UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Adriano de Souza

FILTRAGEM EM DADOS DE FLUXO CONTÍNUO: UMA ABORDAGEM VOLTADA ÀS PRESTADORAS DE SERVIÇO TELEFÔNICO FIXO COMUTADO

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Carlos Becker Westphall, Dr.

FLORIANÓPOLIS, SETEMBRO DE 2002.

FILTRAGEM EM DADOS DE FLUXO CONTÍNUO: UMA ABORDAGEM VOLTADA ÀS PRESTADORAS DE SERVIÇO TELEFÔNICO FIXO COMUTADO

Adriano de Souza

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____

Arthur Ronald de Vallaris Buchsbau, Dr.

Banca Examinadora

_____

Carlos Becker Westphall, Dr.

_____

Carla Merkle Westphall, Dra.

_____

Carlos Aurelio Faria Rocha, Dr.

“Tudo posso Naquele que me fortalece.”

Filipenses 4:13

SUMÁRIO

RESUMO

O serviço de telecomunicações que, por meio de transmissão de voz e de outros sinais, destina-se à comunicação entre pontos fixos determinados, utilizando processos de telefonia denomina-se Serviço Telefônico Fixo Comutado (STFC). As Prestadoras do STFC cobram dos seus assinantes tarifas relativas aos serviços por eles utilizados. Desta forma, todos os serviços utilizados pelos assinantes devem ser registrados e processados, resultando um fluxo contínuo de dados com volume expressivo. Esses dados precisam ser filtrados antes de serem enviados para as diversas áreas dentro de uma Prestadora de STFC. Por exemplo, a área de faturamento deve receber apenas os registros faturáveis. Neste contexto, este trabalho apresenta uma proposta para a criação de funções de filtragem de dados e sua execução de forma eficiente.
Na primeira etapa é apresentada a definição de uma nova linguagem de programação chamada ICLang, voltada para a criação de filtros e transformações sobre dados. A segunda etapa define o interpretador da linguagem. Em seguida é apresentada uma forma de melhoria na eficiência e introduzidas novas funcionalidades na ICLang. Extendendo o contexto de filtragem de dados, são apresentados mecanismos para sua distribuição e introdução de processos envolvendo segurança.

Palavras-chave:
STFC, Bibliotecas Dinâmicas, CORBA

ABSTRACT

The service of telecommunications that, through voice transmission and other signals, is designed for communication between certain fixed points, using telephony processes is called Commuted Fixed Telephone Service (STFC). The companys of STFC charge from your subscribers relative tariffs to the services for them used. In this way, all the services used by the subscribers must be registered, resulting a continuous flow of data with expressive volume. Those data need to be filtered before being sent for the several areas inside of a company of STFC. For example the area of revenue should just receive billable registers. In this context, this work presents a proposal for the creating functions of filtering of data and its implementation efficiently. First stage shows the informal definition of a new programming language call ICLang, focused on the creation of filters and transformations on data. The second stage down the interpreter of the language. The third stage is presented a form of improvement in efficiency and introduced new features in ICLang. Extending the context, is presented in the fourth stage security and distribution.

Keywords:
STFC, Shared libraries, CORBA

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

ANATEL	: Agência Nacional de Telecomunicações
API	: Application Programming Interface
BOA	: Basic Object Adapter
CORBA	: Commom Object Request Broker Architecture
DTD	: Document Type Definition
GIOP	: General Inter-ORB Protocol
HTML	: Hypertext Markup Language
IDL	: Interface Definition Language
IIOP	: Internet Inter-ORB Protocol
IOR	: Interoperable Object Reference
MICO	: MICO is CORBA
OMG	: Object Management Group
ORB	: Object Request Broker
POA	: Portable Object Adapter
STFC	: Serviço Telefônico Fixo Comutado
XML	: Extensible Markup Language
XSL	: Extensible Stylesheet Language
XSLT	: XSL Transformations

1 INTRODUÇÃO

Os serviços de telecomunicações desempenham um papel importante no desenvolvimento da economia e da sociedade moderna. As prestadoras de serviços de telecomunicações capazes de absorver e satisfazer a demanda desses serviços de forma eficiente contribuem não só no desenvolvimento desse setor, mas promovem o desenvolvimento econômico e social.

É apresentada a seguir uma breve visão histórica sobre o desenrolar de alguns fatos relevantes associados ao Serviço Telefônico Fixo Comutado[1] (STFC). Esse conhecimento é a base para compreender a situação atual desse serviço, suas necessidades e problemas, a fim de apontar soluções e direcionamentos.

1.1 Motivação

Em julho de 1997 foi criada a Agência Nacional de Telecomunicações (ANATEL) com a função de órgão regulador das telecomunicações.

No ano de 1998 foram aprovados os planos de outorgas e as metas de qualidade e de universalização a serem cumpridas pelas entidades que prestariam o STFC, chamadas de Prestadoras de STFC. O Plano Geral de Outorgas, entre outros fatores, dividiu o território brasileiro em áreas, estabeleceu as regras para as concessões e definiu as modalidades do STFC destinado ao uso do público em geral em: serviço local, serviço de longa distância nacional e serviço de longa distância internacional.

As metas de qualidade definem um conjunto de indicadores que expressam, por exemplo, os níveis aceitáveis para a qualidade do serviço, para o atendimento à

[1] **Serviço Telefônico Fixo Comutado (STFC)** *(Resolução 85/98)* é o serviço de telecomunicações que, por meio de transmissão de voz e de outros sinais, destina-se à comunicação entre pontos fixos determinados, utilizando processos de telefonia.

solicitação de reparos, de erros na emissão de contas. As metas de universalização visam não somente aumentar a oferta do STFC aos consumidores economicamente rentáveis, mas disponibilizá-lo às pessoas, locais e instituições com pouco acesso ao serviço ou que não o possuem. O passo seguinte aconteceu em 29 de julho de 1998, quando foram privatizadas empresas que formavam o Sistema TELEBRÁS por meio da venda das ações de propriedade da União Federal, que, como acionista majoritária, exercia o controle dessas empresas por intermédio do Poder Executivo.

Desde então, atingir as metas de qualidade e universalização significa não somente cumprir as exigências vigentes em contrato. A antecipação de metas permite que as prestadoras dirijam-se a outras áreas e serviços. Dessa forma, a busca por novas concessões, lucratividade, competitividade, interesses estratégicos, defesa dos interesses do consumidor, está inserida num cenário em que mudanças não são raras. As prestadoras necessitam que os diversos setores, que compõem o seu negócio, estejam aptos a absorver eficientemente tais modificações. Para tanto, faz-se necessário que seus funcionários, equipamentos e *software* também estejam preparados.

