando como exemplo um simulador militar, pode-se citar o módulo que controla um veículo, ou o que realiza os cálculos de tiro. Esses componentes são mais fáceis de serem portados para a nova *engine*, uma vez que o trabalho é reimplementar a lógica no novo simulador sem se preocupar com amarras à antiga *engine*.

Em alguns casos, o componente pode ser dependente de outro, como é o caso de um módulo de armamentos, que depende do módulo de cálculos de tiro para saber se um dado tiro acertou o alvo. Um estudo completo e prévio do simulador deve ser feito antes de começar a substituição, de modo a analisar essas dependências e organizar a melhor ordem para a reconstrução. Em alguns casos, tendo dois ou mais componentes dependentes entre si, se torna necessária suas reimplementações em modo paralelo para que a substituição ocorra ao mesmo tempo.

Para elementos que estão mais conectados à *engine*, não é possível assumir que estes são um componente como os demais. Uma funcionalidade presente em todas *game engines* é o motor de física, a tornando amarrada à *engine* e necessita de um tratamento adicional durante a substituição dos componentes. Deve ser testada a viabilidade de inserir a física da nova *engine* aos elementos que a utilizam. Caso necessário, os componentes gráficos devem ser modelados e desenvolvidos tendo como base a nova *engine*. Este é um trabalho que deve ser preciso para que não ocorra problemas na integridade da simulação. Porém, essa tarefa não entra no escopo deste trabalho.

5 DESENVOLVIMENTO DOS ADAPTADORES

Os adaptadores são apresentados e discutidos neste capítulo. As *game engines* Delta3D e Unity foram escolhidas, voltadas ao simulador onde deve ser realizado os testes práticos, tendo por objetivo a substituição do descontinuado Delta3D por Unity. A maior parte foi desenvolvida em C++, com exceção dos *scripts* Unity, em C#. O OpenSplice DDS é usado para o desenvolvimento da abordagem, foi escolhido por ser a implementação DDS mais completa e popular do mercado (ANGELO CORSARO, 2009), contando com amplo suporte e diversos materiais na Internet.

Os adaptadores foram construídos independente de simulador, ou seja, sem qualquer modificação em códigos internos do sistema para o qual foi projetado. Este é um dos objetivos principais da metodologia apresentada, já que diminui a complexidade e o risco de adição de *bugs* em funcionalidades já existentes. A camada de abstração, apesar de ser opcional, é desenvolvida para diminuir o trabalho na construção dos adaptadores e permitir um visual mais limpo em seus códigos, seguindo a metodologia proposta. Os tipos de mensagens do Delta3D foram usadas para fins de desenvolvimento.

O capítulo está disposto em quatro seções. A seção 5.1 apresenta o tratamento das mensagens para viabilizar a interoperabilidade entre a rede das duas *engines*. A camada de abstração ao DDS é discutida na seção 5.2. As seções 5.3 e 5.4 apresentam o desenvolvimento dos adaptadores, para Delta3D e Unity, respectivamente.

5.1 MENSAGENS

Partindo do Delta3D como *engine* a ser substituída, é necessário que suas mensagens sejam recriadas em IDL, juntamente com as mensagens específicas do simulador. O formato IDL é usado pelo DDS para declarar tipos de mensagens que serão usados durante a comunicação. Ao compilar, são gerados códigos na linguagem destino da aplicação, no nosso caso, o C++.

Os tipos de mensagens reimplementados em IDL devem possuir os mesmos campos do tipo original para que não haja perda de informações nesta comunicação entre *engines*. Um exemplo de código IDL, apresentando dois tipos de mensagens Delta3D, é mostrado na Figura 5.1.

Contudo, pode haver casos de tipos de mensagens com atributos complexos e que não são possíveis de serem representados em IDL, como é o caso de objetos da própria *engine* sendo passados pela mensagem. Um exemplo é a mensagem *GameEventMessage* do Delta3D, no qual seu único campo é um ponteiro para um objeto *GameEvent* que não pode ser representado em IDL por não ser um tipo primitivo.

