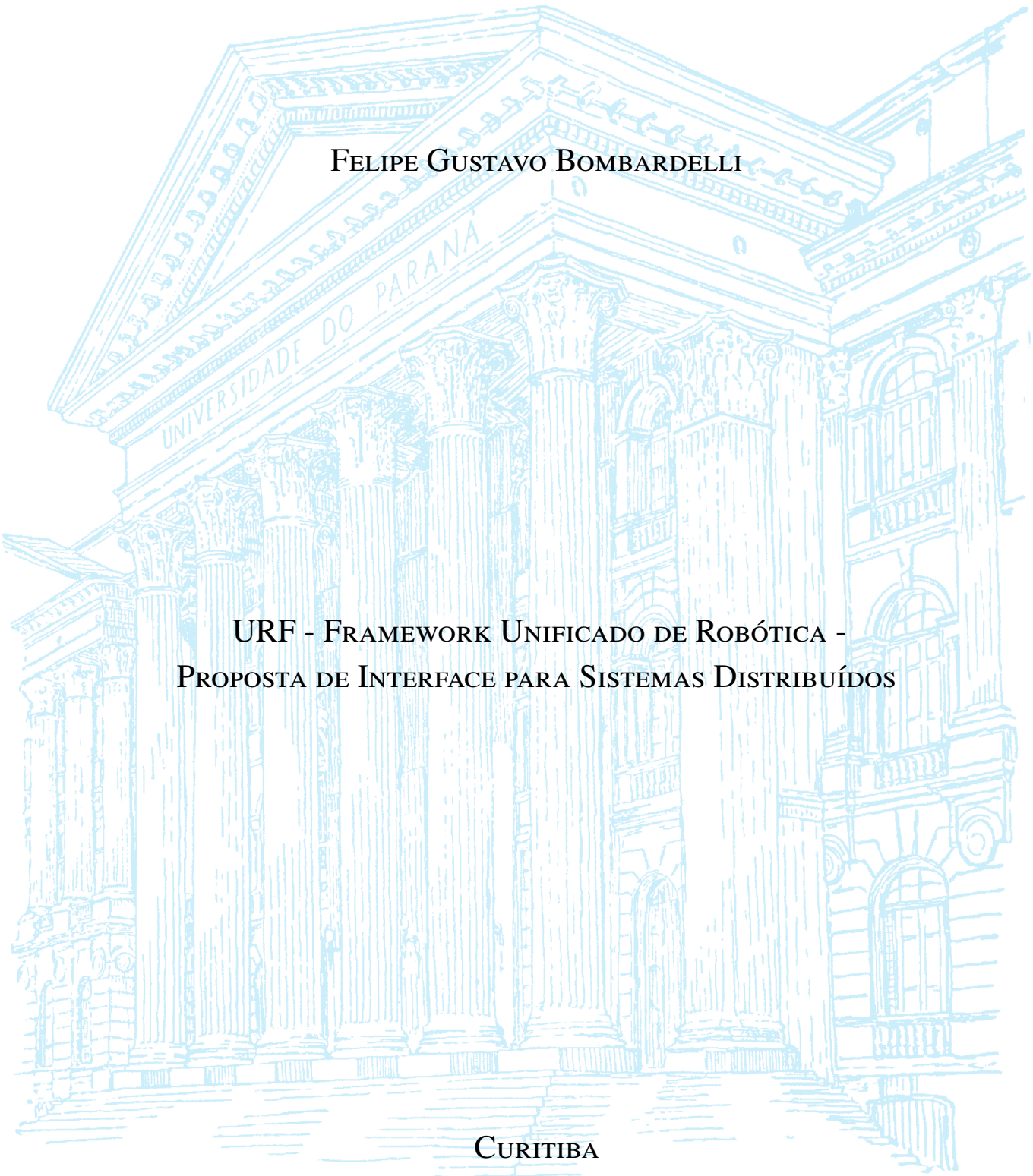
Universidade Federal do Paraná



Felipe Gustavo Bombardelli

URF - Framework Unificado de Robótica - Proposta de Interface para Sistemas Distribuídos

Curitiba
2017

Felipe Gustavo Bombardelli

URF - Framework Unificado de Robótica - Proposta de Interface para Sistemas Distribuídos

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Eduardo Todt.

Curitiba
2017

B695u Bombardelli, Felipe Gustavo
URF – Framework unificado de robótica: proposta de interface para sistemas distribuídos / Felipe Gustavo Bombardelli. – Curitiba, 2017.
76 f. : il. color. ; 30 cm.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2017.

Orientador: Eduardo Todt.

1. Sistemas distribuídos. 2. Frameworks de robótica. 3. Interfaces. I. Universidade Federal do Paraná. II. Todt, Eduardo. III. Título.

CDD: 629.892

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
Setor CIÊNCIAS EXATAS
Programa de Pós-Graduação INFORMÁTICA

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **FELIPE GUSTAVO BOMBARDELLI** intitulada: **Framework Unificado para Robótica - Proposta de Interface para Sistemas Distribuídos.**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua **aprovação** no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

Curitiba, 30 de Agosto de 2017.

EDUARDO TODT
Presidente da Banca Examinadora (UFPR)

FERNANDO SANTOS OSORIO
Avaliador Externo (USP/SC)

ROBERTO PEREIRA
Avaliador Interno (UFPR)

CARLOS ALBERTO MAZIERO
Avaliador Interno (UFPR)

Dedico esta dissertação a minha Família e principalmente aos meus Pais, Gledi e Clovis Bombardelli

Agradecimentos

Agradeço ao meu Orientador, Eduardo Todt, pela sua grande orientação, liberdade de escolha nos caminhos seguidos no trabalho, pelas diversas dicas e seu enorme incentivo e apoio, que mesmo em momentos difíceis, acreditou em mim e no meu trabalho. Também agradeço aos meus amigos do laboratório do VRI, principalmente Leonardo Amaral, Valber Zacarkim, Caroline Cordeiro, Cides Bezerra, Rayson Laroca, pelas diversas dicas sobre a estrutura do texto, apoio material e pelos conselhos nas dificuldades, além dos inúmeros cafés e cervejas discutindo os mais diversos assuntos. Também agradeço a Christina Pahl pela ajuda na qualificação deste trabalho. E por fim, aos meus pais e ao meu irmão que são as pessoas responsáveis pela possibilidade de realização desta dissertação.

Resumo

Em robótica, sistemas distribuídos têm sido utilizados para organizar os projetos de software em módulos, facilitando o desenvolvimento, compartilhamento, manutenção e testes. No entanto, esta modularização traz o ônus da comunicação entre os módulos, que deve ser padronizada e sincronizada, motivando o desenvolvimento de diversos *frameworks* com o objetivo de superar estas dificuldades. Ainda, o código dos módulos tem diversas linhas de *overhead* referentes ao *framework* utilizado, o que limita compartilhamento de seus módulos entre diferentes *frameworks*. Neste trabalho é proposto um *framework* que busca unificar a interface de acesso aos dados, o que permite encapsular diferentes *frameworks* como recursos. Como resultado, o *framework* proposto apresenta uma grande melhora na transparência de acesso aos dados, em relação aos *frameworks* existentes, já que diferentes estruturas podem ser acessadas pela mesma interface, possibilitando a criação de uma árvore de recursos bastante flexível.

Palavras-chave: Sistemas distribuídos, frameworks de robótica, interfaces.

Abstract

On robotics, distributed systems have been used to organize the software projects in modules, facilitating the development, sharing, maintenance and testing. However, this modularization brings an increased complexity of communication between the modules, which must be synchronized and standardized, motivating the development of various efforts to overcome these difficulties. In addition, the module code has several overhead lines over the framework used, which limits sharing of its modules between different frameworks. This work proposes a framework that aims to unify the data access interface, which allows to encapsulate different frameworks in resources. As a result, the proposed framework presents a great improvement in the transparency of access to data, in relation to existing frameworks, since different structures can be accessed by the same interface, making possible the creation of a very flexible resource tree.

Keywords: Distributed systems, robotics frameworks, interfaces.

Sumário

Lista de Figuras

Lista de Siglas

API	Application programming interface. xvii, 61, 63, 68, 72, 74, 76
BSD	Berkeley Software Distribution. 56
CDE	Controle-Data-Execução. xv, xvi, 23, 25–28, 39, 40, 46, 62–65, 81
CORBA	Common Object Request Broker Architecture. xvi, 46, 49, 50
CSV	Comma-Separated Values. xvii, 31, 68, 72, 73, 75, 81
DDS	Data Distribution Service. xvi, 45, 46
FPU	Float Point Unit. 35, 38
IDL	Interface Description Language. 46, 53, 60
IPC	Inter-Process Communication. 61
JSON	JavaScript Object Notation. 30, 31, 67, 71, 75, 81
LBA	Logical Block Addressing. xv, 29
LCM	Light Commmunication and Marshalling. 49, 52, 53, 58–60
LMC	Light Communications and Marshalling. 59
MOM	Message-Oriented Middleware. 45
OMG	Object Management Group. 45, 49
OROCOS	Open RObot COntrol Software. 53, 54, 60
POSIX	Portable Operating System Interface. 22, 81
ROS	Robot Operating System. xvii, 23, 41, 47, 56–60, 62, 64, 66, 67, 75, 76, 81
RPC	Remote Process Call. 43, 56–58, 60
SIMD	Single Instruction Multiple Data. xv, 34

SLAM Simultaneous Localization And Mapping. 53
SSDP Simple Service Discovery Protocol. 34
SSE Streaming SIMD Extensions. 34, 35, 38
SSH Secure Shell. 68

TCP Transmission Control Protocol. 55

UDP User Datagram Protocol. 55
ULA Unidade Lógica Aritmética. xvi, 34, 38, 40
URF Unified Robotic Framework. xvii, 23, 61, 63, 64, 66, 67, 72, 74, 76
URISC Ultimate Reduced Instruction Set Computer. 32

XML eXtensible Markup Language. 64, 67, 71, 81

YAML YAML Ain't Markup Language. 30, 31, 67, 68, 71, 74–76, 81
YARP Yet Another Robot Platform. xvii, 54–56, 60, 62, 66, 67

Capítulo 1

Introdução

Um simples robô móvel utiliza frequentemente um controlador e um computador. O controlador normalmente com baixo poder de processamento, mas com acesso aos sensores e atuadores. Este lê os dados dos sensores e os envia para o computador com maior poder de processamento, o qual executa algoritmos de maior custo computacional. Posteriormente retorna ao controlador mensagens para manipular os atuadores. Robôs mais complexos podem requerer uma quantidade maior destes componentes.

Estes algoritmos executados podem ter uma complexidade teórica bastante grande, pois muitos algoritmos da robótica envolvem conhecimento do desenvolvedor em diferentes áreas, por exemplo em matemática, estatística, otimização de grafos, modelos de representação de mapa, processamento de imagens e outros. Como cada robô pode ter diversas configurações tanto em tipos de atuadores e sensores, como em números de processadores. Então os algoritmos desenvolvidos para um robô, torna-se fortemente acoplamento a configuração do robô.

Portanto, reescrever estes complexos algoritmos para simplesmente adequá-los a um outro modelo de robô não é uma boa política. Portanto alguns trabalhos como [Makarenko et al., 2007] apontam para a utilização de sistemas distribuídos com abordagens como clientes/servidor e sistemas orientados a serviços etc. Deste modo, pode-se organizar o projeto de software do robô em diversos módulos para, além de outras vantagens, desacoplar o algoritmo de seu robô e permitir um melhor compartilhamento e reuso destes algoritmos.

Seguindo esse caminho um único robô utiliza um sistema distribuído para organizar, modularizar e gerenciar todo seu sistema, seja em seu desenvolvimento, quanto em sua execução. Mas pode-se ampliar os horizontes e agora o sistema distribuído ao invés de organizar e gerenciar um único robô, gerência um conjunto de robôs com diferentes arquiteturas, sejam fixos, com rodas ou com pernas; e com diferentes finalidades. De modo, que vários robôs podem trabalham em conjunto sobre uma tarefa. Cenário este que está em evidência atualmente pela recente ideia da indústria 4.0, que otimiza um velho problema industrial apontado por [Marx, 1996, pp.460]: O princípio peculiar da divisão de trabalho causa um isolamento das diferentes fases de produção. Estabelecer e manter a conexão entre as funções isoladas requer transporte ininterrupto do artigo de uma mão para outra e de um processo para outro. Do ponto de vista da grande indústria, isso se apresenta como uma limitação característica, custosa e imanente ao princípio da manufatura.

Portanto um transparente e eficiente comunicação entre estes diferentes robôs e também de computadores servidores tem se tornado um foco de pesquisa pois isto é um dos componentes para a realização da indústria 4.0. Este problema industrial já foi abordado anteriormente, o qual resultou na linha de produção Toyotista e Fordista. Esta última tem uma linha de produção linear e bastante rígida, o qual pode ser flexibilizada pela substituição das tradicionais esteiras pela

robótica móvel. Por exemplo, se na produção de um carro, a fase de colocação da porta esquerda está ocupada, então o robô o redireciona para a colocação da porta direita.

Contudo o bom funcionamento de toda essa complexa linha de produção requer um monitoramento dessas diversas fases e uma excelente integração entre todos esses sensores, atuadores e controladores, o qual é um dos objetivos da indústria 4.0 para [Pfeiffer et al., 2016] e [Meng et al., 2017]. Além de melhorar o fluxo de matéria prima através do monitoramento automático da produção. Neste sentido, para o futuro dos sistemas industriais é ter uma maior conectividade e interatividade. Portanto um acesso mais integrado e eficiente aos dados distribuídos entre as diversas máquinas tornou-se um problema comum para a indústria.

Portanto este trabalho aborda justamente o problema da integração dos dados e principalmente sua transparência de acesso, ou seja, de existir uma interface única para acessar um arquivo no servidor, de acessar uma imagem capturada pelo robô 1 ou de acessar a posição do robô 2. Esta transparência é tão fundamental para o gerenciamento de todo este complexo sistema industrial quanto para um simples sistema de um único robô.

1.1 Problema

No início da computação os códigos eram desenvolvidos de modo monolítico e sem alguma modulação. Com o tempo, surgiu a necessidade de dividir esses códigos em pequenas partes buscando as seguintes vantagens: reutilização de código já escrito; diminuição da quantidade de erros, visto que reutilizar um código já amplamente testado diminui a possibilidade de erros; facilitar a manutenção, visto que o algoritmo desta pequena parte está apenas em um único lugar e não existe diversas cópias delas espalhadas; compartilhamento desses módulos em diferentes projetos; um grupo de programadores poder dividir o algoritmo e trabalhar em paralelo de forma mais independente possível; Em busca dessas vantagens, elaborou-se diversos paradigmas de programação como, por exemplo, estruturada e orientação a objetos.

Posteriormente, com o surgimento de diversos sistemas operacionais, necessitou-se padronizar algumas funções bastante utilizadas pelos processos para que seu código possa ser facilmente portável para outros sistemas operacionais, o que levou o surgimento do Portable Operating System Interface (POSIX). Fazendo um comparativo, o sistema operacional é o sistema integrador e os processos são seus módulos. Posteriormente, com a criação das redes de computadores, houve a ideia de unir vários computadores para funcionar como um único sistema. Logo cada processo espalhado por diferentes computadores é módulo de um sistema distribuído. Contudo o código desses módulos, principalmente nesses sistemas distribuídos estão bastante acoplados ao sistema utilizado, o que dificulta o reuso de seus códigos.

O grande problema abordado por este trabalho é como aumentar a independência dos módulos de seu sistema integrador para, assim, aprimorar seu reuso e compartilhamento. Dentro deste escopo, o trabalho focaliza em buscar uma interface para melhorar a transparência dos dados para futuramente aprimorar a independência dos módulos.

1.2 Justificativa

Uma maior independência dos módulos de seus sistemas integradores permite um melhor reuso e compartilhamento, o que tem uma enorme importância para a robótica, pois a robótica é uma área de conhecimento bastante ampla e por isso dificulta um desenvolvedor conhecer e programar algoritmos para as diferentes áreas da robótica. Considerando o seguinte caso, um grupo especializado em robótica móvel desenvolveu um excelente algoritmo para

determinar a localização de um robô e um outro grupo de visão computacional desenvolveu um algoritmo de detecção de pessoas, facilmente uma única pessoa pode unir esses dois módulos para criar um robô móvel detector de pessoas. Contudo os módulos devem ser o mais independente do sistema integrador utilizado, caso contrário, haverá necessidade estudar o código e gastar muito tempo para modificá-lo e adequá-lo para o sistema desejado.

1.3 Objetivos

O projeto surgiu com a ideia de utilizar as funções fprintf e fscanf para realizar todas as comunicações entre um aplicativo e o framework Robot Operating System(ROS), o que possibilitaria uma independência maior do código do módulo com o ROS e utilizaria uma interface, o qual é amplamente conhecida e utilizada. Contudo pela extensão do assunto, o escopo foi limitado a uma prospecção do *framework* unificado e a proposta de uma interface capaz de abstrair diferentes recursos em busca de uma maior transparência dos dados.

1.4 Estrutura da Dissertação

O segundo capítulo apresenta o modelo Controle Dados e Execução(CDE). Este modelo serve como base metodológica para o restante do trabalho, pois a maioria das figuras elaboradas seguem o padrão baseado no modelo.

O terceiro capítulo, também sobre fundamentação, apresenta algumas arquiteturas de sistemas distribuídos na ótica do modelo CDE. Já o quarto capítulo apresenta alguns *frameworks* desenvolvidos para sistemas distribuídos alguns para uso genéricos e outros específicos para robótica. Além de exemplos de códigos apresentando sua interação com as principais ferramentas do *framework*.

O quinto capítulo apresenta uma prospecção do *framework* URF, qual sua visão geral, como descrever um sistema, como desenvolver um módulo para ele. Contudo para que o *framework* possa ser elaborado, a questão da transparência dos dados é importante, logo o trabalho foca em como obter essa transparência a partir de uma interface.

O sexto capítulo apresenta exemplos de manipulação de dados através dessa interface. No sétimo capítulo, uma discussão dos problemas identificados na elaboração dessa interface. E por fim a conclusão.

Capítulo 2

Fundamentação

A pesquisa sobre um *framework* unificado de robótica requer um estudo sobre divisão e organização de sistemas. Pois um dos principais objetivos deste *framework* visa obter uma maior transparência e independência dessas pequenas partes dos sistemas para aprimorar sua reutilização em outros sistemas.

O modelo de Von Neumann foi descrito no trabalho [Neumann, 1945] e tornou-se a base teórica para arquiteturas dos processadores. Levando em conta que processos são executados pelos processadores e os sistemas distribuídos em última instância são um conjunto de diversos processadores, que se comunicam através de mensagens e passam um aspecto de um único sistema, então este trabalho buscou no modelo de Von Neumann um meio de organizar, estruturar e estudar os diferentes conceitos. O que leva uma abordagem diferente dos tradicionais livros sobre sistemas distribuídos como [Coulouris et al., 2011] e [Tanenbaum, 1996].

Como o modelo de Von Neumann é específico para os processadores, então foram realizadas pequenas alterações neste modelo para deixá-lo genérico o suficiente para ser utilizado tanto para representar processadores como para representar sistemas distribuídos. Este modelo baseado em Von Neumann é chamado neste trabalho de modelo CDE. E este capítulo tem como objetivo explicá-lo para que seja utilizado nos capítulos posteriores.

2.1 Sistemas Computacionais Digitais

O modelo de Von Neumann descrita no trabalho [Neumann, 1945] indica que os sistemas computacionais digitais são compostos por três principais componentes: Central de Controle(CC), Central Aritmética(CA) e Memória. O trabalho [Turing, 1950] utiliza como base o modelo de Von Neumann, porém ele nomeia os três componentes como: Unidade de Controle, Unidade de Execução e Memória. Estes três componentes são interligados, como mostrado na figura 2.1.

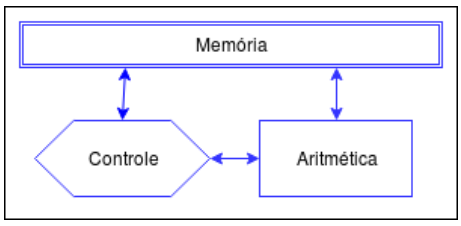


Figura 2.1: Computador digital tem 3 componentes principais: controle, unidade aritmética e memória. Imagem retirada e traduzida do trabalho [Booth e Britten, 1949]

A unidade de controle tem a responsabilidade de receber uma tarefa e distribuí-la para suas unidades de execução. Para [Booth e Britten, 1949], a sequência de execução do controle é definida por: extração da ordem da memória, decodificação da ordem e direcionada para unidade aritmética e memória, finalização da tarefa pela unidade aritmética e localização da próxima ordem e retorna ao primeiro passo de extração da ordem da memória.

Segundo [Turing, 1950], a unidade de execução é a parte que realiza as várias operações individuais envolvidas em um cálculo. Essas operações individuais variam de máquina para máquina. Normalmente, operações razoavelmente longas podem ser feitas, como "Multiplique 3540675445 por 7076345687", mas em algumas máquinas são apenas um simples "Write 0". E por fim, a unidade de memória é um conjunto de bit, bytes ou blocos indexados por números naturais frequentemente iniciado por 0 até o tamanho máximo suportado pelo sistema computacional.