Esta dissertação aborda a problemática do sistema de *software* responsável pelo processamento dos dados advindos das chamadas[2] geradas pelos clientes de uma prestadora. O processamento realizado sobre os dados pode incluir, não exclusivamente, padronizações, transformações e filtragens. As filtragens realizadas em cima desses dados não são fixas e cada vez mais se procuram *softwares* que facilitem a suas definições e atualizações.

De acordo com Hjálmtýsson et al. a idéia de adicionar novas funcionalidades a um programa rodando não é nova. A facilidade de carregamento dinâmico está presente na maioria dos sistemas operacionais e linguagens de programação. A linguagem JAVA, que se tornou extremamente popular, possibilita de forma fácil o acréscimo dinâmico de funcionalidades (FENG et al. 2001). Contudo, tais mecanismos baseados em JAVA

---

[2] Ação realizada pelo chamador a fim de obter comunicação com o equipamento terminal desejado. ANATEL, Glossário.

apresentam pouca eficiência e não se preocupam como o usuário de tal sistema criará as suas funções (filtros).

O ideal é a existência de um mecanismo que permita a criação das funções de filtragem de dados, oferecendo as facilidades de uma linguagem voltada ao conhecimento dos usuários, associado à eficiência de um código compilado.

Partindo do contexto descrito, essa dissertação visa propor um mecanismo que se preocupa basicamente com facilidade e eficiência. O usuário do sistema deve ter facilidade na criação de novas funções de filtragem de dados. Essas funções, quando executadas, devem ser feitas de forma eficiente.

Seguindo esses preceitos, são apresentadas uma linguagem para criação de filtros para dados de fluxo contínuo e algumas propostas para que o código gerado seja eficiente na aplicação desses filtros.

1.2 Objetivos do Trabalho

Este trabalho propõe mecanismos nos quais o usuário possa concentrar seus esforços no conhecimento do que deve fazer e não como deve fazer, propondo soluções nas quais ele possa especificar, de forma amigável, a função de filtragem sobre os dados. Para isso são objetivos desse trabalho:

- a criação de uma nova linguagem, mais fácil de ser aprendida e utilizada;
- a definição e a implementação de um interpretador capaz de aplicar os filtros definidos nessa linguagem;
- o estudo e a proposição de mecanismos para aumentar a eficiência do interpretador;
- propor um mecanismo mais abrangente, que possa ser utilizado por outras aplicações que manipulem um fluxo contínuo de dados, baseado na experiência adquirida;

- o estudo e a utilização de CORBA, bibliotecas compartilhadas e ferramentas XML.

1.3 Organização do Trabalho

No próximo capítulo são apresentadas a revisão da literatura e uma pequena revisão sobre assuntos utilizados no desenvolvimento desta dissertação. Entre os assuntos abordados estão CORBA, bibliotecas compartilhadas, XML e o sistema de bilhetagem.

No capítulo 3 é apresentada a proposta da dissertação. A proposta foi subdivida em etapas, que foram sendo cumpridas partindo-se da proposta inicial e baseadas nas conclusões da fase anterior. Assim, são apresentadas a definição de uma nova linguagem, a definição dos pontos mais relevantes da implementação do interpretador e propostas de melhoria na sua eficiência. Na última etapa, apresenta-se uma proposta mais abrangente utilizando CORBA e XML para criação de funções de manipulação de dados de fluxo contínuo.

2 REVISÃO DE LITERATURA

2.1 CORBA

CORBA é o acrônimo de Commom Object Request Broker Architecture e é o resultado de um consórcio de empresas chamado OMG (Object Management Group). O CORBA é uma arquitetura e infra-estrutura aberta, da qual as aplicações podem utilizar para trabalhar em cooperação através da rede.

Segundo a OMG um dos usos mais importantes do CORBA, e também mais freqüentes, está nos servidores que necessitam assegurar um grande número de clientes conectados com confiabilidade. O CORBA está por detrás de muitos dos maiores *websites* do mundo, mas versões especializadas rodam em sistemas de tempo real e pequenos *embedded systems*.

Linguagens de programação modernas empregam o paradigma de objetos para que uma determinada tarefa seja realizada dentro de um simples processo de um sistema operacional. O próximo passo lógico é distribuir a tarefa em múltiplos processos em uma ou em diferentes máquinas. Aplicações baseadas em CORBA são compostas de objetos; unidades individuais de software que combinam funcionalidades, dados e freqüentemente representam algo do mundo real. Duas aplicações baseadas em CORBA podem interoperar independentemente do sistema operacional, da linguagem de programação e rede nos quais residem (Cf. CORBA BASICS).

2.1.1 Conceitos

O ORB provê a comunicação entre clientes e objetos, ativando transparentemente esses objetos caso não estejam ativos quando há uma requisição destinada a eles. Um objeto CORBA é na verdade uma implementação feita numa

linguagem de programação seguindo certos preceitos do CORBA. Uma requisição é feita pelo cliente para um objeto alvo.

O cliente pode existir separado do objeto ou estar na mesma aplicação. Um servidor CORBA é uma aplicação onde residem um ou mais objetos.

Uma requisição é um pedido de execução de uma determinada operação, num objeto CORBA, por um cliente. Ela tem origem no cliente e é destinada ao objeto alvo no servidor. O objeto alvo, após executar a requisição, retorna os resultados requeridos. Uma referência de objeto é o que permite identificar e localizar um objeto CORBA.

Para poder fazer uma requisição para um objeto, o cliente precisa conhecer a interface oferecida por ele. Portanto, para cada objeto é necessário que seja definida uma interface. Segundo Orfali e Harkey (1997), a interface é escrita em Interface Definition Languague (IDL) e define os serviços fornecidos aos seus clientes em potencial. Qualquer cliente que deseje invocar uma operação de um objeto precisa usar essa interface para determinar a operação que deseja realizar e como repassar os argumentos. Por outro lado, quando a invocação chega ao objeto alvo, a mesma interface é utilizada para reconstruir os argumentos para que o objeto possa realizar a operação e enviar os resultados (Cf. CORBA BASICS).

As definições em IDL são independentes de linguagem de programação, mas são mapeadas para todas as linguagens de programação mais populares, como: C, C++, Java, Cobol, Smaltalk, Ada, Lisp, Python e IDLscript (Cf. OMG IDL).

