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

Rodrigo Pincolini Amaral

UMA COMPARAÇÃO DE TECNOLOGIAS PARA A COMUNICAÇÃO ENTRE SIMULADORES DISTRIBUÍDOS EM AMBIENTE UNITY

Santa Maria, RS
2021

Rodrigo Pincolini Amaral

UMA COMPARAÇÃO DE TECNOLOGIAS PARA A COMUNICAÇÃO ENTRE SIMULADORES DISTRIBUÍDOS EM AMBIENTE UNITY

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

ORIENTADOR: Prof. Raul Ceretta Nunes

COORIENTADOR: Diogo Otto Kunde

480
Santa Maria, RS
2021

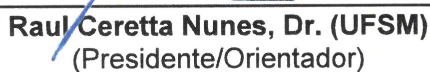
©2021
Todos os direitos autorais reservados a Rodrigo Pincolini Amaral. A reprodução de partes ou do todo deste trabalho só poderá ser feita mediante a citação da fonte.
End. Eletr.: rpamaral@inf.ufsm.br

RODRIGO PINCOLINI AMARAL

UMA COMPARAÇÃO DE TECNOLOGIAS PARA A COMUNICAÇÃO ENTRE SIMULADORES DISTRIBUÍDOS EM AMBIENTE UNITY

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

Aprovado em 10 de Fevereiro de 2021:



Raul Ceretta Nunes, Dr. (UFSM)
(Presidente/Orientador)

Diogo Otto Kunde, Me. (UFSM)
(Coorientador)

Mateus Beck Rutzig, Dr. (UFSM)

Cesar Tadeu Pozzer, Dr. (UFSM)

Santa Maria, RS
2021

AGRADECIMENTOS

Agradeço a toda minha família pela educação, apoio, incentivo e insistência que me proporcionaram mesmo com todas as dificuldades. Agradeço especialmente a minha mãe, Liliana Pincolini, e ao meu pai, Fernando Amaral. Também agradeço a minha segunda mãe, a tia Lourdes Maria Pincolini e ao meu segundo pai, o padrasto Alexandre Lucca.

Agradeço também a minha companheira, Larissa Dornelles Schiar, que me ajudou mesmo nos momentos difíceis durante a pandemia de COVID-19.

Agradeço ao Gabriel Cardoso Santos e ao professor Raul Ceretta Nunes, ambos que possibilitaram minha participação no projeto e crescimento profissional.

Agradeço pela orientação do professor Raul Ceretta Nunes e do Diogo Otto Kunde que me ajudaram muito a possibilitar a realização deste trabalho mesmo em situação de pandemia. Também agradeço a banca, os professores Cesar Tadeu Pozzer e Mateus Beck Rutzig pelo esforço e tempo em avaliar este trabalho.

Agradeço a todos colegas do projeto que facilitaram e ajudaram na minha inserção no projeto, especialmente ao Gabriel Cardoso e ao Diogo Otto Kunde. Também agradeço aos colegas de faculdade que me ajudaram em algum momento, especialmente aos amigos Lucas e João Davi aos quais compartilhei bons momentos.

Agradeço ao projeto SIS-ASTROS e a Universidade Federal de Santa Maria.

RESUMO

UMA COMPARAÇÃO DE TECNOLOGIAS PARA A COMUNICAÇÃO ENTRE SIMULADORES DISTRIBUÍDOS EM AMBIENTE UNITY

AUTOR: Rodrigo Pincolini Amaral
ORIENTADOR: Raul Ceretta Nunes
COORIENTADOR: Diogo Otto Kunde

Quando simuladores precisam trocar dados entre computadores distintos é necessário utilizar alguma tecnologia que permita a conexão entre as instâncias distribuídas da simulação. Estas tecnologias podem variar entre APIs para a simples troca de mensagens, até padrões de interoperabilidade que possuem diversas funcionalidades com abordagens definidas. Com foco em simulações desenvolvidas na *engine* Unity, este trabalho realiza a comparação de quatro tecnologias de comunicação de dados: o *Unity Networking* (UNET), o suporte de rede interna da Unity, o *Tasharem Networking and Serialization Tools* (TNET 3), um *asset* para suporte à programação em rede na Unity, o *High Level Architecture* (HLA), um dos principais padrões de interoperabilidade de simuladores, e o *Data Distribution Service* (DDS), um dos principais padrões de interoperabilidade para aplicações de tempo real. O trabalho destaca os prós e contras das tecnologias, levando em consideração o projeto de um simulador distribuído desenvolvido em Unity e aspectos teóricos e práticos das tecnologias. O resultado aponta que a adoção de uma tecnologia pertencente ao ambiente Unity, como a TNET3, é a mais benéfica no contexto do simulador estudado.

Palavras-chave: Simulação. Simulação Distribuída. Interoperabilidade. Unity. HLA. DDS. UNET. TNET3

ABSTRACT

A COMPARISON OF TECHNOLOGIES FOR COMMUNICATION BETWEEN UNITY-BASED DISTRIBUTED SIMULATORS

AUTHOR: Rodrigo Pincolini Amaral
ADVISOR: Raul Ceretta Nunes
CO-ADVISOR: Diogo Otto Kunde

When simulators need to exchange data between different computers, belonging to the same simulator or to different simulators, some technology is used that allows the connection between the distributed instances of the simulation. These technologies can vary from APIs for the simple exchange of messages to interoperability standards that have several functionalities with defined approaches. Focusing on simulations developed on the Unity Engine, this work compares four interoperability technologies: *Unity Networking* (UNET), Unity's internal network support, *Tasharem Networking and Serialization Tools* (TNET 3), a Unity asset to support network programming at Unity, *High Level Architecture* (HLA), one of the main simulator interoperability standards, and *Data Distribution Service* (DDS), one of major interoperability standards for real-time applications. The work aims to highlight the pros and cons of the technologies taking into account the design of a distributed simulator developed in Unity. The work results highlight that the adoption of a technology belonging to the Unity environment, such as TNET3, is the most adequate in the context of the studied simulator.

Keywords: Simulation. Distributed Simulation. Interoperability. Unity. HLA. DDS. UNET. TNET3

LISTA DE FIGURAS

Figura 2.1 – Cockpit de voo simulado 12
Figura 2.2 – Simulador construtivo SWORD 15
Figura 2.3 – Camada de *sotfware* chamada *middleware* 16
Figura 2.4 – Organização entre simuladores no HLA 18
Figura 2.5 – Exemplo de FOM 19
Figura 2.6 – Exemplo de ilustrativo de código gerado pelo DevStudio 20
Figura 2.7 – Domain no DDS 22
Figura 2.8 – Ilustação de código arquivo IDL 23
Figura 2.9 – *GameObject da Unity* 25
Figura 2.10 – Lógica de aplicação Unity 26
Figura 2.11 – Compartilhamento de *GameObjects* na TNET3 27
Figura 2.12 – Simulador SVTat 29
Figura 2.13 – Funcionalidade suportadas por HLA e/ou DDS 30
Figura 2.14 – HLA/DDS equivalentes 31
Figura 3.1 – Arquitetura 1 33
Figura 3.2 – Arquitetura 2 34
Figura 3.3 – Compartilhamento de objetos na Unity 36
Figura 4.1 – Fluxo de encriptação de dados na Unity 41
Figura 4.2 – Arquitetura de segurança do DDS 42
Figura 5.1 – Estrutura comum implementada utilizando as tecnologias 48
Figura 5.2 – Interface de exportação de *plugin* DLL 49
Figura 5.3 – Fluxo entre DLL e Unity 50
Figura 5.4 – Ciclo de informações usando HLA 51
Figura 5.5 – Ciclo de informações usando DDS 53
Figura 5.6 – Conversão de dados requerida pelo DDS 54
Figura 5.7 – Estrutura implementada utilizando a TNET3 56
Figura 5.8 – Peso das etapas de implementação (tempo médio de cada etapa) 58
Figura 6.1 – *Hardware* dos computadores para teste de desempenho 59
Figura 6.2 – Vazão de dados no teste de *payload* 61
Figura 6.3 – RTT no teste de *payload* 62
Figura 6.4 – Vazão de dados no teste com dados do simulador 63
Figura 6.5 – Uso de CPU do produtor de dados 64

LISTA DE ABREVIATURAS E SIGLAS

Sigla	Significado
API	Application Programming Interface
$ASTROS$	Artillery Saturation Rocket System
$DCPS$	Data Centric Publish-Subscribe
DDS	Data Distribution Service
DLL	Dynamic Link Library
DoD	Departament of Defense
FOM	Federation Object Model
$IEEE$	Institute of Electrical and Electronics Engineers
HLA	High Level Architecture
$HLAPI$	High Level API
IDL	Interface Definition Languague
IoT	Internet of Things
LVC	Live-Virtual-Constructive
OMG	Object Management Group
OMT	Object Model Template
QoS	Quality of Service
$REOP$	Reconhecimento, Escolhe e Ocupação de Posição
RTT	Round-Time Trip
RTI	Run-Time Infrastructure
$SVTat$	Simulador Virtual Tático
$T\&E$	Test and Evaluation
TCP	Transmission Control Protocol
$TNET3$	Tasharem Networking and Serialization Tools 3
UDP	User Datagram Protocol
$UNET$	Unity Networking
$UFSM$	Universidade Federal de Santa Maria
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO **9**
1.1 JUSTIFICATIVA 10
1.2 OBJETIVOS 10
1.2.1 Geral **10**
1.2.2 Específicos **10**
1.3 ORGANIZAÇÃO DO DOCUMENTO 11
2 REVISÃO BIBLIOGRÁFICA **12**
2.1 SIMULAÇÃO 12
2.1.1 Classificação de simulação **13**
2.1.1.1 Simulação viva *13*
2.1.1.2 Simulação virtual *14*
2.1.1.3 Simulação construtiva *14*
2.2 SISTEMAS DISTRIBUÍDOS 14
2.2.1 *Middleware* **15**
2.3 SIMULAÇÃO DISTRIBUÍDA 16
2.3.1 Interoperabilidade **16**
2.3.2 Padrões de interoperabilidade **17**
2.4 HIGH LEVEL ARCHITECTURE 17
2.4.1 Conceitos Básicos **17**
2.4.2 DevStudio **19**
2.5 DATA DISTRIBUTION SERVICE 21
2.5.1 Conceitos Básicos **21**
2.5.2 Geração de código **23**
2.6 ENGINE UNITY 24
2.6.1 Elementos básicos da Unity **24**
2.6.2 UNET **26**
2.6.3 TNET3 .... **27**
2.6.4 Simulador SVTat REOP **28**
2.7 TRABALHOS RELACIONADOS 29
2.7.1 Trabalho 1 .... **29**
2.7.2 Trabalho 2 .... **30**
2.7.3 Trabalho 3 .... **31**
2.7.4 Trabalho 4 .... **32**
3 PROPOSTA COMPARATIVA **33**
3.1 CONTEXTO DO SIMULADOR 33
3.2 REQUISITOS DA COMUNICAÇÃO 35
3.3 A COMPARAÇÃO DE TECNOLOGIAS 36
3.4 PLANEJAMENTO DOS TESTES DE DESEMPENHO 37
4 COMPARAÇÃO DAS PRINCIPAIS FUNCIONALIDADES ENTRE TECNOLOGIAS **39**
4.1 TECNOLOGIAS EXTERNAS E INTERNAS 39
4.2 TOLERÂNCIA A FALHAS E ESCALABILIDADE 40
4.3 SEGURANÇA 40
4.4 POLÍTICAS DE QUALIDADE DE SERVIÇO 42
4.4.1 Garantia de entrega e ordenação de mensagens **42**

4.4.2 Escopo de dados ... **42**
4.4.3 Filtro de dados ... **43**
4.4.4 *Ownership Managment* ... **43**
4.4.5 QoS exclusivas do DDS ... **44**
4.5 GERENCIAMENTO DE TEMPO ... 44
4.6 METADADOS ... 45
4.7 LICENÇA E SUPORTE ... 45
5 COMPARAÇÃO DE IMPLEMENTAÇÕES ... **47**
5.1 IMPLEMENTAÇÃO NO SVTAT REOP ... 47
5.2 USO DE PLUGIN DLL ... 48
5.3 IMPLEMENTAÇÃO HLA ... 51
5.4 IMPLEMENTAÇÃO DDS ... 53
5.5 IMPLEMENTAÇÃO TNET3 ... 56
5.6 VIABILIDADE DAS IMPLEMENTAÇÕES ... 57
6 TESTES DE DESEMPENHO ... **59**
6.1 CENÁRIO DE TESTE ... 59
6.2 TESTES DE *PAYLOAD* ... 60
6.3 TESTES COM ESTRUTURAS DO SIMULADOR ... 61
7 CONCLUSÃO ... **65**
REFERÊNCIAS BIBLIOGRÁFICAS ... **66**

1 INTRODUÇÃO

A simulação é uma imitação de um processo ou sistema do mundo real ao longo do tempo (BANKS et al., 2009). Um jogo virtual de xadrez pode ser compreendido como uma simulação de um jogo existente na vida real. Quando simuladores precisam colaborar em conjunto utiliza-se o emprego da simulação distribuída, que refere-se a conexão entre diferentes instâncias de simulação através de uma rede de computadores (FUJIMOTO, 2000).

Na simulação distribuída, a interoperabilidade entre simuladores pode ser compreendida como a capacidade de um simulador de se comunicar com outro. Usualmente para garantir a entrega de informações entre os simuladores se utiliza um protocolo de comunicação (TOLK, 2012).

Há diversos protocolos desenvolvidos por organizações diferentes, o que resulta em propostas de comunicação distintas. Assim, como um protocolo para a comunicação pode possuir funcionalidades e abordagens distintas, é necessário realizar a escolha da tecnologia correta para o simulador distribuído de modo que melhor supra seus requisitos de comunicação para seu modelo de simulação.

Neste contexto, simuladores distribuídos desenvolvidos na *engine* Unity podem utilizar diversas tecnologias para garantir a comunicação entre suas aplicações. Existem APIs internas da *engine* Unity que são desenvolvidas com o intuito de estabelecer a comunicação somente entre aplicações Unity, como a UNET (*Unity Networking*), bem como APIs desenvolvidas por terceiros como a TNET3.

Por outro lado, há também outras tecnologias que permitem não só comunicação entre aplicações Unity, mas também com aplicações desenvolvidas fora do ambiente Unity. Neste contexto existem tecnologias como o HLA (*High Level Architecture*), proposto como um padrão de propósito geral para interoperar simuladores distribuídos (SOKOLOWSKI; BANKS, 2010), e o DDS (*Data Distribution Service*), proposto como um padrão de propósito geral para interoperar aplicações em tempo real.

Em vista da importância da escolha da tecnologia para a comunicação, este trabalho visa comparar quatro tecnologias no contexto de um simulador desenvolvido no ambiente Unity, chamado SVTat REOP. Esta comparação leva em conta aspectos funcionais de cada tecnologia, avaliando qual tecnologia melhor supre os requisitos do simulador durante a implementação.

1.1 JUSTIFICATIVA

A escolha de uma tecnologia para estabelecer a comunicação entre simuladores Unity sem um estudo prévio pode levar a necessidade de sua substituição e até a descontinuação total do simulador. Além disso, durante o processo de implementação de comunicação de um simulador, as características da tecnologia utilizada influenciam diretamente em como serão supridos os requisitos do simulador distribuído, como seu desempenho, tolerância a falhas, segurança, escalabilidade e extensibilidade (TANENBAUM, 2007), o que pode se tornar um desafio de acordo com a tecnologia adotada. Da mesma forma, características relacionadas à implementação, modelagem e estruturação de dados podem influenciar o tempo de desenvolvimento e a eficiência da solução. Neste contexto, este trabalho avalia tecnologias de referência para desenvolvedores de simulação em Unity no contexto de um simulador real, podendo servir de elemento de estudo e análise, facilitando tomadas de decisão.

1.2 OBJETIVOS

1.2.1 Geral

Este trabalho objetiva avaliar quatro opções de tecnologias para fornecer a comunicação dentro do ambiente de desenvolvimento da Unity. As tecnologias avaliadas são HLA, DDS, UNET e TNET3.

Para isso é realizada uma comparação teórica das tecnologias englobadas e um estudo de caso prático no contexto do simulador SVTat REOP, a fim de estabelecer resultados que possam servir de base para a tomada de decisão da escolha de tecnologia.

1.2.2 Específicos

Mais especificamente, este trabalho final de graduação objetiva:

- Aprofundar os estudos referentes às tecnologias envolvidas, visando prover o conhecimento necessário para a compreensão deste trabalho.
- Realizar uma comparação teórica das funcionalidades de cada tecnologia, expressando suas diferenças, vantagens e desvantagens, sob perspectiva da simulação distribuída.

- Efetuar a implementação de comunicação no simulador SVTat REOP utilizando as quatro tecnologias, servindo como base para o estudo de caso deste trabalho.
- Comparar as implementações levando em conta a viabilidade de implementação, ou seja, como a tecnologia utilizada atendeu os requisitos obtidos durante o desenvolvimento e demostrou maior facilidade ao ser utilizada e integrada ao ambiente Unity.
- Realizar uma comparação de desempenho das tecnologias elencadas no projeto, utilizando diversas métricas de rede e de processamento.

1.3 ORGANIZAÇÃO DO DOCUMENTO

Este trabalho está organizado em seis capítulos. O capítulo 2 apresenta os fundamentos da parte teórica, provendo o embasamento científico sobre os assuntos utilizados neste trabalho e descreve conceitos fundamentais da simulação distribuída e também aborda os trabalhos relacionados. O capítulo 3 descreve o contexto do simulador SVTat REOP e quais os requisitos do simulador no contexto de comunicação. Posteriormente são especificadas as etapas comparativas para atingir o resultado desejado (sustentação técnica para adoção de tecnologia). O capítulo 4 demonstra as principais funcionalidades de cada tecnologia e as distinções entre elas, constituindo uma comparação teórica. O capítulo 5 apresenta as implementações realizadas e demonstra qual foi a mais viável durante a fase de implementação e melhor atendeu a demanda do simulador. Posteriormente, no capitulo 6 são realizados testes de desempenho no contexto do simulador utilizando as tecnologias englobadas. Por fim, o capítulo 7 sintetiza as conclusões do trabalho.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo busca apresentar o embasamento científico necessário para a compreensão deste trabalho. A seção 2.1 fala sobre a simulação no contexto da computação e apresenta os tipos de simulação. Na seção 2.2 é descrito o que é um sistema distribuído e então introduz o conceito de *middleware*. Posteriormente, na seção 2.3 são apresentados os simuladores distribuídos e a interoperabilidade. Com isso, são descritos os padrões HLA (seção 2.4) e DDS (seção 2.5), que constituem parte do trabalho. A *engine* Unity é apresentada na seção 2.6 onde também são apresentadas as tecnologias UNET e TNET3 que fazem parte do ambiente Unity e também são utilizadas neste trabalho. Por último, na seção 2.7, são abordados os trabalhos relacionados.

2.1 SIMULAÇÃO

Na computação, um simulador é compreendido como um *software* que modela o comportamento de um sistema real ou imaginário ao decorrer do tempo (FUJIMOTO, 2000). Um exemplo de simulador são os simuladores de voo como o *Microsoft Flight Simulator 2020* (figura 2.1), que fornece ao usuário a experiência de pilotar uma aeronave, simulando desde as interações físicas da aeronave com o ambiente até as operações mais complexas como o acionamento do motor e pouso por instrumentos.

Figura 2.1 – Cockpit de voo simulado

Fonte: (XBOX, 2021)

No contexto militar, uma das principais utilizações de simuladores ocorre na simulação de combate com o objetivo de treinamento (NICOL, 2011). Um dos principais benefícios da utilização de simuladores no treinamento militar ocorre pela redução de custos, aumento da segurança operacional e maior visibilidade e reprodutibilidade das ações praticadas na simulação (FLETCHER, 2009). Em outras palavras, ao simular uma situação de combate ao qual envolve-se recursos humanos e/ou bélicos retira-se o risco e custo por se tratar de um ambiente parcialmente ou totalmente virtual, onde as sessões de simulação podem ser salvas em algum formato para serem revistas ou reanalisadas pelos militares envolvidos no treinamento.

A simulação no ambiente militar é usada principalmente para criar ambientes de treinamento, jogos de guerra (*war gaming*) e T&E (FUJIMOTO, 2000). Os ambientes de treinamento referem-se a inserir militares em um ambiente simulado onde podem treinar suas habilidades pessoais para uma situação de combate real, como por exemplo inserir um piloto de caça em um simulador que imite as situações reais de combate.

As simulações do tipo *war gaming* referem-se a tipos de simulação que são usualmente usados para avaliar diferentes tipos de estratégias para ataque ou defesa de uma força inimiga. Este tipo de simulação se preocupa em modelar grupos, unidades ou batalhões sendo chamado de simulação de entidades agregadas (*aggregated simulations*).