2.2 Modelo Controle Dados e Execução

O modelo CDE baseado de Von Neumann tem como objetivo representar uma unidade de execução que deve realizar alguma tarefa, esta unidade pode ser um processador, um processo, um sistema operacional, um robô etc. A figura 2.2 mostra os principais componentes e demonstra um padrão, que aparece nas figuras do trabalho, onde os hexágonos indicam unidades de controle, retângulos com duas bordas indicam unidades de dados e retângulos com uma borda indicam unidades de execução.

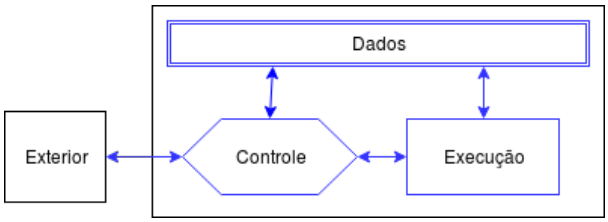


Figura 2.2: Representação do modelo CDE

Importante levar em consideração que uma unidade de execução pode ter unidades de dados, controle e execução, e esta unidade pode ser parte de um sistema maior e desempenhar o papel de controle, dados ou execução neste sistema maior, como exemplificado na figura 2.3, portanto este modelo apresenta uma recursão.

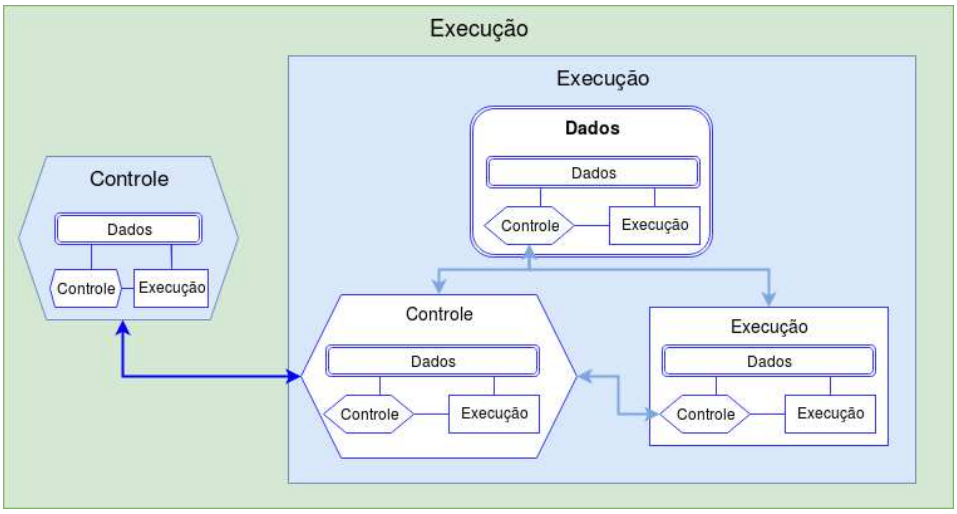


Figura 2.3: Apresentação da existência de recursividade no modelo CDE, onde a unidade

2.2.1 Unidade de Controle

A unidade de controle é responsável de receber ordens de alguma unidade, decodificar a ordem e distribuir para alguma unidade de execução. De forma simples, o código em C a seguir exemplifica e realiza as tarefas de uma unidade de controle, como (1) aguarda a entrada de novos eventos, (2) identificar a unidade que deve executar a evento, (3) receber os parâmetros do evento, (4) chamar a unidade de execução com parâmetros recebidos, retornar o resultado e voltar a ler o próximo evento.

```
1  int main(){
2     int ordem, arg1, arg2;
3     while ( true ){
4        scanf("%d%",&ordem); // (1)
5        if ( ordem == 0 ){   // (2)
6           scanf("%d%",&arg1); // (3)
7           unidade_execucao_A(arg1); // (4)
8        } else if ( ordem == 1 ){ // (2)
9           scanf("%d%",&arg1); // (3)
10          scanf("%d%",&arg2); // (3)
11          unidade_execucao_B(arg1,arg2); // (4)
12       } else {             // (2)
13          int id = fork();
14          if ( id == 0 ){
15             unidade_execucao_C(); // (4)
16             exit(0);
17          }
18       }
19    }
20 }
```

Generalizando, o controle pode receber as ordens de fontes distintas, o formato da mensagem pode variar de acordo com o sistema e finalmente o controle pode permitir a inclusão e exclusão de novas unidade de execução, o que permite em tempo de execução flexibilizar suas funcionalidades. Outra propriedade a ser levado em conta é a execução da unidade de execução, pois pode ser realizada através da mesma *thread*, de uma outra *thread* ou de um novo processo.

2.2.2 Unidade de Execução

A unidade de execução é uma unidade que recebe dados, realiza alguma operação e devolve o resultado. Uma simples função exemplifica uma unidade de execução, como mostrado no código a seguir. A função soma recebe os dados através das variáveis a e b, contém dados internos (no caso a variável res), realiza o procedimento e retorna o resultado.

```
1 int soma(int a, int b){
2    int res = a + b;
3    return res;
4 }
```

Em aplicativos não existem distinções de funções de controle e funções de execução como apresentado, pois é desnecessário. Mas em sistemas distribuído, onde deseja-se separar a fonte dos dados, o formato da mensagem e como é executado, então requer a utilização de um controle para essa flexibilização.

2.2.3 Unidade de Dados

As unidades de dados consistem em um local de armazenamento dos dados e compartilhados entre as unidades de execução e de controle. Em um processador esta unidade de dados é a própria memória, em um processo também é a memória do processador, mas esta pode ser dividida em vários segmentos, como segmento de código, *heap* e pilha dependendo do sistema operacional, e em um objeto, os dados são seus atributos.

Esta seção, para uma melhor didática, exemplifica unidade de dados utilizando códigos em C++. O primeiro código descreve a estrutura Vetor, a qual apresenta uma estrutura de um simples vetor estático com tamanho de 128 inteiros e tem definido a função *get* com um parâmetro, o qual contém o índice de onde deve ser lido. Também a função *set* com dois parâmetros, o índice e o próprio valor a ser escrito.

```
1 struct Vetor {
2    int dados[128];
3
4    int get(int indice){return this->dados[indice];}
5    void set(int indice, int val){this->dados[indice] = val;}
6 };
```

Apesar da linguagem C++ ser compilada, esta estrutura é bastante semelhante a descrição de um dispositivo de memória com 512 bytes com um barramento de 32 bits, dispositivo o qual aparece no modelo de Von Neuman. Como o modelo CDE serve para outros contextos, então a unidade de dados deve ser mais genérica e poder representar qualquer estrutura de dados, seja ela uma árvore de busca, um vetor, uma matriz, uma tabela *hash* etc. Logo uma unidade de dados permite indexar seus itens por *string* e não apenas por números naturais, seu conteúdo pode ser acessado de forma aleatória ou sequencial, seu tamanho pode ser estático ou dinâmico. Essas variações são mais aprofundadas nas subseções a seguir.

Tamanho do Índice

Uma unidade de dados pode ser um vetor de bytes como os pentes de memória do computador, e seu índice varia de 0 até o tamanho máximo e o tamanho de cada item é de exatamente 1 byte. Contudo em discos rígidos o tamanho do item é de 512 bytes, por exemplo e este valor pode variar de acordo com o fabricante. Mas os discos mantêm o índice composto por um número natural de 0 a quantidade de setores, como demonstrado na figura 2.4.

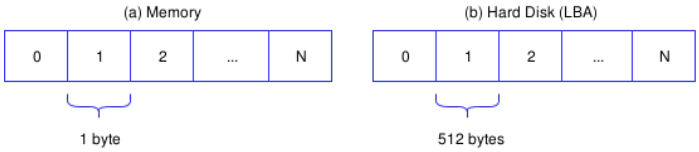


Figura 2.4: (a) Índice com um número de 0 a N e cada elemento é um byte. (b) Um número de 0 a N e cada elemento tem 512 bytes, como acontece nos atuais discos rígidos através da LBA.

Os índices podem ser formatos por dois ou três números, como por exemplo, nas matrizes e estruturas cúbicas, como mostrado a seguir na figura 2.5.

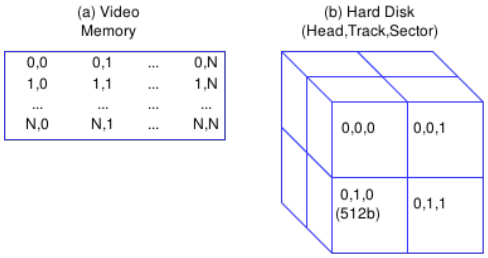


Figura 2.5: (a) Matriz, com dois índices numéricos, como a memória de vídeo ou imagens. (b) Pode ser um cubo, como a endereçamento dos antigos discos rígidos com 3 índices numéricos representando cabeça, trilha e setor.

Também o índice de uma estrutura pode ser composto por N números naturais, como acontece na estrutura *QuadTree*, onde uma matriz é dividida em 4 regiões recursivamente, como demonstrado na figura 2.6.

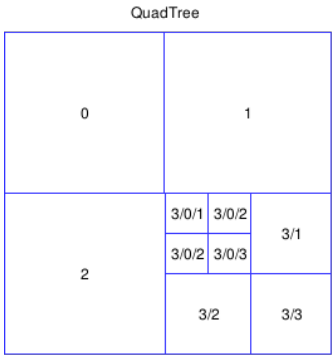


Figura 2.6: Representação de uma Quadtree e o tamanho do seu índice pode variar de 1 a N, dependendo da quantidade de níveis.

E por final, os índices não obrigatoriamente são números naturais, eles podem ser números em pontos flutuantes ou *strings*, este último caso ocorre nos diretórios dos sistemas de arquivos.

Índice Contínuo ou Não-Continuo

Os vetores, que tem um índice numérico de 0 a N, contêm um índice contínuo, pois todos os índices possíveis, entre 0 a N, estão presentes. Isto possibilita uma rápida verificação

se o índice X existe, o que simplesmente é uma verificação se X é maior que 0 e menor que N. Portanto é sempre uma operação de custo O(1). Contudo em índices não contínuos isto não é verdade e, portanto, deve-se fazer uma busca pelos índices para poder verificar se X existe. Essa busca tem um custo dependente da quantidade de elementos e dependente da estrutura de busca utilizada, se for um simples vetor não ordenado, então O(N), se for uma árvore então O(logN).

Itens com ou sem Formatação

Uma unidade de dados pode ser simplesmente um vetor de bytes sem dar alguma indicação se é um inteiro, *string* ou outro. Ou ela mesma já conhece sua estrutura e permite informar como os dados estão estruturados, como o caso de uma estrutura proveniente de um texto em uma linguagem de marcação, como JavaScript Object Notation (JSON).

Localização Interna ou Externa ao Processo

Uma unidade de dados pode ser um segmento de memória na *heap* do processo, portanto pode ser facilmente referenciado e acessado diretamente pelo processo. Contudo a unidade de dados pode referenciar dados localizados em um outro dispositivo, como o caso dos arquivos e logo precisa-se chamar um intermediário, neste caso a estrutura em C(FILE*) junto com as chamadas ao sistema para acessar os dados do arquivo. As memórias externas podem ter um *buffer*, que junto com a política de leitura/escrita preguiçosa pode diminuir a quantidade de transferências de dados.

Tamanho do Item

A unidade de dados pode ser composta de itens de tamanho fixo, como os tradicionais pentes de memória com itens de tamanho de um byte, ou discos rígidos com itens fixo de 512 bytes. Ou seus itens podem ter um tamanho dinâmico, como o caso de um diretório, onde cada arquivo tem tamanhos diferentes.

Coleções e Itens

Uma unidade de dados pode ser apenas um único item ou uma coleção de itens. Contudo existe uma dificuldade teórica em delimitar essas fronteiras, pois um item é uma coleção de bytes e um byte é uma coleção de bits. Contudo cada linguagem de computação define algumas variáveis primitivas, que podem ser facilmente manipulados. Como por exemplo, a linguagem PHP, que não tem tipagem, define valores escalares como inteiro, ponto flutuante, *string* ou booleano. Já os tipos *array*, *object* e *resource* não são escalares, portanto são coleção de itens. Enquanto para YAML Ain't Markup Language (YAML), uma linguagem para intercâmbio de dados baseado em unicode, ela representa suas construções através de três primitivas: valor escalar, sequência e mapeamento.

Escalar é qualquer objeto com estrutura espressada como uma série de caracteres unicode;

Sequencia é uma coleção ordenada de elementos;

Mapeamento é uma coleção sem ordem o qual associa uma chave única a um valor;

O sistema de arquivos é um conjunto de diretórios, que contém arquivos e também outros diretórios. Contudo os arquivos não trazem consigo seu formato, portanto são vistos pelo sistema operacional como apenas um vetor de bytes. Por fim a linguagem C++ utiliza o conceito de contêineres, o qual é um objeto que guarda uma coleção de outros objetos, que são seus elementos. A linguagem utiliza este conceito para abstrair as estruturas de dados como vetor, pilha, árvores, tabelas hash etc, e classifica-os em 4 grandes grupos dependendo do modo de indexação: contêineres de sequência, contêineres associativos e contêineres sem ordem.

Formato dos Registros

Um arquivo ou um *pipe* pode ser visto como um conjunto de registros. Este registro pode ter um formato fixo, ou seja, todos os registros têm os mesmos campos, como ocorre no formato CSV mostrado na figura 2.7. Ou os registros podem ter o formato variável, ou seja, cada registro pode ser diferente um do outro. Portanto ter campos diferentes. Um exemplo disto é o formato JSON, YAML e afins que permitem que cada conjunto de elemento tenha seus próprios campos, contudo cada valor leva consigo o seu índice, caso o índice não for contínuo.

(a) Mensagem Fixa e Formatada

	X (int)	Y (int)	Name (String)
0	0	10	Cozinha
1	20	11	Sala
2	23	15	Sala
	...	...	...
N	120	50	Corredor

(b) Mensagem Variável

```
[
  { x: 0, y: 10, name: "Cozinha", ang: 10.300},
  { y: 11, x: 20, status: "na parede", ang: 16.0},
  { name: "Sala", x: 23, y: 15},
  ....
  {x:120,y:50,name:"Corredor"}
]
```

Figura 2.7: (a) Formato fixo, onde o arquivo tem N registro e todos eles contêm os campos X, Y e Nome, que são respectivamente dois inteiros e uma String. (b) Pode ser vetor de registros variáveis, onde cada registro tem seus próprios campos.

Os modelos de formato fixo são normalmente menores, pois todos os registros compartilham os mesmos índices, e mais rápidos, principalmente se á estiverem no formato binário, logo não precisam ser convertidos. Dentro de um registro, o acesso aos seus dados pode ser feito rapidamente através do índice numérico de 0 a quantidade de campos ou pode-se buscar pelo nome do campo. Contudo os modelos fixos têm o problema de permitir apenas um tipo de mensagem.

Os modelos variáveis permitem um fluxo de dados com diferentes tipos de registros. Contudo perde-se desempenho, pois o acesso a variável é realizado através do nome de seu campo, o que leva a uma busca em seus índices e depende de qual estrutura de busca foi utilizado.

2.3 Relação entre Unidade de Controle e Memória

Nesta seção discute-se a existência de uma máquina, a qual contenha a mais simples unidade de controle e uma ou mais unidades de dados, como mostrado na figura 2.8. As unidades de dados contêm apenas as operações de leitura e escrita e a unidade de controle, por ser simples, tem o menor conjunto de instruções possível. Este conjunto contém apenas instruções de controle de fluxo, como por exemplo as instruções do assembly intel x86: jmp(pula), jne(pula se não igual), je(pula se igual). Lembrando que o controle também deve poder comparar dois elementos da memória, pois seu resultado é utilizado pelas instruções jne e je.

A arquitetura com o menor conjunto de instruções possível é a Ultimate Reduced Instruction Set Computer (URISC), proposto por [Mavaddat e Parhami, 1988] com a finalidade de facilitar o ensino de arquitetura aos alunos, e contém uma única instrução. Esta instrução é definida como: subtraia B por A e pula para C se negativo. Deste modo é possível fazer uma máquina Turing completa apenas com esta instrução. Lembrando que a subtração é uma operação de comparação também, tanto que o código em assembly x86 (sub eax, 10; jne .label) funciona de modo similar à comparação (cmp eax, 10; jne .label). Portanto [Mavaddat e Parhami, 1988] apenas uniu a operação de comparação com operação de pulo na mesma instrução.

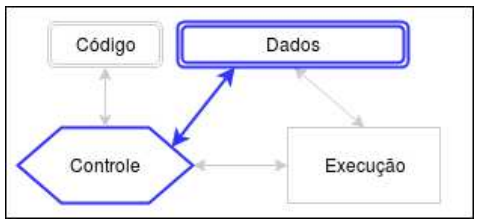


Figura 2.8: Existência de um controle com conjunto mínimo de instruções e a possibilidade de executar apenas operações de escrita e leitura na memória. Portanto sem a existência de uma unidade de execução.

Outra máquina, que tem em sua definição apenas controle de fluxo, comparação de elementos, leitura e escrita de memória é a máquina de Turing, a qual é definida pelo trabalho [Turing, 1936].

Uma maquina de Turing é uma quíntupla $M = (Q, \sum, \Gamma, \delta, q_0)$ onde Q é um conjunto de estados, $\sum$ é um conjunto finito de símbolos, que constituem o alfabeto da fita, $\sum$ é um subconjunto de Γ, δ é uma função parcial de $Qx\Gamma$ para $Qx\Gamma x\{L, R\}$ chamado de função de transição, e q_0 é um único estado pertencente a Q chamado de estado inicial.[Sudkamp, 2005, pp.259-260]

A fita da maquina de Turing tem uma borda a esquerda e estende indefinidamente para a direita. Cada posição da fira pode ser numerada por números naturais iniciando por 0 e cada posição contém um elemento do alfabeto. [Sudkamp, 2005, pp.259-260]

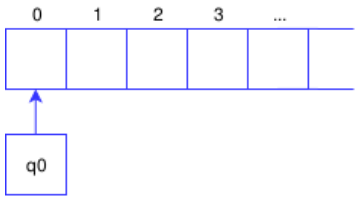


Figura 2.9: Figura retirada do livro [Sudkamp, 2005, pp.260]

A função de transição consiste em três ações: mudança de estado, leitura do simbolo sobre a posição atual do cursor e movimentação do cursor. A direção do movimento é especificada pela ultimo componente da função. O simbolo L (left) indica movimentação a esquerda e R (right) indica movimentação a direita.[Sudkamp, 2005, pp.259-260]

Considerando a seguinte configuração:

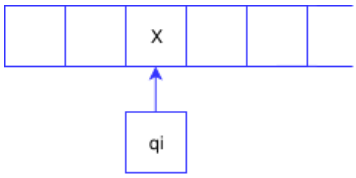


Figura 2.10: Atual estado do exemplo de uma máquina de Turing[Sudkamp, 2005, pp.260]

e uma transição S(qi,x) = [qj, y, L] combinados produzem a nova configuração

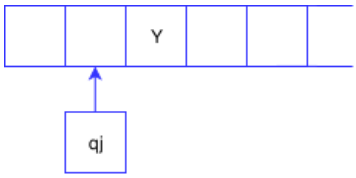


Figura 2.11: Próximo estado do exemplo da máquina de Turing, após executar a função de transição S(qi,x) = [qj, y, L][Sudkamp, 2005, pp.260]

A transição muda o estado de qi para qj, substituindo a simbolo x para y na fita, e move o cursor uma posição para esquerda. A computação pára quando a maquina encontra um estado e um simbolo para o qual não existe nenhuma transição definida. Portanto existe uma termino anormal e possível erro. [Sudkamp, 2005, pp.260]

Existem variações na máquina de Turing, se considerar a utilização de memória com acesso aleatório ao invés de sequencial ou utilizar várias fitas(memórias) ou a memória contiver varias trilhas.

Portanto a beleza da máquina de Turing está em sua simplicidade, pois sem definir qualquer operação matemática, como soma ou subtração, sua máquina é capaz de realizar qualquer operação realizada por um computador moderno.

2.4 Relação entre Unidades de Controle e de Execução

Esta seção explica sobre a relação entre a unidade de controle e a de execução, e focado nos componentes em azul mostrado na figura 2.12. Esta relação pode haver diferenças na quantidade de controladores, na quantidade de unidade de execução, na velocidade do canal de comunicação entre eles e variação no conhecimento do controle sobre a unidade de execução.

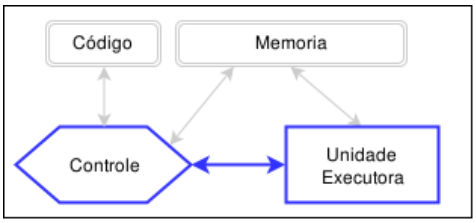


Figura 2.12: Em azul destacado o foco da sessão, a relação entre o controle a unidade de execução. O código normalmente está dentro do controle e não existe uma memória conhecida pelos dois. Figura baseada da figura original do trabalho [Booth e Britten, 1949]

Este nível de conhecimento entre as unidades pode ser grande, portanto fortemente acoplados, como por exemplo, em um processador, onde o controlador e a unidade são imutáveis,

foram desenvolvidos no mesmo projeto, e para cada controlador tem uma unidade de execução própria. Portanto o controlador não precisa alocar e liberar as unidades utilizadas. Além que o controlador e o executor compartilham o mesmo *clock*, desta maneira o controlador sabe, que após tantos *clocks*, a unidade de execução já terminou sua tarefa. Portanto a sincronização é realizada por este *clock* compartilhado.