A Figura 1, baseada na figura da OMG, mostra uma requisição. *Stubs* e *skeletons* servem como *proxies* tanto para clientes como para servidores. Através da interface definida em IDL, o *stub* do cliente pode comunicar-se perfeitamente com o *skeleton* do servidor, mesmo que os dois tenham sido compilados em linguagens de programação distintas ou estiverem rodando em ORBs de diferentes vendedores (Cf. ORB Basics).

Cada instância de objeto tem a sua própria referência, capaz de identificá-lo unicamente dentro do ambiente distribuído. Os clientes fazem uso dessas referências, repassando-as ao ORB, identificando assim a instância exata de suas invocações.

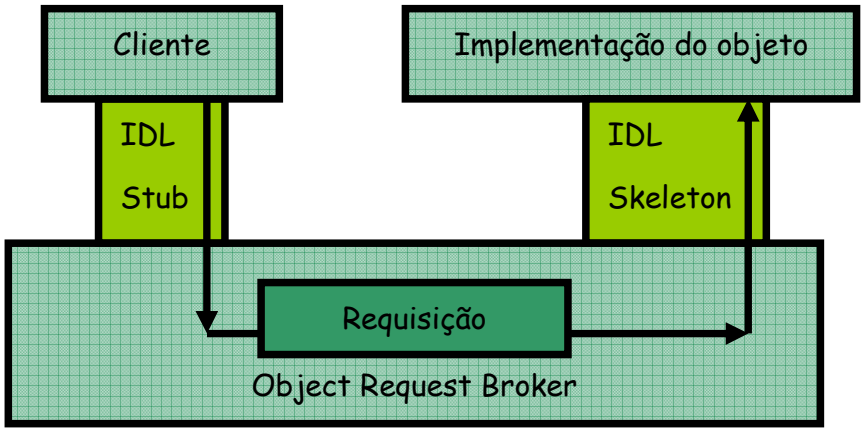

Figura 1 – Fluxo de uma requisição

Conforme Henning e Vinoski (1999), os adaptadores de objetos conectam as implementações dos objetos ao ORB, permitindo ao objeto receber requisições. Até a versão 2.1, o CORBA possuía apenas a especificação do Basic Object Adapter (BOA). O BOA foi o adaptador de objetos original do CORBA, mas devido a alguns problemas na sua especificação foi removido do CORBA para dar lugar ao Portable Object Adapter (POA), introduzido na versão 2.2.

Na versão 2.0 foi introduzida uma arquitetura de interoperabilidade entre os ORBs chamada de General Inter-ORB Protocol (GIOP). GIOP é um protocolo que especifica a sintaxe para transferência e padrões nos formatos de mensagens que permitem que ORBs diferentes comuniquem-se. O Internet Inter-ORB Protocol (IIOP) especifica como o GIOP é implementado sobre TCP/IP.

Para Henning e Vinoski (1999), a interoperabilidade entre ORBs também requer a padronização das referências ao objeto. Este padrão é denominado Interoperable Object Reference (IOR). Um IOR contém o nome do host, uma porta TCP/IP e um identificador do objeto que o identifica unicamente dentro do mesmo host e porta. Os principais mecanismos que podem ser utilizados, em CORBA, para tornar as referências

dos objetos disponíveis, são os seguintes:

- Interoperable Object Reference (IOR) transformadas em strings: Os ORBs podem transformar uma IOR em um string, que pode ser armazenada em um recurso compartilhado como, por exemplo, um arquivo. Clientes podem obter a referência lendo esse arquivo.

- *Naming Service* e outros serviços CORBA:
  CORBA fornece serviços que controlam as referências dos objetos. Através desses serviços um cliente pode obter referência ao objeto.

As duas implementações CORBA utilizadas na dissertação foram MICO (Cf. MICO is CORBA) com mapeamento para a linguagem C++ e DORB para a linguagem Delphi (Cf. DORB).

MICO é implementado em C++, compatível como padrão CORBA 2.3.

O projeto MICO é o protótipo para o projeto DORB. Dessa forma, o DORB, é basicamente uma tradução do MICO C++ para Delphi.

2.2 Bibliotecas Compartilhadas

Segundo Cobb et al., no UNIX, não raramente a compilação é feita utilizando-se da técnica de linkagem estática. Com essa técnica, múltiplos arquivos de objeto, que definem símbolos globais e contêm códigos e dados, são combinados e escritos num único arquivo executável com todas as referências solucionadas. Dessa forma, o executável é um único arquivo auto-suficiente. A linkagem estática ainda é usada em várias circunstâncias, mas tem as seguintes desvantagens:

- Se qualquer uma das bibliotecas usadas pelo programa é atualizada, o programa precisa ser novamente linkeditado para tirar proveito das bibliotecas atualizadas.

- A má utilização do espaço em disco, porque todo programa no sistema contém cópias privadas da função de biblioteca de que precisa.
- A má utilização de memória, porque todos os processos presentes na memória contêm sua própria cópia privada das mesmas funções de biblioteca.
- Degradação do desempenho porque os executáveis são maiores do que precisam, podendo causar maior paginação.

De acordo com Cobb et al., permitir compartilhar as bibliotecas entre programas diferentes é o conceito de bibliotecas compartilhadas. Quando um programa é linkeditado com uma biblioteca compartilhada, o código da biblioteca compartilhada não é incluído no programa gerado. Ao invés disso, uma informação é armazenada dentro do programa, de forma que permita encontrar a biblioteca e carregá-la quando o programa é executado. O código da biblioteca compartilhado é carregado na memória global do sistema pelo primeiro programa que a utiliza e é compartilhado por todos os programas que necessitem.

Há quatro vantagens principais no uso de bibliotecas compartilhadas:

1. Melhor utilização de espaço em disco, porque o código de biblioteca compartilhado não é incluído dentro programas executáveis.
2. Melhor utilização de memória, porque o código de biblioteca compartilhado só é carregado uma vez.
3. O tempo de carregamento pode ser reduzido se a biblioteca compartilhada já foi carregada por outro programa.
4. Clientes podem obter resolução mais rápida de *bugs*, pois podem carregar bibliotecas atualizadas da *Web* e podem reiniciar a aplicação, em vez de esperar pelos novos executáveis.

Quando um programa é linkeditado com bibliotecas compartilhadas, o executável contém uma lista de bibliotecas compartilhadas que são necessárias pelo programa e uma lista de símbolos importadas por ele. O código atual não é incluído.