Por último as simulações de Teste e Avaliação (*Test And Evaluation*) preocupam-se em inserir componentes físicos (como sensores para detectar o lançamento de míssil) dentro do ambiente virtual para avaliarem a efetividade do dispositivo em cumprir sua proposta.

2.1.1 Classificação de simulação

O Departamento de Defesa dos Estados Unidos (DOD, 1998) classifica os simuladores de acordo com seu nível de iteração humana e grau de realismo dos equipamentos utilizados, conhecidos respectivamente como *Live*, *Virtual* e *Constructive* (LVC).

2.1.1.1 Simulação viva

A simulação viva (*live simulation*) envolve pessoas reais operando sistemas reais. Este tipo de simulação tenta ser o mais próximo da realidade, e usualmente envolve equipamentos ou sistemas reais (SOKOLOWSKI; BANKS, 2010). Um exemplo disto é um piloto de aeronave real lançando mísseis em alvos físicos com o intuito de treinamento, testes ou avaliação de capacidade operacional (HODSON; HILL, 2014). Por ser a simulação mais próxima da realidade usualmente possui maiores custos e riscos de realização.

2.1.1.2 *Simulação virtual*

A simulação virtual (*virtual simulation*) envolve pessoas reais operando equipamentos simulados. Este tipo de simulação requer a participação de um humano (*human-in-the-loop*) exercitando suas habilidades (DOD, 1998). Estes sistemas são projetados para inserir o usuário em um ambiente realístico. Um bom exemplo de simulação virtual é a simulação de um *cockpit* de avião para treinar pilotos. Este simulador utiliza uma representação física igual a verdadeira mas utiliza modelos computacionais para gerar as dinâmicas de voo e mudanças de atmosfera/ambiente aos quais o piloto deve responder (HODSON; HILL, 2014). Em comparação com a simulação viva, normalmente possui um menor grau de realismo em alguns aspectos, mas consegue simular operações reais com um menor custo e maior segurança.

2.1.1.3 *Simulação construtiva*

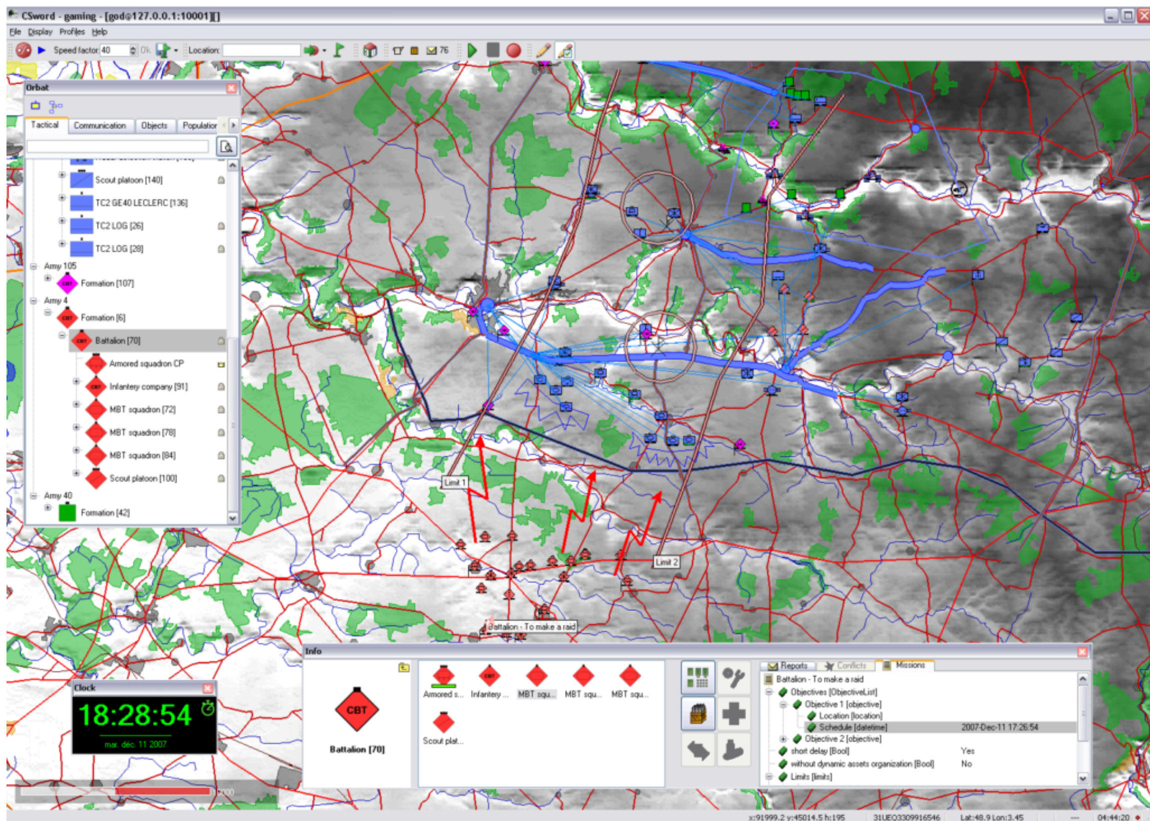
Neste tipo de simulação pessoas reais inserem na simulação pessoas simuladas que operam sistemas simulados (SOKOLOWSKI; BANKS, 2010). Assim, são estabelecidos cenários, parâmetros e comandos decisivos. Então o computador computa (joga) as missões e as consequências destas decisões servem como suporte ao desenvolvimento de táticas, técnicas e procedimentos (FLETCHER, 2009). Este tipo de simulação possui um caráter estratégico. Exemplos de simuladores construtivos são os chamados de *War Gaming*, ao qual preocupam-se com questões estratégicas de batalha, como, por exemplo, no simulador SWORD (figura 2.2) (Synergy Simulation, 2021).

2.2 SISTEMAS DISTRIBUÍDOS

Para TANENBAUM (2007), apesar de a literatura demonstrar definições de sistemas distribuídos não satisfatórios e discordantes entre si, destacam-se partes importantes da definição que serão úteis para o trabalho. A primeira destaca que um sistema distribuído é constituído por componentes (como computadores) autônomos. A segunda é que os usuários deste sistema pensam estar lidando com um sistema único. E por último que estes componentes que formam o sistema distribuído precisam colaborar entre si de alguma forma. Estes componentes podem estar distribuídos em regiões geograficamente diferentes ou em um mesmo lugar tendo como principal objetivo facilitar a troca de informação entre os componentes através de uma rede de computadores.

Dentre os principais desafios na construção de um sistema distribuído estão lidar com a heterogeneidade dos componentes, segurança, escalabilidade, tolerância a falhas,

Figura 2.2 – Simulador construtivo SWORD



Fonte: (Synergy Simulation, 2021)

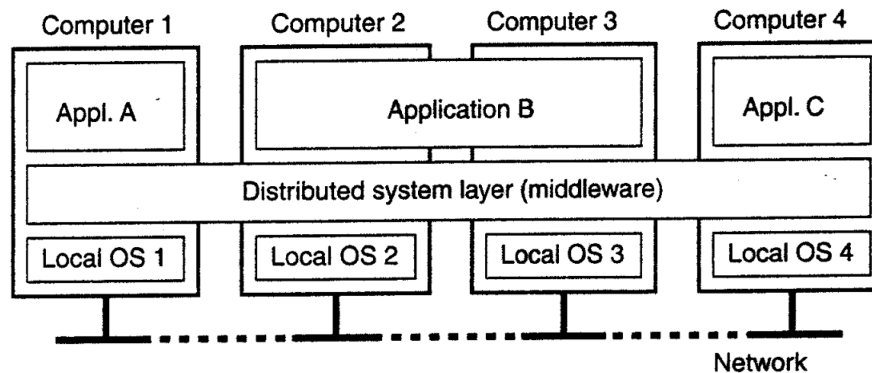
concorrências dos componentes e prover qualidade de serviço (COULORIS, 2009). Os principais exemplos de sistemas distribuídos modernos são os mecanismos de busca pela internet, jogos multijogadores online e sistemas de finança e negociação.

2.2.1 *Middleware*

Como um sistema distribuído pode ser composto por componentes heterogêneos, aos quais podem estar em diferentes sistemas operacionais e redes diferentes, a camada de *software* responsável por interligar os componentes fica usualmente entre a camada superior, onde estão as aplicações dos usuários, e a camada inferior, onde estão os sistemas operacionais e sistemas de comunicação de rede (TANENBAUM, 2007). Na figura 2.3 é demonstrado esta relação, onde a camada que permite este sistema distribuído existir é chamada de *middleware*.

Nesta figura existem quatro computadores conectados, e três aplicações, onde a aplicação B é compartilhada entre os computadores dois e três. O sistema distribuído, através do *middleware*, permite que estes se comuniquem entre si, inclusive entre diferentes aplicações ao mesmo tempo em que esconde as diferenças de *hardware* e sistema operacional entre as diferentes aplicações.

Figura 2.3 – Camada de *sotfware* chamada *middleware*



Fonte: (TANENBAUM, 2007)

2.3 SIMULAÇÃO DISTRIBUÍDA

Para FUJIMOTO (2000) a simulação distribuída se refere a tecnologia que permite que simuladores executados em sistemas computacionais se comuniquem por interconexões de uma rede de computadores. Em outras palavras, esta tecnologia permite que um programa de simulação seja executado em um sistema distribuído. *Fujimoto* também cita que a simulação distribuída tem como principais benefícios a redução do tempo de execução, distribuição geográfica, integração de diferentes máquinas e tolerância a falhas.

2.3.1 Interoperabilidade

O IEEE (1990) define interoperabilidade como a capacidade de dois ou mais sistemas ou componentes de trocar informações e usar as informações que foram trocadas. Para o Departamento de Defesa dos EUA (DOD, 1998) a interoperabilidade é definida como a capacidade de dois ou mais sistemas ou componentes proverem informações, dados, materiais e serviços e também a capacidade de receber o mesmo de outros sistemas permitindo a operação efetiva desses sistemas em conjunto.

Existem vários níveis de interoperabilidade entre dois sistemas que variam de nenhuma interoperabilidade a interoperabilidade total (TOLK; MUGUIRA, 2003). Não é a proposta do trabalho exemplificar estes níveis, mas vale ressaltar que a interoperabilidade engloba diferentes técnicas que circulam em torno da troca de informações.

2.3.2 Padrões de interoperabilidade

Padrões de simulação distribuída, também conhecidos como protocolos de interoperabilidade, permitem que a execução de modelos independentes (simuladores) interoperem, normalmente através de uma rede de computadores, de modo a simular colaborativamente um cenário ou ambiente (WAINER; AL-ZOUBI, 2010). Como os aplicativos interoperáveis podem ser desenvolvidos em momentos diferentes por pessoas diferentes ou organizações, a padronização do protocolo de interoperabilidade torna-se um pré-requisito para tais sistemas operarem com sucesso (TOLK, 2012). Estes padrões podem possuir diversas funcionalidades e diferentes abordagens em torno da interoperabilidade, e por isso suprir os requisitos do simulador de diferentes maneiras.

2.4 HIGH LEVEL ARCHITECTURE

O HLA (*High Level Architecture*) é um padrão cujo objetivo é permitir a interoperabilidade entre os simuladores e o reuso de dados da simulação (IEEE, 2010a). Este padrão foi criado em 1995 e tornou-se o padrão utilizado em todas atividades de simulação do Departamento de Defesa dos Estados Unidos (DoD). O HLA é importante porque é visto como o padrão de proposta geral para a simulação distribuída, tanto para simulações virtuais como construtivas.

A especificação inicial do HLA é descrita como HLA 1.3 sendo originalmente criada pelo DoD. No ano 2000 sofreu padronização pela IEEE descrita como HLA 1516-2000, até evoluir em seu atual estado conhecida como HLA 1516-2010 *EVOLVED*.

O HLA fornece uma *framework* ao desenvolvedor, possibilitando a estruturação e modelagem dos sistemas de simulação. A especificação do HLA é descrita pelos seguintes documentos: HLA *Federate Interface Specification* (IEEE, 2010c), *HLA Framework and Rules Specification* (IEEE, 2010a) e *HLA Object Model Template* (IEEE, 2010b).

2.4.1 Conceitos Básicos

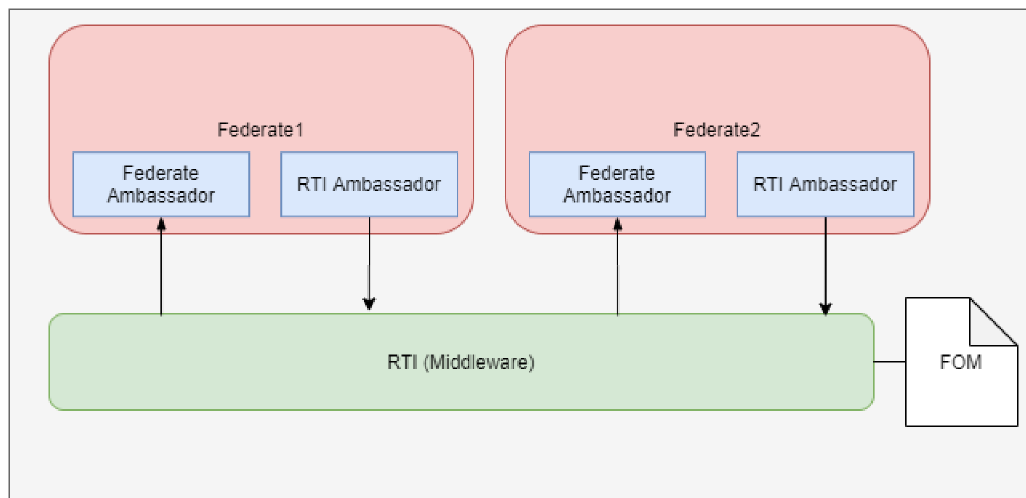
No HLA um simulador é denominado federado e o conjunto de federados chama-se federação. As iterações entre simulações em uma execução de federação são controladas pelo RTI (IEEE, 2010a). O RTI (*Run-Time Infrastructure*) é especificado como o *middleware* responsável por integrar as simulações fornecendo uma *API* para gerenciar as federações, federados, objetos, propriedades e o tempo. Uma federação deve conter um FOM (*Federation Object Model*) que específica os tipos dados da simulação e como será trocada a informação entre os federados, basicamente sendo um arquivo XML responsável

pela modelagem da simulação. Este arquivo é regrado pelo OMT (*Object Model Template*).

O HLA utiliza o modelo *Publish-Subscribe* para a disseminação de dados, onde uma interação (*Interation Class*) é compreendida como uma mensagem trocada entre federados e uma instância de objeto (*Object Class*) é compreendida como uma instância com atributos e um identificador único. Estes são os principais dados que são descritos no FOM.

No IEEE (2010a) é especificado duas importantes classes: o *RTI Ambassador* e o *Federate Ambassador*. O *RTI Ambassador* permite invocar funções disponíveis pelo RTI como possibilitar a entrada em uma federação ou criar um objeto, enquanto o *Federate Ambassador* limita-se a receber eventos que ocorreram no RTI (como a criação de um objeto remoto que foi assinada) através de *callbacks*. A figura 2.4 ilustra esta relação descrita e mostra como normalmente um sistema distribuído está disposto utilizando HLA.

Figura 2.4 – Organização entre simuladores no HLA



Fonte: O Autor

O RTI (*Run time Infrastructure*) é definido em seis áreas de acordo com as normas apresentadas, sendo eles:

- **Federation management:** permite criar e deletar a execução de federações e permite que um federado entre ou saia de uma federação.
- **Declaration management:** permite aos simuladores declararem suas intenções em publicar objetos ou assinar objetos produzido por outros simuladores.
- **Object management:** permite aos simuladores criar e deletar objetos assim como produzir atualização de atributos sobre objetos.
- **Ownership management:** permite gerenciar o simulador que tem direito de modificar os atributos de um objeto e também possibilita mudar esse direito em tempo de execução.

- **Time management:** permite gerenciar a sincronização de tempo entre os federados permitindo simulações baseados em tempo ou eventos, bem como o recebimento de mensagem ordenadas por tempo (*Time Sent Order*).
- **Data distribution management:** serviço que atua na otimização do tráfego entre os simuladores, reduzindo os dados a serem processados e/ou serem enviados pela rede.

2.4.2 DevStudio

No contexto deste trabalho foi utilizada uma ferramenta de geração de código para o HLA, chamada de DevStudio (Pitch Technologies, 2021). Esta ferramenta lê os dados do FOM que modelam a simulação e cria uma API que integra o modelo de simulação diretamente com a API fornecida pelo HLA. Assim é removida a necessidade do programador de implementar as instâncias de objetos (*Object Class*), interações (*Interaction Class*), processos de decodificação e codificação e outros na linguagem de programação utilizada. Isto permite ao programador se preocupar apenas em desenvolver a lógica da aplicação utilizando os recursos fornecidos pelo HLA para um dado FOM. A figura 2.5 ilustra um exemplo de FOM.

Figura 2.5 – Exemplo de FOM

```
<objects>
    <objectClass>
        <name>Carro</name>
        <attribute>
            <name>nome</name>
            <dataType>HLAASCIIstring</dataType>
        </attribute>
        <attribute>
            <name>ano_produzido</name>
            <dataType>HLAinteger32BE</dataType>
        </attribute>
    </objectClass>
</objects>
<interactions>
    <interactionClass>
        <name>InteraçãoMensagem</name>
        <parameter>
            <name>Mensagem</name>
            <dataType>HLAASCIIstring</dataType>
        </parameter>
    </interactionClass>
</interactions>
```

Fonte: O Autor

O FOM ilustrado na figura declara um objeto que pode ser instanciado (*Object Class*) chamado Carro. Este objeto possui como atributos um 'nome' do tipo *HLAASCIIstring*, que representa uma *string* de caracteres com codificação ASCII e o atributo 'ano_produzido' representado por um inteiro de 32 bits chamado de *HLAinteger32BE*. Na declaração de interações, a interação *InteraçãoMensagem* possui como parâmetros apenas uma mensagem do tipo *HLAASCIIstring* que pode ser trocada entre federados.

Figura 2.6 – Exemplo de ilustrativo de código gerado pelo DevStudio

```
int main()
{
    /*** Entra na federação ***/

    Federate federado;

    federado.JoinFederation("Federação1");

    /*** Subscribe to HLAVehicle ***/

    HLACarroListener listener;

    federado.SubscribeCarro(listener);

    /*** Publica um Veiculo e atualiza/modifica seu atributo ano_produzido ***/

    HlaCarro carro = federado.CreateCarro("VeiculoNome", 2010);

    carro.UpdateAno_Produzido(2011);

    /*** Sai da federação ***/

    federado.ResignFederation();
}
```

Fonte: O Autor

Na figura 2.6, está representado o uso de uma adaptação do código que seria gerado pelo DevStudio. Primeiro a execução do programa conecta-se a uma federação como um federado. Posteriormente é realizada uma inscrição para receber objetos remotos do tipo 'Carro' através da classe *HLACarroListener* que irá receber via *callbacks* os eventos relacionados a este objeto. Por último, é publicada uma instância objeto Carro e então atualiza-se seu atributo 'ano_produzido', este objeto e seus atributos serão distribuídos para quem assinou este objeto na federação. Posteriormente o programa sai da federação. Este é o tipo de abstração provido pelo DevStudio.

2.5 DATA DISTRIBUTION SERVICE

O *Data Distribution Service* (DDS) é um protocolo implementado na forma de *middleware*, também referenciado como API, desenvolvido para conectividade centrada em dados. O DDS é padronizado pela *Object Management Group* (OMG). Ele integra componentes de um sistema fornecendo conectividade de dados de baixa latência, extrema confiabilidade e uma arquitetura escalável requeridas por aplicativos de negócios e da Internet das Coisas (IoT) (OMG DDS PORTAL, 2020).

O objetivo do DDS é facilitar a distribuição eficiente de dados em um sistema distribuído, focando em sistemas de tempo real (*real time system*) (PARDO-CASTELLOTE, 2003), aos quais são sensíveis ao tempo e atraso de processamento. Diferente do HLA, o DDS não é focado em simulação distribuída, tendo uma proposta mais generalizada.

O DDS também utiliza a modelo *publish-subscribe* como o HLA. A especificação principal é a DDS v1.4 (OMG DDS 1.4, 2015), sendo descrita em duas camadas: uma inferior para a distribuição de dados, chamada DCPS (*Data-Centric-Publish-Subscribe*) usada para a entrega eficiente dos dados aos destinatários adequados, e uma superior (opcional), chamada DLRL (*Data Local Reconstruction Layer*), que permite a reconstrução local de dados e uma integração mais simples na camada de aplicação. Neste trabalho o foco é na camada DCPS.

2.5.1 Conceitos Básicos

Apesar de o DDS não ser focado em simulações, pode-se considerar para o contexto deste trabalho que um conjunto de simuladores é conhecido como domínio (*Domain*). O domínio é onde as entidades devem inicialmente se conectar. Estas entidades são conhecidas como participantes do domínio (*DomainParticipant*). Assim como o HLA, o DDS utiliza a disseminação de dados *Publish-Subscribe*.