Figura 5.1 – Implementação de mensagens Delta3D em IDL

```
module Message
{
    struct ActorUpdate
    {
        string name;
        string actorTypeName;
        string actorTypeCategory;
        string updateParameter;
        long actorType;
        string prototypeName;
        long prototypeID;
    };
    #pragma keylist ActorUpdate name

    struct LogAvailableLogs
    {
        string<1024> logList;
    };
    #pragma keylist LogAvailableLogs logList;

    ...
}
```

Fonte: Autor

Quando isso acontece, é necessário reprogramar na nova *engine* a classe do objeto em questão, o que também se torna útil para caso este objeto seja usado por algum componente futuro. Após a reprogramação, os atributos deste objeto são inseridos na IDL no lugar dele. Caso um dos atributos seja outro objeto, a mesma regra deve ser aplicada, e assim sucessivamente até que tenham apenas tipos primitivos.

Sabendo disso, é necessário um tratamento adicional ao enviar e receber tipos de mensagens que possuem este problema de complexidade. O adaptador, ao recriar uma mensagens desse tipo, precisa instanciar o objeto assim como no tipo de dado original, e então copiar para ele os atributos recebidos na mensagem.

5.2 CAMADA DE ABSTRAÇÃO

O uso de uma camada de abstração ao padrão DDS facilita o desenvolvimento, uma vez que implementando esta camada, ela serve para ambos adaptadores. Além de abstrair o DDS ao desenvolvimento do adaptador, ela pode possuir características mais específicas para facilitar o uso no simulador que está sendo trabalhado, como o uso de políticas QoS comuns pré-definidas, ou partições para organizar o envio das mensagens de acordo com a função de um cliente na simulação.

Além disso, criar uma classe para um participante ou tópico ajuda com que o código possa ser reutilizado e se mantenha de forma organizada, deixando os adaptadores com um visual mais limpo e sem complexidade, para que o foco do código se torne o tratamento e conversão das mensagens, independente da sintaxe do DDS.

Há duas classes principais que compõem a camada de abstração: uma para um participante e outra para tópicos. O participante armazena objetos únicos a um cliente, como o domínio, o *publisher* e o *subscriber*. Esses objetos são usados para todas as mensagens trocadas, não importando o tipo da mensagem. A Figura 5.2 mostra algumas declarações presentes na classe.

Figura 5.2 – Trecho de código do participante DDS na camada de abstração

```
...

class APIDDS {
private:
    dds::domain::DomainParticipant dp = dds::core::null;
    dds::pub::Publisher pub = dds::core::null;
    dds::sub::Subscriber sub = dds::core::null;

...

    // Altera o domínio em que as mensagens estão sendo trocadas
    bool ChangeDomain(std::string domain);

    // Define um novo padrão de política QoS para o publisher
    bool SetPublisherQoS(std::string presentation,
        std::string partition, std::string group_data);

    // Define um novo padrão de política QoS para o subscriber
    bool SetSubscriberQoS(std::string presentation,
        std::string partition, std::string group_data);

...
```

Fonte: Autor

No *constructor* é feita a criação dos objetos da classe, que não se alteram depois disso. Há um método para alterar o domínio em que o participante se encontra, caso haja uma separação de domínios por tipos de clientes, por exemplo. A política QoS para o *publisher* e o *subscriber* pode ser alterada a qualquer momento também. Esta é a classe principal da camada de abstração, a qual deve haver uma instanciação por cliente.

A segunda classe é a do tópico. Uma vez que o número de tópicos é, no mínimo, igual ao número de tipos de mensagens, e cada tópico deve possuir seu próprio *DataReader* e *DataWriter*, é necessário uma classe para organizar tudo isso sem que haja inúmeras instâncias de cada objeto. Cada tipo deve possuir uma instância desse objeto, o qual será usado para enviar e receber mensagens. Um trecho da classe é apresentada na Figura 5.3.