Também pode-se imaginar também um aplicativo desenvolvido em OpenMPI para multiplicação de matrizes distribuída em um *cluster*. Como o código deste aplicativo foi desenvolvido no mesmo projeto, e o mesmo código compilado é utilizado por todos os *hosts*, então o controlador (computador controlador) conhece todas as funcionalidades dos executores (computadores executores) e seu protocolo de comunicação.

Por final, pode-se imaginar em uma situação onde o controle apenas tem uma comunicação de fluxo de bytes com a unidade sem conhecimento de suas funcionalidades e formatos das mensagens. Neste caso, deve-se utilizar um serviço de descobrimento como o Simple Service Discovery Protocol (SSDP).

Outra propriedade é a conexão entre o controle e unidade. Esta pode ser permanente, como ocorre nos processadores, ou temporária como ocorre na internet quando um computador requisita uma página de um servidor.

Nas próximas subseções, analisa-se como é a relação se houver muitos controles e uma única unidade de execução e se houver um único controle e várias unidade de execução.

2.4.1 Um Controlador e muitos Executores Idênticos

O controlador pode ter diversas unidades de execução idênticas, ou seja, disponibilizam o mesmo conjunto de funções, utilizando o conceito de classe da orientação à objetos, as unidades idênticas são instâncias da mesma classe como demonstrado na figura 2.13. As múltiplas unidades de mesma classe permitem que a tarefa possa ser dividida e cada unidade execute uma parte da tarefa paralelamente.

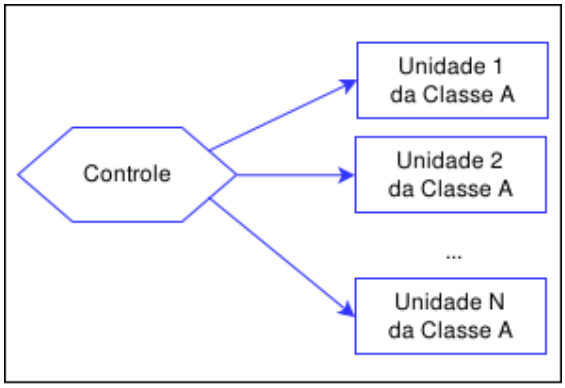


Figura 2.13: Um controlador distribui a mesma tarefa para diversas unidades de execução. Isto é possível pois todas as unidades têm a mesma função. Modelo utilizado normalmente em processadores SIMD, onde o controlador envia a mesma instrução para diversas unidades, contudo sobre dados diferentes.

Este modelo bastante comum em processadores SIMD, como por exemplo na unidade de execução Streaming SIMD Extensions (SSE) ou placas de vídeo, os quais tem N ULA e realizam N somas paralelamente.

Outra vantagem de utilizar diversas unidades com mesma funcionalidade é o aumento da disponibilidade. Por exemplo, um sistema distribuído pode colocar diversos computadores oferecendo o mesmo serviço para caso algum desconecte ou desligue, tenha outro computador para substituí-lo e manter o serviço disponível.

2.4.2 Um Controlador e muitos Executores Distintos

O controlador pode ter diversas unidades de execução de diferentes classes, portanto com diferentes funcionalidades. Desta maneira o controle tem à disposição um conjunto maior de funcionalidades. Um exemplo disto, foi adição da unidade Float Point Unit (FPU) aos processadores, a qual fornece instruções para manipulação de números de ponto flutuante. O mesmo aconteceu com a adição do SSE. Portanto a cada unidade distinta inserida houve um incremento no conjunto de instruções do processador.

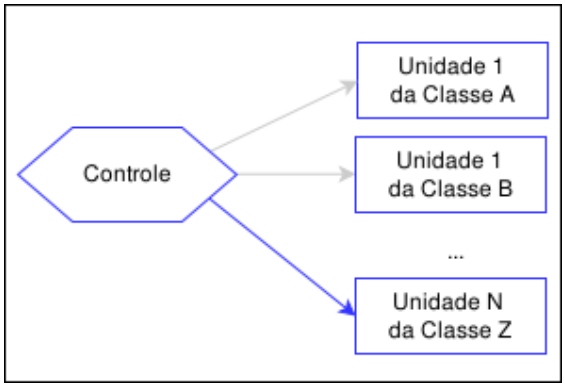


Figura 2.14: Um controlador tem seu conjunto de instruções aumentado devido as unidades de diferentes classes. Contudo somente pode enviar a tarefa para as unidades que contém a função necessária, não permitindo uma paralelização.

Embora a adição aumente as funcionalidades do controlador, não necessariamente permite a execução paralela de uma tarefa, como o caso mostrado na sessão 2.4.1.

2.4.3 Muitos Controladores e um Executor

Em diversos casos, uma unidade de execução ou de memória podem ser requisitados por diversos controles paralelamente, como por exemplo vários processos acessarem o mesmo disco ou a mesma região de memória, então existe o típico problema de concorrência.

Uma das soluções para o problema de concorrência é a duplicação da unidade. Como por exemplo em casos de pequenos dados na memória, que podem ser duplicados para cada *thread* do processo, logo evita a leitura concorrente. Contudo em outros casos, este dado pode ser muito extenso, o que implica um custo muito alto em sua duplicação. Ou até mesmo a unidade pode ser um dispositivo físico, como um disco rígido o que inviabiliza sua duplicação.

Outra solução possível é serializar a unidade e desta maneira compartilhar um mecanismo de gerenciamento de acesso à unidade. Este mecanismo pode ser realizado pelos próprios controles, como demonstrado na figura 2.15, ou existir uma unidade específica para gerenciar a unidade, como demonstrado na figura 2.16.

No primeiro modelo, todos os controladores podem acessar diretamente a unidade de execução, contudo eles devem compartilhar o mesmo mecanismo de controle de acesso, que

pode ser um segmento de memória compartilhado significando um semáforo. Outro detalhe que controlador deve conhecer o protocolo de comunicação com o executor. Então usualmente o controle de acesso e o protocolo de comunicação são implementados em forma de bibliotecas para ser compartilhados entre vários processos.

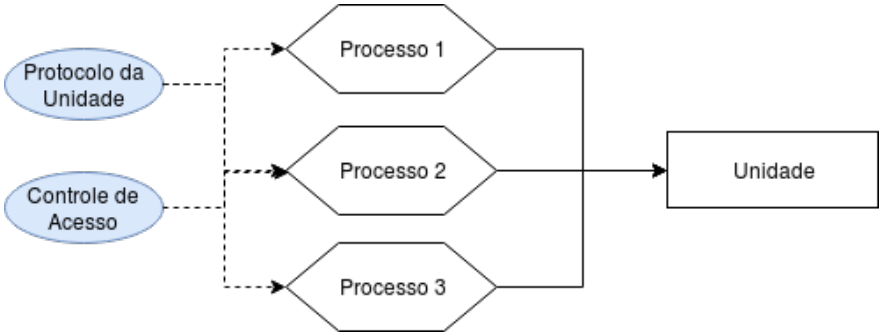


Figura 2.15: Na figura existem três controles, que acessam a mesma unidade, como por exemplo, três processos acessarem diretamente um disco rígido.

No segundo modelo, insere-se uma nova unidade de execução intermediaria, a qual tem como responsabilidade gerenciar os acessos e utilizar o protocolo adequado com a unidade. Assim todos os processos acessam diretamente esta unidade intermediaria, e somente ela acessa o dispositivo.

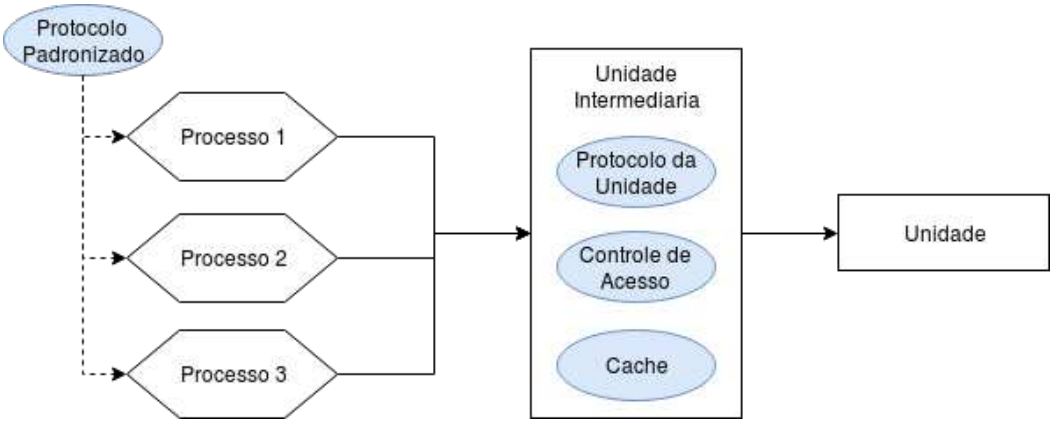


Figura 2.16: Neste outro caso, existe uma unidade intermediaria e somente ela que acessa a unidade. A unidade intermediaria gerencia o controle de acesso e já tem o protocolo necessário de comunicação com a unidade, além de permitir adição de um cache para diminuir a quantidade de comunicação. Contudo a unidade intermediaria deve ter uma interface conhecida pelos processos.

O intermediário resolve o problema do controle de acesso e do protocolo de comunicação com a unidade primaria. Além que é possível criar um gerenciamento de protocolos de modo que o intermediário conhece uma gama de protocolos e seleciona o protocolo mais adequado para a unidade, como ocorre frequentemente nos sistemas operacionais com o gerenciamento de *drivers*. Embora solucione o protocolo da unidade intermediaria com a unidade principal, os controladores devem conhecer o protocolo de comunicação com o intermediário. Portanto deve-se padronizar uma interface.

2.5 Relação entre Unidade de Controle e o Código

Esta sessão apresenta em como o código é estruturado dependendo do controle e como o código também pode influenciar no comportamento do controle.

O código, ou melhor, o algoritmo é um conjunto finito de instruções, que definem como um determinado problema pode ser resolvido. O desenvolvedor pode ter diversos modos de como estruturar um algoritmo dependendo de qual paradigma ele usará, que pode ser através da não estruturada, estruturada, orientação a objetos, dirigida a eventos e outros.

O primeiro detalhe é como o controle acessa o código. Se o código está em memória e pode ser acessado aleatoriamente, então o controle pode ter instruções de controle de fluxo e pular para outras instruções formando laços de repetição ou estrutura condicionais. Este é o ambiente tradicional dos aplicativos.

Agora se for considerado, que este controle está esperando alguma requisição vindo de um outro processo, então ele executa apenas a requisição corrente e não permite que este controle possa voltar cinco requisições atrás e realizar um laço de repetição. Neste caso utiliza-se frequentemente uma programação dirigida a eventos e o controle fica sempre aguardando alguma mudança no código(na porta de entrada das requisições).

2.5.1 Programação Estruturada

A programação estruturada restringe os mecanismos de fluxo de controle e os classifica em três tipos: sequencial, condicional e de repetição.

A programação estruturada permite a divisão do aplicativo em diversas funções. Cada função tem um código e seu controle pode ser realizado pelo mesmo controlador da função chamadora e deste modo cria-se uma pilha de controles, como mostrada na figura 2.17. Todos os dados desses diversos controladores empilhados são guardados na mesma pilha. Contudo, caso a função seja executada através de uma *thread*, então é criado um novo controle com uma pilha própria.

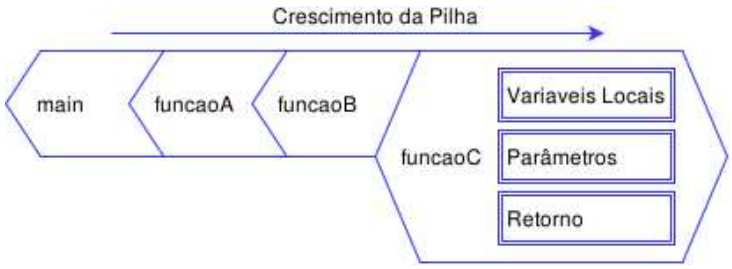


Figura 2.17: A execução de um programa, o qual inicia pela função main, o qual chama a funçãoA, que por sua vez chama a funçãoB e por final chama a funçãoC. Os 4 controles compartilham o mesmo processador e como cada controlador requer um espaço indefinido de memória, então utiliza-se frequentemente uma estrutura de pilha na memória principal.

Cada controlador nesta pilha tem 3 unidades de dados, as variáveis locais os parâmetros e o endereço de retorno. Quando o controle termina sua execução, então as unidades de dados são liberadas. Nos processadores atuais estes dados são alocados na pilha do processo, mas pode ser que futuramente elas sejam alocadas dentro do processador.

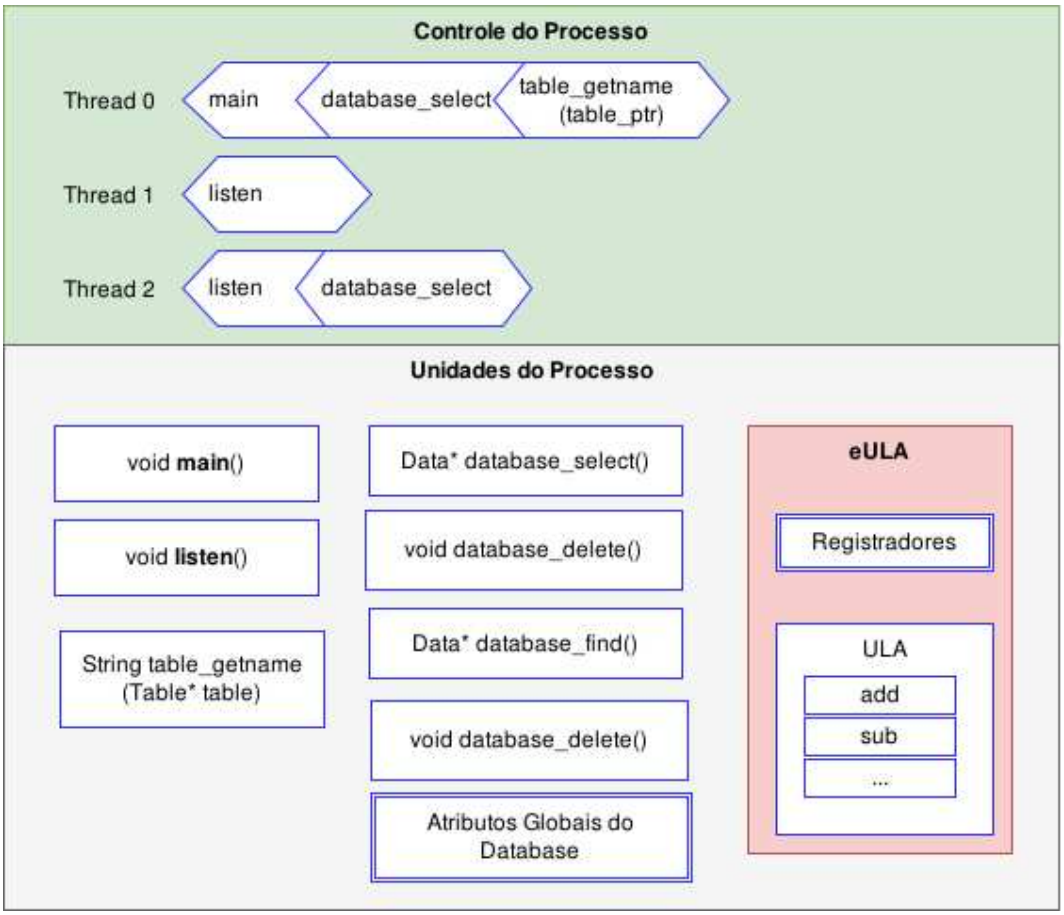


Figura 2.18: Um aplicativo pode ter diversas *threads*, como mostrado neste exemplo na parte de controle do processo, e cada *thread* tem a sua disposição as unidades do processo, que consistem das funções e variáveis globais do sistema, além de utilizar a ULA do processador.

De modo geral, o aplicativo tem a estrutura semelhante ao exemplo apresentado na figura 2.18. O aplicativo pode ter diversas *threads* e cada uma delas tem o mesmo conjunto de funções, de variáveis globais e de dispositivos fornecidos pelo processador (ULA, FPU, SSE) a disposição. O sistema operacional possibilita que cada *thread* utilize os dispositivos do processador sem concorrência.

A programação estruturada inicialmente foi pensada para ter uma única *thread*. Posteriormente criou-se mecanismo para criação de novas *threads* sem alteração nas diversas linguagens através de chamadas de função. Na linguagem C/C++ existe bibliotecas para esta finalidade, que teve sua padronização realizada pelo padrão POSIX. Contudo a utilização dessas chamadas implica que o código serial do aplicativo seja diferente do código paralelo, o que dificulta sua manutenção.

Sobre este problema, o grupo de desenvolvedores da biblioteca OpenMP observou essa deficiência na linguagem fortran e sua solução foi alterar a linguagem para permitir linhas de códigos que indiquem como deve ser feito o paralelismo de um determinado bloco de código. Em C/C++, as alterações realizadas pelo OpenMP foram feitas através de diretivas ao compilador, permitindo que as linhas de código OpenMP não prejudiquem compiladores sem o módulo OpenMP. Já em outras linguagens como Scala, essas diretivas já são nativas à linguagem.

2.5.2 Programação Orientada a Objetos

A programação orientada a objeto para [Rentsch, 1982] tem uma difícil definição formal e sua possível definição gira em torno do conceito de objeto, contudo a parte principal do objeto não é o que ele é, e sim como ele é visto pelo exterior. Então a visão bem aceita deste paradigma é dada por [Booch, 1986], que organiza qualquer sistema através de uma coleção de objetos.

A orientação a objeto permitiu agrupar um conjunto de funções que operam sobre uma mesma estrutura de dados formando assim um objeto, ou na nomenclatura do modelo CDE, uma unidade de execução. Este objeto ou unidade de execução pode conter uma unidade de dados(atributos), diversas unidades de execução (funções, instâncias ou referências de outros objetos) e pode ter um controlador. Este controlador é inexistente nos objetos sobre aplicativos com uma única *thread*, pois todos os códigos são executados por esta mesma *thread*, contudo em sistemas distribuídos onde os objetos estão em diversos *hosts*, existe a necessidade de um controle, o qual é uma unidade especifica para realizar a leitura dos eventos e a distribuição das tarefas.

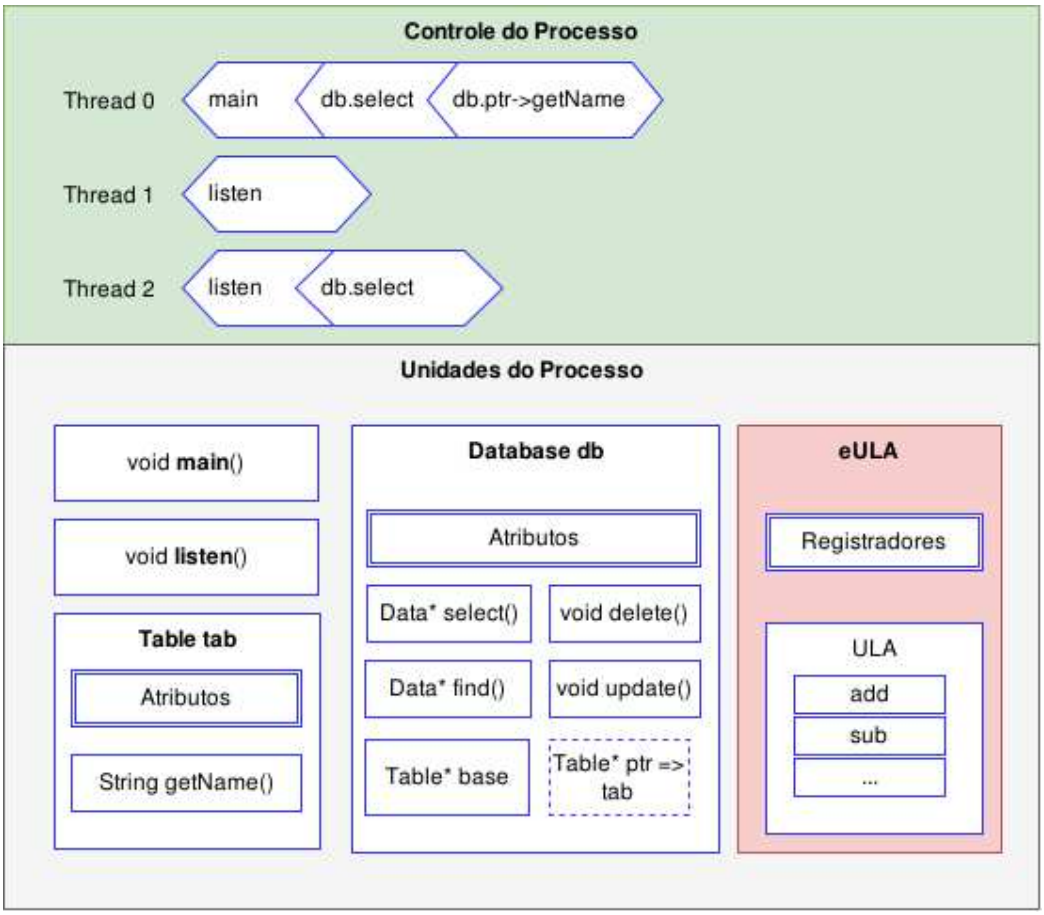


Figura 2.19: O aplicativo elaborado em orientação a objeto é semelhante ao estruturado, mas difere-se por permitir que uma unidade contenha um conjunto de funções e uma memória compartilhada entre este conjunto de funções. Deste modo as funções da base de dados e variáveis globais da figura 2.18 foram unidas no mesmo objeto.