Os tipos de dados publicáveis/assináveis são chamados de tópicos (*Topics*). Um tópico pode conter uma chave primária que o identifique, sendo possível assim ter múltiplas instâncias de um mesmo tópico. Tópicos com identificadores são chamados de *Keyed Topic*. Caso não seja especificado um identificador, o Tópico age como se fosse um *Singleton*, possuindo uma única instância.

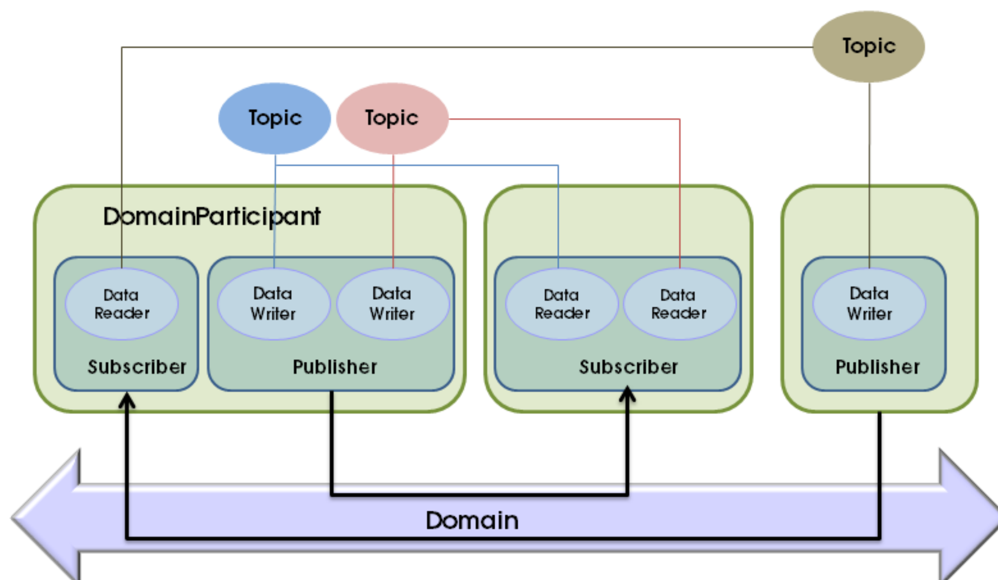
Para a declaração de Tópicos, utiliza-se um arquivo do tipo IDL (*Inteface Definition Language*), com sintaxe semelhante ao C, podendo ser em certos aspectos equivalente ao FOM do padrão HLA.

A API do DDS em questão, DCPS, é composta por cinco módulos (descritos a seguir): *Infrastructure* (Infraestrutura), *Topic* (Tópico), *Publication* (Publicação), *Subscription* (Inscrição) e *Domain* (Domínio). A figura 2.7 ilustra a relação dos módulos descritos abaixo

e também demonstra como é organizado um sistema distribuído utilizando o DDS.

- **Topic module:** o módulo de tópico contém as classes *Topic* e *TopicListener*, um *Topic* é compreendido como um tipo de dado publicável ou assinável enquanto o *TopicListener* permite escutar eventos de um determinado tópico que foi assinado.
- **Subscription module:** o módulo de subscrição contém o *Subscriber* e o *DataReader*, um *Subcriber* pode conter vários *DataReaders*. Cada *DataReader* assina um único tipo de *Topic*.
- **Publication module:** o módulo de publicação contém o *Publisher* e o *DataWriter*, um *Publisher* pode conter vários *DataWriters*. Cada *DataWriter* publica um único tipo de *topic*.
- **Domain module:** o módulo de domínio refere-se a entidades que estão no sistema distribuído. O *Domain* é onde todos os participantes da simulação se conectaram. Estes participantes são chamados de *DomainParticipant*. Um *DomainParticipant* usualmente é constituído de um *Publisher* e um *Subscriber*
- **Infrastructure module:** o módulo de Infraestrutura compreende as classes *DCPSEntity, Listener, Condition, Waiset e QoS Policy*. Estes funcionam como classes herdadas que serão utilizadas pelas outras camadas. Por exemplo, a classe *Listener* será herdada por outra classe para receber notificações de determinado Tópico, a classe QoSPolicy fornece opções de Qualidade de Serviço, etc.

Figura 2.7 – Domain no DDS



Fonte: (PARDO-CASTELLOTE; FARABAUGH; WARREN, 2015)

2.5.2 Geração de código

Diferente do HLA, o DDS prove a geração de código embutida em sua própria implementação e especificada em seu padrão. No HLA é comumente usado algum *sotfware* terceiro para a geração do código porque a API provida deixa na mão do programador implementar funcionalidades como os dados descritos no FOM, a codificação/decodificação parcial destes dados.

Como citado, o DDS utiliza o arquivo IDL onde são declarados seus tópicos. O gerador de código provido chama-se IDLPP (*Interface Definition Language Pré Processor*) que lê o arquivo IDL durante a compilação do programa que está sendo utilizado e gera as funcionalidades que são providas na camada DCPS para os tópicos declarados.

Figura 2.8 – Ilustação de código arquivo IDL

```
module MyModule
{
    struct Carro
    {
        string nome;

        int ano_produzido;
    };
    #pragma keylist Carro nome

    struct Mensagem
    {
        string mensagem_;
    };
};
```

Fonte: O autor

O arquivo IDL na figura 2.8 declara os mesmos dados de simulação do FOM representado na figura 2.5. O tópico Carro possui o atributo 'nome' do tipo *string* e o atributo 'ano_produzido' do tipo *int*. Abaixo de sua declaração está descrito que este tópico será identificado pelo seu atributo 'nome', ou seja, que este atributo será sua chave primária, através da declaração ao compilador "*#pragma keylist Carro nome*". Assim este tópico pode ser instanciado múltiplas vezes, similarmente ao Carro declarado no FOM anterior.

Também foi declarado um tópico Mensagem, que possui um único atributo chamado 'mensagem_'. Como não foi especificada nenhuma chave primária este tópico possui uma única instância, similarmente a Interação '*IteraçãoMensagem*' declarada no FOM anterior.

2.6 ENGINE UNITY

A Unity é uma *game engine*, também chamada de *grapichs engine* (motor gráfico), focada em criar aplicações 2D e 3D, mais popularmente usada para a criação de jogos, possuindo suporte multiplataforma utilizando a linguagem C# e *javascript*.

Uma *game engine* é uma coleção de ferramentas (incluindo bibliotecas de baixo nível, editores de interface de usuário e ferramentas de gerenciamento de multimídia) que facilitam o trabalho de desenvolver um jogo (Messaoudi; Simon; Ksentini, 2015), e por isso a Unity pode ser considerada uma *framework* para criação de jogos. Uma *game engine* possui diversos módulos para facilitar e abstrair a criação da aplicação, dentre eles estão módulos de física, *scripting*, *input*, inteligência artificial, *multimedia*, *rendering* e *networking*:

- **Scripting:** permite que o desenvolvedor desenvolva a lógica da aplicação utilizando a linguagem de *script* da *game engine* ao qual está integrada com os componentes da aplicação.
- **Física:** permite simular o comportamento de objetos do mundo virtual de acordo com as leis da física observadas na realidade.
- **Inteligência Artificial:** permite simular comportamentos de objetos virtuais que dão a ilusão de inteligência.
- **Multimedia:** permite a reprodução de áudio e outros tipos de mídia.
- **Rendering:** fornece um *pipeline* gráfico para a geração de elementos 2D e 3D.
- **Input:** permite capturar o *input* de dispositivos como mouse, teclado e controles ao quais poderão ser refletidos na lógica da aplicação.
- **Networking:** permite conectar diferentes instâncias de uma aplicação, dependendo da aplicação é conhecida como modo *multiplayer*.

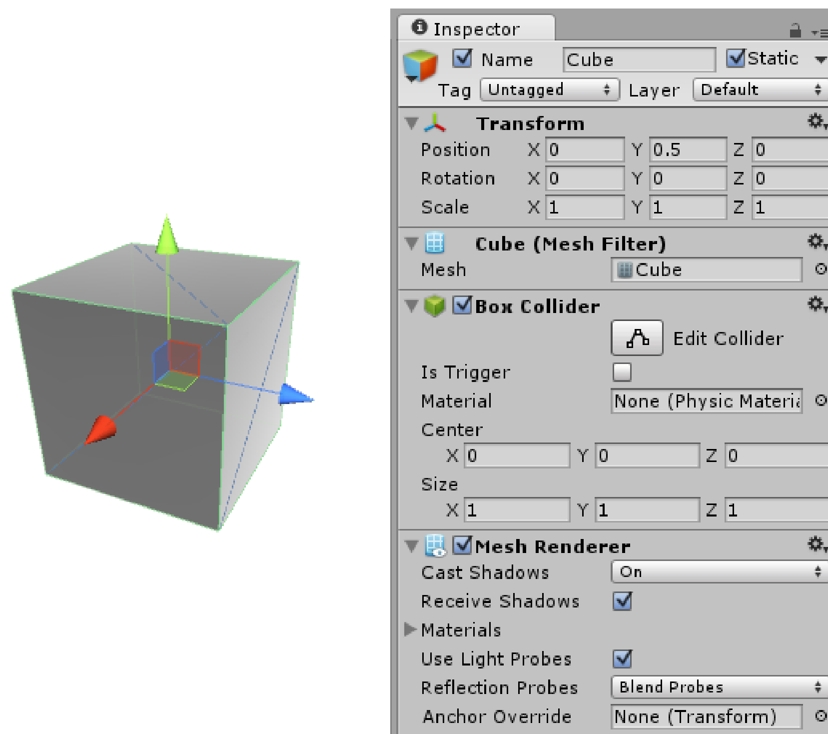
2.6.1 Elementos básicos da Unity

Existem alguns conceitos básicos para entender o funcionamento da Unity durante a fase de desenvolvimento. A Unity fornece o editor para que os programadores desenvolvam as suas aplicações. Uma aplicação na *engine* Unity é inicialmente dividida entre cenas (*Scenes*). Pode-se compreender uma cena como um único nível, onde em cada cena há um ambiente com obstáculos, objetos e desafios diferentes permitindo que a aplicação seja dividida em pedaços (UNITY, 2020).

Uma cena contém uma coleção de *GameObjects* que se organizam entre si em estrutura de árvore n-ária. Estes objetos são usados para descrever entidades de seu jogo e desenvolver a lógica da aplicação. Usualmente uma cena é composta por diversos *GameObject*.

GameObjects são *containers* para Componentes (*Components*). Componentes podem variar desde objetos 3D, *scripts* na linguagem C# até componentes de simulação física. Podemos compreender que componentes adicionam propriedades e moldam os *GameObjects* que compõem a cena. O uso destes elementos básicos objetiva uma fácil organização e desenvolvimento da lógica da aplicação Unity sendo parte do ambiente fornecido.

Figura 2.9 – *GameObject da Unity*



Fonte: (UNITY, 2020)

A figura 2.9 representa um *GameObject* chamado *Cube* (Cubo) que possui quatro componentes. O primeiro componente é o *Transform*. Este componente é usado para guardar a sua posição, rotação e escala e sempre estará presente em *GameObjects*. O segundo componente (*Cube* (*Mesh Filter*)) especifica os vértices que formam o Cubo. O terceiro componente (*Box Collider*) é um componente físico para realizar testes de colisões deste cubo. E o último componente (*Mesh Renderer*) fornecerá a renderização dos vértices e especificações de materiais do cubo junto ao *pipeline* gráfico da Unity. Basicamente este *GameObject* forma um cubo que será desenhado no cenário e foi descrito pelos seus componentes. Assim uma cena é formada por vários *GameObjects* que podem representar qualquer objeto do ambiente Unity, sendo ele renderizado ou não.

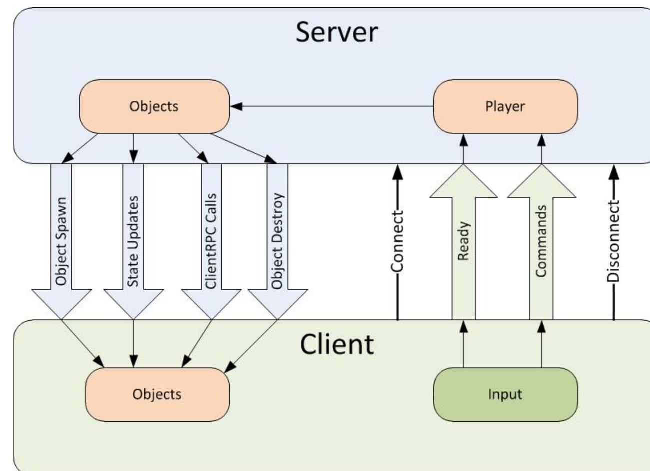
2.6.2 UNET

A UNET (Unity Network) é a rede interna utilizada pela *engine* Unity para integrar suas aplicações. Segundo a documentação da Unity (UNITY, 2020), a UNET provê a HLAPI (*High Level API*), que é uma API que fornece os requisitos mais comuns para estabelecer a comunicação entre as aplicações Unity.

Por ser uma API utilizada somente para aplicações Unity, ela permite somente que aplicações Unity se comuniquem. Ela funciona permitindo o compartilhamento de objetos nativos da própria Unity (*GameObjects*) entre as aplicações. O *GameObject* da figura 2.9 poderia ser facilmente compartilhado entre aplicações Unity caso fosse utilizado a UNET. Ela utiliza a arquitetura cliente-servidor, onde os clientes são as aplicações Unity que compartilham seus objetos.

O servidor fica responsável por fazer o controle dos objetos e manter a consistência da lógica da aplicação (figura 2.10). Quando os clientes necessitam criar algum *GameObject* é requisitado ao servidor que então redistribui aos outros clientes. Este mesmo processo ocorre ao atualizar um *GameObject*.

Figura 2.10 – Lógica de aplicação Unity



Fonte: (UNITY, 2020)

Atualmente, a UNET está sendo descontinuada (UNITY, 2020) e não sofrerá mais manutenção que permitirá atualizações, inserção de novas funcionalidades ou correções de erros. A Unity está desenvolvendo um nova API para substituir a UNET, mas no período de desenvolvimento deste trabalho esta API ainda não está disponível e por isso não pôde ser inserida na comparação realizada neste trabalho. No contexto do trabalho, a UNET não pode ser usada devido a limitações em suas funcionalidades (isso será discutido no capítulo 3).

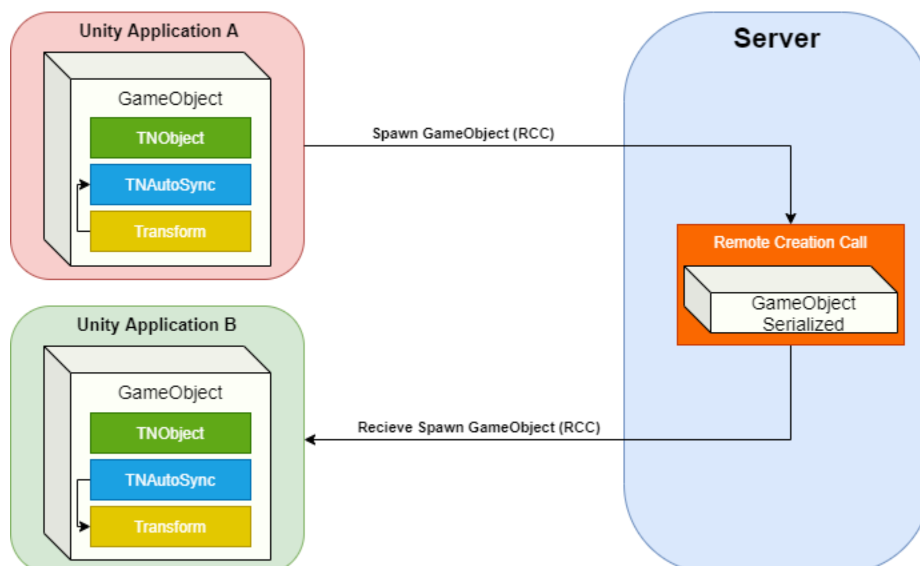
2.6.3 TNET3

A Unity possui uma loja onde é possível obter *assets* que podem ser utilizados para diversas finalidades. A TNET3 (*Tasharem Networking and Serialization Tools 3*) é um *asset* que busca praticar o mesmo papel que a UNET. De acordo com o site da desenvolvedora, a TNET3 é uma ferramenta de rede para a Unity e uma extensão para o Editor da Unity, leve, fácil de usar e altamente escalável (TASHAREM, 2016). Ele usa a arquitetura cliente-servidor fornecendo uma API de programação para estabelecer a comunicação entre aplicações Unity. Como a UNET está sendo descontinuada da *engine* Unity (UNITY, 2020), neste trabalho optou-se pela inclusão desta API que possui potencial similar e suporte.

A TNET3 utiliza um servidor onde os clientes (aplicações Unity) se conectam e este servidor fornece diversas funcionalidades. Ele funciona principalmente permitindo que objetos nativos da Unity (*GameObjects*) sejam compartilhados entre as aplicações e posteriormente que o estado destes objetos sejam sincronizados entre os clientes.

A UNET e TNET3 não são padrões de interoperabilidades e sim APIs específicas para estabelecer a comunicação de aplicações Unity. Diferente do HLA e DDS, na TNET3 os dados compartilhados não são escritos em algum documento, mas sim modelados no ambiente Unity através dos *GameObjects*. A figura 2.11 demonstra uma aplicação A tornando um *GameObject* local em remoto, assim esse objeto irá ser compartilhado com a aplicação B. Este objeto pode ser por exemplo o *Cube* da figura 2.9 ou o objeto *Vehicle* das figuras 2.8 e 2.5 caso sejam modelados no ambiente Unity.

Figura 2.11 – Compartilhamento de *GameObjects* na TNET3



Fonte: O autor

Nesta figura, a TNET3 utiliza uma chamada de procedimento remoto denominada RCC (*Remote Creation Call*) para tornar este *GameObject* remoto e compartilha-lo com a aplicação B. Esta chamada é serializada e enviada pela rede junto com o *GameObject* e então ao chegar ao seu destino é deserializada. Todo *GameObject* remoto é identificado

por um componente chamado *TNObject* (em verde na figura). Para sincronizar o estado de *GameObjects* remotos, ou seja, atualiza-los, a TNET3 oferece o componente chamado *TNAutoSync* (em azul) que permite sincronizar parte dos componentes que formam um *GameObject* e especificar o intervalo de atualização. O exemplo mais trivial é permitir a sincronização do componente *Transform* (em amarelo) que especifica a posição, escala e orientação de um *GameObject*, tal como visto nesta figura. O componente *TNAutoSync* irá ler o estado dos componentes locais do *GameObject* da aplicação A, por exemplo o *Transform*, e então irá enviá-lo pela rede para que atualize os objetos remotos deste.

A TNET3 também oferece o uso de chamada de funções remotas (*Remote Function Call*) entre os componentes que são *scripts*, permitindo que ao invés de realizar uma atualização de período fixo, as atualizações possuam um comportamento descrito pelo programador, permitindo uma otimização e melhor expressividade de comportamento.

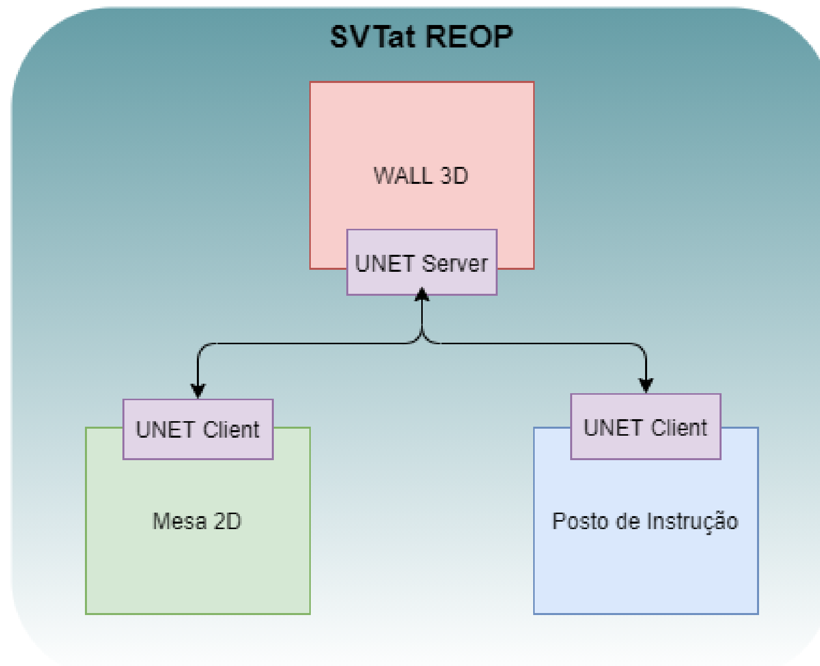
2.6.4 Simulador SVTat REOP

O Simulador Virtual Tático de Reconhecimento, Escolha e Ocupação de Posição (SVTat REOP) é um simulador em desenvolvimento pela Universidade Federal de Santa Maria (UFSM) em parceria com o Exército Brasileiro, para operar no Sistema Integrado de Simulação ASTROS (SIS-ASTROS). Este simulador tem como objetivo treinar militares em questões de planejamento (tático) e execução (operacional) no uso de armas de artilharia especificamente viaturas lançadoras de mísseis e foguetes que são pertencentes ao ASTROS (*Artillery Saturation Rocket System)*.