Então, se um das bibliotecas compartilhadas é atualizada, os programas usando aquela biblioteca não precisam ser re-linkeditados, pois terão a versão atual da biblioteca automaticamente na próxima execução. Isto permite que novas versões de bibliotecas que contêm correções possam ser disponibilizadas sem a necessidade de uma nova versão inteira da aplicação.

O carregamento dinâmico permite que uma aplicação carregue um novo módulo em seu espaço de endereço, seja para adicionar uma nova funcionalidade ou implementar uma interface existente, sem a necessidade de linkagem com uma biblioteca. Isto permite a chamada de funções contidas dentro de bibliotecas compartilhadas.

As funções em C/C++ que implementam o carregamento dinâmico são:

1. void *dlopen (const char *filename, int flag);

Essa função procura o módulo especificado no espaço de endereço do processo e retorna um manipulador para o módulo carregado. Se o módulo já estiver carregado não será novamente carregado, mas um manipulador igual é devolvido. O manipulador é usado para chamadas subseqüentes para dlsym() e dlclose().

2. void *dlsym(void *handle, char *symbol);

Essa função procura o símbolo especificado no módulo carregado e retorna seu endereço.

3. int dlclose (void *handle);

Essa função é usada para fechar acesso a um módulo carregado com dlopen ().

4. const char *dlerror(void);

Essa função é usada para obter informação sobre o último erro que aconteceu numa chamada a uma das funções acima especificadas.

2.3 XML

XML é um conjunto de regras aplicáveis sobre um conjunto de dados texto, de forma a deixá-los estruturados, mas XML não é uma linguagem de programação. Com XML pode-se gerar dados, lê-los e assegurar que a estrutura deles não é ambígua. (Cf. W3.XML in 10 points)

Assim como em HTML, XML faz uso de *tags* e atributos. Dessa forma as palavras ficam entre ' < ' e ' > ' e os atributos seguem a forma nome = "valor". HTML especifica *tags* e atributos indicando como o texto deve ser apresentado pelo *browser*. XML utiliza *tags* para delimitar os dados e deixa a interpretação por conta da aplicação que lê os dados. Um "<p>" num arquivo XML, não pode ser assumido com um parágrafo. Dependendo do contexto, pode ser um preço, um parâmetro, uma pessoa.

Uma vantagem de um formato de texto é que permite as pessoas, se necessário, olharem para os dados sem o programa que os produziu, isto é, ler o arquivo no formato de texto com o editor de texto favorito. Arquivos em formato de texto também permitem depurar aplicações mais facilmente. Diferentemente de HTML, as regras para arquivos XML são rígidas. Um *tag* esquecido ou um atributo sem aspas torna um arquivo XML inutilizável. Quando um arquivo XML contiver um erro a aplicação tem que parar e informar o erro.

Levando em conta que os arquivos XML estão em formato texto com *tags* para delimitar os dados, arquivos XML quase sempre são maiores que os equivalentes em formato binário.

XML 1.0 é uma especificação que define como *tags* e atributos são usados. Além da especificação XML 1.0, existe um conjunto crescente de módulos que oferecem serviços úteis para realizar tarefas freqüentes. Entre estas ferramentas o XSLT, usado para arranjar, adicionar ou apagar tags e atributos.

Desta forma, XML permite criar uma linguagem de marcação, com elementos e atributos que melhor se ajustam à informação que se deseja manter. O processo de

definição formal de uma linguagem em XML é chamado de modelagem de documentos. Uma das formas de modelar um documento é utilizando-se de DTD (Definições de Tipo de Documento) que descrevem a estrutura de um documento de forma declarativa. O modelo de documento determina quais documentos estão de acordo com a linguagem.

2.4 Chamada Telefônica

As prestadoras de STFC vivem da exploração dos serviços dos quais possuem concessão, sejam eles na modalidade de serviço local, serviço de longa distância nacional ou serviço de longa distância internacional. Independente da modalidade, o foco desta dissertação está na aplicação de filtros sobre o conjunto de informações que constituem as chamadas telefônicas.

Uma chamada telefônica é definida, segundo a ANATEL (Glossário Termos Técnicos de Telecomunicações), como uma ação realizada pelo chamador a fim de obter comunicação com o equipamento terminal desejado. Chamador é quem origina a chamada e é também conhecido como Assinante A ou Originador. Quem recebe a chamada é conhecido como Chamado, Asssinante B ou Destino.

Segundo Luca (2002), para que as prestadoras possam cobrar dos seus usuários essas chamadas, é necessário registrar os dados das chamadas telefônicas, afim de permitir que posteriormente seja feito o cálculo dos custos das chamada e emissão de contas. O responsável por registrar todas as informações necessárias chama-se sistema de bilhetagem. As centrais telefônicas, além de servir como nó de comutação de chamadas, também podem executar a função de bilhetagem, mas nem todas executam esta função. As centrais com função de bilhetagem serão chamadas de centrais bilhetadoras.

Algumas das informações que uma central bilhetadora precisa registrar:

- quem originou a chamada (Assinante A);
- quem é o destino da chamada (Assinante B);

- qual é a data da chamada (Data Início);
- qual é a hora de início da chamada (Hora Início);
- qual a duração da chamada (Duração);
- o completamento ou não da chamada (Fim de seleção).

A central registra essas e outras informações da chamada num registro conhecido como bilhete ou CDR (Call Detail Recording). A central bilhetadora armazena então este CDR juntamente com outros numa memória de massa. É este conjunto de CDR que constitue um arquivo e que deve ser filtrado, dependendo das informações nele contidas.

Este mesmo arquivo pode alimentar diferentes sistemas dentro de uma prestadora, mas geralmente contendo apenas informações úteis ao sistema em questão. Deste modo, é indesejável à área de faturamento receber bilhetes que não são faturáveis, provocando um processamento desnecessário. Os bilhetes não faturáveis, por exemplo, por congestionamento na rede, são extremamente úteis para área de rede detectar os motivos e horários de maior congestionamento, enquanto que os faturáveis não possuem muita utilidade. Daí a importância da filtragem de bilhetes evitando o processamento de informação desnecessária.

3 A PROPOSTA

A proposta está dividida em etapas ordenadas cronologicamente. Cada etapa possui os seus próprios objetivos e peculiaridades, mas seu inter-relacionamento se torna evidente no final e no começo de cada etapa. São apresentados apenas os aspectos considerados mais relevantes de cada uma.