Na figura 2.20 apresentada lado a lado cada conceito e pode-se dizer que existe uma grande semelhança entre eles. Então este trabalho enxerga essas diferentes estruturas como um

subdomínio do modelo CDE, contudo cada uma tem algumas peculiaridades como as funções não tem um controle, pois o compilador/linkador encarregam-se de juntar a chamada objeto.metodo() à unidade correspondente. Enquanto que o sistema de herança da orientação a objetos é realizado pela união das unidades, ou seja, união de seus dados e seus métodos, e caso ocorra algum conflito de índice, então requer um polimorfismo.

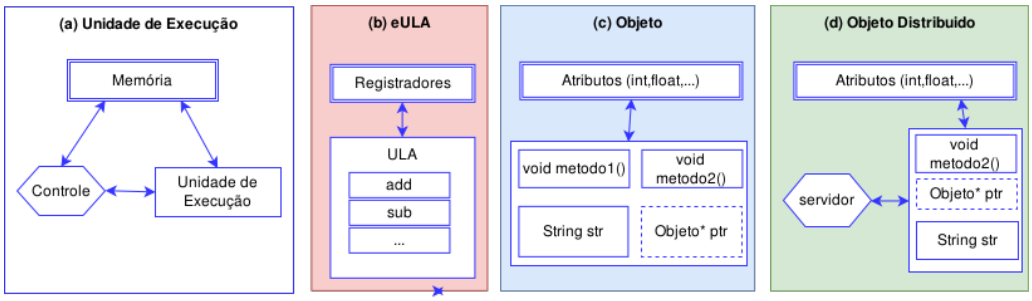


Figura 2.20: (a) Modelo CDE; (b) Representação da ULA com seu banco de registradores (c) Um objeto em um aplicativo.

2.5.3 Programacão dirigida a Eventos

A programação dirigida a eventos é bastante utilizada em aplicações com interfaces gráficas, onde o usuário pode gerar diversos eventos como clicar em um botão, mover o mouse sobre uma área, digitar um texto. O aplicativo dirigido a eventos, como por exemplo um aplicativo gráfico, tem uma *thread* que fica lendo novos eventos, seja através do sistema acordá-lo quando chegar um novo evento(interrupção) ou se ele mesmo a cada tempo verificar se chegou um novo evento (*polling*).

Esta unidade, que lê os novos eventos, é denominada na filosofia deste trabalho como controlador, como demonstrado na figura 2.21, pois ele lê as instruções(eventos) e envia as requisições às unidades de execução para realizar o processo vinculado ao evento. Desta maneira, o controlador, quando recebe um evento, busca em uma tabela, qual função ou qual objeto manipulará aquele determinado evento. Portanto nesta etapa a programação dirigida a eventos é descritiva e deve relacionar os tipos de eventos com as determinadas funções ou métodos. Já na parte de programação da unidade de execução (funções e objetos) a programação volta a ser imperativa.

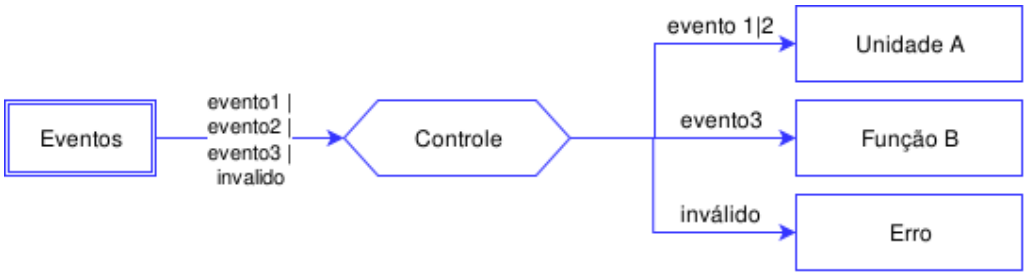


Figura 2.21: Estrutura geral de um sistema de evento, onde o controle aguarda por um evento e dado sua relação entre evento e unidade, o controle envia o evento para ser processado para a unidade de execução correspondente.

O controlador pode executar a tarefa da unidade de execução na mesma thread, o que implica que os eventos são executados serialmente e se um entrar em loop infinito, o controlador

deixa de ler os novos eventos. Outra política é o controlador abrir uma nova *thread* para executar a tarefa, o que deixa livre o controlador somente com a responsabilidade de ler os novos eventos e poder compartilhar a memória do processo com a *thread*. Também é possível o controlador abrir um novo processo para a tarefa, o que permite o aplicativo não ser finalizar em caso ocorra uma falha de segmentação na tarefa, contudo não existe mais o compartilhamento da memória do processo.

Em bibliotecas para interface gráficas como Qt e GTK, existem funções que permitem relacionar um evento a uma função *callback*. Este mesmo mecanismo ocorre em *frameworks* de sistemas distribuídos como ROS, que também existem funções no *framework* para relacionar um tópico a uma função *callback*.

Capítulo 3

Fundamentação - Modelo de Controle-Dados-Execução em Sistemas Distribuídos

O livro [Tanenbaum, 1996] define o conceito de sistema distribuído como vagamente caracterizado como uma coleção de computadores independentes que aparecem aos usuários como um único computador. Já o livro [Coulouris et al., 2011] define conceito de sistema distribuído como componentes de hardware ou software localizados em computadores interligados em rede, que se comunicam e coordenam suas ações apenas enviando mensagens entre si. O livro [Coulouris et al., 2011] também indica consequências importantes devido esta definição, como concorrência, inexistência de relógio global e falhas independentes.

3.1 Middleware Procedural

A ideia do Remote Process Call (RPC), segundo [Birrell e Nelson, 1984], é simples e baseia-se no fato, que as chamadas de rotinas são bem conhecidas e sua transferência de dados e de controle são bem entendidas dentro de um contexto de um único processo. Portanto a proposta de [Birrell e Nelson, 1984] é justamente estender este mesmo mecanismo para prover uma transferência de dados e de controle através da rede.

Esta extensão da chamada local para a remota utiliza dois conceitos chaves, o primeiro chamado de stub, o qual é responsável por interpretar e empacotar os parâmetros da rotina e posteriormente desempacotar o resultado recebido da rotina, e o segundo conceito RPC-runtime é responsável por enviar para a rede uma requisição, aguardar e receber o resultado. A figura 3.1 demonstra exatamente este mecanismo, onde a máquina chamadora realiza uma chamada local para um stub empacotar. Após o empacotamento o stub envia o pacote para o RPC-Runtime o qual transmite-o e se mantém em espera até receber o resultado. Enquanto isso, o RPC-Runtine da máquina servidora recebe o pacote, o stub do servidor desempacota e a função desejada é executada.

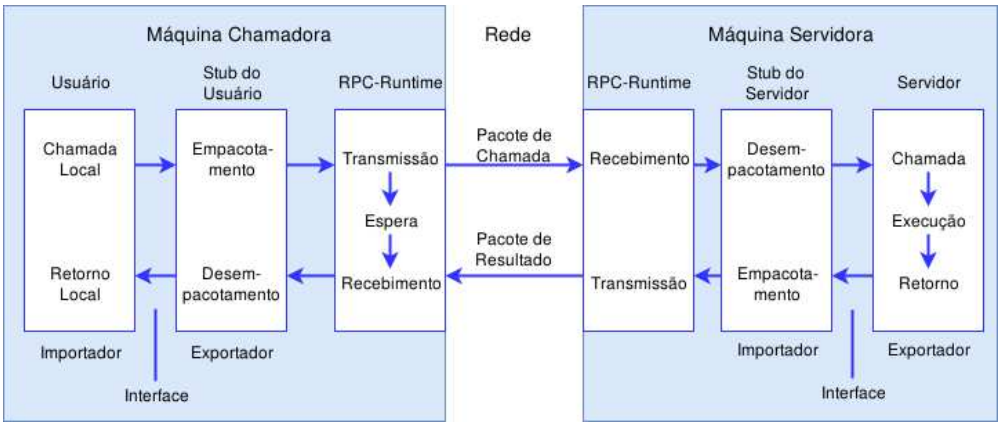


Figura 3.1: Figura traduzida do trabalho [Birrell e Nelson, 1984]

Após a execução, o stub recebe o resultado, realiza o empacotamento, passa para o RPC-Runtime e envia para maquina chamadora. A máquina chamadora recebe o pacote, depois é direcionado para o stub, desempacotado e o valor do resultado vai para a variável desejada pelo desenvolver na memória do aplicativo.

Nas figuras 3.2 e 3.3 é feito um paralelo entre uma chamada local e uma chamada remota. A diferença que existe uma comunicação não tão rápida e, portanto, os parâmetros são enviados junto com a requisição e o processo precisa tem um controle especifico, que fique escutando a porta de conexão no processo servidor. Por fim, precisa-se haver um compartilhamento entre a interface.

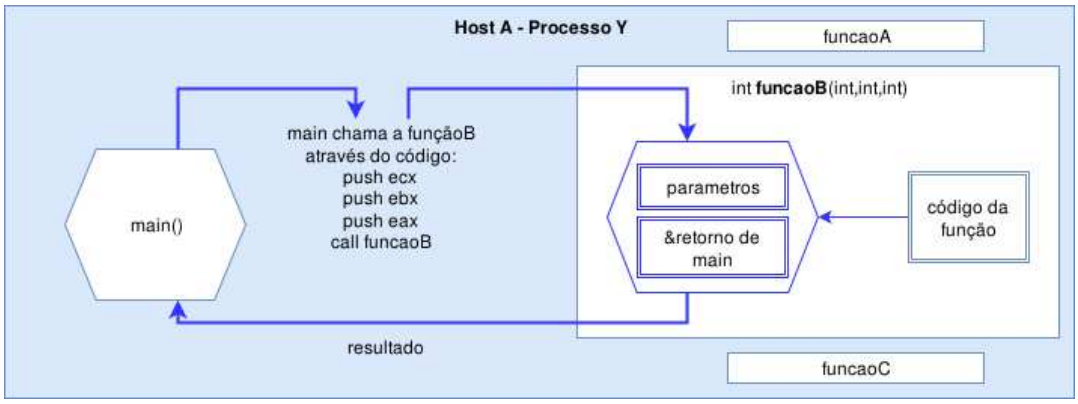


Figura 3.2: Processo Y está executando a rotina main e a rotina chama a funçãoB. A chamada é realizada pelo código *assembly* x86 no meio do diagrama, o qual vai carregar 3 números na pilha e executar a instrução call. Esta instrução guarda o endereço do retorno na pilha e passa a executar o código da funçãoB, portanto o código da função main deixa de ser executado. Após o termino da funçãoB, as variáveis locais, parâmetros e endereço de retorno são eliminados da pilha. E main recebe o resultado por algum meio dependendo do compilador utilizado.

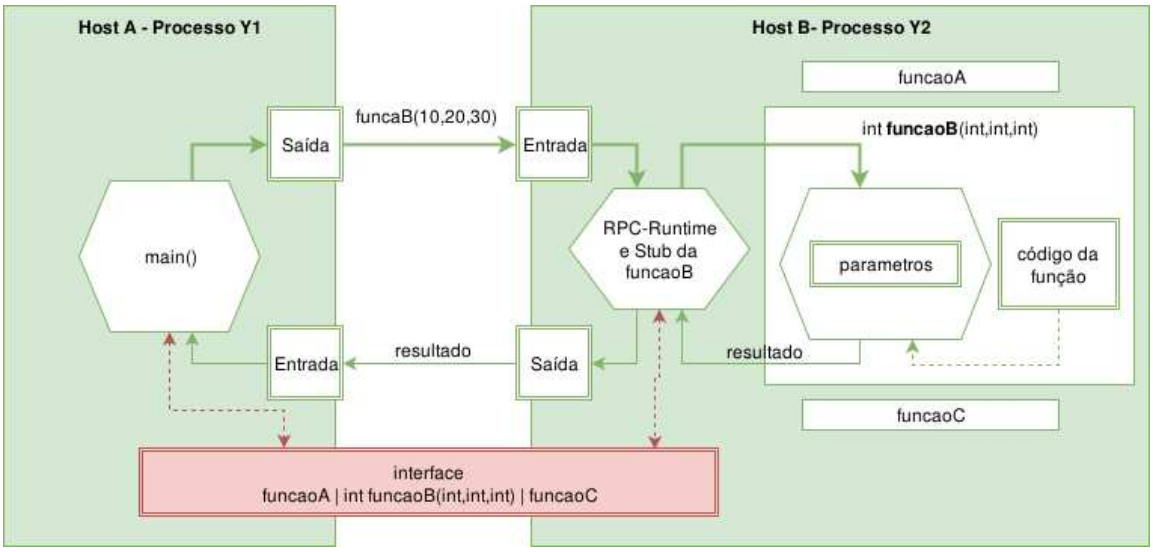


Figura 3.3: O processo Y1 no Host A chama a funcaoB no Host B. A rotina main de Y1 escreve o pacotefuncaoB(10,20,30) na porta de saída. Em seguida o RPC-Runtime, que está monitorando a entrada, recebe o pacote, desempacota, e envia a requisição para funcaoB. A função devolve o resultado para o Runtime e este devolve para o Host A. A interface deve ser compartilhada pelos dois *hosts* para haver compatibilidade

.

3.2 Middleware Orientado a Mensagem

Alguns sistemas distribuídos são mais focados sobre o fluxo de dados e principalmente sobre independência da origem dos dados. Por exemplo, um módulo que realiza um processo sobre uma imagem não precisa conhecer, se os dados vêm de uma câmera, de um vídeo ou pela rede. Este apenas precisa receber os dados, realizar um processamento e envia o resultado para algum outro módulo, que não necessariamente foi o módulo que lhe enviou os dados, como ocorre geralmente no procedural.

Um exemplo simples deste conceito pode ser vista em um *pipe* sobre o sistema Linux, como por exemplo no comando a seguir.

```
1 ls -lh | tr -s ' ' | cut -d ' ' -f 2;
```

O comando ls lista todos itens de um diretório e envia estes dados ao comando tr, o qual tem a função de remover os caracteres em branco repetidos e depois repassa para o comando cut. Cada um deles podem ser executados isoladamente e os três desconhecem quais operações serão realizadas sobre o resultado de seus dados. Aqui existe uma diferença importante com o modelo procedural, o qual o cliente conhece exatamente a função realizada pelo servidor. Contudo o modelo de pipeline do Linux é simplório, pois não permite vários processos ler sobre um único *pipe* e não permite o processo abrir diversas saídas, assim como não permite a adição e remoção de processos em tempo de execução do *pipe*.

Então a seguir é apresentado o DDS, o qual é um dos mais populares padrões Message-Oriented Middleware (MOM). Este é definido pelo Object Management Group (OMG) no documento [Omg, 2006] e seu design é mostrado na figura 3.4 e a seguir a descrição dos seus 6 principais conceitos:

Domínio Aplicações no DDS envia e recebe dados dentro do mesmo domínio. Um domínio é um espaço virtual que conecta determinadas publicantes e assinantes. Somente aplicativos com o mesmo domínio podem comunica-se e esta restrição ajuda a isolar e otimizar a comunicação [Omg, 2006].

Publicador O publicante é responsável por publicar os dados e disseminá-lo para todos os assinantes relevantes no domínio [Omg, 2006].

Assinante O assinante é responsável por receber os dados publicados e torná-los disponível para o aplicativo. Ele pode receber diversos tipos de mensagem e portanto o aplicativo deve usar um leitor de dados anexado ao assinante [Omg, 2006].

Tópico Um tópico associa um nome único no sistema a um tipo de dados. Logo muitos publicantes podem escrever dados sobre um tópico e muitos assinantes podem ler a partir deles. Usualmente os tópicos são tipados possibilitando uma verificação de compatibilidade em tempo de compilação e mais eficiência, já que o aplicativo conhece exatamente o tipo de dados. CORBA e DDS utilizam a linguagem Interface Description Language (IDL) para descrever o formato destes dados [Omg, 2006].

Leitor de Dados recebe os dados de um assinante e entrega os dados desempacotados para a aplicação e cada leitor está vinculado a um particular tópico.

Escritor de Dados é usado pelo código do aplicativo para publicar os valores para o sistema. Cada escritor de dados está vinculado a um tópico específico. O escritor de dados é responsável por empacotar os dados e passá-lo para a transmissão [Omg, 2006].

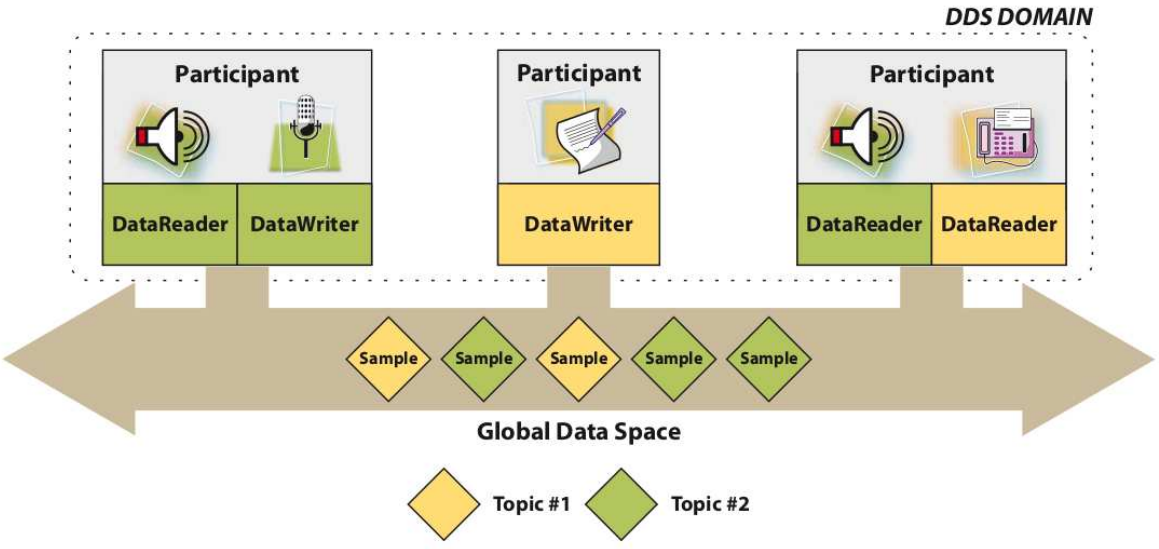


Figura 3.4: Modelo de comunicação para o DDS, onde os participantes fazem parte do mesmo domínio e compartilham dois tipos de mensagens representados pela cor amarela e verde. Figura retirada do trabalho [Tijero, 2012].

Os middleware orientados a mensagem sobre a ótica do modelo de CDE podem ser representados por uma cadeia de unidades de execução, contudo tem uma importante diferença na relação entre o modelo procedural(cliente-servidor), onde o cliente envia o comando a ser

executado e seus respectivos parâmetros ao servidor. Já no modelo orientado a mensagem, a unidade de execução envia apenas os dados, portanto ele desconhece qual função será realizado sobre estes dados, o que permite ao controle mudar seu serviço sem qualquer alteração na unidade de execução publicadora. Outra diferença, que o resultado não necessariamente retorna para o cliente, ele pode ir para um outro servidor.

Pode-se construir um modelo assinante-publicante utilizando como base o modelo cliente-servidor. Contudo a construção de um modelo cliente-servidor através do modelo assinante-publicante é mais difícil, pois normalmente o cliente recebe uma resposta do servidor, caso que nem sempre ocorre no modelo assinante. Por isso alguns *frameworks* como ROS disponibilizam os dois modelos, os tópicos (modelo assinante-publicante) e os serviços (modelo servidor-cliente).

Capítulo 4

Trabalhos Relacionados

Este capítulo apresenta implementações de *middlewares* para sistemas distribuídos e está divido em duas seções, a primeira exemplifica alguns *middlewares* de uso geral, os quais podem implementar o modelo procedural, modelo de transação, orientado a objeto, de componente, ou orientado a mensagens. E a segunda sessão apresenta alguns exemplos de *frameworks* específicos para robótica.