O simulador engloba três tipos de estações: o Posto de Instrução e a Mesa Tática, que fornece uma representação 2D do terreno, e o Wall, que fornece uma representação 3D do terreno. Este simulador utiliza a *engine* Unity como motor gráfico e é adotado como estudo de caso neste trabalho.

Cada estação pode representar uma instância do SVTat REOP e pode executar em computadores diferentes das demais. Na versão atual, internamente as estações são conectadas via UNET. Também é importante ressaltar que o Wall é responsável por agir como servidor para conectar as estações, o que implica que para existir uma instância do SVTat REOP deve haver pelo menos a estação Wall instanciada. Uma instância do SVTat pode conter no máximo uma instância de cada estação. A figura 2.12 demonstra a atual arquitetura do simulador entre as estações.

Figura 2.12 – Simulador SVTat



Fonte: O autor

2.7 TRABALHOS RELACIONADOS

Esta seção irá demonstrar quatro trabalhos relacionados.

2.7.1 Trabalho 1

O primeiro trabalho intitula-se *A Comparison of Simulation and Operational Architectures* (MARTINEZ et al., 2012). Esse artigo busca comparar as arquiteturas de interoperabilidade HLA e DDS. O trabalho inicialmente descreve as principais características das arquiteturas destacando suas diferenças em requisitos necessários para a simulação distribuída.

Utilizando como base os serviços fornecidos pelo HLA (vide seção 2.4), o artigo tenta demonstrar que o DDS suporta a grande parte dos recursos que o HLA (conhecido como o padrão de propósito geral para simulação) suporta, apesar de algumas diferenças.

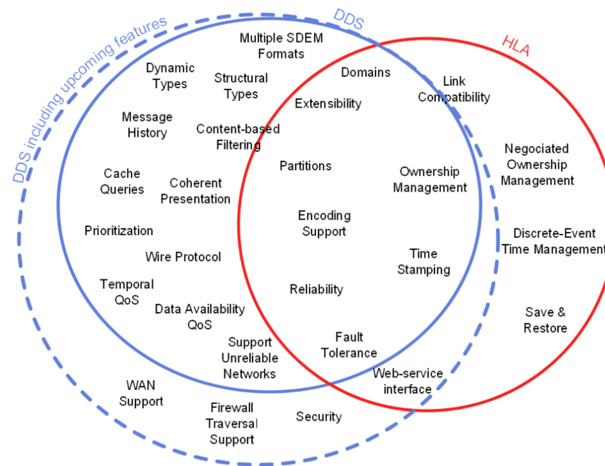
Posteriormente, o trabalho busca demonstrar as partes em que o DDS se sobressai, como o suporte a uma diversidade de políticas de Qualidade de Serviço (QoS), dentre outros. Mas também destaca que alguns mecanismos, como o *Time Managment*, que estão ausentes no DDS e presentes no HLA e são considerados requisitos para a simulação distribuída.

O artigo conclui que apesar das diferenças, os padrões compartilham muitas similaridades e que ambas possuem funcionalidades distintas e úteis na simulação distribuída.

Com a figura 2.13 os autores demonstram as funcionalidades em comum bem como as funcionalidades distintas dos padrões.

Ressalta-se que neste trabalho relacionado foram comparados os padrões o HLA e o DDS, não focando em comparar o uso de alguma tecnologia para integrar simuladores construídos na Unity. Este trabalho relacionado não tem nenhum estudo de caso, sendo totalmente teórico.

Figura 2.13 – Funcionalidade suportadas por HLA e/ou DDS



Fonte: (MARTINEZ et al., 2012)

2.7.2 Trabalho 2

O segundo trabalho intitula-se *A Comparison and Mapping of Data Distribution Service and High-Level Architecture* (JOSHI; GERARDO-PARDO; CASTELLOTE, 2006). Esse trabalho também busca comparar os padrões de interoperabilidade HLA e DDS. O modo de comparação é parecido ao trabalho 1, mas diferentemente, no início, os autores se esforçam para estabelecer um mapeamento entre os dois padrões, por exemplo ao demonstrar que uma instância de um objeto no HLA é equivalente a um *Topic* no DDS. A figura 2.14 demonstra parte das relações estabelecidas pelos autores.

Apesar dos trabalhos serem parecidos, este segundo se aprofunda mais em detalhes relacionados a *API* de programação dos padrões. Os autores concluem que o DDS é suficiente para atender a uma larga classe de problemas que são resolvidos pelo HLA. Também ressaltam que a *API* de *Time Management* provida pelo HLA é um aspecto positivo daquele padrão e que as políticas de QoS correspondem ao aspecto positivo do DDS. Assim como o trabalho anterior, é um trabalho comparativo teórico apenas dos padrões sem o contexto de comparar o uso de alguma tecnologia para comunicar simuladores que utilizem a Unity.

Figura 2.14 – HLA/DDS equivalentes

HLA	DDS
HLA-OMT	DDS-DLRL
HLA-RTI (IFSpec)	DDS-DCPS
HLA-Rules	-
Federation	Domain
Federate	Participant / Application
RTIAmbassador	***DomainParticipant, Publisher, DataWriter, Subscriber, DataReader***
FederateAmbassador	***Listener classes***
Object class	***Keyed Topic***
Interaction class	***Topic*** (no keys)
Update attribute	Write "keyed" instance
Reflect attribute	Read/Take "keyed" data samples
Send interaction	Write “non-keyed” instance
Receive interaction	Read/Take “non-keyed” samples

Fonte: (JOSHI; GERARDO-PARDO; CASTELLOTE, 2006)

2.7.3 Trabalho 3

Este trabalho relacionado intitula-se *Design, Implementation, and Performance Evaluation of HLA in Unity* (SÖDERBÄCK, 2017). Seu principal objetivo é investigar o uso de um *plugin* DLL para interoperar um simulador Unity. Este trabalho investiga quais os requerimentos e desafios necessários para desenvolver um *plugin* para a Unity e posteriormente também demonstra como a performance é afetada pelo uso de um *plugin* DLL.

Este trabalho utiliza um simulador de tanques, onde a implementação do *plugin* DLL, que torna o simulador Unity capaz de se conectar ao ambiente distribuído, foi utilizada como estudo de caso.

O trabalho conclui que existem alguns desafios ligados à implementação da DLL como a necessidade de recriar estruturas de dados para que estas possam ser passadas entre a DLL (escrita em C++) e a Unity (utiliza C#). Esta questão está ligada a como cada linguagem codifica os dados e como devem ser representadas as estruturas para que o processo de *Unmarshal/Marshal* (processo de transformação da representação em memória de um objeto) dos dados entre linguagens funcione corretamente.

Na parte de desempenho, foi comparada a Unity com diferentes números de tanques instanciados com o uso do *plugin* e posteriormente sem o seu uso. Foram usados métricas como o FPS do simulador e o RTT (*Round-Trip Time*). Os testes demonstraram que a perda de desempenho ao escalar a simulação ocorre em grande parte pelo uso de processamento da Unity e não do uso do *plugin* DLL.

No contexto deste trabalho, este se relaciona na parte de implementação do HLA

e os testes de desempenho realizados. Este trabalho limita-se a comparar o uso de um *plugin* DLL no ambiente Unity, ao qual também será abordado no capítulo 5 onde haverá o uso de uma DLL durante a fase de implementações.

2.7.4 Trabalho 4

Este trabalho intitula-se *Data Distribution Service (DDS): A Performance Comparison of Open Splice and RTI Implementations* (Bellavista et al., 2013). Este trabalho objetiva comparar a performance em rede de duas implementações do DDS. A comparação ocorre em uma LAN onde um *DomainParticipant* produz dados a diversos assinantes para possibilitar o stress da rede e a coleta de resultados. São realizados testes em dois ambientes, um que possui uma conexão de apenas 100 *Mbps* e outra que chega até 1 *Gbps*.

Nos testes são utilizadas métricas como RTT (*Round-Time Trip*), *Throughput* (taxa de transferência), número de mensagens recebidas por intervalo de tempo, uso de CPU e memória. O teste realizado é conhecido como teste de *payload*, onde *array* de *bytes* com tamanhos arbitrários são enviados de modo a perceber como as métricas de teste se comportam, possibilitando comparar as implementações.

O trabalho desta seção se relaciona com a parte de testes de desempenho deste trabalho final de graduação, porém limita-se a comparação apenas de implementações do DDS em relação a desempenho, utilizando testes de *payload* sem estruturas de dados específicas.

3 PROPOSTA COMPARATIVA

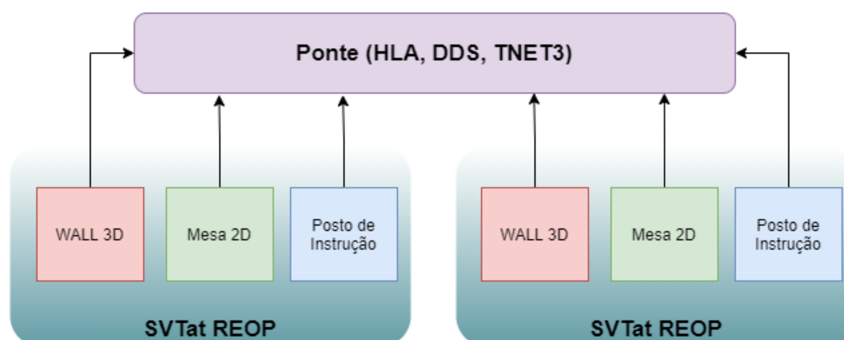
Esta capítulo mostra a proposta para realizar a comparação das tecnologias para a comunicação. Na seção 3.1 é explicado o contexto de implementação no SVTat REOP e as possíveis arquiteturas para comunicação a serem utilizadas na implementação. Assim na seção 3.2 é discutido os requisitos para implementar a arquitetura de comunicação escolhida. Na seção 3.3 é descrito como é realizada a comparação entre as tecnologias. Por último a seção 3.4 fala sobre os testes de desempenho.

3.1 CONTEXTO DO SIMULADOR

No âmbito do SIS-ASTROS tornou-se um requisito possibilitar a comunicação entre duas instâncias do SVTat REOP. Atualmente o simulador se comunica apenas entre suas estações através da UNET (ver figura 2.12) e também possui suporte para se comunicar com outros tipos de simuladores desenvolvidos fora do ambiente Unity, utilizando o padrão HLA.

Existem duas arquiteturas possíveis para permitir que o simulador se comunique entre si. A primeira é permitir que exista um único meio de comunicação que habilite que múltiplas instâncias do SVTat REOP se comuniquem entre si, inclusive entre suas estações. Nesta arquitetura, o servidor UNET da WALL é removido e todas as estações, inclusive de diferentes instâncias do SVTat REOP, devem ser autônomas e utilizar o mesmo meio de comunicação, tal como demonstrado na figura 3.1.

Figura 3.1 – Arquitetura 1



Fonte: O Autor

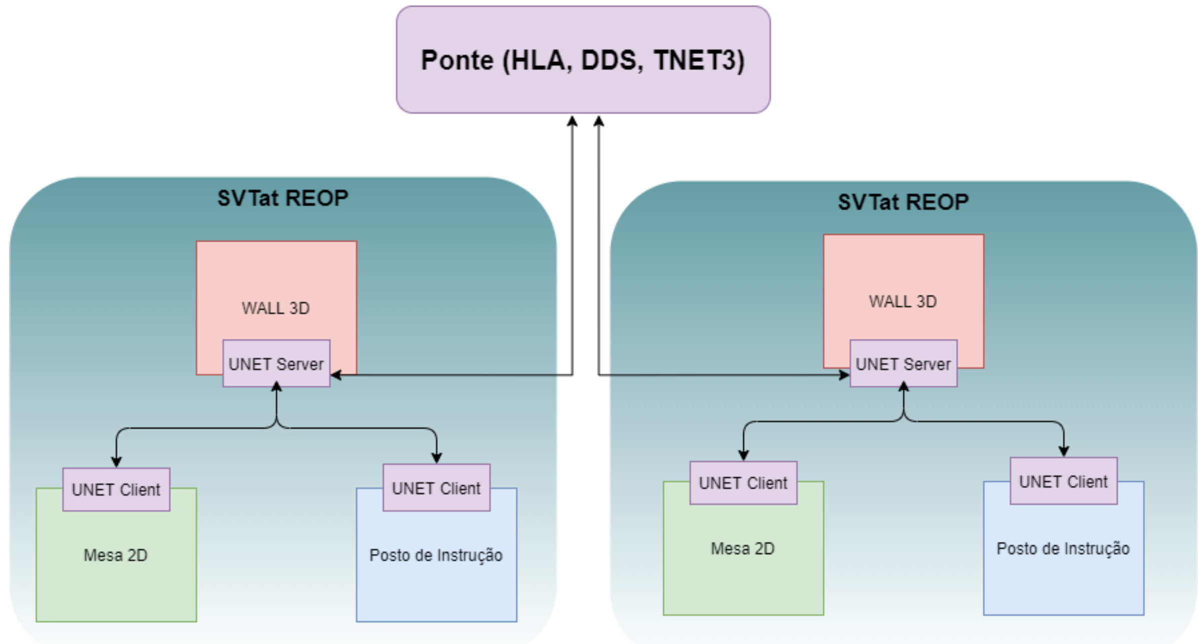
A figura mostra duas instâncias do SVTat REOP conectadas por uma ponte. Ressalta-se que como a tecnologia utilizada pode ser HLA, DDS ou TNET3, optou-se por chamar o meio de comunicação entre os simuladores de ponte (meio de comunicação entre as partes). Nesta arquitetura a estação WALL não age mais como servidor para suas estações e

então uma ponte permite que N instâncias do SVTat REOP se comuniquem diretamente. Deste modo as estações somente conseguem se comunicar pela ponte e não precisam utilizar o servidor UNET do WALL como ordenador da simulação.

Apesar desta arquitetura permitir a comunicação entre diferentes instâncias do SVTat REOP, considerando o projeto deste simulador, ao utilizá-la haveria uma necessidade de reprogramar a rede interna utilizada (UNET) pelas estações, exigindo a reprogramação de cada estação para suportar a nova tecnologia. Outros tipos de comunicações com outros simuladores que estão presentes no SVTat, dependem diretamente da implementação utilizada pela UNET e também teriam que ser reprogramados.

Uma segunda arquitetura para permitir o estabelecimento da comunicação entre instâncias do SVTat REOP e manter o atual estado do simulador (interconexão entre estações com UNET) é criando uma ponte responsável apenas pela conexão dos servidores das estações WALL. Nesta arquitetura, todo evento remoto que ocorrer entre as estações de uma instância do SVTat REOP será capturada pelo servidor UNET no WALL e retransmitindo através da ponte para outros servidores UNET no WALL de outras instâncias do SVTat REOP. Esta ponte basicamente trabalha refletindo eventos entre servidores UNET das estações WALL existentes.

Figura 3.2 – Arquitetura 2



Fonte: O autor

A figura 3.2 demonstra esta arquitetura, a estrutura original do simulador permanece a mesma mas agora existe uma ligação do servidor UNET do WALL com uma ponte externa, permitindo assim que os servidores UNET de cada WALL se comuniquem entre si e realizem uma simulação em conjunto. Uma vez que cada simulador WALL se comunique entre si, basta que este reflita os eventos remotos a todas estações que pertencem a sua

instância SVTat REOP.

Considerando que o esforço para reprogramação das estações do SVTat REOP pode ser grande e levar a problemas hoje não previstos, este trabalho optou por manter a estrutura interna do SVTat REOP e explorar a segunda opção de arquitetura.

Uma desvantagem de utilizar esta arquitetura é que não é possível utilizar a UNET para estabelecer a comunicação entre os simuladores SVTat REOP, pois a UNET somente permite que uma aplicação Unity se conecte a um único servidor UNET por vez, além de não permitir a instanciação de mais de um servidor por aplicação Unity. Como o WALL já possui o servidor UNET torna-se impossível utilizar apenas esta tecnologia na arquitetura adotada. Isto ocorre por que as classes da UNET são desenvolvidas usando o padrão de *Singletons*.

Vale ressaltar que a adoção desta arquitetura não leva em consideração a possibilidade da descontinuação da UNET, já que neste caso vale mais a pena adoção da outra arquitetura onde se removeria a UNET e se criaria toda a comunicação entre um único meio para todas estações. Assim, este trabalho foca apenas na aquisição do novo requisito do simulador, desconsiderando o problema da descontinuação da UNET.

3.2 REQUISITOS DA COMUNICAÇÃO

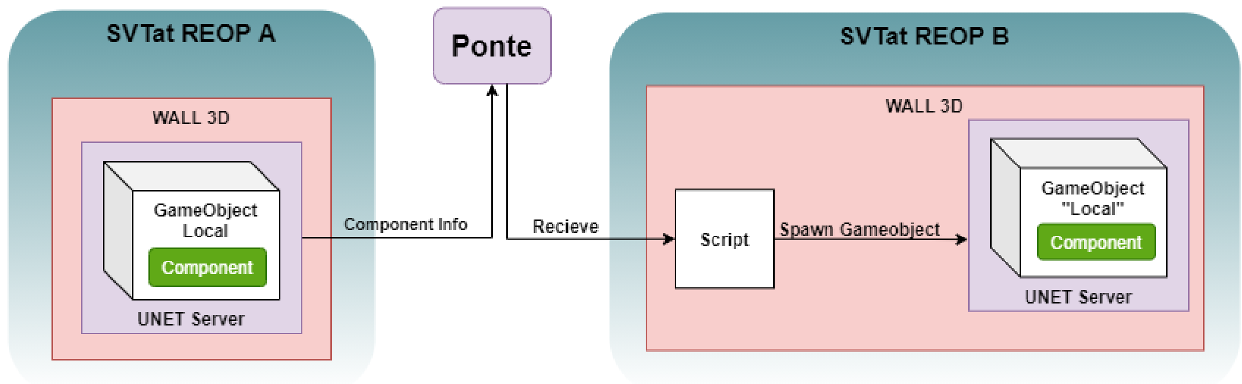
Ao escolher a arquitetura 2 o principal requisito é permitir que a estação WALL de cada instância SVTat seja capaz de enviar informações sobre eventos ocorridos em seu servidor UNET (que contêm e controla as trocas de informações entre as estações) como também receber o mesmo de outras instâncias.

Deste modo, no WALL, ao receber informações remotas, como um objeto que representa um carro blindado, poderá refletir isto entre suas estações. A partir desta informação serão criados objetos do mesmo modo que objetos locais são criados. Como as estações de uma instância perceberão este objeto remoto como se fosse local, se reduz a necessidade de programação adicional nas estações para que este objeto seja reconhecido e se integre à simulação.

Como visto anteriormente, a UNET trabalha pela troca de informações principalmente através de objetos nativos da Unity, como os *GameObjects*. No simulador diversos objetos que fazem parte da simulação, como baterias e viaturas, são modelados em termos de *GameObjects*. Como são os componentes destes *GameObjects* que o compõem, são eles que devem ser compartilhados através das tecnologias (ponte). A figura 3.3 ilustra esta troca de informações.

Como citado anteriormente (vide seção 2.6.2), o servidor UNET gerencia todos os *GameObjects* que foram tornados remotos e compartilhados entre as três estações. Assim, nesta figura é mostrado o simulador SVTat A enviando um *GameObject* local, que

Figura 3.3 – Compartilhamento de objetos na Unity



Fonte: O autor

pode representar um veículo, por exemplo, ao simulador SVTat B. Este *GameObject* é lido a partir do servidor UNET contido no WALL e a informação útil que está presente no componente (em cor verde) é enviada através da ponte ao SVTat B.

Ao receber essa informação no simulador SVTat B, um *script* lê os dados recebidos e cria (realiza o *spawn*) este *GameObject* no servidor UNET, do mesmo modo que *GameObject* locais são criados em um único simulador e suas estações. Assim este *GameObject* é compartilhado entre todas as estações. Posteriormente, também é necessário manter este *GameObject* que foi recebido remotamente atualizado, assim periodicamente haverá o envio de informações do componente pertencente ao *GameObject* do SVTat A para o SVTat B. No capítulo 5 é discutido os detalhes sobre cada implementação e como cada uma atingiu este requisito.

3.3 A COMPARAÇÃO DE TECNOLOGIAS

Para comparar as tecnologias DDS, HLA e TNET3 será implementada no simulador SVTat REOP a comunicação interna usando estas tecnologias, servindo como estudo de caso. A arquitetura usada será manter o SVTat REOP em seu estado atual e criar uma ponte para interligar diferentes instâncias deste simulador, através das estações WALL. Para comparar as tecnologias, este trabalho está dividido em três partes.

