# After First Build

Depois da primeira build são gerados 4 executáveis

* Activity Manager
* Digital Twin Adapter
* Serial Apdapter
* Simulator

E na mesma pasta deve se encontrar a biblioteca do MuJoCo, para que seja possível rodar a simulação

* mujoco.dll

Para rodar a stack é indicado que se siga os passos:

Abrir 4 terminais na pasta bin.

## No primeiro:

./activity\_manager.exe

E então entre o ip do broker e a porta a ser usada. O ip pode ser obtido com o comando ipconfig

IPv4 Address. . . . . . . . . . . : 192.168.0.17

A porta pode ser escolhida a critério do usuário, nesse exemplo usaremos a 9091.

## No segundo:

Executar a simulação, passando o arquivo do modelo como parâmentro

.\simulator.exe ..\model\Scorbot-ERVII\Scorbot-ERVII.xml

## No terceiro:

Executar o Digital Twin Adapter

.\digital\_twin\_adapter.exe "192.168.0.17:9091"

## No quarto (opcional):

Executar o Serial Adapter que publica as posições destino para o robô

.\serial\_adapter.exe "192.168.0.17:9091"

## Movimentando o robô pelo Serial Adapter:

No terminal do Serial Adapter entre na opção de escrita digitando w +[Enter] . Então digite a junta a ser movimentada ex.: 2 + [Enter] , por fim digite a posição da junta ex.: 1 + [Enter].

## Movimentando o robô pelo browser:

Abra uma nova guia no Chrome(ou outro browser), aperte F12 para abrir o console, então digite as seguintes linhas

var activityManagerSocket = new WebSocket("ws://192.168.0.17:9091");

activityManagerSocket.send("positionUpdate(&\*\*&)2 0");

A primeira linha abre uma coneção com o Activity Manager na URL 192.168.0.17:9091, a segunda envia um comando de movimentação para a junta ‘2’ ir para a posição ‘0’

O [digital twin(escolher nome)] é uma solução de software de código aberto, desenvolvido em um esforço conjunto, entre alunos e professores, do grupo de estudos focado em engenharia para indústria da Pontifícia Universidade Católica do Rio Grande do Sul.

A solução busca ser capaz de simular a física de um braço robô, bem como simular seu software de controle, planejar sua programação em ambiente virtual e apresentar cópia de movimentos em tempo real.

O [Digital Twin] é composto principalmente por um serviço de simulação de física, que usa o simulador MuJoCo, o qual é open-source e amplamente utilizado em pesquisas na área de robótica; Um controlador, lógica de controle para a movimentação do robô; O Activity Manager, um broker de mensagens pub/sub que utiliza websockets (futuramente UDP multicast?); Position Info Channel, canal para comunicação com o controlador por shared memory; E modelos de publishers de informações de posição em c++, Javascript e Python.

Interface gráfica do usuário, Diagrama

Descrição gerada automaticamente

# Activity Manager

Diagrama

Descrição gerada automaticamenteActivity Manager é um broker que implementa(implementará) uma arquitetura de publisher subscriber. Um processo que se conectar ao mesmo, terá cada mensagem sua encaminhada para todos os demais processos conectados ao Activity Manager.

A padronização das mensagens é bem simples, sempre começando pelo nome do tópico, seguido do separador ”(&\*\*&)” e então a mensagem a ser passada. Caso não esteja usando nossa biblioteca do c++ é necessário montar essa mensagem manualmente.

Uma imagem contendo Texto

Descrição gerada automaticamente

# Digital Twin Adapter

O digital Twin Adapter no momento funciona apenas como uma ponte que recebe mensagens de posição atual do robô real, e encaminha para a simulação.

Ele recebe as informações através de uma coneção com o Activity Manager, representada pela classe IActivityManagerConnector, e as encaminha para o controlador da simulação através da classe IPositionInfoChannel.

# Simulation

É o simulador basíco disponível no repositório do MuJoCo, adaptado para rodar a lógica de controle implementada na biblioteca **sim\_controller.**

O simulador vem acompanhado de um arquivo Scorbot-ERVII.xml que modela o robô ER-VII. Mas é póssível usar qualquer arquivo de modelo aceito pelo MuJoCo: https://mujoco.readthedocs.io/en/latest/modeling.html

# Controller

O controlador recebe posições de destino para cada junta por um canal em shared memory, e exerce força nas juntas da simulação, para que o robô se posicione conforme.

A função SimController::Init é executada logo no começo da simulação, é responsável por publicar informações do robô no canal de shared memory:

this->mPositionInterface->writeJointsQuantity(m->nq);

Então inicializa os parâmetros do algoritmo usado para o controle para cada junta:

this->mWeigths[0].p = 0.4;

Já a função SimController::Step é executada a cada atualização da simulação e é o que define a força a ser aplicada a cada junta (data->ctrl[i]), tomando como parâmetro a posição desejada da junta (this->mPositionInterface->read(i)) e a posição atual da junta (data->qpos[i]):

data->ctrl[i] = this->mWeigths[i].p \* (- this->mPositionInterface->read(i) - data->qpos[i]);

# Position Info Channel

Canal de posição é usado para transmitir a posição destino do robô, do Digital Twin Adapter para o controlador da simulação. A fim de minimizar a latência foi escolhido uma transmissão por shared memory.