4.1 Middlewares para Uso Geral

Existem diversos middlewares para uso geral como Java RMI, DCOM, CORBA, ICE, os quais são implementações de middlewares de objetos distribuídos, e como Light Commmunication and Marshalling (LCM) e ZeroMQ, que são implementações de middlewares orientados a mensagem. Nesta sessão é apresentada uma implementação do padrão CORBA e uma implementação do modelo orientado a mensagens pelo middleware LCM. Na sessão a seguir, existem os *frameworks* específicos para robótica, os quais tem poucas diferenças com LCM, já que são orientados a mensagem.

4.1.1 Common Object Request Broker Architecture (CORBA)

CORBA é um padrão criado pela OMG para estabelecer e simplificar a troca de dados entre sistemas distribuídos. Por ser um padrão aberto existem diversas implementações para a linguagem C++, como OmniORB, Orbacus, The ACE ORB (TAO). Esta sessão apresenta um código implementado em OmniORB.

O modelo CORBA implementado pelo OmniORB está representado na figura 4.1. Onde o sistema distribuído pode ter vários processos e cada processo pode ter vários objetos instanciados, os quais fornecem uma gama de métodos para outros processos no sistema.

Cada processo que disponibiliza um objeto ao sistema distribuído deve inicializar um ORB, o qual desempenha o papel de controle, e portanto mantém-se lendo a porta e para cada requisição, desempacota a requisição, busca o objeto, executa o método desejado, empacota o resultado e envia para destinatário.

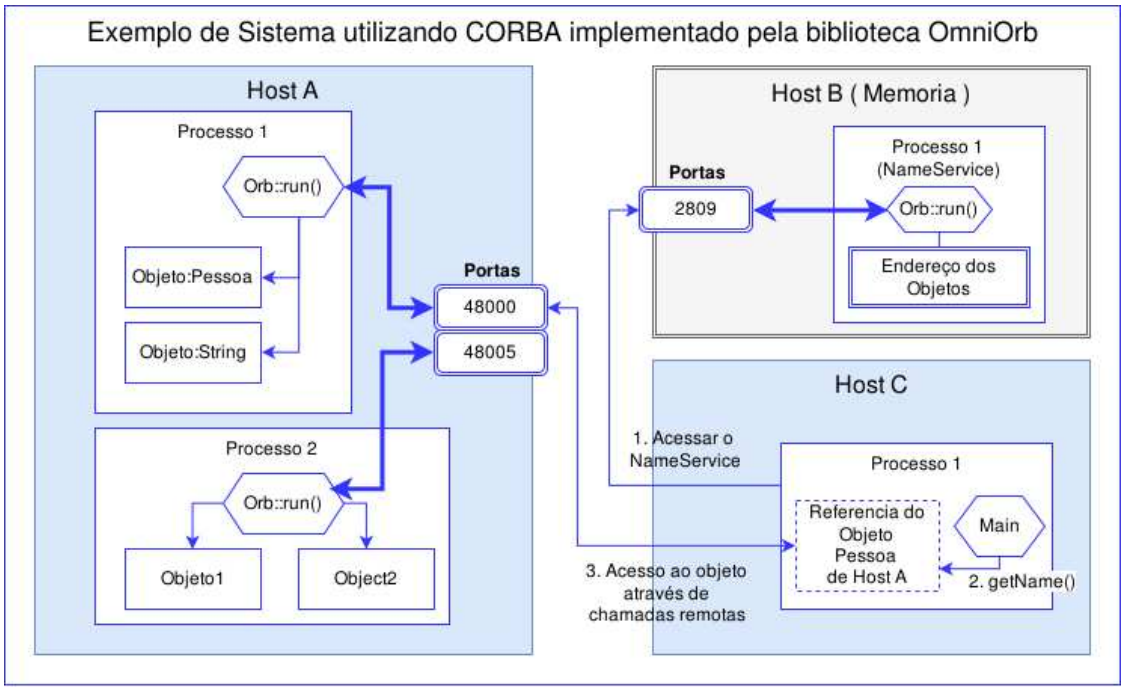


Figura 4.1: Exemplo de sistema do CORBA. No exemplo existem três *hosts*, o primeiro chamado de A tem dois processos e cada um disponibiliza dois objetos para o sistema. O *host* B fornece o serviço *Namespace*. E o *host* C tem um processo, o qual busca a referência do objeto em B e depois acessa diretamente A e requisita o método getName() do objeto Pessoa.

O endereçamento do objeto é realizado através do IOR, o qual é um vetor de bytes com um formato definido pelo CORBA. Um exemplo deste IOR pode ser visto a seguir:

```
IOR:010000002b00000049444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e
67436f6e746578744578743a312e3000000100000000000000700000000101020000d000000
3139322e3136382e312e33310000f90a0b0000004e616d655365727669636500300000000
0000000800000001000000005454410100000001c000000010000000100010001000000010 0
0105090101000100000009010100035454410800000006bed8c59010045e3
```

Este vetor de bytes contém as seguintes informações:

```
Type ID: "IDL:omg.org/CosNaming/NamingContextExt:1.0"
Profiles:
1. IIOP 1.2 192.168.1.31 2809 "NameService"
         TAG_ORB_TYPE omniORB
         TAG_CODE_SETS char native code set: ISO-8859-1
                       char conversion code set: UTF-8
                       wchar native code set: UTF-16
                       wchar conversion code set: UTF-16

         TAG_OMNIORB_PERSISTENT_ID 6bed8c59010045e3
```

A linha mais importante é a terceira, onde é descrito qual protocolo utilizar (IIOP 1.2), qual o endereço do host (192.168.1.31) e qual a porta do ORB (2809) e o nome do objeto desejado ("NameService"). Deste modo o CORBA pode endereçar qualquer objeto dentro de uma rede e portanto para criar uma comunicação com algum objeto precisa-se somente deste IOR.

Com o sistema do IOR funcionando é possível criar o serviço denominado pelo CORBA como *NameService*, o qual tem a funcionalidade de armazenar todos os IOR dos objetos

publicados e indexá-los por uma string. Esta funcionalidade é a mesma que a unidade de memória desempenha no modelo de Von Neumann. Por isso, na figura 4.1 está representado como memória.

O código abaixo demonstra um exemplo do padrão CORBA implementado pela biblioteca OmniORB. O código na linha 9 define um objeto distribuído chamado de UnidadeAB, o qual recebe como herança a classe UnidadeIdl, que está definida no arquivo unidade_idl.hh e foi gerada automaticamente pelo OmniOrb a partir do arquivo de interface do objeto.

A interface do objeto foi gerada automaticamente e permite a classe UnidadeAB definir o código das funções da interface, que no exemplo abaixo são os métodos funcaoA e funcaoB.

```
1
2  #include "unidade_idl.hh"
3  #include <omniconfig.h>
4  #include <iostream>
5
6  using namespace std;
7
8
9  class UnidadeAB : public POA_Example::UnidadeIdl {
10 public:
11   inline Server() {}
12   virtual ~Server() {}
13
14   virtual char* funcaoA(const char* mesg){
15     // Executa a tarefa A
16   }
17
18   virtual char* funcaoB(const char* mesg){
19     // Executa a tarefa B
20   }
21 };
22
23
24 int main(int argc, char** argv) {
25    // Inicializa o OMNI CORBA
26   CORBA::ORB_var    orb = CORBA::ORB_init(argc, argv);
27   CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
28   PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
29
30     // Instancia a Unidade
31   PortableServer::Servant_var<Server> server = new Unidade();
32   poa->activate_object(server);
33
34     // Ativa o Servidor
35   PortableServer::POAManager_var pman = poa->the_POAManager();
36   pman->activate();
37
38     // Fica escutando as requisições
39   orb->run();
40   orb->destroy();
41   return 0;
42 }
```

E por final, a função main do código acima, precisa inicializar o ORB (lembrando que necessita apenas de um ORB por processo), depois criar uma instância do objeto UnidadeAB, ativar o servidor e colocar o processo em escuta.

4.1.2 Lightweight Communications and Marshalling (LCM) - 2010

LCM foi apresentado no trabalho [Huang et al., 2010] e implementa um middleware orientado a mensagem sem nenhum recurso para chamadas remotas. A seguir é apresentado um exemplo de unidade de execução, o qual recebe dados de dois tópicos e executa a tarefa relacionada ao seu respectivo tópico, como mostrado na figura 4.2. Neste exemplo, pode-se analisar como o middleware relaciona os tópicos com uma unidade de execução, como o controle do middleware é declarado no código, como o aplicativo acessa os dados do tópico.

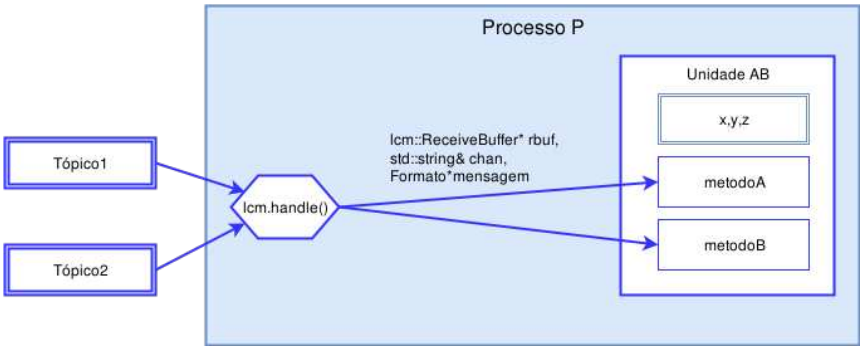


Figura 4.2: Processo P recebe os dados de dois tópicos (1 e 2) e a função lcm.handle recebe as requisições e o redireciona para os métodos A ou B, que fazem parte do objeto UnidadeAB, que tem x,y,z como seus atributos.

O código a seguir apresenta a declaração da unidadeAB na linha 5, o qual tem duas funções: metodoA e metodoB. Essas funções são relacionadas aos tópicos 1 e 2 nas linhas 33 e 34 do código a seguir. Portanto quando o controle do LCM receber algum evento do tópico1, ele executa o metodoA do objeto unidade instanciado na pilha da função main e quando receber o evento do tópico2 então executa o metodoB.

O controle é executado no laço while pela função lcm.handle() o qual verifica se veio algum novo evento e chama sua respectiva função. E permite ao desenvolvedor modificar este controle utilizando o corpo do *while*, pois a cada momento termina a execução lcm.handle, verifica-se ocorreu error, executa o corpo do *while* e volta para o *handle*.

```
1  #include <stdio.h>
2  #include <lcm/lcm-cpp.hpp>
3  #include "formato.hpp"
4
5  class UnidadeAB {
6     int x,y,z;
7
8   public:
9     ~Handler() {}
10
11    void metodoA(
12       const lcm::ReceiveBuffer* rbuf,
13       const std::string& chan,
14       const Formato* mensagem
15    ){
16       // Tarefa da metodo A
17    }
18
19
20    void metodoB(
```

```
21          const lcm::ReceiveBuffer* rbuf,
22          const std::string& chan,
23          const Formato* mensagem
24      ){
25          // Tarefa da metodo B
26      }
27  };
28
29  int main(int argc, char** argv){
30      lcm::LCM lcm;
31
32      UnidadeAB unidade;
33      lcm.subscribe("topico1", &UnidadeAB::metodoA, &unidade);
34      lcm.subscribe("topico2", &UnidadeAB::metodoB, &unidade);
35
36      while(0 == lcm.handle()){
37          // código de controle do aplicativo
38      }
39
40      return 0;
41  }
```

A estrutura Formato, descrita no arquivo formato.hpp, é gerado automaticamente pelo LCM a partir de um arquivo IDL, o qual descreve sua estrutura. O arquivo IDL permite que o LCM converta seu formato descrito para sua descrição na linguagem de programação utilizada no projeto, no exemplo anterior, a linguagem C++. Se outros módulos utilizarem esta estrutura, então o arquivo IDL deve ser compartilhado para todos os módulos, que o utilize.

4.2 Frameworks para Robótica

As situações mais frequentes da robótica normalmente consistem em poucos computadores próximos localmente. Alguns microcontroladores são responsáveis por manipular os diversos dispositivos do robô e um computador mais potente para as tarefas mais pesadas, como processamento de imagem e algoritmos de Simultaneous Localization And Mapping (SLAM). Deste modo, a maioria dos middlewares abordaram o problema utilizando o paradigma de assinante/publicante.

4.2.1 Open RObot COntrol Software (OROCOS) - 2001

Um dos mais antigos *frameworks* para robóticas de código aberto, o qual foi apresentado por [Bruyninckx, 2001]. Segundo [Orocos, 2017], a ideia nasceu em dezembro de 2000 após duas décadas de experiências decepcionantes com tentativas em uso de softwares de controle robótico comerciais para pesquisas avançadas em robótica. E estas decepções foram causadas pela falta de acesso às camadas mais profundas do controle de hardware.

Open RObot COntrol Software (OROCOS) é organizado em três componentes: Kinematics and Dynamics Library (KDL), Bayesian Filtering Library (BFL) e Real-Time Toolkit (RTT). Esta sessão apresenta apenas o RTT, pois somente ele está relacionado à modularização e à comunicação entre os componentes do *framework*.

O RTT organiza seu sistema como um conjunto de componentes interligados, como demonstrado na figura 4.3. Estes componentes enviam mensagens através das portas de saída e de entrada, que são semelhantes aos tópicos da orientação a mensagem. Uma vantagem deste

framework é a possibilidade de adoção de políticas de *realtime* em cada uma delas. O *framework* permite manipular essas conexões para permitir ao desenvolvedor criar a cadeia de componentes como queira. Contudo cada componente deve seguir o padrão estipulado pelo *framework*, o qual é descrito na figura 4.4.

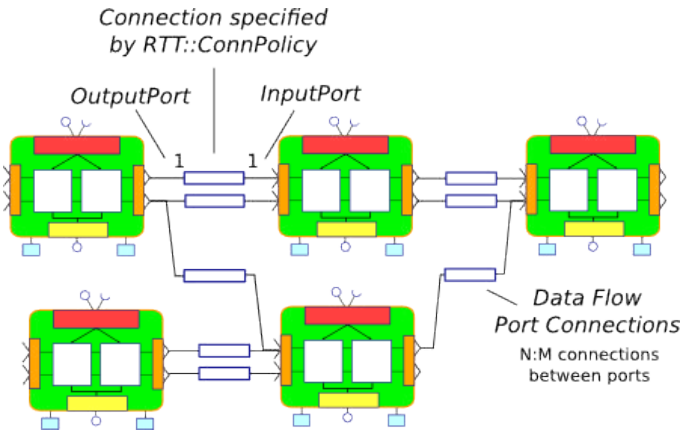


Figura 4.3: A figura mostra uma cadeia de componentes OROCOS, onde componentes conectados enviam dados através dos fluxos de entrada e saída. Figura retirada de [Bruyninckx, 2001].

A base conceitual deste *framework* advém do modelo deste componente. A interface de fluxo de execução desempenha o papel de controle, o qual fica aguardando alguma requisição ou evento e repassa para as funções em C/C++, *callbacks*. O fluxo de entrada, de saída, interface de configuração são as unidades que permitem a leitura e escrita de dados.

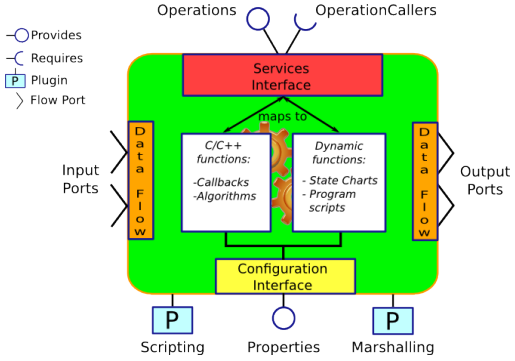


Figura 4.4: Componente base do OROCOS. Figura retirada de [Bruyninckx, 2001]

Portanto o modelo do OROCOS é bastante similar ao modelo Von Neumann, contudo tem uma variedades maior de componentes, sendo muitos destes uma subclasse da unidade de memória (fluxo de entrada,saída, propriedades) e subclasses de unidades de execução(*callbacks*,*scripts* etc).

4.2.2 Yet Another Robot Platform (YARP) - 2006

YARP é um conjunto de bibliotecas com o objetivo de separar os diferentes dispositivos, que formam o robô, de seu processamento e comunicação.

YARP implementa um middleware orientado a mensagem com recursos para chamadas remotas. Ele utiliza uma estrutura chamada de *bottle* para montar a mensagem. Esta estrutura é

uma fila e aceita os tipos naturais, inteiros, ponto flutuantes e strings. Devido a esta estrutura, o YARP não requer o compartilhamento do formato da mensagem explicitamente, o que torna a compilação do modulo mais independente. Contudo qualquer alteração na ordem das variáveis implica na alteração nos códigos dos módulos. O envio das mensagens do YARP pode ser realizado através de memória compartilhada, User Datagram Protocol (UDP), Transmission Control Protocol (TCP) ou *multicast*. A decisão de escolher qual é a melhor forma de transmissão fica por padrão a cargo do *middleware*.

A figura 4.5 apresenta um exemplo de unidade de execução, o qual recebe dados de dois tópicos e executa a tarefa relacionada ao seu respectivo tópico, como mostrado na figura 4.5. Neste exemplo, pode-se analisar como o middleware relaciona os tópicos com uma unidade de execução, como o controle do middleware é declarado no código, como o aplicativo acessa os dados do tópico.

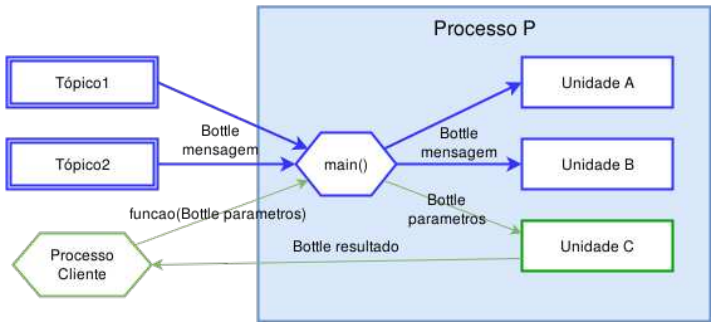


Figura 4.5: Representação da unidade de execução do YARP, o qual é descrita em C++ usando as funções deste framework.

O código a seguir tem a inclusão da biblioteca padrão do YARP na linha 1, declaração das funções unidadeA, unidadeB e unidadeC nas linhas 7, 11, 15 respectivamente. As três unidades recebem um parâmetro do tipo Bottle, o qual contém os dados recebidos do tópico ou os parâmetros recebidos pela chamada remota. As unidades A e B como são designadas para responder os eventos dos tópicos, e os tópicos não requerem nenhuma resposta, então a saída é void. Já a unidadeC devolve um Bottle, o qual contém o resultado da execução.

```
1  #include <yarp/os/all.h>
2  #include <stdio.h>
3
4  using namespace yarp::os;
5
6
7  void unidadeA(Bottle& mensagem){
8     // Tarefa da unidade A
9  }
10
11 void unidadeB(Bottle& mensagem){
12    // Tarefa da unidade B
13 }
14
15 Bottle unidadeC(Bottle& parametros){
16    Bottle resultado;
17    // Tarefas da unidade C
18    resultado.addString("resultado da operacao foi");
19    resultado.addInt(10);
20    return resultado;
```

```
}

int main(){
   Network yarp;

   // Criacao do tópico 1
   BufferedPort<Bottle> topico1;
   topico1.open("/topico1");

   // Criacao do tópico 2
   BufferedPort<Bottle> topico2;
   topico2.open("/topico2");

   // Criacao do serviço funcao
   RpcServer rpc;
   rpc.open("funcao");

   while (true) {
      // Leitura do tópico 1
      Bottle* mensagem1 = topico1.read();
      if (mensagem1!=NULL) {
         unidadeA(*mensagem1);
      }

      // Leitura do tópico 2
      Bottle* mensagem2 = topico2.read();
      if (mensagem2!=NULL) {
         unidadeB(*mensagem2);
      }

      // Leitura das requisições da funcao
      Bottle parametros;
      rpc.read(parametros,true);
      Bottle resultado = unidadeC(parametros);
      rpc.reply(resultado);
   }
}
```

A simplicidade do YARP está na utilização da estrutura Bottle para serializar os dados. Deste modo o código pode adicionar qualquer informação na estrutura e envia-lo para um tópico ou para um RPC como parâmetro, além de poder ler os dados recebidos pelos tópicos. Outros *frameworks* utilizam-se desta técnica, contudo pode haver erros durante a execução, caso um modulo resolva alterar a ordem dos dados da mensagem.

4.2.3 Robot Operating System (ROS) - 2009

ROS é um dos mais populares *frameworks* de robótica tanto que por uma busca pelas palavras “ROS“ e “robot“ no IEEE Xplorer(http://ieeexplore.ieee.org) obtém-se uma quantidade de 647 artigos relacionados. Ele é um *framework* de componente lançado sobre a licença Berkeley Software Distribution (BSD) e de acordo com [Quigley et al., 2009] seus objetivos filosóficos são: comunicação *peer-to-peer* entre os processos, suporte a diversas linguagens, ser modular, gratuito e de código aberto. Na mesma abordagem utilizada na sessão sobre YARP, aqui é apresentado como é o código de uma unidade de execução no framework ROS.