A classe faz o uso de *template* do C++, já que precisa ser específica para cada tipo de mensagem em questão. Há o método para envio e recebimento das mensagens daquele tópico. Para o recebimento, é necessário passar por parâmetro um *vector* onde serão guardadas as mensagens recebidas, como forma de retorno. Além disso, um inteiro para determinar o número máximo de mensagens desejadas, e o modo de leitura – manter as mensagens no *buffer* de leitura ou retirar dele –. O retorno de todas as funções é um booleano que indica se houve êxito no comando.

Figura 5.3 – Trecho de código do tópico DDS na camada de abstração

```
...

template <typename T>
class APITopicDDS : public virtual dds::sub::DataReaderListener<T> {
private:
    dds::topic::Topic<T> topic = dds::core::null;
    dds::pub::DataWriter<T> dw = dds::core::null;
    dds::sub::DataReader<T> dr = dds::core::null;

    APIDDS participant;

...

    // Define o participante, onde há o domínio, pub e sub
    bool SetParticipant(APIDDS participant);

    // Envia uma mensagem
    bool WriteMessage(T message)

    // Recebe uma mensagem
    bool ReadMessage(std::vector<T>& messages, int max, int mode)

    // Define um novo padrão de política QoS para tópicos
    bool SetTopicQoS(std::string durability,
        std::string reliability, std::string deadline);

...
```

Fonte: Autor

5.3 ADAPTADOR DELTA3D

Apesar de possuir o módulo *dtNet*, específico para a comunicação e que utiliza a biblioteca GNE, o adaptador Delta3D foi desenvolvido baseado na comunicação alternativa implementada no módulo *dtGame*, uma vez que este é modelo o usado no simulador onde serão realizados os testes. Seu funcionamento é simples: uma mensagem é enviada pelo método *SendMessage* do *Game Manager*, e então a mensagem é recebida no método *ProcessMessage* de todos os clientes.

De modo a filtrar os tipos de mensagem de interesse de cada cliente, é implementada uma sequência de estruturas condicionais dentro do método *ProcessMessage* que, ao receber uma mensagem, entra na sua condicional correspondente para fazer o processamento.

O adaptador deve conter condicionais para todas as mensagens trocadas dentro da rede da *engine*, uma vez que o outro lado deve receber toda a comunicação como se fosse parte do mesmo sistema. Dentro de cada condicional, a mensagem é convertida do tipo Delta3D para DDS, e então escrita no tópico correspondente para ser enviada à outra *engine*. A Figura 5.4 mostra o processamento de um tipo de mensagem.

As mensagens enviadas pela outra *engine* que foram recebidas devem ser lidas, as quais serão guardadas em um *vector* que é percorrido logo na sequência. Cada mensagem possui seus campos copiados para o tipo do Delta3D. Então, a mensagem é enviada pela rede do Delta3D. Dessa forma, a mensagem que foi enviada pela outra *engine* percorre a rede interna e é distribuída à todos seus clientes, como se fosse uma mensagem

Figura 5.4 – Envio de uma mensagem pelo adaptador Delta3D

```
void ProcessMessage(const Message& message) {
    ...
    if (message.GetMessageType() ==
        dtGame::MessageType::INFO_TIME_CHANGED) {

        MessageDDS::InfoTimeChanged msg;
        msg.timeScale = message.timeScale;
        msg.simulationTime = message.simulationTime;
        msg.simulationClockTime = message.simulationClockTime;

        timeChangedTopic.WriteMessage(msg);
    }
    else if (message.GetMessageType() ==
        dtGame::MessageType::REQUEST_SET_TIME) {
        ...
    }
    else if (message.GetMessageType() ==
        dtGame::MessageType::REQUEST_PAUSE) {
        ...
    }
        ...
```

Fonte: Autor

enviada por um cliente Delta3D próprio. A Figura 5.5 mostra este processamento.