A primeira parte do trabalho irá mostrar os principais benefícios identificados em cada tecnologia. Esta parte do trabalho irá coletar as informações nas especificações destas tecnologias, não fazendo parte do estudo de caso das implementações no simulador. Esta parte também servirá de complementação para a parte de comparação das implementações e, posteriormente, para a conclusão do trabalho, sendo uma parte de caráter teórico/conceitual. Esta parte está descrita no capítulo 4.

A segunda parte do trabalho visa comparar as implementações e demonstrar qual

foi a mais viável, ou seja, qual tecnologia melhor supriu os requisitos e ocasionou o menor esforço para ser implementada. Como visto anteriormente, o principal requisito é o compartilhamento de informações sobre objetos da Unity, entre as estações WALL. Esta parte está no capítulo 5.

A terceira e última parte é a realização de testes de desempenhos entre as implementações, permitindo avaliar a capacidade e impacto de cada tecnologia no ambiente de simulação distribuído. Os testes de desempenho estão no capítulo 6.

3.4 PLANEJAMENTO DOS TESTES DE DESEMPENHO

Para a realização dos testes de desempenho, serão consideradas as principais métricas utilizadas para avaliar a performance em rede de computadores, tais como:

- **Throughput:** número total de *bytes* recebidos por segundo;
- **Round-Trip Time:** intervalo de tempo que uma mensagem leva para ir para um destinatário e voltar a sua origem;
- **Uso de CPU:** o quanto alguma aplicação consome de processamento;
- **Uso de memória:** o quanto alguma aplicação consome de memória; e
- **Vazão de mensagens/atributos:** quantas mensagens a tecnologia consegue transmitir por instante de tempo, independente do tamanho da mensagem.

A principal métrica é a vazão de dados (*Throughput*) porque esta métrica permite visualizar qual tecnologia suporta o maior envio de informações. No contexto do simulador isto permite que a simulação mantenha-se escalável ao aumentar o número de informações trocadas entre o simulador.

O RTT é uma métrica que permite medir a responsividade na troca de mensagens entre os simuladores, principalmente quando um simulador depende da resposta de outro. O motivo para utilizar somente o RTT e não a latência é que torna-se difícil sincronizar as máquinas, mesmo que localmente. Assim o uso do RTT independe de sincronização dos relógios das máquinas, já que se pode utilizar somente o relógio da entidade que mandou a mensagem, calculando o tempo entre a ida e retorno da mensagem.

O uso de CPU e memória torna-se importante no contexto de escalabilidade, já que a tecnologia utilizada estará dividindo o processamento e memória da máquina com o simulador. Por último a vazão de mensagens é importante, principalmente quando falamos de atualizações de atributos de objetos (*Keyed Topic*, *Object Class* ou *GameObjects*), quanto mais mensagens por segundo é possível receber, maior a capacidade de receber

atualização dos atributos de instâncias de objetos. Normalmente atualizações de atributos são da ordem de poucos *bytes*. O capítulo 6 entrará em mais detalhes sobre o *setup* dos testes de desempenho.

4 COMPARAÇÃO DAS PRINCIPAIS FUNCIONALIDADES ENTRE TECNOLOGIAS

Este capítulo demonstra as principais funcionalidades de cada tecnologia e exibe as distinções entre suas funcionalidades. Nem todas as funcionalidades das tecnologias podem se tornar um requisito durante a implementação (descrita no capítulo 5) mas devem pesar na escolha da tomada de decisão, já que simuladores estão sujeitos a mudanças e, portanto, novos requisitos podem existir.

Este capítulo está dividido em seis seções. A primeira seção 4.1 demonstra as diferenças em utilizar tecnologias internas e externas ao ambiente Unity. A segunda seção 4.2 demonstra alguns benefícios da organização em rede das tecnologias em relação a tolerância a falhas e escalabilidade na simulação distribuída. A terceira seção (4.3) aborda sobre a segurança provida por cada tecnologia durante a troca de informações na simulação. As seções 4.4, 4.5, e 4.6 falam sobre as políticas de QoS fornecidas pelas tecnologias. Por último, a seção 4.7 destaca detalhes sobre a questão de licenciamento e suporte das tecnologias.

4.1 TECNOLOGIAS EXTERNAS E INTERNAS

As tecnologias HLA e DDS são desenvolvidas para interoperar aplicações distribuídas. Essas aplicações podem ser desenvolvidas em diferentes plataformas (sistemas operacionais) e em diferentes linguagens. Como essas tecnologias não se limitam ao escopo da Unity, neste trabalho são denominadas de tecnologias externas. Por outro lado, a TNET3 é uma tecnologia desenvolvida exclusivamente para estabelecer a comunicação entre aplicações Unity e, portanto, limita-se a simuladores desenvolvidos na Unity, sendo assim chamada de tecnologia interna.

Como as tecnologias externas podem ser utilizadas em diferentes simuladores elas estabelecem um meio comum para a comunicação. No caso do HLA todos os simuladores utilizam o FOM. A adoção de uma tecnologia externa tem como benefício a possibilidade de se comunicar com outros tipos de simuladores. Ao utilizar a TNET3 somente é possível estabelecer a comunicação com simuladores Unity.

Outro fator da adoção de uma tecnologia externa é a necessidade de transformar objetos representados no ambiente Unity (*GameObject*) em objetos publicáveis. Como na Unity as representações de entidades/unidades são dadas através de *GameObjects*, para que seja possível enviar estes tipos de informação utilizando os padrões HLA e DDS, deve-se transformar as informações contidas nos componentes dos *GameObjects* para tópicos (no DDS) ou *Object Class* (no HLA).

No caso da TNET3 não há necessidade de transformação já que estes objetos

podem ser nativamente compartilhados em rede.

4.2 TOLERÂNCIA A FALHAS E ESCALABILIDADE

Dentre as tecnologias englobadas, HLA e TNET3 possuem um processo central separado dos clientes que assumem responsabilidades após a conexão. No HLA, após a conexão, o RTI abre conexões diretas entre os federados para a troca de objetos, interações e outros tipos de informação. Neste caso, o RTI fornece o papel de gerenciador, permitindo por exemplo, que os federados se conheçam após a conexão. Quando um federado entra atrasado na simulação o RTI notifica os federados já existentes, assim os federados podem notificar diretamente o novo federado sobre seus objetos usando a conexão direta entre os federados. Como o RTI não é usado para trocar informações de objetos da simulação, existe um sobrecarregamento menor sobre o RTI já que é utilizado para notificação de eventos entre os federados. O RTI também é responsável pelo gerenciamento de tempo entre os federados.

No caso da TNET3, ela utiliza a arquitetura cliente-servidor e seu servidor é responsável por trafegar todo tipo de dado. Isso pode levar a um sobrecarregamento maior quando muitos dados estão trafegando na simulação.

O DDS é o único que não necessita de um processo único e separado dos clientes para estabelecer a conexão ou trafegar os dados da simulação. O DDS é conhecido por ser totalmente distribuído, ou descentralizado. Assim, o DDS implementa o descobrimento dinâmico de novos participantes. Quando algum *DomainParticipant* descobre outro, eles guardam uma referência para poder estabelecer a conexão e posteriormente realizar a troca de dados direta.

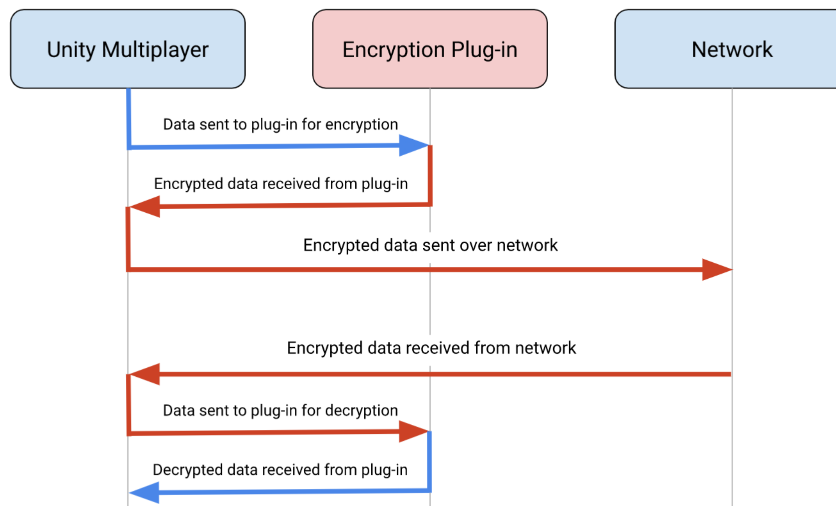
Utilizar processos centrais como HLA e TNET3 implica principalmente para a TNET3 um sobrecarregamento do processo quando há muitos dados ou objetos trafegando, afetando a escalabilidade da simulação. Outro fator é a tolerância a falhas, tanto no HLA como na TNET3 a falha de seus processos centrais implica na falha total da simulação distribuída porque a execução da simulação distribuída depende destes processos. O DDS, por outro lado, não sofre de nenhum desses problemas, pois não possui uma arquitetura centralizada.

4.3 SEGURANÇA

Segurança pode ser um requisito da simulação distribuída, principalmente no contexto militar onde tende-se a possuir informações sigilosas que trafegam pela rede. De

acordo com a documentação da Unity (UNITY, 2020), não existem mecanismos de segurança que encriptem os dados e por isso deve ser utilizado um *plugin* adicional para prover isto e evitar ataques a suas aplicações. A lógica deste tipo de *plugin* é exemplificada na figura 4.1. Dentre as tecnologias alvo deste trabalho, a única que fornece suporte a segurança dos dados é o DDS.

Figura 4.1 – Fluxo de encriptação de dados na Unity

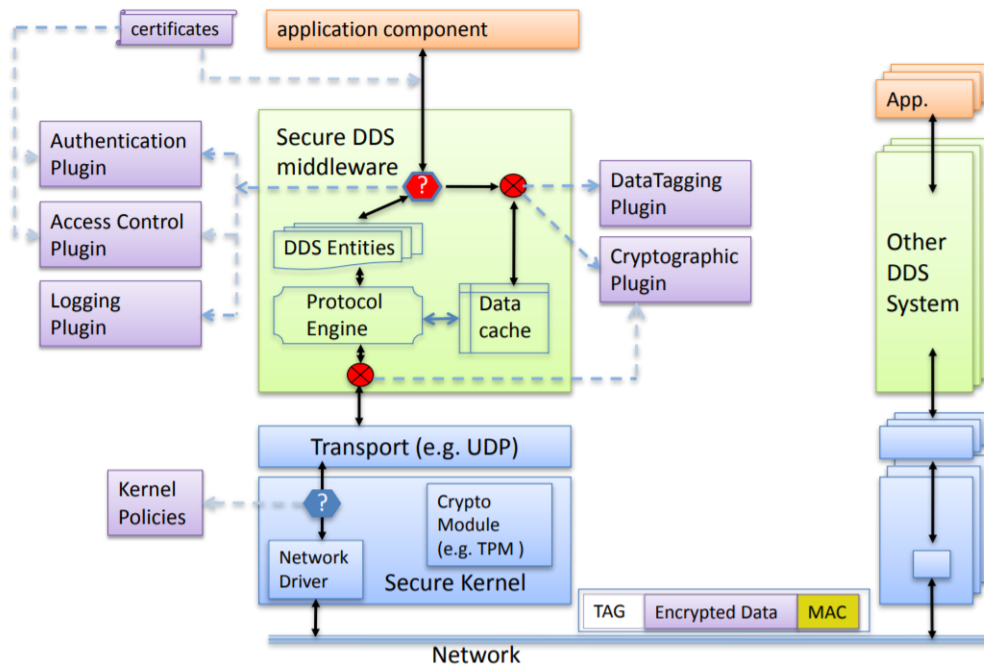


Fonte: (UNITY, 2020)

Segundo a especificação de segurança do DDS (OMG DDS 1.1, 2018), é fornecido ao usuário uma camada chamada *Security Model and Service Plugin Interface* (SPI) demonstrada na figura 4.2, e que fornece as seguintes funcionalidades:

- **Authentication Service Plugin:** permite verificar a identidade da aplicação que invoca operações, facilitando a autenticação entre os participantes;
- **Access Control Service Plugin:** permite fornecer políticas para a tomada de decisão sobre operações que um participante autenticado pode tomar, como, por exemplo, qual tópico pode ser publicado;
- **Cryptographic Service Plugin:** implementa operações de criptografia e descriptografia, *hashing* e assinaturas digitais;
- **Logging Service Plugin:** permite a auditoria de todos eventos relevantes para a segurança; e
- **Data Tagging Service Plugin:** permite adicionar uma marcação (*TAG*) a pacotes de dados.

Figura 4.2 – Arquitetura de segurança do DDS



Fonte: (OMG DDS 1.1, 2018)

4.4 POLÍTICAS DE QUALIDADE DE SERVIÇO

Esta seção demonstra as políticas de qualidade de serviço de cada tecnologia, todas tecnologias possuem em algum nível opções para ajuste de QoS. Assim também será demonstrado distinções entre as *QoS* amplamente suportadas. A tecnologia que mais tem destaque em oferecer as políticas de qualidade de serviço é o DDS, que será o principal foco desta seção.

4.4.1 Garantia de entrega e ordenação de mensagens

Todas as tecnologias suportam QoS que garantem a entrega de mensagens bem como a entrega ordenada por envio destas mensagens.

4.4.2 Escopo de dados

O DDS implementa a noção de partições (*Partitions*) que permitem que determinados tópicos assinados sejam notificados apenas para participantes (*DomainParticipant*) de determinadas partições. Isto serve como delimitador de informações permitindo a otimização de tráficos entre um mesmo tópico.

A TNET3 permite o uso de diferentes canais para o fluxo de informação funcionando

de forma similar as partições. Assim um *GameObject* que foi instanciado remotamente só trafega por um único canal, funcionando também como uma otimização de tráfego. O HLA é o único que não fornece esse tipo de mecanismo deste modo particular.

4.4.3 Filtro de dados

DDS e HLA provêem meios para a filtragem de dados. O DDS possui um mecanismo mais complexo para filtrar os dados (de tópicos) através de consultas SQL que podem ser aplicadas a dados já recebidos. Como o dado é filtrado somente ao ser recebido não funciona como um otimizador de tráfego.

O HLA possui as chamadas *Regions* que servem para filtrar atributos de instância de objetos ou interações. Este tipo de filtro é especificado no FOM e define intervalos de dados para que este dado seja válido e possa ser transmitido ao ambiente distribuído. Como este tipo de filtro é aplicado antes da publicação, ele funciona como uma otimização de tráfego. Os tipos de filtros que podem ser aplicados são especificados nas normas do HLA e possuem uma expressividade mais limitada do que a semântica SQL provida pelo DDS.

A TNET3 não possui mecanismos que permitam o filtro de dados já que trabalha com objetos nativos da Unity.

4.4.4 *Ownership Managment*

O *Ownership Managment* controla qual entidade (Cliente, Federado ou *DomainParticipant*) tem direito de editar/atualizar os atributos de determinada instância de objeto (Tópico, *GameObject* ou *Object Class*). Todas tecnologias suportam esta tecnologia, mas com distinções.

No HLA um federado pode ser dono de atributos de uma instância de objeto e não da instância inteira. Ele suporta que apenas um federado seja dono do atributo do objeto por vez e permite a negociação de *onwership* entre os federados em tempo de execução.

No DDS não existe *onwership* por atributos mas pelo tópico inteiro. Não existe negociação de *ownership* em tempo de execução. Ao criar os tópicos é especificado se ele é exclusivo ou compartilhado, ou seja, no DDS é possível que diversos *DomainParticipant* compartilhem o *onwership* de um mesmo tópico.

A TNET3 não possui exatamente esta noção, porém em tempo de execução é possível verificar se o cliente atual foi o criador do objeto, podendo facilmente simular o comportamento em que apenas o cliente criador do *GameObject* possa editar seu estado. Assim, negociações de *onwership* podem ser implementadas, se necessário.

4.4.5 QoS exclusivas do DDS

Esta seção exemplificará as principais políticas de QoS exclusivas do DDS.

- **Lifespan:** permite especificar o atraso máximo para o assinante receber determinados tópicos. Assim, tópicos considerados antigos são descartados.
- **Presentation:** permite especificar a ordem do recebimento de tópicos, ou seja, atualização de tópicos, mesmo que de diferentes tipos sejam recebidos na ordem de envio.
- **Destination Order:** permite especificar a ordem do recebimento de atualização de um tópico, de acordo com qual *DomainParticipant* realizou esta atualização ou por ordem de tempo.
- **Resource Limits:** permite especificar o máximo de instâncias de tópicos a serem mantidos em memória.
- **Time based filter:** permite especificar um intervalo mínimo e máximo para que tópicos sejam recebidos. Em outras palavras especifica qual a frequência em que um assinante pode receber os tópicos.
- **Durability:** controla o tempo de vida dos tópicos existentes em um domínio.
- **History:** especifica o tamanho do histórico para guardar o recebimento de tópicos (atualizações subsequentes de um mesmo tópico).
- **Transport Priority:** especifica a importância sobre um tópico a ser transmitido.

4.5 GERENCIAMENTO DE TEMPO

Em questões de entrega de mensagens, as tecnologias HLA e DDS possuem suporte para entregar mensagens ordenadas por tempo (*Time Stamp Order*). Na TNET3 não existem mecanismos de sincronização baseada em tempo.

O HLA é o único que possui especificado em seu padrão o gerenciamento de tempo (*Time Management*) que permite que cada federado possua seu próprio relógio que determina qual a sua posição na linha do tempo de simulação. Ao mandar mensagens (instâncias de objetos ou interações), outros federados somente receberão a mensagem se seu relógio estiver no mesmo instante da simulação que do federado que publicou esta

informação. O RTI é responsável por guardar o relógio de cada federado e garantir que a troca esteja correta. Por exemplo, quando um federado deseja avançar seu relógio ele deve pedir ao RTI.

Outra funcionalidade do HLA são os pontos de sincronização. Isto permite que os federados anunciem eventos permitindo que todos os federados se sincronizem. Um exemplo é criação de um ponto de sincronização onde todos os federados carregam o cenário da simulação. Cada federado ao carregar o cenário deverá anunciar que atingiu o ponto de sincronização. Assim somente quando todos os federados declararem que chegarem ao ponto de sincronização a execução continua. Todos estes mecanismos providos pelo HLA permitem que simulações sensíveis ao tempo sejam executadas sem a necessidade de implementação adicional. Nas outras tecnologias isto deve ser implementado.

4.6 METADADOS

A troca de informações adicionais está explicitamente descrita nas tecnologias HLA e DDS. No HLA existe o *User Supplied Tag* consistindo em informações adicionais que podem ser trocadas entre os federados, ao gerar atualizações sobre instâncias de objetos. No DDS é fornecida uma QoS que permite que informações adicionais sejam trocadas entre os tópicos, chamada TOPIC_DATA. A TNET3 não fornece este tipo de mecanismo, entretanto como fornece o uso de chamadas de funções remotas, torna-se trivial a implementação de troca de informações adicionais.

4.7 LICENÇA E SUPORTE

A TNET3 é fornecida pela loja de *assets* da Unity sendo necessário comprá-la. O suporte é dado via o fórum da empresa sem limitações. Não há garantias que a TNET3 não seja descontinuada futuramente, já que a Unity esta em constante evolução e melhores tecnologias de comunicação podem surgir.

A versão do *middleware* HLA (RTI) utilizada é provida pela *Pitch Technologies* (Pitch Technologies pRTI, 2021). Ao contrário da TNET3 existem limitações em seu uso. Deve-se comprar uma licença para poder utilizar o RTI. A licença é por federado de uma simulação, limitando o número de federados que podem estar conectados ao RTI, ou podendo gerar aumento de custo em licenças. Outro fator é que o suporte deve ser renovado periodicamente, não sendo vitalício. Como o HLA é um padrão largamente usado na simulação existe uma menor chance de ser descontinuado.

Para o DDS, neste trabalho foi utilizada a versão OpenSplice, sendo esta uma ver-

são gratuita e liberada para a comunidade. Também existe a versão que pode ser comprada, onde então pode-se haver suporte para o uso comercial. Também existe a versão chamada OpenDDS, sendo esta uma versão aberta do DDS mantida pela OMG. O suporte comercial também deve ser requisitado à OMG, caso necessário. No caso do OpenDDS também existe uma menor tendência a descontinuação já que é um padrão mantido e criado pela própria OMG.