A figura 4.6 apresenta um exemplo de unidade de execução, o qual recebe dados de dois tópicos e executa a tarefa relacionada ao seu respectivo tópico. Como existe mecanismo de RPC, então existe uma declaração desta no exemplo.

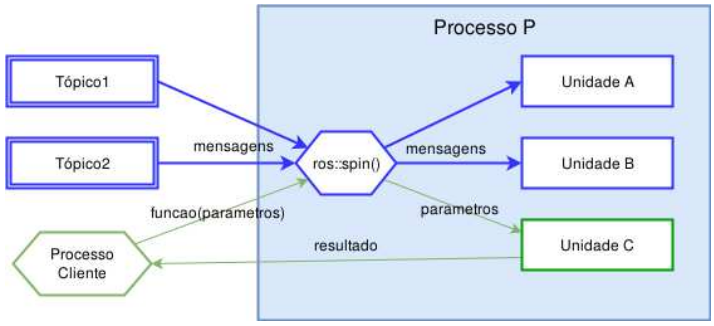


Figura 4.6: Representação da unidade de execução do ROS, a qual é descrita em C++ usando as funções deste framework.

No código abaixo mostra como implementar a figura 4.6 em C++ utilizando o ROS. E basicamente ele inicializa o *framework* e depois conecta uma *string*, seja ela o nome do tópico ou o nome do serviço, a uma função frequentemente chamada de *callback*. O número 1000 nas chamadas do método n.subscribe indica o tamanho do *buffer*, que neste caso pode armazenar 1000 mensagens em espera de execução.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void unidadeA_callback(const std_msgs::String::ConstPtr& msg) {
  // tarefa da unidade A
}

void unidadeB_callback(const Formato& msg) {
  // tarefa da unidade B
}

void unidadeC_callback(Request &parametros, Response &resultado) {
  // tarefa da unidade C
}

int main(int argc, char **argv){
  // Inicializacao das rotinas do ROS no Processo,
  // o qual recebe o nome de Processo P
  ros::init(argc, argv, "Processo P");
  ros::NodeHandle n;

  // Conexão dos eventos recebidos por topico1 com a rotina unidadeA_callback
  ros::Subscriber sub1 = n.subscribe("topico1", 1000, unidadeA_callback);

  // Conexão dos eventos recebidos por topico2 com a rotina unidadeB_callback
  ros::Subscriber sub2 = n.subscribe("topico2", 1000, unidadeB_callback);

  // Conexão do serviço chamado funcao com a rotina unidadeC_callback
  ros::ServiceServer service = n.advertiseService(
    "funcao", unidadeC_callback
  );

```

```
33   // Executa o controle do ROS
34   ros::spin();
35
36   return 0;
37 }
```

ROS assim como o LCM também requer o compartilhamento do formato da mensagem explicitamente, o que torna a compilação do modulo mais dependente, como demonstrado na figura 4.7.

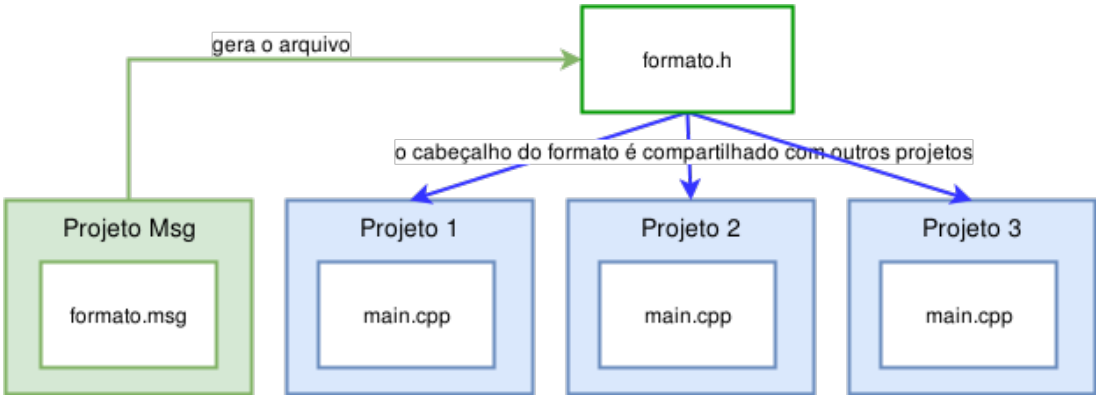


Figura 4.7: O projeto Msg contém a descrição da mensagem e o ROS faz o parsing e gera automaticamente a estrutura para a linguagem utilizada nos projetos, neste caso a linguagem C++ (formato.h). Então os diferentes projetos devem incluir o arquivo formato.h em seus códigos.

O framework do ROS é bastante completo pois tem mecanismo de RPC, mecanismo de orientação a mensagem, leitura de parâmetros, mecanismo padronizado de serialização, um sistema de compartilhamento de módulos já compilados através de um repositório, entre outros recursos. Além de uma grande documentação comparada aos outros *frameworks*.

4.2.4 Genom3 - 2010

A principal ideia do Genom3 é desacoplar o algoritmo central das linhas de código utilizadas pelos *middleware* para realizar o encapsulamento e a comunicação dos dados, como descrito em [Mallet et al., 2010].

O Genom3 organiza sistema em diversos componentes e cada componente pode ter um conjunto de funções escrita em uma qualquer uma linguagem suportada pelo framework. A seguir apresenta-se um exemplo de descrição de um componente chamado “simple“, o qual tem uma única função chamada add com dois números (*double*) de entrada com 0.0 como valor padrão e um outro número (*double*) como resultado. A palavra reservada lang indica que os “codels“ são escritos em C e a linha 7 “codel simple_add...“ indica que ali deve ser chamado o *codel* “simples_add.

```
1 component simple {
2   lang "c";
3   function add(in double x = 0. : "First number",
4                in double y = 0. : "Second number",
5                out double r =: "Result")
6   {
7     codel simple_add(in x, in y, out r);
8   };
```

```
9 };
```

Este componente é parseado pelo Genom3, o qual cria um arquivo de código em C com todos os protótipos das funções no formato correto da linguagem e permite ao desenvolvedor escrever os algoritmos dos *codels* na linguagem desejada. A seguir apresenta o componente simples descrito anteriormente e agora parseado pelo Genom3. O *framework* gerou todo o código apresentado e na linha 11 colocou o cabeçalho da função relativa ao codel simple_add e espera-se que o desenvolvedor coloque o algoritmo do *codel* simple_add na linha 8. Importante notar que os parâmetros descritos na função do componente devem ser traduzidos para o formato entendido pela linguagem C. Neste caso o tipo double utilizado pela variável existe na linguagem C, portanto não precisa criar uma *struct* e seus valores de entrada(in) são passados por cópia e os valores de saída(out) são passados como ponteiros.

```
1  #include "acsimple.h"
2
3  #include "simple_c_types.h"
4
5
6  /* --- Function add ---------------------------------------------------- */
7
8  /** Codel simple_add of function add.
9   *
10  * Returns genom_ok.
11  */
12 genom_event
13 simple_add(double x, double y, double *r, genom_context self) {
14   /* skeleton sample: insert your code */
15   /* skeleton sample */
16   return genom_ok;
17 }
```

Genom3 tem a descrição dos componentes e os códigos de cada *codels*, assim ele consegue que os *codels* não contenha nenhuma linha relacionada a tarefa de comunicação entre os componentes. Esta tarefa fica sobre responsabilidade do genom3, o qual antes de chamar o *codel* simple_add, por exemplo, ele realiza a leitura dos dados, desempacotamento e a chamada da função. Após a conclusão, ele realiza o empacotamento e envia os dados. Esta comunicação pode ser feita por qualquer middleware, seja pelo ROS, pelo Light Communications and Marshalling (LMC) ou qualquer um que tenha um *template* elaborado para o Genom3. Portanto, deste modo, ele cumpre o objetivo de desacoplar o código central do middleware.

Sob a ótica do modelo Von Neumann, este framework faz uma separação bastante explicita do controle e da unidade de execução. Enquanto os framework mostrados anteriormente criam funções especificas na linguagem utilizada para relacionar um evento a uma unidade de execução, Genom3 cria uma linguagem específica para descrever o controle descrito pelo arquivo (.gen) e utiliza outra linguagem já conhecida, como C, C++, Python para descrever as unidades de execução. Como o Genom3 tem as informações sobre toda a estrutura do sistema e tem a capacidade de gerar código, então isto possibilita esta separação do código central da comunicação.

4.3 Resumo

LCM apresenta um middleware orientado a mensagens, o qual possibilita o controle relacionar não apenas funções, mas também métodos de objetos aos eventos gerados pelos seus

tópicos. O formato da mensagem é compartilhado em tempo de compilação. Portanto o projeto, antes de compilar, deve possuir os formatos das mensagens, seja já na linguagem alvo ou por IDL. Um detalhe negativo ao LCM é o fato dele não prover serviços RPC, mas ainda assim ele pode ser facilmente utilizado como base para frameworks de robótica.

ROS apresenta um *framework* bastante completo, já que provê serviços RPC, modelo publicante-assinante e dados permanentes. E tanto LCM quanto ROS utilizam uma linguagem própria para descrever os formatos das mensagens, portanto deve existir o compartilhamento desses formatos antes da compilação do projeto. Mas esta característica permite validar e acessar diretamente as estruturas de dados utilizadas para comunicação dos dados.

YARP apresenta um *framework* com RPC e tópicos. A compilação de seus módulos é mais isolada pois não necessita compartilhar em tempo de compilação o formato das mensagens utilizadas. Contudo este isoladamente permite que ocorra erros durante a execução, quando um módulo enviar uma mensagem com tipos de dados ou ordem errada da mensagem.

Genom3 explora a diferença entre a programação do controle com a programação das unidades de execução. A programação do controle consiste em elaborar uma tabela que une o tipo do evento com a unidade de execução, então Genom3 possui uma linguagem para descrever o controle e os formatos das mensagens utilizadas. Deste modo possibilitou o desenvolvimento das unidades de execução em qualquer linguagem e sem grandes inserções de código do *framework* nas funções e objetos.

OROCOS apresenta um modelo para os seus componentes, o qual é demonstrado na figura 4.4. Este modelo é bastante parecido com o modelo de Von Neumann. Pois apresenta um fluxo de entrada, de saída, as funções centrais, seja as de C/C++, *callbacks*, funções dinâmicas e *scripts* são as unidades de execução e a interface de serviços desempenha o papel de controle, já que recebe uma requisição e mapeia para alguma função. A diferença do modelo do OROCOS do Von Neumann é não considerar que as portas entradas, de saída (dados temporários) e a interface de configuração (dados permanentes) fazem parte do mesmo elemento, a memória do módulo.

Capítulo 5

Prospecção para Proposta de Framework Unificado de Robótica (URF)

Cada *frameworks* de robótica apresentado no capítulo anterior tem suas próprias interfaces para realizar o acesso dos seus tópicos, dos seus atributos permanentes, dos serviços e normalmente utiliza a interface do sistema operacional para o acesso dos arquivos e diretórios do *host*. Isto afeta o requisito de transparência de acesso sobre os recursos dos *frameworks* e contamina o algoritmo com linhas de código para realizar estes acessos. Desta maneira, os módulos só são compartilháveis sem alguma alteração no código se utilizarem o mesmo middleware, ou se for colocado no sistema algum tópico intermediário que faça a comunicação entre os diferentes *frameworks*.

Genom3 resolve este problema abordando a descrição do controle juntamente com geração de código automático, o que possibilita uma alta transparência e utilização de qualquer middleware, desde que tenha sido feito o *template* do middleware para o Genom3. Contudo a portabilidade é afetada, pois a mudança para um outro middleware requer uma nova compilação do aplicativo.

5.1 Visão Geral

O URF tem como objetivo diminuir ao máximo a dependência do código do módulo com o sistema ou middleware utilizado. Para alcançar este requisito o *framework* URF idealiza um modo de desenvolvimento do módulo, o qual requer um sistema de transparência dos dados unificado. Pois o sistema integrador pode definir quais recursos o módulo pode utilizar. Contudo essa relação precisa de uma interface, que abrange uma grande gama de unidade de dados.

O módulo tem pode acessar os diferentes tipos de recursos através de uma única API e permite a publicação de diferentes recursos em uma mesma árvore. O sistema integrador pode definir quais os recursos são inicializados na árvore de recursos de um módulo e o módulo acessa essa árvore como se fosse um sistema de arquivo, contudo específico do módulo, que pode apontar para diferentes recursos. Isso possibilita maior independência do módulo, maior substituição de recursos e maior reuso e compartilhamento dos módulos.

Este conceito de árvore de recursos é demonstrado na figura 5.1, onde existe um processo sobre um sistema operacional qualquer, o qual fornece aos processos um sistema de arquivo e uma base de dados. Atualmente estes recursos são acessados diretamente através das chamadas ao sistema ou por Inter-Process Communication (IPC). Já no URF, estes recursos são referenciados na árvore de recurso do próprio processo e de preferências acessados através desta árvore, a qual é uma unidade de memória.

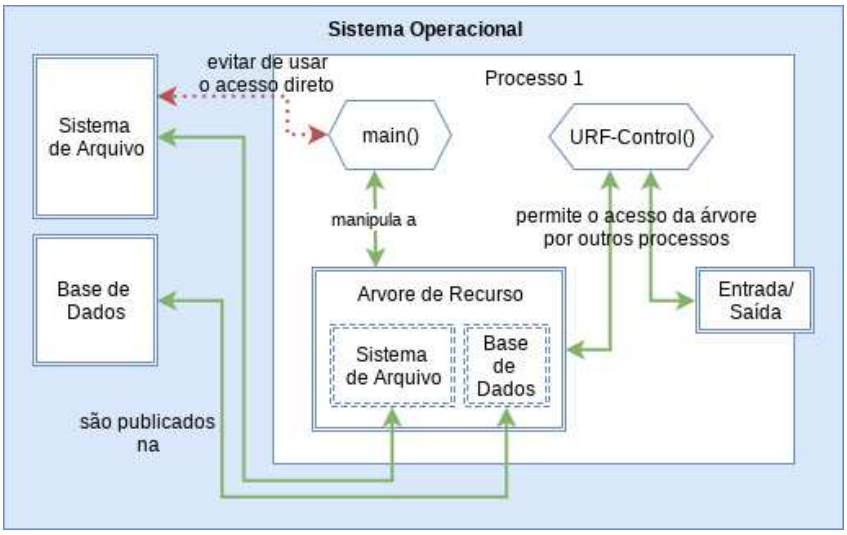


Figura 5.1: Cada processo tem uma árvore de recursos, onde são publicados os recursos internos e externos, como por exemplo, sistemas de arquivos, base de dados, segmento de memória do processo e outros. Também o processo tem uma thread específica (URF-Control) para compartilhar os dados desta árvore com outros processos.

Esta árvore de recursos é montada antes da execução da função "main" através de um *script* passado por uma variável de ambiente. Desta maneira o processo criador pode manipular de como o processo inicializado deve montar a árvore de recursos. Por exemplo, o processo criador pode requisitar que o índice /pose seja um link para o arquivo(./poses.csv) ou um tópico do middleware ROS ou YARP.

O processo também deve permitir um meio de compartilhar sua árvore de recursos com outros processos. Então o processo precisa ter uma outra *thread*, que executa a função do URF-Control o qual fica responsável em ler as requisições por algum meio(memória compartilhada, pipe, socket) e devolver os dados requisitados.

Com a árvore montada e compartilhada, este processo pode inicializar outros processos e disponibilizar os recursos de sua árvore como quiser a seus processos filhos. Assim como o sistema operacional fornece o sistema de arquivos para todos os seus processos, o processo pai pode fornecer seu próprio sistema de arquivos(árvore de recursos) para os processos filhos. Como demonstrado na figura 5.2. Como o modelo CDE é recursivo, então todo este sistema pode ser uma unidade de execução de outro modelo.

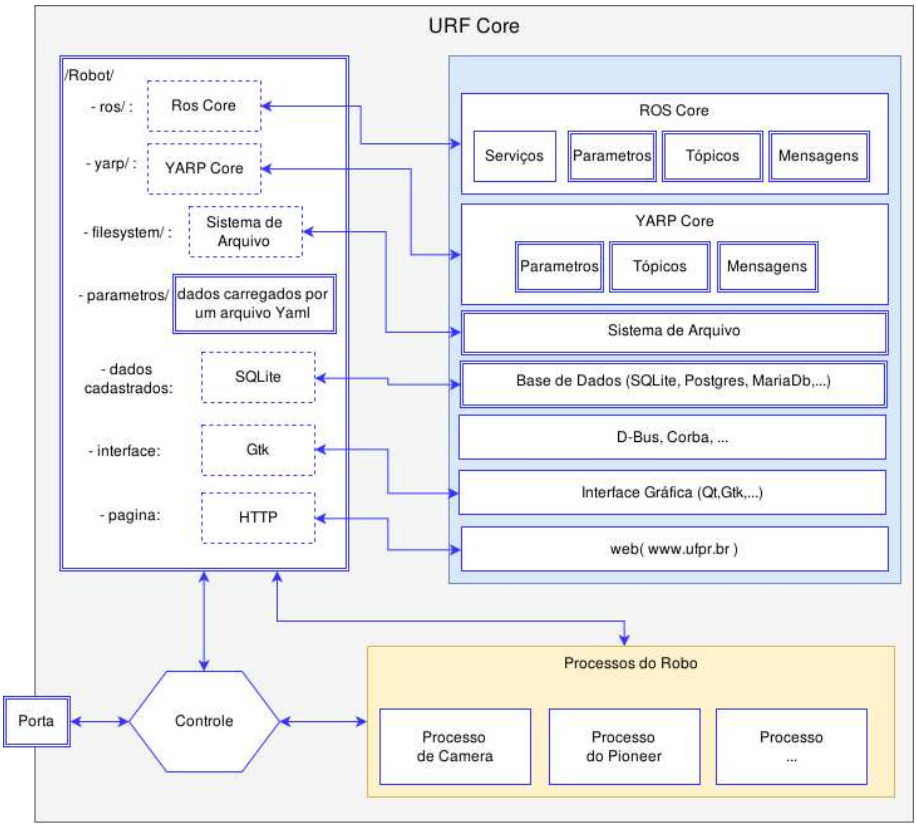


Figura 5.2: Existe um processo core, o qual elabora os dados do sistema com os recursos disponíveis, como base de dados, frameworks de sistemas distribuídos e outros. Então ele pode criar novas unidades de execução(processos) e compartilhar seus recursos com eles.

O desafio desta abordagem é elaborar uma API que suporte os mais diversos recursos. Então a primeira questão é definir o que é um recurso. A resposta desta questão foi buscada no modelo CDE e por este motivo os capítulos da fundamentação teórica e trabalhos relacionados desta dissertação foram revisados através da ótica deste modelo.

A definição de recurso utilizada neste trabalho é a definição de uma unidade de execução. Este modelo pode ser completo e ter os três componentes (controle, execução e dados) como por exemplo os processadores e o objetos distribuídos; ou incompleto como os disco rígido (controle e dados) ou objeto de um aplicativo (dados e execução).

O *framework* divide o desenvolvimento em dois ambientes, o ambiente do desenvolvimento do módulo em C++, que futuramente pode ser em outras linguagens, e o ambiente de integração permite a criar sistemas a partir da união dos módulos. Esta união foi elaborada sobre o modelo CDE e buscou uma grande flexibilização em como relacionar as unidades e também buscou um modo de um sistema já descrito se tornar uma unidade de execução de um sistema maior.

5.2 Descrição dos Sistemas

O *framework* URF idealiza dois ambientes possíveis de desenvolvimento, o primeiro é o desenvolvimento dos módulos em C++, que é utilizado em um sistema integrador. E o segundo ambiente é o sistema integrador, que descreve quais módulos são utilizados e como

eles são interligados. Esse segundo ambiente foi baseado no conceito dos arquivos *launch* do *framework* ROS, o qual utiliza eXtensible Markup Language (XML) para descrever todos os módulos necessários de um sistema juntamente com seus parâmetros. Também o ROS fornece a possibilidade de remapear os tópicos de um determinado módulo para que ele fique compatível com a nomenclatura utilizada sem a necessidade de alteração do código.

No caso do URF, existirá o comando urf-start, que vai ler um arquivo que contém toda a descrição do sistema e inicializar todo o sistema. Um exemplo de chamada deste comando no Linux está apresentado a seguir.

```
user@host:project# urf-start sistema.urf
```

A descrição do sistema, formulada para o URF, foi toda elaborada sobre os conceitos do modelo CDE, então os conceitos controle, unidade e dados fazem parte do cerne da descrição. A unidade de execução como fornecedora de algum serviço e o controle como um intermediário entre a unidade externa e a unidade de execução com o objetivo de isolar os diferentes protocolos de comunicação e modos de execução da unidade de execução. Portanto permitir um melhor compartilhamento desta unidade de execução.

A seguir existe um exemplo desta descrição, que descreve um sistema onde existe apenas uma unidade, o qual é um processo chamado de pioneer. Este processo tem a variável init, que indica como o processo deve ser inicializado, neste caso, ele deve usar o *driver* urf-process e criar um processo a partir do executável pioneer.