Figura 5.5 – Recebimento de mensagens pelo adaptador Delta3D

```
...
success = timeChangedTopic.ReadMessage(messages, 10, 1);

if (success) {
    while(!messages.empty()) {
        MessageDDS::InfoTimeChanged msg = messages.back();

        dtCore::RefPtr<Message> msg =
            CreateMessage(dtGame::MessageType::INFO_TIME_CHANGED);
        msg->timeScale = msg.timeScale;
        msg->simulationTime = msg.simulationTime;
        msg->simulationClockTime = msg.simulationClockTime;

        gameManager->SendMessage(msg);

        messages.pop_back();
    }
}
...
```

Fonte: Autor

5.4 ADAPTADOR UNITY

Para o adaptador Unity, a lógica segue a mesma do adaptador Delta3D. As mensagens recebidas têm seus campos transferidos para uma mensagem Unity, e enviada ao servidor que encaminha aos clientes interessados. Porém, uma vez que o código DDS se encontra na linguagem C++ e o Unity em C#, não é possível utilizar diretamente o DDS, o que inclui os seus tipos de dados. Com isso, torna necessário criar uma classe para cada

tipo de mensagem diferente, onde serão guardadas as informações para serem repassadas ao DDS. A Figura 5.6 mostra a implementação de uma mensagem.

Figura 5.6 – Implementação de uma mensagem em Unity

```
public class InfoTimeChangedMessage : MessageBase
{
    public float timeScale;
    public double simulationTime;
    public double simulationClockTime;
}
```

Fonte: Autor

A incompatibilidade de linguagem de programação entre Unity e o DDS usado também impede que a camada de abstração seja usada de forma direta. Para contornar isso, é desenvolvida uma *unmanaged* DLL que deve ser utilizada no projeto como plugin para prover as funções necessárias para a comunicação DDS. O uso de uma DLL como plugin no Unity possibilita que códigos de linguagens diferentes de C# sejam importados e usados em um projeto, como é o caso da camada de abstração ao DDS implementada em C++. Para evitar a reimplementação em C#, este método foi explorado.

Para que o código C++ seja compilado e guardado na DLL, é necessário que ele esteja com sintaxe baseada em C. Com isso, alguns ajustes na API foram necessários para a compilação em DLL. Após a importação para o projeto Unity, suas funções devem ser declaradas para serem usadas pelos *scripts* C#. Então, é possível chamar a função normalmente como se fosse um método nativo de C#. A Figura 5.7 mostra a forma com que os métodos foram declarados para que fossem compilados em DLL.

Figura 5.7 – Criação de DLL para uso em Unity

```
#ifdef APIDDSDLL_EXPORT
#define APIDDSDLL_API __declspec(dllexport)
#else
#define APIDDSDLL_API __declspec(dllimport)
#endif

extern "C" APIDDSDLL_API bool ChangeDomain(static char* domain);

extern "C" APIDDSDLL_API bool SetPublisherQoS(static char* presentation,
    static char* partition, static char* group_data);

...
```

Fonte: Autor

6 TESTES PRÁTICOS

A partir do desenvolvimentos dos adaptadores foram realizados testes práticos em um simulador para treinamento militar. Este simulador foi desenvolvido em Delta3D, *game engine* descontinuada e sem suporte; e com base na abordagem apresentada, tem por objetivo ser reconstruído na *engine* Unity. Então, com o cenário de substituição da tecnologia do Delta3D por Unity foram desenvolvidos os adaptadores para as duas *engines*, apresentados anteriormente no capítulo 5. Eles foram usados nos testes práticos para verificar a viabilidade da aplicação da abordagem.

O simulador dos testes se apresenta amarrado ao sistema operacional Debian 6, versão antiga que limita a atualização de pacotes essenciais para o desenvolvimento. Com isso, foi necessário trabalhar com versões de pacotes antigas no lado do Debian, enquanto no outro lado, do Windows, é preciso que tudo esteja o mais atualizado possível, já que esse é o objetivo do processo: rejuvenescer. Essa diferença de versões fez com que fosse necessário o uso de uma versão mais antiga do OpenSplice DDS, o que pode tirar algumas funcionalidades presentes nas versões mais recentes, mas que não devem trazer problemas para o objetivo do trabalho.