5 COMPARAÇÃO DE IMPLEMENTAÇÕES

Este capítulo foca em demonstrar como foram realizadas as implementações no simulador SVTat REOP, discutindo a viabilidade de cada implementação, ou seja, qual melhor atendeu os requisitos do simulador na implementação e necessitou menos esforço para ser programada. A primeira seção 5.1 descreve quais as estruturas do simulador foram consideradas durante a implementação. A seção 5.2 descreve a etapa de implementação dos *plugins* DLL para DDS e HLA. As seções seguintes (5.3, 5.4 e 5.5) descrevem a etapa de implementação utilizando as tecnologias HLA, DDS e TNET3, respectivamente. Por último, a seção 5.6, baseada nas seções anteriores, ressalta a viabilidade de cada implementação.

5.1 IMPLEMENTAÇÃO NO SVTAT REOP

Como o simulador tem como objetivo o treinamento de militares sobre viaturas lançadoras de mísseis/foguetes, uma das principais unidades no SVTat são os veículos (ou viaturas). Portanto, para a implementação foram considerados seis veículos pertencentes a bateria ASTROS. O motivo para considerar apenas os veículos de uma bateria se dá porque no escopo de teste não é necessário implementar a comunicação entre todos os módulos/entidades do simulador.

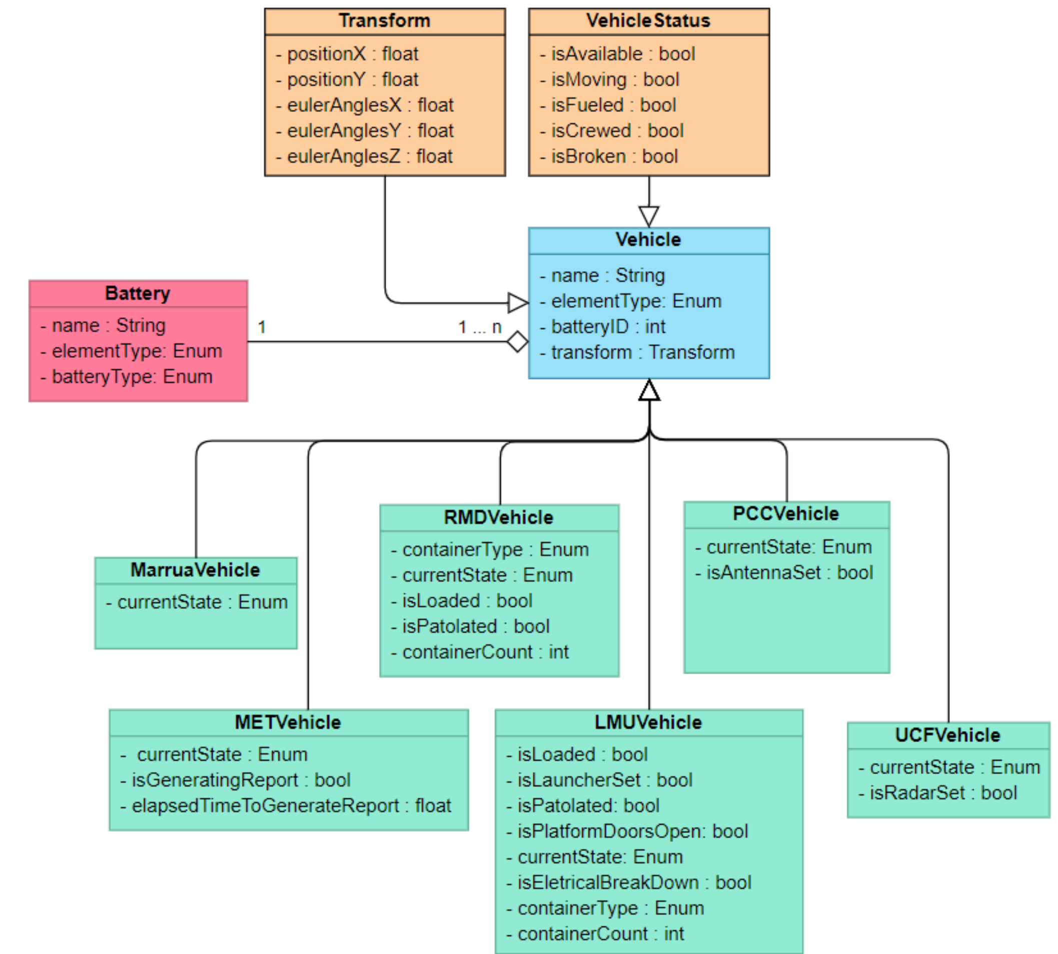
Os veículos que foram considerados para este trabalho são parte do Sistema ASTROS e chamam-se PCC (Posto de Comando e Controle), UCF (Unidade Controladora de Fogo), LMU (Lançadora Múltipla Universal), RMD (Remuniciadora), MET (Posto Meteorológico) e a viatura Marruá (viatura leve de reconhecimento).

Na figura 5.1 está ilustrado em um diagrama de classe seis viaturas (em verde) que herdam a classe base *Vehicle*. Também há a bateria (em rosa) que pode possuir diversos objetos do tipo *Vehicle*. O significado dos atributos de cada entidade não será explicado já que não é necessário para o entendimento do trabalho.

No simulador SVTat estas entidades, tanto baterias como veículos, estão representadas a partir de *GameObjects*. A estrutura descrita na figura 5.1 faz parte dos componentes que compõem o *GameObject* que representa estas viaturas/baterias. Também existem outros componentes que são responsáveis por outras tarefas, como a representação 3D da viatura, cálculos físicos e sincronização dos dados de rede na UNET. Entretanto, estes tipos de componentes não precisam ser compartilhados pois são instanciados pelo próprio simulador durante o recebimento de um objeto remoto.

Assim, como citado anteriormente, o principal requisito é permitir que os componentes que representam as viaturas/baterias (que são compartilhadas entre as estações

Figura 5.1 – Estrutura comum implementada utilizando as tecnologias



Fonte: O autor

através da UNET) sejam compartilhados em rede, para que nos simuladores remotos este dado seja recebido e usado para a criação de viaturas/baterias locais. Posteriormente, também estes dados precisam ser mantidos atualizados/sincronizados em rede, para que as mudanças de estado das viaturas e baterias sejam refletidas em outros simuladores. Este requisito está ilustrado na figura 3.3.

5.2 USO DE PLUGIN DLL

O primeiro passo de implementação presente nos padrões HLA e DDS e ausente na API TNET3 é a criação de um *plugin* DLL. Isto é um requisito para a utilização destas tecnologias por causa da linguagem nativa utilizada pelas implementações do DDS e HLA. A DLL permite que o código escrito em C++ seja importado e utilizado pela Unity que utiliza a linguagem C#. A DLL é criada a partir do código gerado pelos geradores de códigos utilizados pelos padrões HLA e DDS. Estes geradores recebem como entrada os

arquivos IDL/FOM que foram modelados com os dados da figura 5.1.

O uso de uma DLL implica em manter uma parte do código separado da Unity. Na DLL deverá ser desenvolvida a lógica para a troca de dados, de modo que forneça os serviços necessários como a criação e atualização de objetos e o registro de *callbacks* para recebimento de novos objetos/atributos de objetos. Isto permitirá que a Unity desfrute das funcionalidades dos padrões HLA e DDS mesmo que implementados em uma linguagem diferente. Posteriormente se desenvolve uma interface de exportação na DLL, possibilitando que a Unity utilize as funcionalidades fornecidas pela DLL (demonstrado na figura 5.2).

Figura 5.2 – Interface de exportação de *plugin* DLL

```
extern "C"
{
    __declspec(dllexport) void Connect(char* host, int port, OnBatteryDeclaredCallback c0, OnLMUVehicleDeclared c1, OnPCCVehicleDeclared c2, OnUCFVehicleDeclared c3, OnRemuV
        OnUpdateElementAngleCallback c9, OnUpdateVehicleStatusCallback c10, OnUpdateUCFStatusCallback c11, OnUpdateLMUStatusCallback c12, OnUpdateMarruaStatusCallback c13, O

    __declspec(dllexport) void Disconnect(){ ... }

    __declspec(dllexport) int CreateBattery(char* name, int nvehicles, DevStudio::BatteryType::BatteryType type, float x, float y, float xx, float yy, float zz){ ... }

    __declspec(dllexport) int CreateLMU(BaseInstance instance, VehicleNetStatusAttribute_Aux status, LMUNetStatusAttribute_Aux other){ ... }

    __declspec(dllexport) int CreatePCC(BaseInstance instance, VehicleNetStatusAttribute_Aux status, PCCNetStatusAttribute_Aux other){ ... }

    __declspec(dllexport) int CreateUCF(BaseInstance instance, VehicleNetStatusAttribute_Aux status, UCFNetStatusAttribute_Aux other){ ... }

    __declspec(dllexport) int CreateRemu(BaseInstance instance, VehicleNetStatusAttribute_Aux status, RemuNetStatusAttribute_Aux other){ ... }

    __declspec(dllexport) int CreateMarrua(BaseInstance instance, VehicleNetStatusAttribute_Aux status, MarruaState::MarruaState other){ ... }

    __declspec(dllexport) int CreateMeteo(BaseInstance instance, VehicleNetStatusAttribute_Aux status, MeteoNetStatusAttribute_Aux other){ ... }

    __declspec(dllexport) void UpdateVehiclePosAngleName(int ID, float x, float y, float xx, float yy, float zz, char* name, VehicleType type){ ... }

    __declspec(dllexport) void UpdateBattPosAngleName(int ID, float x, float y, float xx, float yy, float zz, char* name){ ... }

    __declspec(dllexport) void UpdateLMUStatus(int ID, VehicleNetStatusAttribute_Aux status, LMUNetStatusAttribute_Aux other){ ... }

    __declspec(dllexport) void UpdatePCCStatus(int ID, VehicleNetStatusAttribute_Aux status, PCCNetStatusAttribute_Aux other){ ... }

    __declspec(dllexport) void UpdateUCFStatus(int ID, VehicleNetStatusAttribute_Aux status, UCFNetStatusAttribute_Aux other){ ... }

    __declspec(dllexport) void UpdateRemuStatus(int ID, VehicleNetStatusAttribute_Aux status, RemuNetStatusAttribute_Aux other){ ... }

    __declspec(dllexport) void UpdateMarruaStatus(int ID, VehicleNetStatusAttribute_Aux status, DevStudio::MarruaState::MarruaState other){ ... }

    __declspec(dllexport) void UpdateMeteoStatus(int ID, VehicleNetStatusAttribute_Aux status, MeteoNetStatusAttribute_Aux other){ ... }
```

Fonte: O autor

Além de gerar a DLL, também há necessidade de importar a DLL na Unity e garantir que a Unity use-a de maneira correta. Isto ocorre quando já é possível utilizar as funções para criar objetos e atualizá-los assim como receber objetos remotos da DLL e, principalmente, permitir que este objeto ao ser recebido, possua uma representação na linguagem C#.

Existem diversos motivos que justificam manter estes dados na linguagem C#. O primeiro é a manutenibilidade e legibilidade do código, já que manipular a informação diretamente no código da Unity torna a implementação mais fácil e trivial. Outro motivo para manter uma representação destes dados, principalmente ao recebê-los da DLL é para permitir que sejam consumidos pela Unity posteriormente. Como estes dados são recebidos via *callbacks* não é possível invocar funções da Unity para que estes dados sejam consumidos diretamente. Isto ocorre por que estas *callbacks* são invocadas por uma *thread* diferente da *thread* principal da aplicação Unity, não sendo permitido o uso de funções da Unity por outras *threads*. Assim guarda-se este objeto para consumo posterior. Todo o fluxo do uso da DLL está representado na figura 5.3.

Uma desvantagem existente ao transformar objetos entre linguagem (conhecido como *Unmarshaling/Marshaling*) é a diferença de representação de classes na memória. Alguns tipos de dados, principalmente os mais complexos, são representados de diferentes maneiras na memória pelas linguagens C# e C++ e, portanto, não podem ser passados diretamente pelas chamadas de funções da DLL. Por causa disto é necessário recriar parcialmente os dados que foram gerados pelo geradores de código na DLL. Assim, ao recriar estes dados de outra forma, em ambas as linguagens torna-se possível passar classes/estruturas diretamente pelas funções da DLL.

Esta desvantagem citada, ocorre principalmente ao utilizar classes que possuem tipos de dados que são nativos da linguagem C++. Inclusive foi necessário transformar as classes C++ para *structs* de modo que a transformação funcione, ou seja, foi necessária a reescrita de algumas estruturas geradas pelo gerador na DLL. Tipos primitivos como *floats*, inteiros e *string* não trazem este tipo de problema.

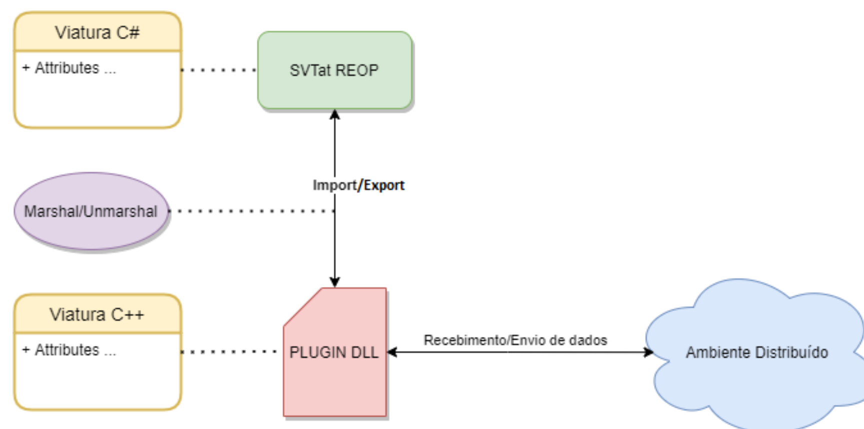


Figura 5.3 – Fluxo entre DLL e Unity
Fonte: O autor

Outro fator que contribuiu com o problema de transformação entre dados, se dá pelos geradores de código. Os geradores criam as classes para representar os dados descritos nos arquivos FOM e IDL. Entretanto estas classes possuem atributos e funções adicionais, não especificados nestes arquivos mas que são utilizados internamente por estes geradores para o funcionamento do código.

Estes dados e funções adicionais, além de não serem úteis para o consumo no ambiente de simulação na Unity, dificultam que a classe seja compartilhada diretamente pela DLL, devido a seus componentes adicionais. Esta é a razão pelo qual na figura 5.2 os atributos, tanto na parte de criação como na de atualização de objetos, estão divididos em algumas estruturas que não foram geradas diretamente pelos geradores de código. Este empecilho também foi encontrado no trabalho relacionado 3 (seção 2.7) durante a integração da Unity à DLL.

Apesar destes passos adicionais na implementação, o uso de DLL traz alguns benefícios. Uma vantagem de utilizar uma DLL é o seu desacoplamento. Como a DLL é uma

parte de código separado que fornece uma série de funções a serem exportadas, é mais fácil realizar a manutenção ou substituição desta DLL pois está desacoplada do ambiente Unity, bastando apenas cumprir com o comportamento gerado pelas funções exportadas.

Esta DLL também pode ser reutilizada em outros simuladores, mesmo que estes não sejam desenvolvidos na Unity, bastando apenas realizar a importação das funções.

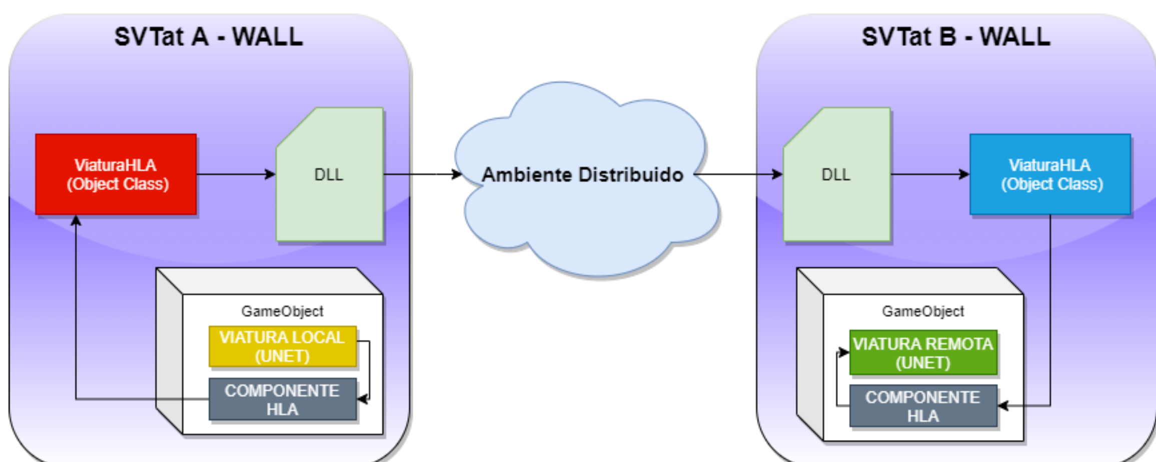
5.3 IMPLEMENTAÇÃO HLA

Para a implementação do HLA foi utilizado o *software* DevStudio que prove a geração de código, este software gera códigos na linguagem C++. O *middleware* utilizado é fornecido pela empresa *Pitch Technologies* (Pitch Technologies pRTI, 2021). A versão do HLA utilizada chama-se HLA 1516 EVOLVED.

No HLA cada estação WALL pertencente a uma instância SVTat REOP foi representada por um federado que se conecta na federação. Os componentes que representam as viaturas e baterias (equivalentes aos dados da figura 5.1), foram representados no HLA através do uso de *Object Class*, que representa nesta implementação as instâncias de viaturas e baterias.

Para a implementação da comunicação entre as estações WALL, cada *GameObject* que representa uma viatura recebeu um componente, que é um *script* responsável por ler os dados do componente que representa a viatura e que precisa ser compartilhado entre as instâncias do SVTat REOP. Este *script* após realizar esta leitura irá transformar estes dados em um *Object Class* para que torne-se um objeto publicável, então irá publicá-los ao ambiente distribuído.

Figura 5.4 – Ciclo de informações usando HLA



Fonte: O Autor

Na figura 5.4 é possível ver este processo inicial na estação WALL do SVTat A (a esquerda). Esta figura ilustra o envio de um *GameObject*, que representa uma viatura, do

WALL do SVTat A para o WALL do SVTat B.

No SVTat A, o componente e *script* em cinza (Componente HLA) lê os dados do componente em amarelo (Viatura Local UNET), que representa os dados da viatura pertencente ao servidor UNET. Após esta leitura ele transforma este dados para um *Object Class* (em vermelho), que é um objeto que pode ser enviado para a DLL para ser publicado. Posteriormente a publicação, o componente HLA também torna-se responsável por ler o estado da viatura local (em amarelo) e invocar funções da DLL para o envio de atualizações sobre o componente que representará a viatura.

Após este objeto chegar ao WALL do SVTat B, o processo oposto ocorrerá. Quando a DLL notificar via *callback* o recebimento de uma nova viatura (em azul), esta viatura será salva em uma lista. Isto é necessário por que *callbacks* são invocadas por uma *thread* diferente da *thread* Unity, e uma *thread* diferente não pode invocar as funções da Unity.

Quando a *thread* da Unity perceber a existência de uma nova viatura nesta lista, será instanciado um novo *GameObject* no servidor UNET do SVTat B, que representara a mesma viatura do SVTat A, ao qual será populada com os dados recebidos remotamente (em azul).

Estes dados serão inseridos no componente que representa a viatura (em verde). Nesta caso também haverá um componente HLA (em cinza), mas este será responsável por ler periodicamente as viaturas recebidas e atualizações destas viaturas (em azul), que estarão disponíveis na lista citada, e inseri-los no componente da viatura (em verde). Todos estes processos permitiram que as estações recebam esta viatura e a percebam como um objeto local.

Todo este processo também foi aplicado para as baterias. Por parte do HLA não houve qualquer limitação em atender os requisitos para implementar a comunicação entre instâncias SVTat. Para esta implementação foram seguidos os seguintes passos:

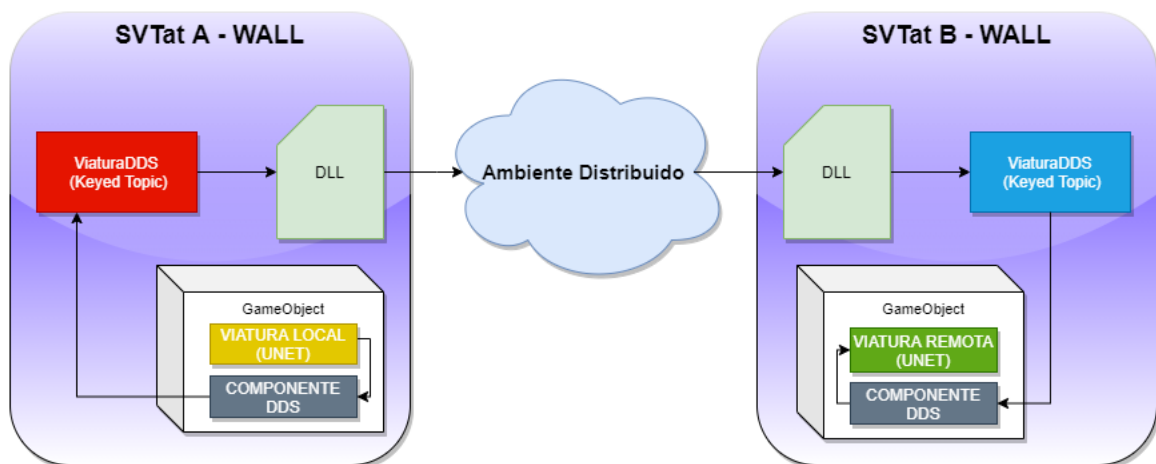
- **1:** criação do documento FOM para a geração, através do DevStudio, das classes que serão utilizados na DLL;
- **2:** desenvolvimento da lógica interna da DLL e definição das funções exportadas. Também houve a necessidade de recriar algumas classes devido a problemas entre transformação de dados entre DLL e Unity;
- **3:** no ambiente Unity, importar as funções da DLL e realizar a criação de classes que representam as viaturas/baterias do HLA na linguagem C# (na figura anterior está na cor vermelha/azul) que são recebidas ou mandadas através da DLL; e
- **4:** desenvolvimento da lógica interna do simulador, ou seja, integrar na simulação objetos locais e remotos. Para isso desenvolve-se o componente HLA (cor cinza) que será inserido junto ao *GameObject* das viaturas/baterias e permitirá o envio/recebimento de dados, bem como mantê-los atualizados.