Na primeira etapa é apresentada a definição informal de uma nova linguagem de programação, chamada ICLang. Esta linguagem é voltada para a criação de filtros e transformações sobre dados, tendo como pontos fundamentais à legibilidade e à redigibilidade.

A segunda etapa define o interpretador da linguagem. O interpretador não utiliza diretamente a ICLang como código fonte, mas sim uma linguagem intermediária. Essa linguagem intermediária é apresentada, assim como alguns detalhes referentes à implementação do interpretador.

Com o objetivo de tornar o interpretador mais eficiente e a linguagem criada mais funcional, são apresentados na terceira etapa, uma forma de melhoria na eficiência e introduzidos novas funcionalidades na ICLang.

Em busca de uma abrangência maior, é apresentado na quarta fase um mecanismos para aumentar ainda mais a eficiência da filtragem bem como a sua distribuição e introdução de mecanismos de segurança.

Nos itens seguintes cada uma das etapas será descrita detalhadamente.

3.1 Definição da linguagem ICLang

As linguagens de programação pressupõem, na implementação de um programa, além do domínio do problema, um conhecimento prévio da linguagem escolhida para implementação do programa. É através deste conhecimento específico, que o usuário, a

partir do conhecimento do problema, poderá criar um programa que representa a realidade. Um obstáculo encontrado é que mesmo a escrita de um programa simples requer uma série de conhecimentos especializados por parte do usuário que muitas vezes não tem tempo, interesse, aptidão ou disposição para adquirir.

A linguagem proposta, chamada de ICLang, tem como pontos fundamentais de projeto: ser de propósito específico e voltada ao conhecimento do usuário do domínio do problema e facilitar tanto a sua redigibilidade como a legibilidade.

A ICLang não é uma linguagem de propósito geral. Ao contrário, ela é bastante específica e voltada essencialmente para filtragem de dados.

O usuário alvo da ICLang é quem detém os conhecimentos do domínio do problema. É ele que conhece o formato dos arquivos advindos das centrais bilhetadoras, os valores aceitáveis e o significado de cada elemento de informação, bem como, os filtros que devem ser aplicados sobre os dados, para que estes possam ser enviados para os sistemas destinos. É desejável que a linguagem aproxime-se o máximo da linguagem natural na qual o usuário expressaria a aplicação de um filtro. Essa é uma característica importante não apenas para que o usuário tenha facilidade em escrever um programa, mas também que ele possa ser facilmente compreendido. Isto inclui funcionários que pertençam a outras áreas ou hierarquias, como o seu chefe, técnico da central e funcionários responsáveis por outros sistemas.

A ICLang utiliza poucas construções, que terão basicamente a função de avaliação de um determinado conjunto de dados, previamente conhecido. Dessa forma, na avaliação de um CDR, uma função de filtragem invariavelmente retornará um valor lógico verdadeiro ou falso, indicando se o CDR deve ou não ser filtrado, como mostrado na Figura 2. Caso o CDR não deva continuar o processo, uma mensagem de erro deve ser retornada, explicitando o motivo pelo qual o CDR foi filtrado.

A ICLang é uma linguagem declarativa, pois diz o que fazer e não como fazer. Um exemplo disso pode ser na comparação da informação contida num CDR com uma constante. O usuário não se preocupa como ou onde está armazenado este valor nem como será a feita a comparação.

ValorDoCampo('Duracao') > Hora(000003)

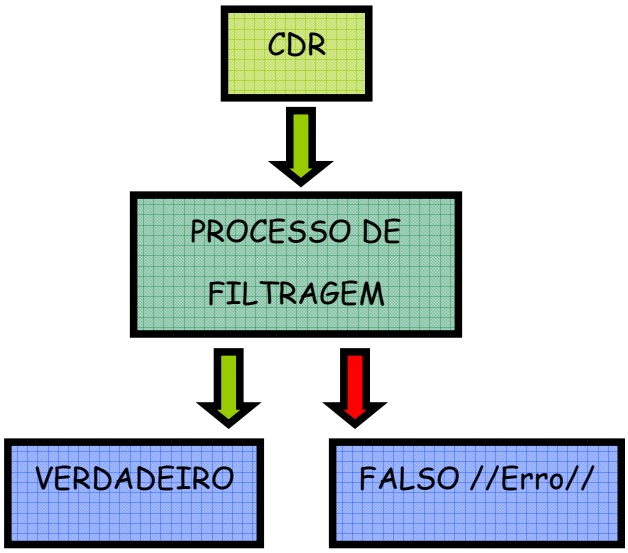

Figura 2 - Aplicação de um filtro sobre um CDR

Um programa consiste essencialmente de duas partes, a descrição dos dados que serão manipulados e a descrição das ações que serão realizadas sobre os dados. Os dados são representados por valores contidos nos campos dos CDRs ou em constantes e as ações serão definidas pela combinação de funções e operadores.

O alfabeto da ICLang consiste de símbolos básicos classificados em letras, dígitos e símbolos especiais.

letras = [a .. z] | [A .. Z]

dígitos = [0 .. 9]

símbolos especiais= "=" | "!" | "<" | ">" | "(" | ")" | "," | "/"

Palavras usadas para representar os símbolos especiais são palavras reservadas e não podem ser usadas para outro propósito. Na ICLang letras maiúsculas e minúsculas são consideradas iguais.

3.1.1 Tipos de dados

Os tipos de dados determinam o conjunto de valores que um campo ou constante podem assumir e o conjunto de operações que podem ser aplicados sobre eles. Os tipos definidos na ICLang são:

a) Data

Campos do tipo Data representam datas válidas no formato AAAAMMDD. Utilizado para datas como Data de início da chamada.

b) Hora

Campos do tipo Hora representam horas válidas no formato HHMMSS. Utilizado para horas como hora de início e fim da chamada.

c) Texto

Campos do tipo Texto representam qualquer caracter ASCII. A sua ordem é dada pela ordem dos caracteres ASCII em que letras e dígitos são ordenados de forma semelhante à ordem alfabética e numérica. Pode representar o nome de um campo ou um valor de um campo.

d) Numérico

Campos numéricos representam números inteiros sem sinal. Utilizado para representar valores discretos, podendo representar códigos para diversos campos de informação, como um código de área, código de assinante.