```
Processo pioneer : Unidade {
   init: {
      driver: urf-process;
      command: ./pioneer
   }

   data: {
      "base/odom": {
         driver: "ros"; topic: "pose"; msg: Odometry, mode="write"
      }
      "base/sonars": {
         driver: "ros"; topic: "sonars"; msg: PointCloud, mode="write"
      }
   }
}
```

A variável data contém informações de como deve ser elaborado a unidade de dados dentro do processo odometry. Nessa unidade de dados tem um diretório chamado base, que contém dois itens odom e sonars. Esses dois itens requerem o *driver* ros para criar um novo publicador de acordo com os parâmetros descritos em cada um. A função deste módulo é ler os dados dos dispositivos de odometria e sonares de sua base *pioneer* e publicá-los nos respectivos itens, sonars e odom.

A seguir um outro exemplo possível, porém mais complexo que o anterior, onde descreve-se um sistema chamado de Robo. Ele tem em sua unidade de dados um item chamado stdout, o qual recebe o tradicional *driver* de entrada e saída da biblioteca stdio. Esse sistema contém três processos: o ros-core, o qual é necessário para a utilização dos tópicos do ROS; o processo pioneer, que já foi descrito anteriormente; o processo Log, o qual vai ler os dados publicados no itens odom e sonars e redirecionará para o item stdout da unidade Log, o qual é redireciona para o stdout da unidade Robo, o qual é um redireciona para a tradicional função printf.

```
Sistema Robo : Unidade {
```

```
    data: {
       stdout: {driver: stdio}
    }

    Processo ros-core : Unidade {
       init: { driver: process; command: "roscore -p 3333" }
    }

    Processo pioneer : Unidade {
       init: {
          driver: urf-process;
          command: ./odometry
       }

       data: {
          odom: {
             driver: "ros"; topic: "pose"; msg: odometry; mode="write"
          }
          sonars: {
          driver: "ros"; topic: "sonars"; msg: PointCloud; mode="write"
          }
       }
    }

    Processo Log : Unidade {
       init: {
          drv: urf-process;
          cmd: ./view;
       }

       data: {
          odom: {
             driver: "ros"; topic: "pose"; msg: odometry, mode="read"
          }
          sonars: {
             driver: "ros"; topic: "sonars"; msg: PointCloud; mode="read"
          }
          stdout: {driver: "link"; unit:"Robo"; url: "stdout"}
       }
    }

}
```

Por fim, considerando a recursividade do modelo CDE, um sistema pode ser uma unidade de execução de outro sistema. Então o código a seguir exemplifica uma possível situação. Nessa situação, deseja-se criar um servidor utilizandoo protocolo http de maneira que, quando ele receber uma requisição, o sistema Web deve retornar como resultado a odometria da unidade Robo descrito anteriormente. Então o sistema tem o processo http, o qual realiza o papel de controle e executa um simples servidor http. Este importa o processo robo, o qual foi descrito anteriormente, contudo foi adicionado nesta instância uma unidade de controle para permitir a comunicação dele com o servidor http. Por fim, deve ser descrito a relação entre o controle http e o robo, ou seja, quando o http receber um determinado evento, ele deve realizar uma requisição à unidade robo e para isso ele deve conhecer o protocolo de comunicação.

```
Sistema Web : Unidade {
   init: {driver: urf-system;}

```

```
    Processo http : Controle {
        init: {
            driver: urf-process;
            cmd: ./http
        }
    }

    Processo robo : Unidade {
        import: { file:"sistema_robo.urf"; unidade: robo }

        Control {
            port: { driver: pipe; format: text }
        }
    }

    Relacao {
        src: http; dst: robo;
        event: "robo"
    }
}
```

Existem outros diversos detalhes para levar em conta na descrição dos sistemas, contudo esses três exemplos permitem clarear um horizonte futuro para o desenvolvimento do URF, que permite uma grande flexibilidade e reuso dos módulos sem que haja alguma alteração no código da unidade de execução. Isso ocorre devido a separação dos conceitos de unidades de controle, execução e dados, e fornecer mecanismo de alteração do controle e dos dados de unidade de execução.

5.3 Desenvolvimento dos Módulos

Na seção anterior foi apresentado de como é a descrição do sistema integrador dos diversos módulos. Nesta seção é apresentado como é idealizado o código de um módulo para que ele seja o mais independente possível do sistema integrador. O código a seguir exemplifica um módulo, onde a função main desempenha o papel de um controlador. Pois ele aguarda novos eventos da entrada padrão, lê seus dados, que no caso representa a posição de um robô, realiza alguma operação sobre eles e retorna os valores x,y do robô pela saída padrão. Este é um código desejado para desenvolvimento de um módulo para o sistema URF, pois ele não contém linhas de código em ROS, YARP e outros *frameworks*, o que possibilita o URF mudar o *framework* utilizado quando inicializar este processo.

```
#include <stdio.h>
#include <urf.h>

int main(){
    float robot_x, robot_y, robot_th;
    while ( isOk ){
        urf_fscanf(stdin, "%d %d %d",robot_x,robot_y,robot_th);
        ...
        urf_fprintf(stdout, "%d %d", robot_x, robot_y);
    }
}
```

Entretanto existem diversos problemas nesta mudança, os quais são identificados no código a seguir. O problema P1 consiste, que antes de inicializar este processo, deve-se indicar

um arquivo de *boot*, que contém uma lista de *frameworks* utilizados e também a descrição da unidade de dados do aplicativo, o qual vai indicar ao processo de como e para onde os dados devem ir, quando realizar uma operação urf_fscanf ou urf_fprintf sobre o stdin ou stdout. Os problemas p2, p5 e p6 são códigos que são realizados ocultamente e não devem estar no código do aplicativo, senão afetará a substituição dos *frameworks* utilizados.

O problema P2 deve inicializar e configurar os *frameworks* indicados no arquivo de *boot* ocultamente e permitir uma flexibilização em sua configuração, pois cada *frameworks* tem diferentes variáveis e opções para sua configuração. O problema P5 consiste em executar ocultamente a função do controle do *framework*, por exemplo, a função ros::oncespin() do framework ROS. E por fim o problema P6, que deve finalizar os *frameworks* inicializados ocultamente.

```
// P1: Definição da Unidade de Dados do Aplicativo;

#include <stdio.h>
#include <urf.h>

int main(){
   // P2: Inicialização dos frameworks utilizados;
   float robot_x, robot_y, robot_th;
   while ( isOk ){
      // P4: Transparência dos diversos tipos de dados;
      urf_fscanf(stdin, "%d %d %d",robot_x,robot_y,robot_th);
      ...
      urf_fprintf(stdout, "%d %d", robot_x, robot_y);
      // P5: Execução de um passo do controle do framework utilizado;
   }
   // P6: Finalização dos frameworks utilizados;
}
```

O problema P4 consiste em adaptar qualquer tipo de entrada ou saída para a interface utilizadas pelas funções urf_fscanf e urf_fprintf, ou seja, o urf_fscanf pode receber dados de um arquivo, de um pipe, de um tópico do *ROS*, de um tópico do YARP etc, enquanto o urf_fprintf pode enviar dados também para uma diversidade de tipos de saída. Além para onde os dados devem ir, o formato da mensagem pode mudar, e dependendo do caso, a mensagem requer uma tradução. Todos esses detalhes de onde, como, para onde são descritos no arquivo de *boot*, a seguir segue um exemplo.

```
stdin: {driver: ros, topic: pose, msg: Odometry}
stdout: {driver: filesystem, url: saida.txt}
```

5.4 Interface com as Unidades de Dados

As unidades de dados como visto no capítulo 2.2.3 podem apresentar uma grande variedade de comportamentos. O que leva a necessidade do *framework* URF prover uma única interface capaz de fornecer a comunicação com qualquer um desses tipos de unidades de dados. Essa abstração é essencial para o objetivo de independência do módulo com o sistema integrador, pois permite ao sistema integrador flexibilizar os modos de comunicação entre as unidades e também ao módulo utilizar um único código para diversos tipos de comunicação.

Primeiramente o trabalho dividiu esses diversos as unidades de dados em: Dados proveniente de linguagem de marcação e comunicação entre processos, como YAML, JSON, XML; Unidades que realizam uma ponte para dados localizados em outros dispositivos e

gerenciados em forma de sistema de arquivos; Unidades que realizam uma ponte para dados provenientes de outros *frameworks* de robótica, como mostrado na figura 5.3.

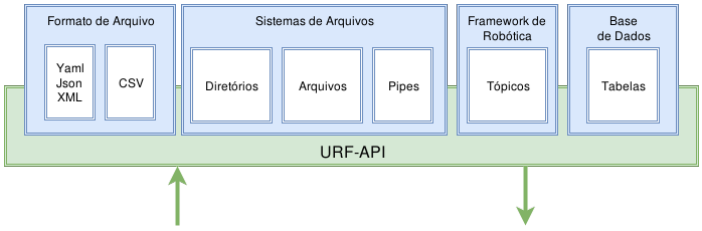


Figura 5.3: API deve abranger a manipulação de estruturas geradas por YAML, JSON e XML, também arquivos CSV, sistemas de arquivos e tópicos dos frameworks de robótica. Já as bases de dados é opcional, mas para tê-los em mente para trabalhos futuros.

Pois essas três classificações abrange uma grande variedade de tipos. Os dados provenientes de linguagem de marcação são dados localizados na memória do processo e pode conter itens, vetores e mapas. Já a ponte para sistemas de arquivos permite utilizar os sistemas de arquivos do próprio sistema operacional ou de algum sistema remoto através do protocolo Secure Shell (SSH), por exemplo. E por fim dados provenientes de *frameworks* de robótica o qual permite enviar dados através dos fluxos definidos por estes *frameworks*. Portanto o objetivo dessa interface proposta é permitir manipular sistema de arquivos como se fosse uma estrutura YAML ou manipular fluxo de dados da robótica como se fosse arquivos CSV.

A interface proposta para encapsular essas diferentes estruturas é mostrada na imagem 5.4. Existe a classe base chamada de unidade, a qual representa uma unidade de execução, logo tem função para ler seus atributos, métodos e metadados e realizar uma chamada de procedimento. Pode-se notar, que basicamente são as principais propriedades de um objeto, portanto pode-se encapsular os objetos distribuídos.

A unidade de dados também é uma unidade de execução, portanto recebe por herança as funções da classe Unit. E por final, a classe Data tem duas especializações: Item e Collection. A seguir uma breve descrição de cada método da classe Unit:

Unit::close finaliza uma unidade;

Unit::attributes retorna uma unidade de dados, o qual permite percorrer pelos atributos da unidade;

Unit::call permite chamar uma função fornecida pela unidade. Recebe o nome do comando e uma memória de parâmetros e devolve outra unidade de dados como resultado.

Unit::getType retorna se é somente uma Unit, uma Data, Item ou Collection;

Unit::metadatas retorna uma unidade de dados, o qual permite percorrer os metadados da unidade;

Unit::methods retorna uma unidade de dados, o qual permite percorrer todas as funções da unidade;

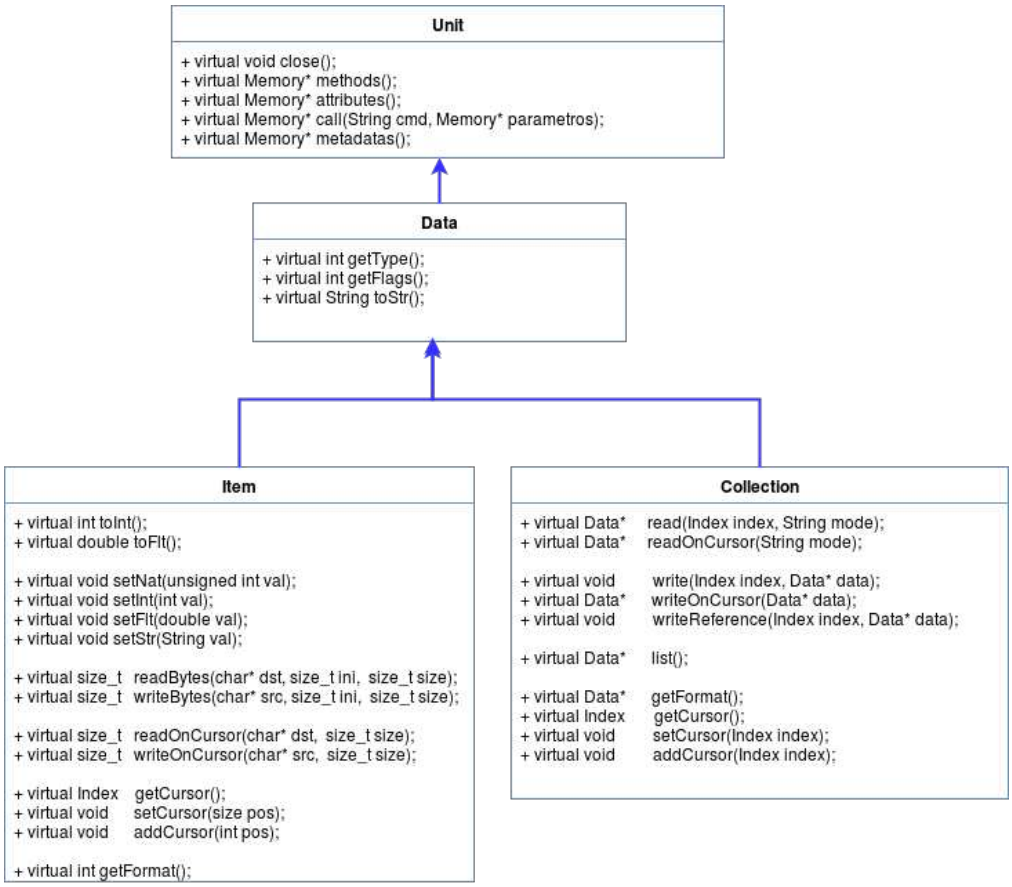


Figura 5.4: Interface proposta

A unidade de dados por sua vez tem uma função que define quais atributos ela tem, se tem acesso aleatório, se o tamanho do elemento é fixo, se tem formato e outros. Estes atributos implicam em alguns efeitos, mas não alteram a interface de acesso. A seguir uma breve descrição de cada método da classe Data:

Data::getFlags retorna os atributos dos dados. Se ela é externa ou interna, se ela é formatada ou não, se é um arquivo ou um pipe, se o elemento é fixo ou variável, ...;

Data::toStr converte a unidade de dados para um *string*. Para um item simplesmente realiza uma cópia dos dados, já para um conjunto, os dados são serializados em uma única *string*.

A principal característica da classe Item é permitir uma manipulação de dados como números, pontos flutuantes, *strings* e vetor de bytes.

Item::toNat converte os dados para um inteiro;

Item::toInt converte os dados para um inteiro;

Item::toFlt converte os dados para um ponto flutuante;

Item::setNat seta o item com um número natural;

Item::setInt seta o item com um número inteiro;

Item::setFlt seta o item com um ponto flutuante;

Item::setStr seta o item com uma *string*;

Item::setVet seta o item com um vetor de bytes;

Item::readBytes lê uma quantidade de bytes indicando a posição inicial e o tamanho do segmento;

Item::writeBytes escreve uma quantidade de bytes indicando a posição inicial e o tamanho do segmento;

Item::readOnCursor lê uma quantidade de bytes iniciando pelo índice indicado pelo cursor;

Item::writeOnCursor escreve uma quantidade de bytes iniciando pelo índice indicado pelo cursor;

Item::getCursor recebe a posição do cursor;

Item::setCursor movimenta a posição do cursor;

Item::getFormat retorna se o item é simplesmente um vetor de bytes binário, um número natural, inteiro, um ponto flutuante ou uma string;

A principal função da classe Collection é gerenciar um conjunto de itens, que podem ser acessados pelos seus métodos read ou write.

Collection::read Lê um item do conjunto apontado pelo índice;

Collection::write escreve um item no conjunto apontado pelo índice;

Collection::readOnCursor Lê um item do conjunto apontado pelo cursor da unidade;

Collection::writeOnCursor escreve um item no conjunto apontado pelo cursor da unidade;

Collection::getCursor recebe a posição do cursor;

Collection::setCursor movimenta a posição do cursor;

Collection::getFormat retorna o formato do conjunto, caso exista este formato;

Capítulo 6

Aplicação da Interface

Este capítulo exemplifica algumas ações de manipulação dos dados utilizadas em tradicionais bibliotecas, que manipulam arquivos YAML, JSON, CSV, em sistemas de arquivos e em recursos de dados fornecidos pelos *frameworks* de robótica, como os tópicos.

6.1 Interface sobre YAML

Esta seção demonstra, que a interface é capaz de manipular uma estrutura do tipo YAML, JSON ou XML. Nestes casos, basicamente, existem alguns formatos escalares como números naturais, inteiros, ponto flutuantes ou *string*, e conjuntos de elementos como sequência (vetores de índices 0 até o tamanho máximo) ou um associativo(estrutura de busca). Estes dois conjuntos podem indexar uma variável escalar ou outro conjunto e o mais importante, todos os dados já estão na memória do processo.

Qualquer biblioteca que realiza o *parsing* de dados representados em linguagens como YAML, XML e JSON cria uma estrutura na memória do aplicativo, onde armazena todos os dados. A própria linguagem de marcação possibilita uma tipagem simples das variáveis em *string*, inteiro, ponto flutuante através de sua sintaxe, como por exemplo uma *string* obrigatoriamente deve começar por aspas.

Portanto essas bibliotecas fornecem funções para obter o tipo da variável e também para obter seu conteúdo, em formato de *string*, inteiro, ponto flutuante, vetor ou um mapa. Quando o aplicativo requerer o conteúdo de uma determinada variável nesta estrutura, a biblioteca retorna todo o seu conteúdo.

A seguir segue um exemplo de arquivo Yaml com os dados de um usuário. Esse exemplo é utilizado para mostrar como esses dados são manipulados pela biblioteca Yaml-C++ e logo em seguida como esses mesmos dados são manipulados através da interface proposta.

```
1 Yaml
2 id: 10
3 name: "fulano"
4 home: "/home/fulano"
5 score: 20.2
6 database: {
7    name: "db"
8    data: [10,20,30,40]
9 }
```

A seguir o código utilizando a biblioteca Yaml-C++, onde na primeira linha realiza o *parsing* do arquivo e cria uma estrutura em memória para armazenar todos as informações lida

do arquivo. Após como realizado a leitura de variáveis do tipo inteiro, ponto flutuante, *string* e também de uma coleção contida de dentro da coleção. Essa biblioteca utiliza o instrumento de *template* fornecido pela linguagem C++ para permitir os diferentes tipos de saída da função as().

```
YAML::Node collection = YAML::LoadFile("user.yml");
  int  id = collection["id"].as<int>();
 float score = collection["score"].as<float>();
string name = collection["name"].as<std::string>();
YAML::Node database = collection["database"];
```

A seguir exemplo do código utilizado pela interface, onde na primeira linha realiza o *parsing* do arquivo pela instanciação do objeto DriverYamlCpp, que utiliza em seu núcleo a biblioteca Yaml-C++ e o encapsula para a interface do URF. Nas linhas seguintes, o código requisita a leitura da variável id pela função read(), converte o ponteiro Data* para Item* pela função item() e requisita a leitura dos dados como um inteiro. As outas linhas seguem o mesmo padrão.

```
Collection* collection = new DriverYamlCpp("user.yml");
int    id = collection->read("id")->item()->toInt();
float score = collection->read("score")->item()->toFlt();
string name = collection->read("name")->item()->toStr();
Collection* database = collection->read("database");
```

A seguir um código com a biblioteca Yaml-C++ para verificar o tipo de um nó da estrutura. Nesta biblioteca cada item é um nó e ele pode *Null*, *Scalar*, *Sequence*, *Map* e *Undefined*.

```
if ( node.Type() == Scalar ) {
   ...
} else if ( node.Type() == Sequence ){
   ...
} else if ( node.Type() == Map ){
   ...
}
```

A seguir o mesmo código anterior, contudo utilizando a interface proposta para verificar se é um item ou um subconjunto. Lembrando que o parâmetro mode da função read foi ocultada por um valor *default*.

```
Data* data = collection->read("id");
if ( data->isItem() ){
   ...
} else if ( data->isCollection() ){
   Collection* collection = data;
   if ( collection->isSeq() ){
      ...
   } else if ( collection->isMap() ){
      ...
   }
}
```

6.2 Interface sobre CSV

Um arquivo CSV é um conjunto de linhas e cada linha é um conjunto de somente escalares, como demonstrado na figura 6.1. Contudo a API proposta permite que dentro de uma linha tenha um outro conjunto.

	X (int)	Y (int)	Name (String)
0	0	10	Cozinha
1	20	11	Sala
2	23	15	Sala
	...	...	...
N	120	50	Corredor

Figura 6.1: Exemplo de arquivo CSV

Exemplo abaixo realiza uma leitura aleatória, portanto o desenvolvedor define qual linha deve ser lida. Logo após lê o escalar do índice 1(y) e o converte para *string*. Depois o escalar do índice "x"e o converte para inteiro.