5.4 IMPLEMENTAÇÃO DDS

Para o DDS foi utilizada a versão *OpenSplice* desenvolvida pela empresa *ADLINK-IST* (ADLINK, 2021), também utilizado na linguagem C++. Nesta implementação cada *DomainParticipant* representou uma instância do WALL pertencente a um SVTat REOP que está conectada a um *Domain*. Foi utilizado o mesmo modo que o HLA para estabelecer a comunicação entre as instâncias SVTat REOP. A figura 5.5 representa duas instâncias do SVTat REOP, onde o SVTat A envia uma viatura ao SVTat B.

Conforme a figura, no SVTat A, também foi criado um componente (em cinza) que será adicionado em cada viatura, sendo este responsável por ler os dados do componente que representa a viatura na UNET (em amarelo), para que neste caso o transforme para um *Keyed Topic* (em vermelho), possibilitando o envio a DLL para a publicação. Do mesmo modo, este componente também é responsável por ler periodicamente os atributos do componente que representa a viatura local (em amarelo) e enviar estas atualizações via DLL.

Figura 5.5 – Ciclo de informações usando DDS



Fonte: O Autor

No SVTat B ocorre o mesmo processo realizado no HLA. Esta instância de tópico é recebida pela DLL. Então a DLL encaminha essa instância (em azul) através de *callbacks* registradas na conexão, que ficará salva em uma lista contendo todas as viaturas.

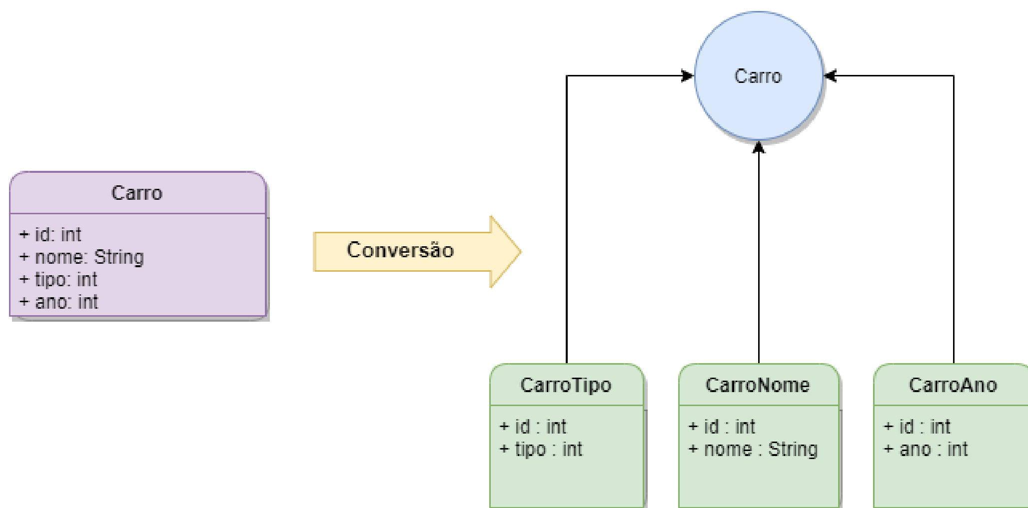
Assim a Unity poderá acessar este novo tópico recebido e instanciará um novo *GameObject* no servidor UNET para representar a viatura recebida. Este novo *GameObject* terá seu componente que representa a viatura (em verde) populado com os novos dados recebidos. Posteriormente o componente DDS (em cinza) será responsável por ler as atualizações de dados recebidas desta viatura, através do acesso a lista, e refleti-los no componente da viatura (em verde). Todo este processo também é realizado para os tópicos do tipo Bateria.

Apesar do processo ser semelhante ao do HLA, o DDS demonstrou menos flexi-

bilidade ao atender alguns requisitos de interesse. Apesar dos veículos/baterias serem devidamente representadas no arquivo IDL, o DDS não permite a atualização individual dos atributos de um tópico. Quando deseja-se atualizar um atributo de um tópico é preciso atualizar o objeto inteiro mesmo que apenas um de seus atributos tenha mudado. Isto ocorre porque o DDS trata um tópico como uma mensagem e não como um objeto com atributos.

O maior problema em não permitir a atualização de atributos de um tópico é a necessidade de mandar o objeto inteiro pela rede cada vez que o objeto mudar de estado. Isto tende a gerar um problema de performance em rede já que todo o tópico terá que trafegar novamente de um simulador a outro.

Figura 5.6 – Conversão de dados requerida pelo DDS



Fonte: O autor

Uma possível solução para este problema está descrita na figura 5.6. A classe hipotética chamada Carro, em cor roxa, demonstra um exemplo de como uma classe seria trivialmente descrita nas tecnologias HLA e TNET3. No DDS se a classe fosse descrita deste modo, utilizando apenas um tópico, quando ocorre a necessidade de atualização de algum atributo todo o tópico Carro teria que ser mandado pela rede.

A solução é demonstrada através da conversão da classe roxa para as três classes verdes. Cada classe pode carregar um atributo do objeto Carro e um id que identifica a qual instância de Carro este atributo pertence. Deste modo o problema de tráfego é mitigado. Note que ao invés de haver apenas um tópico equivalente a Classe Carro (em roxo), haverão três tópicos que representam o objeto Carro. Porém, esta solução acarreta em outros problemas. Haverá um maior esforço do programador tanto para modificar/remodelar a representação do objeto no arquivo IDL, quanto na hora de implementar. Esta parte deve ser levada em consideração em todas etapas de implementação, pois além de tornar o código mais difícil de realizar a manutenção, também o torna menos legível.

Como o tópico carro não pode ser representado da sua maneira trivial, ao receber

este tópico na Unity é necessário juntar as partes fragmentadas e juntá-las novamente no objeto original (cor roxa). O mesmo processo ocorre ao enviar o objeto pela rede, mas desta vez o objeto deve ser dividido entre vários tópicos, que trafegaram individualmente pela rede.

Outro problema ocorre quando há necessidade de que os tópicos que representam os objetos precisem chegar de maneira ordenada. Por exemplo, caso um veículo chegue antes da declaração de sua bateria não é possível instanciar o veículo já que o mesmo depende de informações ligadas a sua bateria. Existem duas soluções para este problema. A primeira é guardar os tópicos e aguardar que outros tópicos dependentes cheguem. A segunda é usar uma QoS para que as mensagens cheguem em ordem de *timestamp* (TSO). Desse modo, as mensagens chegam ordenadas pelo instante de tempo em que foram enviadas.

Todos estes processos aumentam a complexidade do código e o trabalho que o programador tem em implementar algo que contorna este problema. Este método teve que ser aplicado a todas as estruturas de dados das viaturas/baterias que foram modeladas neste trabalho. Além deste problema, não houve qualquer outra limitação em atender os requisitos de implementação. Para esta implementação foi seguido os seguintes passos:

- **1:** criação do documento IDL para a geração das classes através do gerador de código nativo (IDLPP) que serão utilizados na DLL, levando em conta o problema na atualização de atributos;

- **2:** desenvolvimento da lógica interna da DLL, definindo as funções exportadas. Assim como a implementação HLA, houve a necessidade de recriar algumas classes devido ao problema de transformação de dados entre DLL e Unity. Esta parte também tornou-se significativamente mais complicada devido ao problema na atualização de atributos dos tópicos;

- **3:** no ambiente Unity, importar as funções da DLL e realizar a criação de classes que representam as viaturas/baterias no DDS na linguagem C# (na figura 5.5 está na cor vermelha ou azul) que são recebidos ou mandados através da DLL. Esta etapa também foi afetada pelo problema de atualização de tópicos, já que ao receber os tópicos fragmentados é necessário juntá-los na representação normal; e

- **4:** desenvolvimento da lógica interna do simulador, ou seja, integrar objetos C# (na figura 5.5 em vermelho ou azul) que são provindos do DDS juntos as viaturas UNET do simulador WALL. Para isso desenvolve-se o componente DDS (cor cinza) que será inserido junto ao *GameObject* das viaturas/baterias e permitirá o envio/recebimento destes dados e suas atualizações.

5.5 IMPLEMENTAÇÃO TNET3

A TNET3 é fornecida pela loja de *assets* da Unity e produzida pela empresa *Tasharem*, sendo nativo a linguagem C#, a mesma utilizada na Unity. Nesta tecnologia cada instância WALL foi representada como um cliente que se conecta ao servidor da TNET3.

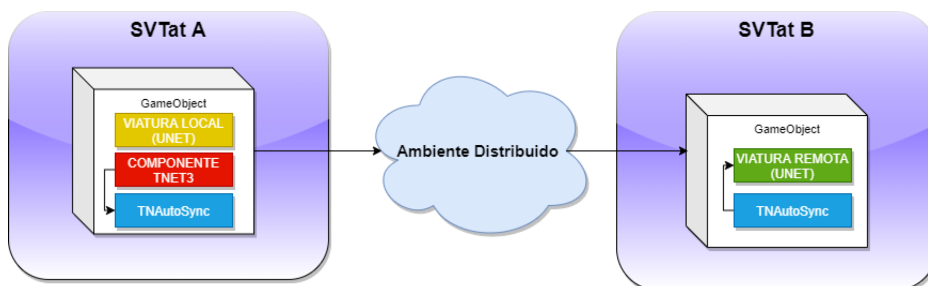
Esta tecnologia trabalha de maneira diferente dos padrões HLA e DDS porque é diretamente integrada ao ambiente Unity. Assim a troca de informações se dá por objetos nativos da Unity (*GameObjects*). Estes objetos podem ser compartilhados e atualizados sem a necessidade de programação adicional. No caso dos padrões DDS e HLA foi necessário converter os componentes de *GameObjects* que representam as viaturas/baterias e transformá-los para tópicos/*Object Class*, tanto na parte de envio como recebimento. Na TNET3 não existe a necessidade desse tipo de transformação.

Para a implementação houve apenas a necessidade de enviar o *GameObject* que representa a viatura atráves da chamada RCC (*Remote Create Call*). Na figura 5.7 está ilustrado todo processo, onde compartilha-se um *GameObject* que representa uma viatura, do WALL do SVTat A para o WALL do SVTat B.

Para tornar isto possível, no SVTat A, foi adicionado um componente (em vermelho) que é um *script* responsável por dar *spawn* neste *GameObject* através da RCC (*Remote Create Call*), ou seja, tornar este *GameObject* remoto, compartilhando-o entre os clientes. Após isto, este *GameObject* irá chegar automaticamente ao SVTat B com apenas o componente requerido que representa a viatura (em verde). Isto é possível por que através da RCC é possível modificar o *GameObject* que irá ser instanciado em outros clientes.

Para que o SVTat A mantenha este *GameObject* atualizado em outros clientes, também é adicionado um componente chamado *TNAutoSync* (em azul) no objeto original. Este componente permite selecionar quais componentes ou partes de componente serão automaticamente refletidos nos *GameObjects* que foram compartilhados com outros clientes. Dessa forma, o componente que representa a viatura local (em amarelo) foi selecionado para ser atualizado. Assim periodicamente o componente da viatura (em amarelo) será refletido no mesmo componente do SVTat B, o componente viatura remota (em verde).

Figura 5.7 – Estrutura implementada utilizando a TNET3



Fonte: O autor

Como pode-se notar a TNET3 fornece uma API que exige um menor esforço do

programador Unity para estabelecer a troca de informações entre aplicações Unity. Para esta implementação foram seguidos os seguintes passos:

- **1:** criação de um componente (*script*), responsável por tornar o *GameObject* que representa a viatura remota. Cria-se a RCC que corresponde a um código que será invocado em todos clientes, para criação de um *GameObject*;
- **2:** adição do componente *TNAutoSync* para permitir a atualização do componente que representa os dados da viatura; e
- **3:** após receber uma viatura remota, desenvolver a lógica interna do simulador para que este *GameObject* seja utilizado corretamente na simulação do mesmo modo que as viaturas locais.

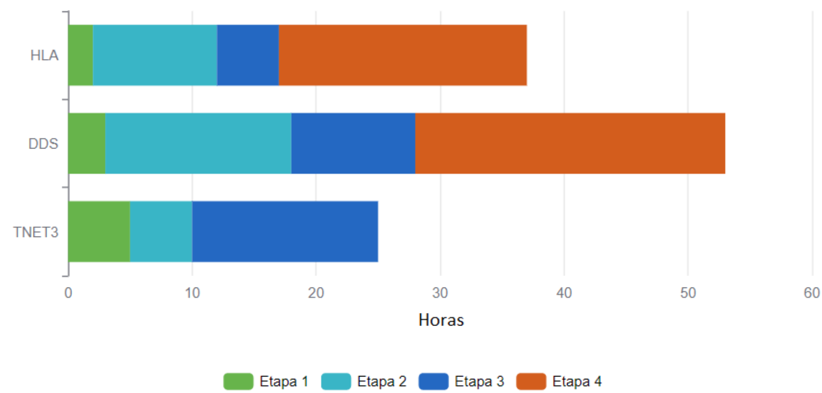
5.6 VIABILIDADE DAS IMPLEMENTAÇÕES

Todas as implementações foram realizadas com sucesso e atenderam aos requisitos de interesse. Entretanto, algumas implementações se demonstraram menos viáveis do ponto de vista de programação. A implementação menos viável foi a que utilizou o DDS. O principal motivo foi a ausência da possibilidade de atualizar atributos de tópicos. Isto fez com que houvesse a necessidade de modificar toda a estrutura da figura 5.1 para que tópicos passem a desempenhar o papel de atributos e não de instâncias de objetos, além da necessidade de ajustar os passos de implementação levando em conta este aspecto.

A TNET3 se mostrou a tecnologia mais ágil para implementar a comunicação entre as instâncias do SVTat. Ela permite que os objetos nativos da Unity sejam serializados e compartilhados em rede com apenas um comando. Também permite que a atualização desses objetos seja dada a partir da adição de um componente de sincronização, onde somente é necessário selecionar quais os atributos de componentes serão sincronizados em redes, sem a necessidade de implementação adicional. Outro fator importante é a reutilização dos dados já existentes no simulador. A TNET3 permite que componentes, *GameObjects* e classes, sejam reaproveitados na comunicação já que eles podem ser serializados e compartilhados pelas chamadas de procedimento remoto.

Utilizando as tecnologias externas, há a necessidade de programar a integração entre componentes e tópicos/*Object Class*, para que possa criar estes objetos na rede e mantê-los atualizados. Também há a necessidade da criação do *plugin* DLL e a sua integração ao ambiente Unity.

Figura 5.8 – Peso das etapas de implementação (tempo médio de cada etapa)



Fonte: O autor

O gráfico de barras na figura 5.8, representa as etapas de implementação. Foi atribuído um valor a cada etapa do processo de implementação (citadas anteriormente em cada seção de implementação). Esta pontuação levou em conta o tempo médio gasto em cada etapa de implementação.

6 TESTES DE DESEMPENHO

Este capítulo irá mostrar os resultados dos testes de desempenho realizados sobre as tecnologias. A primeira seção (6.1), descreve o cenário de teste que foi utilizado. A seção 6.2 demonstra os resultados do teste de *payload*. Por último a seção 6.3 demonstra os resultados de desempenho utilizando as estruturas do simulador.

6.1 CENÁRIO DE TESTE

O cenário de teste é constituído por uma rede local (*LAN*), conectada por cabos *Ethernet CAT 5E*, que suportam uma conexão de 1 *Gigabit*/s (cerca de 100 MBytes/s). Nesta *LAN* estão presentes três computadores que farão parte do teste de desempenho. Seus *hardwares* estão representados na figura 6.1.

O computador 1, por conter o melhor hardware, foi utilizado para manter o servidor da TNET3 ou o RTI utilizado no padrão HLA. No caso do DDS este computador não foi diretamente utilizado. O computador 2 foi utilizado como SVTat A (instância que envia as atualizações) e o computador 3 utilizado como SVTat B (instância que recebe as atualizações). O cenário de teste foi realizado com dois simuladores, pois no contexto do SIS-ASTROS esta será a situação real.

Figura 6.1 – *Hardware* dos computadores para teste de desempenho

	PROCESSADOR	MEMÓRIA
COMPUTADOR 1	Core i5-9400F	12 GB
COMPUTADOR 2	Core I3-9100F	8 GB
COMPUTADOR 3	Core I5-4460	8 GB

Fonte: O autor

Foram realizados dois tipos de testes. O primeiro chamado de teste de *payload*, onde dados de tamanhos arbitrários (*array* de *bytes*) são disparados do computador 2, passam pelo computador 1 e chegam ao computador 3. O objetivo deste teste de stress é observar como cada tecnologia lida com propagação de informações de tamanhos diferentes.

O segundo teste funciona de maneira similar, mas foca em calcular o desempenho utilizando as estruturas do simulador que foram implementadas e perceber como as tecnologias se comportam ao escalar o número de viaturas e baterias durante a simulação. Como as atualizações destas entidades são da ordem de poucos *bytes* é esperado que ocorra uma fragmentação maior entre os pacotes e consequentemente uma menor vazão

de dados. Todos os testes utilizaram o UDP, já que este é o tipo de transporte usualmente utilizado em aplicações de alto desempenho, e também utilizado para disseminação de atributos de objetos.

Estes dois tipos de testes não foram realizados numa instância real do simulador SVTat REOP. Ao invés disso foram criadas aplicações separadas que utilizaram as mesmas estruturas de dados do simulador. No caso do HLA e DDS foi criada uma aplicação na Unity que importa as DLLs e as utiliza da mesma forma que o simulador. Na TNET3 uma aplicação Unity separada do simulador também foi criada. Todas estas aplicações foram desenvolvidos a fim de habilitar uma execução da Unity sem gráficos (*headless mode*). O motivo para isto se dá pela divisão de processamento entre o simulador (tanto parte de processamento gráfico quanto de processamento na CPU) que dificulta que as tecnologias sejam estressadas ao seu limite. O simulador ocupa recursos como cálculos de física e a renderização de objetos, assim como a própria Unity por si só consome parte do processamento na utilização de seu pipeline gráfico. Da forma realizada, as aplicações citadas acima imitam os mesmos comandos gerados pelo simulador no ambiente distribuído, mas sem a necessidade de divisão de processamento, melhorando a qualidade da avaliação de desempenho pretendida.

Ressalta-se que não foi ativada nenhuma politica de qualidade de serviço entre as tecnologias, como a camada de segurança do DDS ou o gerenciamento de tempo do HLA, evitando assim testes de desempenho injustos. Naturalmente isso deve ser levado em conta por eventuais interessados.

Para os cálculos de RTT foi utilizado o tempo local, provido pela maquina local, removendo a necessidade de sincronização entre os computadores durante o teste de desempenho. Assim, ao enviar uma mensagem é calculado o tempo que ela leva desde o envio até o seu retorno, baseado no tempo local.