A arquitetura do simulador é bem dividida em clientes e componentes, o que ajuda a aplicação da substituição por partes. Todo cliente possui os seus componentes, e cada componente é composto por uma ou mais classes. Ao iniciar um cliente do simulador, ele cria os componentes correspondentes àquele cliente. Deste modo, os componentes cuja reimplementação é necessária para o funcionamento de um determinado cliente, na nova *engine* estão demonstrados explicitamente.

O simulador implementa seu próprio sistema de *Publish-Subscribe*, tornando toda a lógica de comunicação dos componentes voltada para este padrão, facilitando a integração com DDS. As mensagens trocadas entre componentes são, em partes, próprias do Delta3D, juntamente à mensagens específicas do simulador.

Todas mensagens específicas do simulador foram criadas a partir do método de declarar mensagens disponibilizado pela *engine* Delta3D. As mensagens criadas desta forma não possuem atributos complexos, ou seja, puderam ser facilmente criadas também como tipos de dados do DDS. Por outro lado, algumas mensagens Delta3D precisaram de um esforço adicional para a recriação de tipo de dados mais complexos, conforme mostradas na seção 5.3.

Além disso, foi possível analisar as dependências encontradas nos componentes do sistema. O Delta3D possui um *Game Manager* que é usado por quase todos módulos do simulador. Ele é responsável por encaminhar as mensagens, armazenar os componentes e modelos que estão no cenário etc. Com isso, é essencial que seja recriado no outro lado antes de começar a substituição modular, porém isso não entra no escopo deste trabalho.

Uma vez que o foco inicial é o desenvolvimento dos adaptadores, os testes buscaram verificar unicamente a efetividade da solução em termos de ser possível fazer a troca de mensagens entre *engines* diferentes, mantendo o sistema interoperável. Esta é uma das linhas a serem aprofundadas para, posteriormente, alcançar o rejuvenescimento do simulador a partir da troca de *engine*.

Após as dificuldades apresentadas anteriormente, a integração do adaptador Delta3D ao simulador foi feita, junto do adaptador Unity a um projeto rodando no Windows. Para testar a comunicação entre adaptadores o componente de gerenciamento de mapas foi escolhido, uma vez que ele é o primeiro passo para se iniciar uma nova simulação no sistema.

A partir da implementação em IDL das mensagens utilizadas pelo componente, juntamente ao desenvolvimento de uma interface para escolha de mapa por parte do usuário, foram feitos os testes de comunicação. Foi testado o envio de comandos de escolha de mapa para o adaptador Delta3D presente na máquina da simulação, o qual recebeu e distribuiu para o componente de mapas que executou normalmente como um comando da simulação.

De modo a testar a bidirecionalidade da comunicação, foi feito um teste ao conectar o cliente, em Unity, com o servidor do simulador em Delta3D. A partir do cliente, foi enviada uma mensagem para comunicar o servidor de que havia um novo cliente, testando a comunicação Unity para Delta3D. No recebimento, o lado Delta3D fez o envio para o lado Unity de uma confirmação de conexão. Tais experimentos asseguram que é possível fazer uma comunicação para ambas direções.

7 CONCLUSÃO

Este trabalho propôs, a partir da apresentação de uma metodologia, a construção de adaptadores de comunicação para facilitar a substituição da *game engine* de um sistema de simulação distribuído. O uso desta técnica deve ajudar nos casos em que se torna necessária a troca de *game engine*, por motivos de falta de atualizações e suporte, busca por um desempenho técnico e/ou gráfico superior à *engine* corrente, ou demais fatores, sem a necessidade de interromper as atividades do simulador durante o período de substituição.