3.1.2 Declaração de Campos e Variáveis

A declaração dos campos juntamente com o seu tamanho, identificador, localização dentro de um CDR é feita externamente à linguagem. Essas informações sobre os campos, juntamente com as descrições dos erros são dados externos que precisam ser repassados para que um programa possa verificar os aspectos léxicos, sintáticos e semânticos da linguagem.

Qualquer variável utilizada dentro do programa precisa ser declarada como um campo. Abaixo são descritos alguns campos com seu tipo, tamanho e a informação de um CDR que serão utilizados nos exemplos:

'Assinante A', Texto, 18, '014492784121'
'Data de Início', Data, 8, '05092002'
'Duração', Hora, 6, '000045'
'Fim de Seleção', Numérico, 2, '01'

3.1.3 Funções de Manipulação de Campos

As funções de manipulação de campos permitem que usuário retorne os dados contidos num determinado campo, escolha partes dessa informação ou calcule a quantidade de caracteres que compõem o campo. Em todas as funções leva-se em consideração que a informação contida nos campos já está sem o caracter de preenchimento. As funções de manipulação são as seguintes:

a) ValorDoCampo

A função *ValorDoCampo* retorna as informações contidas no campo especificado pelo usuário através do seu nome. O tipo retornado pela função está associado ao tipo definido para o campo utilizado como parâmetro da função.

Exemplos:

ValorDoCampo('Assinante A'): '014492784121', Texto

ValorDoCampo('Duração'): '000045', Hora

ValorDoCampo('Data da chamada'): '05092002', Data

b) SubCampo

Com a função SubCampo é possível retornar porções da informação contida no campo especificado. É necessário o nome do campo, o índice e o tamanho. Campos do tipo TEXTO e NUMÉRICO retornam o mesmo tipo. Campos do tipo HORA e DATA retornam tipo NUMÉRICO.

Exemplos:

SubCampo('Assinante A',2,5): '14492', Texto

SubCampo('Duração',1,4): '0000', Numérico

c) RestoDoCampo

A função RestoDoCampo retorna a porção de um campo a partir de uma posição até o fim do campo especificado. Campos do tipo TEXTO e NUMÉRICO retornam o mesmo tipo. Campos do tipo HORA e DATA retornam tipo NUMÉRICO.

Exemplos:

RestoDoCampo('Assinante A',7) = '2784121', Texto

RestoDoCampo('Duração',5) = '00', Numérico

d) TamanhoDoCampo

A função TamanhoDoCampo retorna o tipo NUMÉRICO que é a quantidade de caracteres do campo especificado.

Exemplos:

TamanhoDoCampo('Assinante A'): '12', Numérico

TamanhoDoCampo('Duração'): '6', Numérico

TamanhoDoCampo('Data da chamada'): '8', Numérico

3.1.4 Expressão

Expressões em ICLang são regras que retornam valores lógicos. Elas consistem de um ou mais operandos (valores de campos ou constantes) combinadas através de operadores ou através do retorno de funções lógicas.

Na IcLang, não existe a noção de precedência de operadores. A precedência de operadores precisa ser explicitada pelo usuário através do uso de parênteses.

3.1.4.1 Operadores Relacionais

Operadores relacionais =, !=, <, <=, >, >=, sempre tem um retorno lógico. A sua utilização só é definida quando ambos os operandos são do tipo Data, Hora, Numérico ou Texto.

A forma de comparação é determinada pelo tipo dos operandos. Dessa forma o resultado da comparação de duas datas, duas horas ou dois numéricos será data pela ordem numérica dos valores envolvidos. No caso da comparação de operandos do tipo Texto o resultado será dependente da ordem dada pelo conjunto de caracteres ASCII. Não é definida nenhuma operação relacional com valores lógicos.

Exemplos:

TamanhoDoCampo('Assinante A') != 0

ValorDoCampo('Duração') > Hora(000003)

ValorDoCampo('Data da chamada') = Data(05092002)

3.1.4.2 Operadores Lógicos

Os operadores Lógicos E e OU sempre tem um retorno lógico e tem como operando dois valores também lógicos. Não é definida nenhuma precedência entre estes dois operadores. Qualquer precedência deve ser explicitada através do uso de parênteses.

Exemplos:

TamanhoDoCampo('Assinante A') != 0 E ValorDoCampo('Duração') > Hora(000003)

SubCampo('Assinante A',1,3) = '014' OU SubCampo('Assinante A',1,3) = '021'

3.1.4.3 Constantes

Constantes são os casos particulares de valores que os tipos básicos podem assumir:

a) Constante do tipo DATA

Essa data precisa ser válida e estar no formato AAAAMMDD.

Exemplos:

Data(20020907) – 7 de setembro de 2002.

Data(20020101) – 1 de janeiro de 2001.

b) Constante do tipo HORA

Essa hora precisa ser uma hora válida no formato HHMMSS. Pode representar uma hora ou uma duração.

Exemplos:

Hora(000003) – 0 horas, 0 minutos e 3 segundos.

Hora(182703) – 18 horas, 27 minutos e 03 segundos.

c) Constante do tipo NUMÉRICO

Deve ser um número inteiro sem sinal.

Exemplos:

02 - o valor 2

123423 - o valor 123423

d) Constante do tipo TEXTO

Qualquer caracter entre aspas, exceto aspas.

Exemplos:

'Assinante A' – nome de um campo contido dentro de um CDR.

'002' – texto 002

e) Constante do tipo LÓGICA

Uma constante desse tipo pode assumir somente dois valores distintos pré-definidos: *VERDADEIRO* ou *FALSO*. A constante *VERDADEIRO* indica que o CDR está de acordo com as especificações do usuário, devendo seguir o resto do processamento. A constante *FALSO* indica que o CDR não está de acordo com as especificações do usuário. Nesse caso uma mensagem indicando o não cumprimento, deve vir logo depois entre os caracteres '//'.

Exemplos:

VERDADEIRO

FALSO //'Duração igual a zero'//

```
FALSO //'Código inválido'//
```

3.1.4.4 Funções Lógicas

As funções lógicas retornam valores lógicos, isto é, são capazes de gerar valores Verdadeiro ou Falso.

3.1.4.4.1 Função Lógica Se

A função Se especifica a escolha de execução entre duas expressões, com base no valor gerado pela expressão de condição. A função *Se* sempre tem uma cláusula *Senao* associada, não existindo a forma sem o *Senao*.