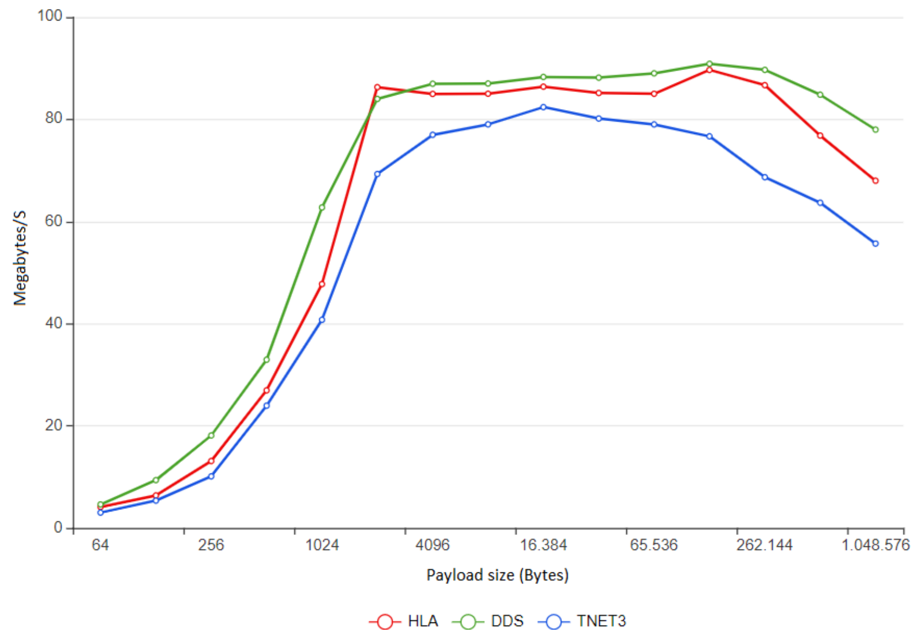
6.2 TESTES DE *PAYLOAD*

Os teste de *payload* foram realizados durante dois minutos. O motivo para considerar um tempo longo se dá pelos algoritmos internos de cada tecnologia que podem dinamicamente ajustar propriedades internas e otimizar o tráfego. Além disso, um tempo longo também gera resultados mais precisos, dado que haverão mais amostras de dados.

A figura 6.2 demonstra os resultados do teste de *payload*, considerando de 64 *bytes* a 1 *MByte* para o tamanho de *payload*. Conforme ilustra a figura, todas tecnologias obtiveram um desempenho aceitável, se considerado o limite da banda provida pela conexão. A TNET3 alcançou o teto de banda em 82 *MBytes*/s, enquanto as tecnologias DDS e HLA obtiveram desempenhos parecidos, mas com a vazão máxima de 88 *MBytes*/s.

Também é possível notar que a TNET3 demora mais para atingir sua vazão máxima

Figura 6.2 – Vazão de dados no teste de *payload*



Fonte: O autor

e também é a primeira a perder sua vazão conforme o tamanho do *payload* aumenta. O motivo para nenhuma tecnologia ter atingido a vazão máxima de 100 *MBytes*/s se dá pelo fato de *bytes* que são gastos pelos pacotes UDP e outras informações de controle utilizadas internamente por cada tecnologia.

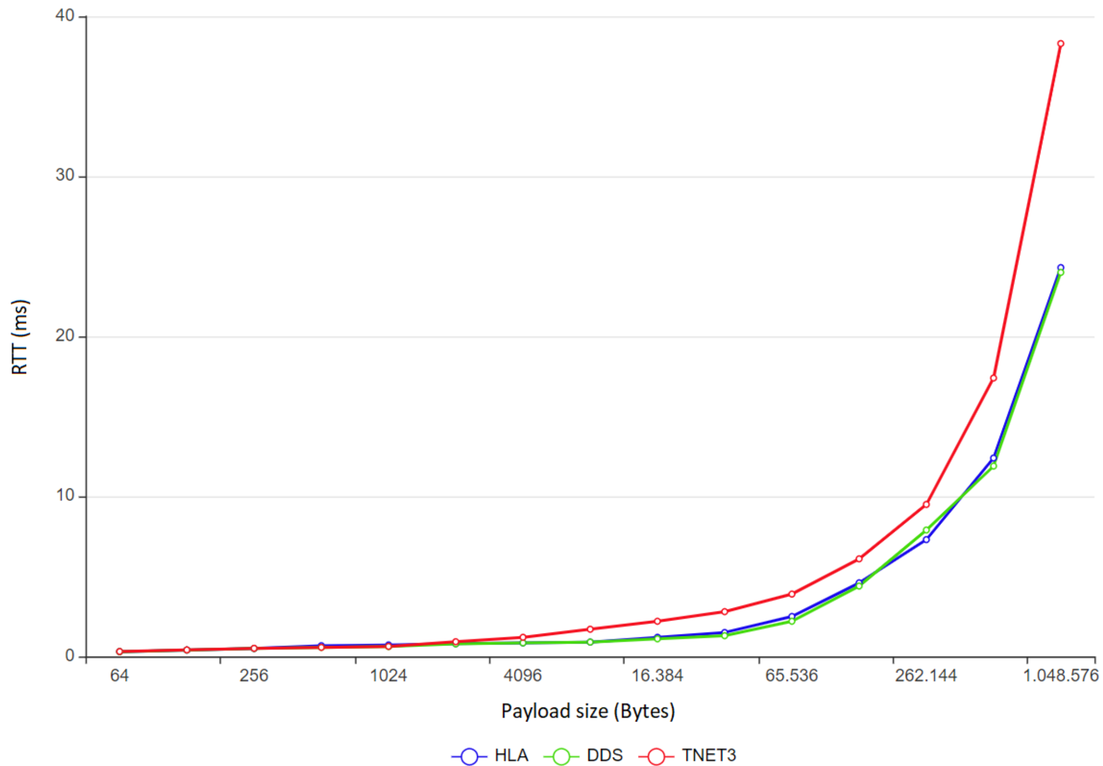
A figura 6.3 demonstra o RTT de cada tecnologia para um dado tamanho de *payload*. A TNET3 demostrou obter um maior RTT em média, principalmente ao aumentar o tamanho do *payload*. Isso ocorreu devido a necessidade do dado passar por um intermediador (seu servidor).

A diferença entre DDS e HLA demostrou-se desprezível, neste caso variando na ordem de *nanosegundos*. Como as aplicações de teste estão conectadas em rede local, assume-se que o gráfico demonstra o *overhead* introduzido pela implementação de cada tecnologia ao enviar dados.

6.3 TESTES COM ESTRUTURAS DO SIMULADOR

Este teste foi realizado com as estruturas da figura 5.1, onde foi escalado o número de veículos ao mesmo tempo que todos os veículos sofreram o máximo de atualização possível ao decorrer do teste. O teste utilizou os seis veículos modelados para a bateria e utilizou o mesmo número de instâncias de veículos por rodada, a fim de facilitar a comparação de vazão de dados. Considerando os seis veículos, um de cada tipo, como uma bateria, eles somam um total de 374 *bytes* por bateria (ou a cada seis veículos).

Figura 6.3 – RTT no teste de *payload*



Fonte: O autor

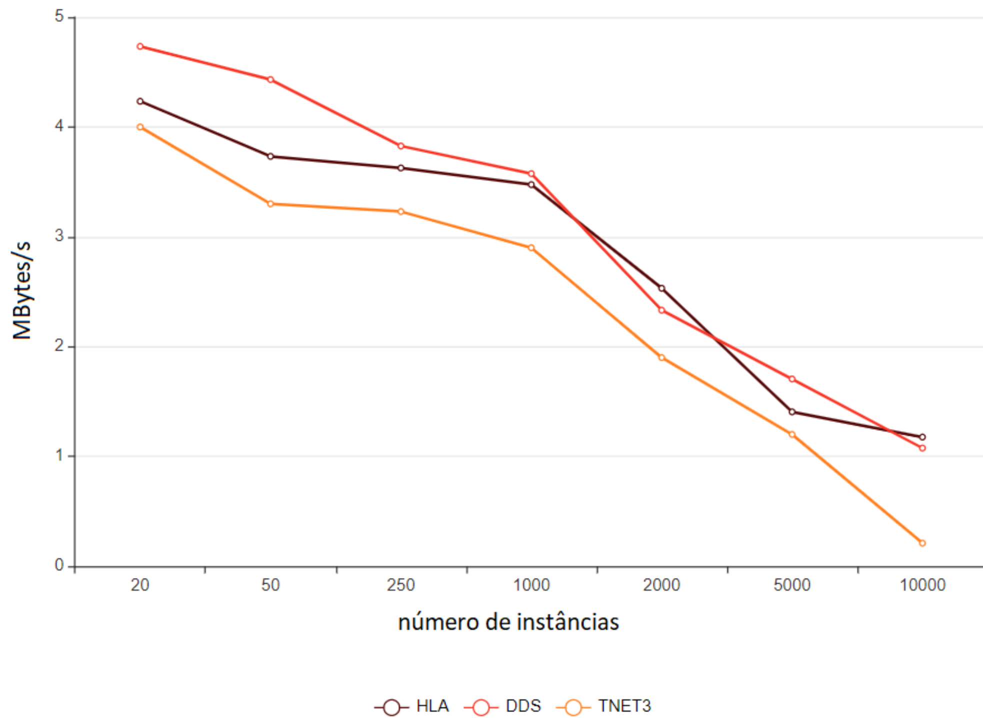
A figura 6.4 demonstra a vazão de dados média para um dado número de instâncias de veículo utilizando as tecnologias. Pode-se notar que a vazão de dados torna-se bem baixa devido a fragmentação de atributos em pacotes únicos. Ao aumentar o número de instâncias também aumenta-se o *overhead* sobre as tecnologias que necessitam procurar as instâncias e atualizá-las.

Mesmo com esta vazão baixa, ao considerar cinco mil instâncias de veículos na figura, pode-se notar que no geral a vazão de dados se mantém em torno de 1.2 *MBytes*/s, o que seria suficiente para manter 3.208 baterias atualizadas por segundo. Como cada bateria tem seis veículos, isto é o equivalente a dizer que poderiam existir 19 mil veículos inteiramente atualizados por segundo.

Podemos ver que para um menor número de instâncias o DDS inicialmente se destaca. A partir de mil instâncias existe uma proximidade de desempenho entre HLA e DDS. Neste caso a TNET3 demostrou obter uma menor vazão de dados desde o início. Mesmo com esta menor vazão ainda pode ser utilizada com um número de instâncias grande, ao considerar em torno de duas mil instâncias, de acordo com gráfico, possui uma taxa de 1.9 *MBytes*/s, que possibilita atualizar inteiramente cerca de 30 mil veículos por segundo.

Ressalta-se que no contexto de um simulador real, atualizações são feitas periodicamente, usualmente quando o atributo muda, ou uma vez por *frame*, o que não põem a tecnologia sob stress. Este teste apenas mostra o limite de cada tecnologia mesmo que

Figura 6.4 – Vazão de dados no teste com dados do simulador

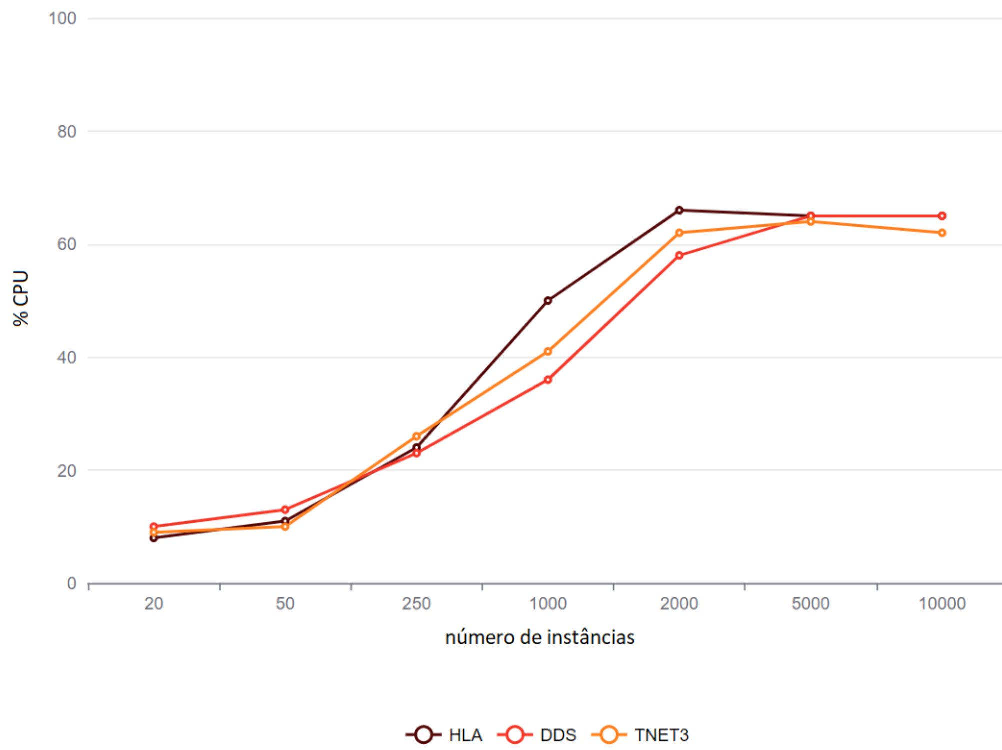


Fonte: O autor

no contexto de simulação real isto não seja utilizado.

Outro fator a se considerar é que no contexto real, atributos normalmente são atualizados em conjunto, o que diminui o *overhead*. A figura 6.5 demonstra o uso de CPU de cada tecnologia para um dado número de instâncias de veículos. Conforme se nota, todas as tecnologias atingem a um determinado limite do uso de processamento, sem uma diferença significativa entre as tecnologias.

Figura 6.5 – Uso de CPU do produtor de dados



Fonte: O autor

7 CONCLUSÃO

Este trabalho comparou as principais funcionalidades das tecnologias HLA, DDS e TNET3. Excluindo a UNET, que está sendo descontinuada e que está limitada a operar com um único servidor, foi demonstrado que o DDS e HLA suportam uma maior variedade de funcionalidades do que a TNET3. Isto se dá por que a TNET3 é desenvolvida exclusivamente para a Unity, onde usualmente não há o requisito de funcionalidades adicionais que são úteis na simulação distribuída. Nas implementações realizadas estas funcionalidades não foram diretamente exploradas, pois não foram necessárias, mas considerando que simuladores podem mudar e obter novos requisitos, uma tecnologia com melhor suporte pode ser considerada.

Das implementações realizadas neste trabalho, conclui-se que a TNET3 demonstra ser a tecnologia mais viável por que abstrai processos de compartilhamento de objetos Unity. A utilização de tecnologias externas exigem passos e solução de problemas adicionais que precisam ser resolvidos pelo programador, principalmente ao utilizar o DDS.

Os teste de desempenho apontaram que, em média, a TNET3 possui a pior performance em termos de aspectos de rede, mas que é capaz de garantir uma fluidez de atualizações na simulação, dado que suporta um número suficiente de instâncias de objeto para simulações no âmbito do SVTat REOP. Esta característica é sustentada mesmo considerando o pior cenário, onde cada atributo é enviado em único pacote. Naturalmente, no contexto de uma implementação real, otimizações deverão ser feitas, fazendo com que as diferenças de desempenho possam ser praticamente imperceptíveis, principalmente por que o uso de processamento do simulador irá limitar as tecnologias de interoperabilidade antes que seja possível identificar as suas distinções de performance.

Finalmente, chega-se a conclusão de que no contexto do simulador SVTat REOP torna-se interessante a utilização de uma tecnologia como a TNET3. Ela resulta num baixo impacto no modelo de programação do SVTat REOP, necessitando um menor esforço e modificação para que o simulador receba a sua funcionalidade adicional requerida. Apesar das outras tecnologias oferecerem possibilidades de maior flexibilidade e funcionalidade, a comunicação necessária entre dois simuladores iguais e programados em Unity não justifica a adoção das tecnologias externas (não nativas da Unity), mesmo estas oferecendo muitas facilidades para o desenvolvimento de simulações distribuídas. Naturalmente, cabe destacar que esta conclusão é válida para um cenário de os simuladores estarem na mesma rede local.

Para trabalhos futuros, pode ser interessante investigar outras tecnologias internas, já que podem possuir funcionalidades adicionais e fornecer níveis de performance superiores.

REFERÊNCIAS BIBLIOGRÁFICAS

ADLINK. **ADLINK OpenSplice**. ADLINK, 2021. Acesso em 18 jan. 2021. Disponível em: <https://www.adlinktech.com/en/vortex-opensplice-data-distribution-service>.

BANKS, J. et al. **Discrete-Event System Simulation**. [S.l.: s.n.], 2009. ISBN 9780136062127.

Bellavista, P. et al. Data distribution service (dds): A performance comparison of opensplice and rti implementations. In: **2013 IEEE Symposium on Computers and Communications (ISCC)**. [S.l.: s.n.], 2013. p. 000377–000383.

COULORIS. **Distributed Systems: Concepts and Design, 4/e**. Pearson Education, 2009. ISBN 9788131718407. Disponível em: <https://books.google.com.br/books?id=O-FyIBTjVOYC>.

DEPARTAMENT OF DEFENSE. **DoD modeling and simulation (MS) glossary**. [S.l.], 1998. 183 p. Acesso em 24 nov. 2020. Disponível em: <https://apps.dtic.mil/dtic/tr/fulltext/u2/a349800.pdf>.

FLETCHER, J. D. Education and training technology in the military. **Science**, v. 323, p. 72 – 75, 2009.

FUJIMOTO, R. M. **Parallel and Distributed Simulation Systems**. [S.l.: s.n.], 2000.

HODSON, D.; HILL, R. The art and science of live, virtual, and constructive simulation for test and analysis. **The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology**, v. 11, p. 77–89, 2014.

IEEE. Ieee standard computer dictionary: A compilation of ieee standard computer glossaries. **IEEE Std 610**, p. 1–217, 1990.

____. Ieee standard for modeling and simulation (m s) high level architecture (hla)– federate interface specification. **IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000)**, p. 1–378, 2010a.

____. Ieee standard for modeling and simulation (m amp;s) high level architecture (hla)– framework and rules. **IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)**, p. 1–38, 2010b.

____. Ieee standard for modeling and simulation (m s) high level architecture (hla)– object model template (omt) specification - redline. **IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000) - Redline**, p. 1–112, 2010c.

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

MARTINEZ, J. et al. A comparison of simulation and operational architectures. In: . [S.l.: s.n.], 2012.

Messaoudi, F.; Simon, G.; Ksentini, A. Dissecting games engines: The case of unity3d. In: **2015 International Workshop on Network and Systems Support for Games (NetGames)**. [S.l.: s.n.], 2015. p. 1–6.

NICOL, J. **Fudamentals of Real-Time Distributed Simulation**. [S.l.: s.n.], 2011. ISBN ISBN 9780986841408.

OMG DDS 1.1. **Data Distribution Service (DDS) Security Especification 1.1**. OMG, 2018. Acesso em 02 dez. 2020. Disponível em: <https://www.omg.org/spec/DDS-SECURITY/1.1/PDF>.

OMG DDS 1.4. **DDS: Data-Distribution Service for Real-Time Systems version 1.4.** OMG, 2015. Acesso em 02 dez. 2020. Disponível em: <https://www.omg.org/spec/DDS/1.4/PDF>.

OMG DDS PORTAL. **OMG DDS PORTAL**. OMG, 2020. Acesso em 02 dez. 2020. Disponível em: <https://www.omg.org/omg-dds-portal/>.

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.

PARDO-CASTELLOTE, G.; FARABAUGH, B.; WARREN, R. An introduction to dds and data-centric communications. 01 2015.

Pitch Technologies. **Pitch Developer Studio**. Pitch Technologies, 2021. Acesso em 18 jan. 2021. Disponível em: <http://pitchtechnologies.com/developer-studio/>.

Pitch Technologies pRTI. **Pitch pRTI**. Pitch Technologies, 2021. Acesso em 18 jan. 2021. Disponível em: <http://pitchtechnologies.com/prti/>.

SÖDERBÄCK, K. **Design, Implementation, and Performance Evaluation of HLA in Unity**. 2017.

SOKOLOWSKI, J. A.; BANKS, C. **Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains**. [S.l.: s.n.], 2010. ISBN ISBN 978-0-470-48674-0.

Synergy Simulation. **MASA SWORD**. Synergy Simulation, 2021. Acesso em 18 jan. 2021. Disponível em: <https://synergy-simulation.com/vendors/masa-group/>.

TANENBAUM. **Distributed Systems: Principles and Paradigms**. USA: Prentice-Hall, Inc., 2007. ISBN 9780132392273.

TASHAREM. **TNET3 Tutorials**. TASHAREM, 2016. Acesso em 02 dez. 2020. Disponível em: <https://www.tasharen.com/forum/index.php?topic=13953.0>.

TOLK, A. **Engineering Principles of Combat Modeling and Distributed Simulation**. Wiley, 2012. ISBN 9781118180303. Disponível em: <https://books.google.com.br/books?id=n7n9HXjikbgC>.

TOLK, A.; MUGUIRA, J. The levels of conceptual interoperability model. In: . [S.l.: s.n.], 2003.

UNITY. **Unity Manual**. Unity, 2020. Acesso em 02 dez. 2020. Disponível em: <https://docs.unity3d.com/Manual/UNet.html>.

WAINER, G.; AL-ZOUBI, K. An introduction to distributed simulation. In: ____. [S.l.: s.n.], 2010. p. 373 – 402. ISBN 9780470590621.

XBOX. **Microsoft Flight Simulator**. XBOX, 2021. Acesso em 18 jan. 2021. Disponível em: <https://www.xbox.com/pt-BR/games/microsoft-flight-simulator>.