A abordagem apresentada demonstra que é possível estruturar de forma fácil o processo de substituição com adaptadores *plug and play* para realizar a troca de mensagens entre diferentes *game engines*. Este é o ponto chave da metodologia, uma vez que é preciso que os sistemas, independente de *engine*, trabalhem de maneira interoperante para que o simulador não deixe de atender à demanda de uso durante o período de substituição dos componentes.

Os adaptadores desenvolvidos neste trabalho se mostraram capazes de desempenhar sua função, garantindo que todo processo de substituição ocorra de maneira mais simples, uma vez que não se torna necessário fazer alterações no código-fonte do simulador no qual está sendo trabalhado, apenas construir o adaptador que reconheça e traduza suas mensagens.

O uso desta técnica trás benefícios também quando se trata de um simulador com informações confidenciais em seu código já que, não sendo necessário alterar códigos já existentes, os desenvolvedores dos adaptadores apenas precisam ter conhecimento dos tipos de mensagens que são trocadas entre componentes, deixando partes com informações sensíveis apenas àqueles que vão reimplementar tal trecho na nova *engine*.

Por conta de ter seu foco na construção dos adaptadores, este trabalho se limita a testar a comunicação para a metodologia introduzida, restringindo os testes a um cenário simples, uma vez que não houveram estudos aprofundados em cima das etapas de compartimentação e preparação, para que fosse possível concretizar a substituição integral de um módulo.

Adicionalmente, o trabalho se limita no desenvolvimento em cima das *game engines* Delta3D e Unity, não sendo feitos testes em quaisquer outras. A aplicação da metodologia de substituição de *engine* para um simulador se restringe a novos estudos em cima deste, de modo a avançar nas etapas restantes que não foram abordadas neste trabalho.

Assim, a técnica de comunicação proposta abre portas para a aplicação da metodologia apresentada focada em rejuvenescer sistemas de simulação distribuídos. Os adaptadores são a essência da abordagem e o foco deste trabalho, portanto, trabalhos adicionais ficam em abertos para concretizar a aplicação completa da metodologia.

7.1 TRABALHOS FUTUROS

Sendo o mecanismo de comunicação o foco deste trabalho, outros pontos devem ser estudados com maior profundidade para complementar a metodologia. A preparação dos modelos para o mundo virtual é uma etapa que pode ser abordada em um trabalho futuro. O ambiente de simulação virtual possui variados modelos, desde terrenos, personagens, veículos, árvores, componentes gráficos, construções etc. Cada um pode possuir seu formato, e até alguns ligados à *engine*, que deve ser buscada a melhor forma de se fazer a reconstrução sem perder a fidelidade da simulação.

Outro ponto crucial, e que merece atenção, é a compartimentação dos componentes. Este trabalho parte da premissa que a lógica de negócio pode ser facilmente modularizada, para então ser reimplementada no outro lado. Porém, podem haver simuladores distribuídos pouco organizados e que possuam mais amarras para impedir esta compartimentalização, necessitando de um estudo sobre as diferentes possibilidades e qual a melhor forma de fazer a modularização em cada caso.

Adicionalmente, um trabalho em cima das *game engines* pode ser feito buscando a melhor forma de desamarrar componentes que estejam ligados à ela e sendo usados em uma simulação. O motor de física, inteligência artificial, animação, entre outros, são exemplos de funcionalidades providas pela *engine* e que podem estar presentes em diversos componentes do simulador, tendo que buscar formas de prover as mesmas funcionalidades para a nova *engine*.

REFERÊNCIAS BIBLIOGRÁFICAS

AEBERSOLD, M. The history of simulation and its impact on the future. **AACN Advanced Critical Care**, v. 27, p. 56–61, 02 2016.

ANGELO CORSARO. **10 Reasons for Choosing OpenSplice DDS**. [S.l.], 2009. Acesso em 04 fev. 2021. Disponível em: <https://pt.slideshare.net/Angelo.Corsaro/10-reasons-for-choosing-opensplice-dds>.

CORSARO, A. **The Data Distribution Service Tutorial**. [S.l.: s.n.], 2014.