```
Collection* csv = new CSV("arquivo.csv");
Collection* linha = csv->read( 0 );
Item* sy = linha->read( 1 );
String y = sy->toStr();
Item* sx = linha->read( 0 );
int   x = sx->toInt();
```

Exemplo a seguir realiza uma leitura sequencial, portanto o desenvolvedor não indica qual linha deve ser lida e sim o próprio arquivo.

```
Collection* csv = new CSV("arquivo.csv");
Index linha_corrente;
Collection* linha = csv->readCursor(&linha_corrente, "r" );
cout << linha_corrent->toStr() << endl;
Item* y = linha->read( Index(1), "r" );
...
```

Um detalhe, que dentro da linha, os itens podem ser referenciados por números de 0 a N, ou pelos nomes deles, caso tenha. A implementação pode verificar no método read, se o índice é um inteiro ou uma string, portanto ele sabe quando buscar por um ou pelo outro. Já na rotina de listar os índices deve ser escolhido apenas um tipo. Este caso ainda requer um estudo melhor.

A parte da leitura é mais tranquila, pois deve ir percorrendo a estrutura e quando chega em um escalar, fica a cargo da implementação da estrutura CSV convertê-lo em *string*, inteiro ou ponto flutuante. Já a escrita é mais complicada pois deve respeitar a formatação da linha.

```
// Preparacao da mensagem (name:"felipe",y:10,x:20)
Collection linha;
linha.write( 0, ConstScalar("felipe") );
linha.write( 1, ConstScalar(10) );
linha.write( 2, ConstScalar(20) );

// Traducao de (name,y,x) para (x,y,name)
Collection* csv = new CSV("arquivo.csv");
Collection* datawriter = csv->getBuffer();
Collection* translate = new Translate(datawriter, "name:0,y:1,x:2");

// Escrita na tradução
translate.writeCursor( linha );
```

Existe a preparação da mensagem e sua ordem é definida pelo desenvolvedor, logo após o código abre o arquivo CSV, adquire um datawriter, o qual contém o formato da mensagem que está sendo utilizada e o passa como parâmetro para o Translate, o qual também recebe o formato da mensagem do desenvolvedor. E por fim a escrita no Translate corresponde a escrita no conjunto csv.

No caso apresentado, o código tem linhas que são utilizadas para realizar esta tradução, o que não é muito elegante. Contudo na proposta do URF, esta parte de tradução é realizada na elaboração da árvore de recursos, pois permite ao *framework* verificar se o programa utiliza o mesmo formato, portanto pode permitir a escrita direta no *datawriter* ou se requer uma tradução, como mostrado a seguir. Onde o índice /csv/pose referência diretamente o *datawriter* ou indiretamente e o índice /csv/pose referência o tradutor.

```
// Escrita direta no datawriter do CSV
/csv/pose => csv.datawriter;

// Escrita indireta no datawriter do CSV
/csv/pose-datawriter => csv.datawriter
/csv/pose => translateTo (csv.datawriter);
```

Desta maneira o aplicativo sempre vai escrever sobre o índice /csv/pose, mas este pode ser alterado no *boot* do processo para que o desenvolvedor do sistema distribuído manipule as comunicações como ele quiser.

6.3 Interface sobre Sistemas de Arquivos

Nesta sessão é apresentado como a API manipula as principais entidades do sistema de arquivos, como os arquivos regulares e diretórios. A manipulação dos arquivos normalmente envolve apenas pequenos pedaços dos arquivos, diferente do modelo YAML, onde simplesmente a escrita substitui o conteúdo anterior pelo novo. A seguir é apresentado um código que manipula um pequeno segmento de dados do arquivo. Como o índice será sempre numérico, não necessita do objeto Index.

```
Collection* dir = new Diretorio("/home/usuario");
Item* file = dir->read("image.bmp","r");

// lê 32 bytes começando pelo índice 10
char buffer[22];
file->readBytes(&buffer,10,32);
```

6.3.1 Verificação da Existência de um Item

Se a unidade de dados tiver um índice contínuo, então basta saber se a posição a ser verificada está dentro do intervalo. Por exemplo, se existe um arquivo de 1Mb, então basta saber se a posição está entre 0 a 1Mb e portanto o custo será desta operação é desprezível e O(1). Caso seja uma unidade de dados de índice não contínuo, então deve ser feito uma busca e o custo da operação varia de acordo com a estrutura utilizada(vetor, árvore binaria, tabela hash).

```
Collection* dir = new Diretorio("/home/usuario");
Data* node = dir->read( "pictures" );
if ( node.isBlank() ){
   // não existe o índice
} else {
   // existe o índice
}
```

6.3.2 Listagem dos Diretórios

A listagem dos índices é uma operação fundamental sobre unidade de dados de índices não contínuos, pois permite conhecer todos os índices lá guardados. Estes índices podem ser inteiros, strings ou ponto flutuantes. Contudo para unidade de dados de índices contínuos não tem tanta necessidade pois utilizam quase sempre números como índices, estes podem ser somados ou subtraídos, e dado o tamanho da unidade pode-se saber todos seus índices.

```
Collection* dir = new Diretorio("/home/usuario");
Data* list = dir->list();
while (true){
   Index index;
   Data* item = list->read(&index);
   if ( item->isBlank() ){break;}
   cout << index.toStr() << ": " << item->toStr() << endl;
}
```

6.3.3 Atributos e Metadados

Nos sistemas de arquivos cada nó tem uma série de atributos como permissão, usuário, grupo, horário de acesso, de modificação e outros. No Linux existe a função stat para a leitura dos atributos e funções especificas para alteração de cada atributo, como chmod, chown, truncate. Estes atributos podem variar de sistema para sistema, contudo boa parte deles estão padronizados pelo POSIX. A seguir há um exemplo dos atributos de um arquivo.

```
mode: 0644,
user: usuario,
group: grupo,
size: 1024,
...
```

Então pode-se notar, que os atributos são uma unidade de dados indexada por uma *string* e seus valores são escalares. Então pode-se utilizar a mesma interface utilizada para os formatos JSON já apresentados.

```
Collection* dir = new Diretorio("/home/usuario");
Data* node = dir->read("arquivo.txt","");
node->attr()->write( "mode", ConstScalar(0755) ); // chmod
node->attr()->write( "user", ConstScalar("usuario") ); // chown
```

6.4 Interface sobre Frameworks

Esta seção apresenta de como a API proposta funciona sobre o *framework* ROS e facilmente pode ser convertido para os outros *frameworks*, exceto o Gemon3, o qual trabalha sobre geração de código e não sobre a execução do módulos e comunicação. A figura 6.2 demonstra que ROS contém três conjuntos: formato das mensagens, tópicos e atributos(parâmetros). Os formatos das mensagens contêm informações de como são estruturadas as mensagens e pode ser acessado de forma semelhante a um arquivo YAML, o qual é mostrado na seção anterior. Os tópicos podem ser acessados de forma semelhante a um arquivo CSV, exceto na questão de inicializar o tópico. Contudo a parte de inicialização é oculta ao módulo, pois este é realizado pelo *framework*, portanto a interface de manipulação de um arquivo CSV e de um tópico é

semelhante. Porém o tópico é um fluxo e não pode ser acessado aleatoriamente, enquanto o arquivo CSV pode ser acessado aleatoriamente.

Os atributos de algum *framework* pode ser manipulado de forma semelhante ao YAML, pois tem um índice composto por uma *string* e seu conteúdo pode ser um inteiro, ponto flutuante, *string* ou um outro mapa indexado por *string*.

Os serviços são unidades de execução e portanto podem ser listadas pelo método Unit::arch() e chamadas através do método Unit::call().

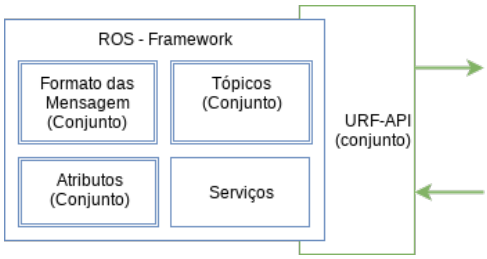


Figura 6.2: ROS fornece um conjunto de formatos de mensagens disponíveis, um conjunto de tópicos abertos, conjunto de atributos e um conjunto de serviços. Todos esses serviços podem ser acessados pela API do URF.

6.5 Resumo

Os exemplos apresentados indicaram como é utilizado a interface para os desenvolvedores dos módulos e como eles manipulam as unidades de dados. Também teve em vista que a interface fique simples e de fácil utilização para qualquer situação. Contudo não exemplifica como encapsular uma estrutura de dados, seja a biblioteca Yaml-C++, ou uma biblioteca que manipule CSV ou uma base de dados como Sqlite3 nesta interface para que seja possível os módulos a utilizarem. Esse detalhe é apresentado na próxima seção.

Capítulo 7

Discussão

Esta seção apresenta alguns problemas identificados na elaboração da interface a partir de problemas encontrados no encapsulamento das funções da biblioteca Yaml-C++ e do sistema de arquivo na interface proposta. Para uma melhor didática, esse capítulo simplifica e indica os três principais problemas, os quais são exemplificados a partir de uma simples estrutura em C++ descrita a seguir, a qual contém um espaço para os dados privados e uma interface para o acesso a esses dados. Sobre o exemplo de código em C++ a seguir, discute-se esses problemas identificados.

```
1 struct VetorA {
2    int dados[128];
3
4    virtual int get(int indice){return this->dados[indice];}
5    virtual void set(int indice, int val){this->dados[indice] = val;}
6 };
```

7.1 Generalização

A primeira dificuldade é generalizar o tipo do conteúdo de int para qualquer classe. Na linguagem C++, pode-se criar um *template* e utilizar a geração de código do próprio compilador para este propósito, contudo isso implica na restrição do desenvolvimento somente sobre a linguagem C++. Portanto o desejado é um conjunto de ponteiros de funções, o qual permite qualquer linguagem realizar um *wrapper* sobre este conjunto.

```
1 struct VetorB {
2    ? dados[?];
3
4    ?   get(? indice){}
5    void set(? indice, ? val){}
6 };
```

Este problema de generalização também ocorre com o índice, pois como visto, pode ser um inteiro, ponto flutuante, *string* ou um conjunto de inteiros. Como o índice é uma estrutura que deve sempre estar na memória do aplicativo. Então pode-se criar facilmente uma estrutura em C++, que armazene inteiros, pontos flutuantes ou *string* de acordo com a necessidade, entretanto deve ser uma estrutura muito bem desenvolvida em vista do requisito de desempenho. Pois o programa frequentemente precisará ler e alterar seu índice, principalmente quando percorrer um vetor ou matriz.

A solução desse problema pela interface foi considerar 4 tipos primários de dados: natural, inteiro, ponto flutuante e *string*. Deste modo a estrutura deve definir as funções para converter seus dados para qualquer um desses tipos. Também uma função que identifique qual é o tipo, para que o programa possa analisar como ler uma estrutura de dados desconhecido.

Porém esta solução tem um problema, pois existem tipos de unidades de dados, que não podem ser convertidos facilmente para qualquer um desses tipos primários, como uma coleção de itens, pois ele contém diversos dados e não pode ser reduzido a um único item sem passar por uma função de serialização. Por isso, as unidades de dados foram classificadas em item e coleção. Embora no item, algumas *strings* não podem ser convertido para inteiro, mas o item consiste em apenas em um valor, apesar deste valor não poder ser convertido para todas as possíveis representações. Em uma coleção, como contém diversos itens, primeiramente deve indicar qual item deve ser lido.

Uma dificuldade sobre a definição de item e coleção é que o item é, por exemplo uma *string*, também pode ser um conjunto de bytes ou palavras. Então torna-se um problema de como manipular o item como se fosse uma coleção.

7.2 Modo de Acesso

Um complexo problema é o modo de acesso a unidade de dados. Este problema resume-se nos dois tipos de protocolos de comunicação *stateful* e *stateless*.

A seguir este problema é exemplificado através de dois códigos em C++. O primeiro descreve a estrutura VetorAleatorio, que contém as funções *get* e *set*. Esta estrutura considera que cada chamada de função como uma transação independente e portanto o índice é passado como parâmetro e não existe concorrência de escrita sobre o índice.

```
1 struct VetorAleatorio {
2    int dados[128];
3
4    virtual int get(int indice){return this->dados[indice];}
5    virtual void set(int indice, int val){this->dados[indice] = val;}
6 };
```

Entretanto a estrutura VetorSequence descrito a seguir tem a função *seek*, que posiciona o cursor sobre um índice; a função *get* sem parâmetros, a qual devolve o valor indicado pelo cursor; e a função *set*, a qual escreve um valor sobre o cursor. E tem como o cursos como um dado necessário entre as transações e portanto utiliza o conceito *stateful*.

```
1 struct VetorSequence {
2    int dados[128];
3    int cursor;
4
5    void seek(int indice){this->cursor = indice;}
6    int read(){return this->dados[cursor++];}
7    void write(int val){this->dados[cursor++] = val;}
8 };
```

O modo utilizado por VetorSequence é ideal para manipular unidades de dados como fluxo de dados, pois permite que própria unidade de dados manipule o cursor. Contudo um dos problemas de ter o índice junto com a unidade é que algumas vezes precisa-se abrir outras conexões para manipular os dados com diferentes *threads*.

7.3 Localização dos Dados

Uma unidade de dados pode ser um instância e ter contido nelas os seus dados ou ela simplesmente ser uma conexão e ter um caminho para obtenção daqueles dados. Por exemplo, na estrutura descrita a seguir chamada de Interno. Ela contém os dados internamente, logo, quando ela for liberada os seus dados também serão. Diferentemente do que ocorre quando a estrutura, a seguir, Externo for liberada, pois como ele é apenas uma conexão para o arquivo através do FILE, então, quando ela for liberada apenas a conexão é liberada e os dados continuam persistentes.

```
1 struct Interno {
2    int dados[128];
3    int cursor;
4
5    void seek(int indice){this->cursor = indice;}
6    int read(){return this->dados[cursor++];}
7    void write(int val){this->dados[cursor++] = val;}
8 };
```

```
1 struct Externo {
2    FILE* fd;
3
4    void seek(int indice){fseek(fd,indice,SEEK_SET);}
5    int get(){int a; fread(&a,1,4,fd); return a;}
6    void set(int val){fwrite(&a,1,4,fd);}
7 };
```

7.4 Resumo

O problema da localização dos dados, que classifica uma unidade de dados entre instância e conexão. Esta por sua vez pode realizar uma conexão com outra conexão e assim por diante. Juntamente com os dois modos de acesso, principalmente o modo sequencial, que precisa manter dados para cada conexão aberta, logo torna-se uma cadeia de conexões nada simples de ser gerenciável e aumenta as possibilidades consideradas por uma interface única. Além que para aumentar a complexidade ainda existe a generalização dos tipos dos índices, tipos de dados, tamanho dos dados e metadados. A interface proposta classifica as unidades de dados em item e coleções. Deste modo tanto item como coleções tem funções de acesso aleatório e acesso sequencial, contudo não garante um bom gerenciamento das diversas conexões, o que pode ocasionar conexões não liberadas e concorrências nos dados, portanto instabilidade no sistema.

Capítulo 8

Conclusão

O projeto surgiu com a ideia de utilizar as funções fprintf e fscanf para realizar todas as comunicações entre um aplicativo e o framework ROS, o que possibilitaria uma independência do código do módulo com o ROS e utilizaria o padrão POSIX como interface, o qual é amplamente conhecida e utilizada. Contudo foram identificados diversos problemas para realização desta ideia, que se resumem principalmente na grande diversidade de tipos de dados e a falta de uma interface capaz de abstraí-las. Resolvida essa questão, pode-se facilmente trabalhar sobre o aspecto da execução de uma tarefa, como divulgar a interface das tarefas, como monitorar seu estado, como conectar uma tarefa a um controle e outros, já que existe uma forma uniforme de comunicação.

Este trabalho apresenta uma prospecção de uma interface que permite manipular tanto essas estruturas provenientes de bibliotecas como YAML, JSON, XML, quanto como CSV, além de sistemas de arquivos. Utilizando isso como base, os *frameworks* são vistos como um conjunto de tópicos, parâmetros, formatos de mensagens, os quais tem um acesso semelhante a essas estruturas YAML e CSV. Portanto criado uma interface única para sistemas de arquivos, arquivos CSV e YAML, pode-se facilmente adaptado para a manipulação de dados de um *framework* de robótica.

Contudo criar uma interface única para diversos tipos de dados, que abstraia sua localização, seu modo de indexação, tipos de dados, tamanho do item, tipo do item , como acessar, já apresentam uma grande dificuldade e existe diversos caminhos. Contudo a maioria dos problemas consegue-se resolver através de uma generalização na passagem dos parâmetros, contudo a localização, tempo de vida e dados concorrentes criam difíceis barreiras para apresentar uma interface definitiva.

Duas importantes contribuições foram apresentadas neste trabalho, a primeira a releitura do modelo de Von Neumann e nomeada de modelo CDE possibilitou uma visão mais abrangente e recursiva do problema, já que um processo é uma unidade de execução, um robô é uma unidade de execução, um grupo de robôs também pode ser uma unidade de execução. A segunda foi a interface, a qual ainda tem problemas, mas resolvido os problemas pode apresentar um acesso uniforme de diferentes formatos de linguagem de marcação, de sistemas de arquivos, de bases de dados, *middlewares* e *frameworks* de robótica.

Como trabalho futuro, fica a resolução do problema da localidade e concorrência da abstração dos dados. Consolidado a abstração dos dados, pode-se desenvolver diversos *drivers* para as diferentes bibliotecas de linguagem de marcação, para o protocolo SSH, para *frameworks* e *middlewares*. Com a comunicação dos dados abstraída, logo fácil interligação entre os módulos, então pode-se começar o desenvolvimento da descrição do sistema integrador do *framework* URF.

Também é necessário criar ferramentas que permitam o fácil compartilhamento e instalação dos módulos e sistemas desenvolvidos.

Referências Bibliográficas

[Birrell e Nelson, 1984] Birrell, A. D. e Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59.

[Booch, 1986] Booch, G. (1986). Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221.

[Booth e Britten, 1949] Booth, A. D. e Britten, K. H. (1949). Principles and progress in the construction of high-speed digital computers. *The Quarterly Journal of Mechanics and Applied Mathematics*, 2(2):182–197.

[Bruyninckx, 2001] Bruyninckx, H. (2001). Open robot control software: the orocos project. Em *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, volume 3, páginas 2523–2528 vol.3.

[Coulouris et al., 2011] Coulouris, G., Dollimore, J., Kindberg, T. e Blair, G. (2011). *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition.

[Huang et al., 2010] Huang, A. S., Olson, E. e Moore, D. (2010). LCM: Lightweight communications and marshalling. Em *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Taipei*, páginas 4057–4062.

[Makarenko et al., 2007] Makarenko, A., Brooks, A. e Kaupp, T. (2007). On the Benefits of Making Robotic Software Frameworks Thin. Em *Communication*, páginas 163–168.

[Mallet et al., 2010] Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S. e Ingrand, F. (2010). Genom3: Building middleware-independent robotic components. Em *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, páginas 4627–4632.

[Marx, 1996] Marx, K. (1996). *O Capital - Tomo 1*. Editora Nova Cultural Ltda.

[Mavaddat e Parhami, 1988] Mavaddat, F. e Parhami, B. (1988). Urisc: The ultimate reduced instruction set computer. *International Journal of Electrical Engineering Education*, 25:327–334.

[Meng et al., 2017] Meng, Z., Wu, Z., Muvianto, C. e Gray, J. (2017). A data-oriented m2m messaging mechanism for industrial iot applications. *IEEE Internet of Things Journal*, 4(1):236–246.

[Neumann, 1945] Neumann, J. v. (1945). First draft of a report on the edvac. Relatório técnico.

[Omg, 2006] Omg (2006). Data-Distribution Service for Real-Time Systems (DDS).v1.2. Relatório técnico, OMG.

[Orocos, 2017] Orocos (2017). Orocos project history. `http://www.orocos.org/orocos/history`. Accessed: 2017-07-21.

[Pfeiffer et al., 2016] Pfeiffer, T., Hellmers, J., Schön, E. M. e Thomaschewski, J. (2016). Empowering user interfaces for industrie 4.0. *Proceedings of the IEEE*, 104(5):986–996.

[Quigley et al., 2009] Quigley, M., Conley, K., Gerkey, B., FAust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R. e Mg, A. (2009). ROS: an open-source Robot Operating System. *Icra*, 3(Figure 1):5.

[Rentsch, 1982] Rentsch, T. (1982). Object oriented programming. *SIGPLAN Not.*, 17(9):51–57.

[Sudkamp, 2005] Sudkamp, T. A. (2005). *Languages and Machines: An Introduction to the Theory of Computer Science (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Tanenbaum, 1996] Tanenbaum, A. S. (1996.). *Sistemas Operacionais Distribuídos*. Prendice Hall, México, 1. ed. edition.

[Tijero, 2012] Tijero, H. P. (2012). *Integrating a real-time Model in Configurable Middleware for Distributed Systems*.

[Turing, 1936] Turing, A. (1936). On computable numbers, with an application to the entscheidungproblem. *Proceedings of the London Mathematical Society*, 42:230–265.

[Turing, 1950] Turing, A. (1950). Computing machinery and intelligence. *Computing Machinery and Intelligence*, 59:433–460.