A estrutura dessa memória compartilhada é de um bloco de header, que contém um mutex e o número de blocos de dados usados no canal, seguido de n blocos de dados, cada contendo a posição de uma junta. O bloco header é definido na classe ChannelHeaderStructure já os blocos seguintes são instâncias da classe ChannelBlockStructure.

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

Descrição gerada automaticamente

Para se conectar com o Position Info Channel deve ser feita uma chamada para a factory responsável por instanciar um objeto de interface:  
positionInterface = PositionInfoChannelFactory::Create("simulationPosition", ConnectionType::Slave);

Então usar o objeto de interface para

positionInterface->writeJointsQuantity

positionInterface->readJointsQuantity

Escrever 10 no bloco 2

positionInterface->write(2, 10)

Ler o valor do bloco 2

positionInterface->read(2)

# Activity Manager Connector

É uma conexão com o broker do projeto. Ela permite mandar e receber mensagens do broker. Para abrir uma conexão é preciso fazer uma chamada para a factory de conexões passando a URL do Activity Manager e um std::function para receber as mensagens.

auto parserCallback = std::bind(&SerialAdapter::Parse, this, std::placeholders::\_1, std::placeholders::\_2);

this->mActivityManagerConnector = ActivityManagerConnectorFactory::Create(activityManagerURL, parserCallback);

Cada conexão é uma abstração para um session de webscokets, essa session pode ser encontrada em client\_websocket\_session.cc e tem pontos importantes como receber mensagens do broker em

ClientWebsocketSession::OnRead

E mandar mensages em  
ClientWebsocketSession::Send

Todas as operações de leitura e escrita são feitas no mesmo contexto, na mesma thread, que fica localizada em

ActivityManagerConnector::ActivityManagerConnector

auto thread = std::thread([contextPtr](){

contextPtr->run();

});

Para postar uma operação nessa thread, são feitas as chamadas assíncronas de

this->mWebsocketStream.async\_write

this->mWebsocketStream.async\_read

this->mWebsocketStream.async\_handshake

socket->async\_connect

mResolver.async\_resolve

Essas funções botam uma lambda(ou outro objeto invocável) em uma fila e as executam uma a uma. E é por esse motivo, por exemplo, que a função ClientWebsocketSession::OnRead enfileira novamente uma leitura (async\_read), pois sempre após uma leitura, outra deve entrar na fila. Assim é garantido que a fila nunca fique fazia, pois isso causaria o termino da thread de execução de contexto.

# Exemplo de modificações no programa

## Sempre que um cliente se desconectar, mandar uma mensagem para todos dizendo “tchau!”

No arquivo server\_handler.cc , na função ServerHandler::Leave adicionar a linha  
this->Broadcast("tchau!");

## Mudar a resposta quando um cliente se conecta

No arquivo server\_handler.cc , na função ServerHandler:: HandleHttpRequest modificar as linhas

ss << mDebugInt++ << " Unknown HTTP-method";

para

ss << mDebugInt++ << "Mova mensagem";

e

http::response<http::string\_body> res{http::status::bad\_request, req.version()};

para

http::response<http::string\_body> res{http::status::ok, req.version()};

Observação, também é possível mudar o conteúdo da mensagem diretamente em  
res.body() = ss.str();

## Mudar a mensagem que um cliente manda a se conectar ao Activity Manager

Quando criar a coneção com o Activity Manager, passar a mensagem

ActivityManagerConnectorFactory::Create(activityManagerURL, "resources/manifest.json", parserCallback, "MensagemAoConnectar");

## Ler a mensagem que foi mandada pelo cliente ao se conectar no Activity Manager

No arquivo server\_handler.cc , na função ServerHandler:: HandleHttpRequest adicionar as linhas

std::cout << req.body();

obs.:

req = request (mandada pelo cliente) , res = response (mandado para o cliente), tomar cuidado para não confundir

## Sempre que um cliente mandar uma mensagem para o tópico PositionUpdata transformar em uma mensagem para PositionUpdate

No arquivo server\_handler.cc , reescrever a função ServerHandler::HandleWebsocketMessage para:

void ServerHandler::HandleWebsocketMessage(ServerWebsocktSession \* session, std::string const& message) {

std::string newMessage = message;

auto space = message.find("(&\*\*&)");

auto match = message.substr(0, space) == "PositionUpdata";

if (match) {

newMessage.at(13) = 'e';

}

this->Broadcast(newMessage);

}

## Para modificar os parâmetros de controle do braço

No arquivo sim\_controller.cc na função SimController::Init , modificar a linha

this->mWeigths[1].p = 2.5;

para

this->mWeigths[1].p = 1.3;

## Adicionar um offset na força do controlador

No arquivo sim\_controller.h na struct JointControl, adicione um membro K

struct JointControl {

uint16\_t p;

uint16\_t k;

};

E então em SimController::Init adicione o offset para ser usado em cada eixo

this->mJointParameters [0].k = 1;

Por último use o valor em SimController::Step

data->ctrl[i] = this-> mJointParameters [i].p \* (- this->mPositionInterface->read(i) - data->qpos[i]);

data->ctrl[i] += this-> mJointParameters [i].k ;