DARKEN, R.; MCDOWELL, P.; JOHNSON, E. Projects in vr: The delta3d open source game engine. **Computer Graphics and Applications, IEEE**, v. 25, p. 10 – 12, 06 2005.

DOD, U. Dod modeling and simulation (m&s) glossary. 1998.

EUGSTER, P. et al. The many faces of publish/subscribe. **ACM Comput. Surv.**, v. 35, p. 114–131, 06 2003.

FUJIMOTO, R. M. **Parallel and distributed simulation systems**. [S.l.]: Citeseer, 2000. v. 300.

GREGORY, J. **Game engine architecture**. [S.l.]: crc Press, 2018.

HEERMANN, D. W. Computer-simulation methods. In: **Computer Simulation Methods in Theoretical Physics**. [S.l.]: Springer, 1990. p. 8–12.

JOSHI, R.; GERARDO-PARDO, P.; CASTELLOTE, P. A comparison and mapping of data distribution service and high-level architecture. 01 2006.

KRAKOWIAK, S. **Middleware Architecture with Patterns and Frameworks**. 2007.

LEWIS, M.; JACOBSON, J. Game engines in scientific research. **Communications of the ACM**, v. 45, p. 27–31, 01 2002.

MCDOWELL, P. et al. Delta3d: a complete open source game and simulation engine for building military training systems. **The Journal of Defense Modeling and Simulation**, SAGE Publications Sage UK: London, England, v. 3, n. 3, p. 143–154, 2006.

____. Delta3d: A complete open source game and simulation engine for building military training systems. **The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology**, v. 3, p. 143–154, 07 2006.

OBJECT MANAGEMENT GROUP. **Data Distribution Services (DDS)**. [S.l.], 2020. Acesso em 10 dez. 2020. Disponível em: <https://www.omg.org/spec/DDS/1.4/PDF>.

____. **DDS Portal: Data Distribution Services**. [S.l.], 2020. Acesso em 10 dez. 2020. Disponível em: <https://www.dds-foundation.org/>.

PAGE, E.; BRIGGS, R.; TUFAROLO, J. Toward a family of maturity models for the simulation interconnection problem. 01 2004.

PARDO-CASTELLOTE, G. Omg data-distribution service: architectural overview. In: . [S.l.: s.n.], 2003. v. 1, p. 242– 247 Vol.1. ISBN 0-7803-8140-8.

PUDER, A.; RÖMER, K.; PILHOFER, F. **Distributed systems architecture: a middleware approach**. [S.l.]: Elsevier, 2011.

RF WIRELESS WORLD. **Advantages of DDS protocol**. [S.l.], 2012. Acesso em 04 fev. 2021. Disponível em: <https://www.rfwireless-world.com/Terminology/Advantages-and-Disadvantages-of-DDS-protocol.html>.

SCHLESSELMAN, J.; PARDO-CASTELLOTE, G.; FARABAUGH, B. Omg data-distribution service (dds): architectural update. In: . [S.l.: s.n.], 2004. p. 961 – 967 Vol. 2. ISBN 0-7803-8847-X.

STEEN, M. van; TANENBAUM, A. S. A brief introduction to distributed systems. **Computing**, Springer, v. 98, n. 10, p. 967–1009, 2016.

TANENBAUM, A. S.; STEEN, M. V. **Distributed systems: principles and paradigms**. [S.l.]: Prentice-Hall, 2007.

TAYLOR, S. J. Distributed simulation: state-of-the-art and potential for operational research. **European Journal of Operational Research**, Elsevier, v. 273, n. 1, p. 1–19, 2019.

UNITY TECHNOLOGIES. **Unity User Manual**. [S.l.], 2020. Acesso em 10 dez. 2020. Disponível em: <https://docs.unity3d.com/2020.1/Documentation/Manual/index.html>.

VBS4. **VBS4**. [S.l.], 2020. Acesso em 10 dez. 2020. Disponível em: <https://vbs4.com/>.