Exemplo:

```
Se (ValorDoCampo('Duracao') > Hora(000003)) Entao
        VERDADEIRO
Entao
        FALSO//'Duração inferior a 3 segundos'//
```

3.1.4.4.2 Função Lógica MudeCampo

A função *MudeCampo* permite que um campo tenha o seu valor modificado de acordo com a necessidades do usuário. Podem ser utilizados constantes ou então valores advindo de outros campos, separados por vírgulas.

Exemplo:

```
Se (ValorDoCampo('Fim de Seleção') = 1 Entao
        MudeCampo('Status Da Chamada','Chamada Completada')
Senao
        MudeCampo('Status da Chamada', 'Chamada não Completada')
```

3.1.4.4.3 Função Lógica Pertence

A função Pertence permite ao usuário verificar se um determinado valor está contido em um determinado conjunto de dados. As estruturas que definem o domínio são Intervalo, Conjunto e ConjuntoDaTabela .

a) Intervalo

Um intervalo é definido com um par de valores, sendo o primeiro menor que o segundo. Utilizar o intervalo numa função Pertence, significa verificar se um determinado valor está contido no intervalo definido. A definição dos pares está associada ao tipo do valor pesquisado.

Exemplo:

```
Se Pertence(ValorDoCampo('Fim de Seleção'),Intervalo(1,2)) Entao
        VERDADEIRO
Senao
        FALSO //'Fim de seleção inválido'//
```

b) Conjunto

Um conjunto define uma quantidade finita de elementos que serão comparados um a um para ver se o valor está contido nesse conjunto. Assim como nos intervalos os

elementos que formam o conjunto seguem as imposições definidas pelo tipo do valor pesquisado. Sendo assim, se o tipo do campo a ser consultado é do tipo numérico, apenas constantes númericas serão aceitas como elementos constituintes do conjunto. Intervalos também podem ser elementos de um conjunto.

Exemplos:

```
Se Pertence(ValorDoCampo('Fim de Seleção'),Conjunto(Intervalo(1,2))) Entao
        VERDADEIRO
Senao
        FALSO //'Fim de seleção inválido'//
```

```
Se Pertence(ValorDoCampo('Fim de Seleção'),Conjunto(Intervalo(1,2), 7, 10)) Entao
        VERDADEIRO
Senao
        FALSO//Fim de seleção inválido//
```

c) ConjuntoDaTabela

Um conjunto da tabela permite verificar se um determinado valor pertence a uma coluna de uma tabela previamente definida pelo usuário. É necessário definir o nome da tabela e nome da coluna onde estão os dados que devem ser comparados. Construções como essas permitem que regras sejam modificadas com uma simples retirada/inserção de dados da uma determinada tabela. Se fosse necessário, por exemplo, filtrar os números de telefone de um conjunto de usuários que originaram a chamada, bastaria criar uma tabela com estes números e utilizar a expressão abaixo. Considerando que a tabela seja chamada de assinantes e a coluna num_fil:

```
Se Pertence(ValorDoCampo('Assinante A'), ConjuntoDaTabela('assinantes', 'num_fil') Entao
        VERDADEIRO
Senao
        FALSO//Assinante inválido//
```

3.1.4.4.4 Função Lógica Não

O operador Não nega um valor lógico.

Exemplo:

```
Se Não(Pertence(ValorDoCampo('Fim de Seleção'),Conjunto(Intervalo(1,2), 7, 10))) Entao
        FALSO//Fim de seleção inválido//
Senao
        VERDADEIRO
```

3.1.5 Forma de integração

A utilização da ICLang pressupõe que a aplicação que irá se utilizar dela, para fazer a aplicação de filtros sobre os CDRs, tenha o conhecimento de como ler um CDR e como ler as informações que o compõem. Isto porque centrais bilhetadoras de diferentes fabricantes podem armazenar as informações de forma e ordem diferente. Para isso, é necessário que essa aplicação permita a definição dos campos que compõem um CDR.

As informações dos campos que compõem um CDR incluem posição de leitura no arquivo advindo da central, o nome dado ao campo, seu tipo e tamanho. O nome do campo é nome especificado nas expressões que fazem referência ao campo como ValorDoCampo e RestoDoCampo. O tipo define quais os valores permitidos e como serão

efetuadas as operações e funções. É também na definição dos campos que será associada a expressão de filtragem daquele campo.

A tabela 1 mostra um exemplo de como poderia ser feita a definição dos campos e sua expressão associada.

Para aplicar a filtragem sobre um arquivo de CDRs, um CDR é lido segundo as definições de seus campos. Então, são aplicadas sobre ele as expressões de filtros, caso existam, seguindo a ordem definida pelos campos. No exemplo do campo Duração, se a avaliação de um CDR retornar FALSO, a mensagem de erro 'Duração inferior a 3 segundos' será associada a este campo, para posterior contabilização de quantos CDRs foram filtrados, quais os motivos da filtragem e a que campos estavam associados.

Filtro para o Sistema de Faturamento			
Nome	Tipo	Posição	Expressão de Filtro Associada ao Campo
Assinante A	Texto	1 –18	-
Assinante B	Texto	19-36	-
Data de Início	Data	37-44	-
Fim de seleção	Numérico	45-46	Se Pertence(ValorDoCampo('Fim de Seleçao'), Intervalo(1,3) Entao VERDADEIRO Else FALSO//'Fim de Seleção Inválido'//
Duração	Hora	47-52	Se (ValorDoCampo('Duracao') > Hora(000003)) Entao VERDADEIRO Else FALSO//'Duração inferior a 3 segundos'//

Tabela 1 – Exemplo de declaração dos campos de um CDR

As mensagens de erro também devem ter sido anteriormente definidas pela aplicação. A mensagem de erro utilizada na expressão do campo Duração, por exemplo, deveria ter sido definida na aplicação, numa interface de definição de Mensagens de Erro.

Da mesma forma, todas as tabelas, juntamente com sua estrutura, devem ter sido previamente definidas para que possam ser utilizadas.

Para cada sistema que se deseje filtrar os dados é necessário uma definição semelhante a da Tabela 1.

3.1.6 Conclusão

A linguagem criada é bastante simples, o que permite sua legibilidade e também a facilidade em escrever uma expressão. Como contém poucas construções, a linguagem é facilmente aprendida pelo usuário.

A facilidade de busca em tabelas permitem que expressões não precisem ser modificadas, bastando alterar apenas o conteúdo das tabelas.

Por ser simples a linguagem também não apresenta operações matemáticas que podem ser necessárias em algumas situações, assim como operações de coersão se tipos, por exemplo, para transformar um campo do tipo numérico para texto e vice-versa.

As construções não suportadas pela linguagem devem ser tratadas externamente, por algum mecanismo oferecido pela aplicação. A linguagem também não oferece facilidades para que possa ser expandida.

3.2 Definição e Implementação do Interpretador

Nessa etapa foi definido e implementado o interpretador para a ICLang. O interpretador não utilizará a mesma linguagem para analisar um CDR , mas sim uma linguagem intermediária, permitindo que o interpretador seja mais eficiente.

A implementação do interpretador leva em conta, que o programa fonte não está mais descrito conforme a especificação da linguagem ICLang. Isso quer dizer que um programa tradutor, recebe um programa em IcLang e a transforma em uma linguagem intermediária que é então interpretada pelo interpretador. A cada novo CDR, o código fonte é lido e são realizadas as operações sobre os dados do CDR. Considera-se também que todos os aspectos léxicos, sintáticos e semânticos já foram resolvidos anteriormente, antes de terem sido convertidos para a linguagem intermediária.

3.2.1 Linguagem Intermediária

Como para cada CDR a expressão é novamente executada, todas as funções e operações são definidas com apenas um caractere. Assim:

'!' = !=	'A' = ConjuntoDaTabela	'N' = Não
'#' = <=	'B' = ConjuntoDaTabelaNCol	'O' = Ou
'(' = (	'C' = Conjunto	'P' = Pertence
')' =)	'D' = Data	'R' = RestoDoCampo
',' = ,	'E' = E	'S' = SubCampo
'<' = <	'F' = Falso	'T' = TamanhoDoCampo
'>' = >	'H' = Hora	'V' = ValorDoCampo
'=' = =	'I' = Intervalo	'Y' = Verdadeiro
'@' = >=	'K' = Se	'Z' = Mensagem de Erro
'[' = Senão	'M' = MudeCampo	

O código fonte a ser interpretado segue a Notação Polonesa, proposta por Jan Lukasiewicz. Nela os operando precedem os operadores, não sendo necessário a

utilização de parênteses para identificar a ordem de precedência. Letras maiúsculas e minúsculas são diferentes. Os paramêtros das funções permanecem inalterados.

Dessa forma o seguinte código em ICLang:

```
ValorDoCampo('Duração') > Hora(000003)
```

é escrito da seguinte forma, utilizando a Notação Polonesa:

```
> ValorDoCampo('Duração')  Hora(000003)
```

Aplicando-se agora apenas uma letra para identificar as funções, conforme a tabela acima:

```
> V('Duração') H(000003)
```

Para a função Se foi inserida uma modificação para facilitar o caso em que a expressão da condição retornar o valor FALSO e for necessário um pulo para o Senão. Para tornar a localização mais eficiente, a expressão de um Senão estará entre colchetes.

A constante FALSO //'Mensagem do erro'// foi subdividida em duas partes. A primeira que identifica a constante e a segunda que identifica a mensagem de erro.

Em ICLang:

```
Se ValorDoCampo('Duração') > Hora(000003) Então
        VERDADEIRO
Senão
        FALSO //'Duração Inválida'//
```

No código fonte do interpretador:

K > V ('Duração') Hora(000003) Y [F M(12)]

No caso acima, 12 é o código do erro associada a mensagem 'Duração Inválida'.

3.2.2 Interpretador

O interpretador tem como função aplicar a expressão sobre um conjunto de dados que formam o CDR, retornando um valor lógico e indicando se o CDR deve continuar o processo ou se contém algum erro explicitado pelo código de erro.

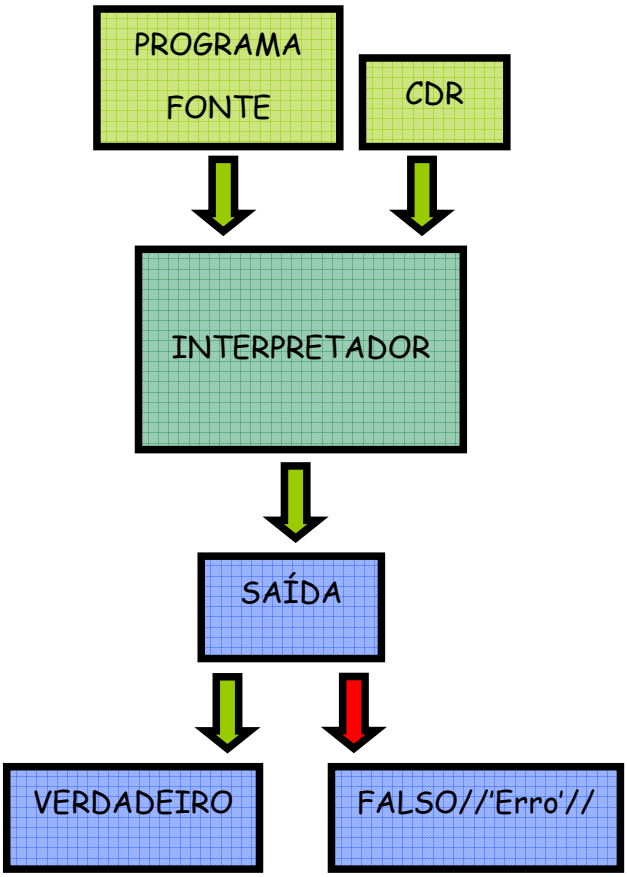

Figura 3 - Funcionamento do Interpretador

O interpretador possui internamente outros objetos, responsáveis por outras funções específicas e que colaboram para a interpretação da expressão. Dessa forma, interpretador é composto por um leitor responsável pela leitura da expressão retornando os *tokens* e um executor, responsável por executar as funções, empilhar operandos e operadores.

Basicamente o interpretador pede para o leitor retornar um *token*. Este *token* é reconhecido pelo interpretador, que verifica, conforme o tipo do *token*, se é necessário buscar parâmetros. Com todos os dados necessários, o interpretador repassa-os ao executor invocando o método adequado. Dessa forma se o interpretador reconhece um *token* como o operador relacional igual, ele invoca o método de empilhar operador relacional do executor. Todo esse processo é repetido até que se chegue ao fim da expressão ou um valor de retorno seja encontrado.

3.2.3 Leitor

A principal função do leitor é retornar um *token*, juntamente com o seu tipo. A expressão é guardada na forma original, ou seja, em notação polonesa e percorrida conforme requisições do interpretador através do método:

```
char Leitor::leiaToken();
```

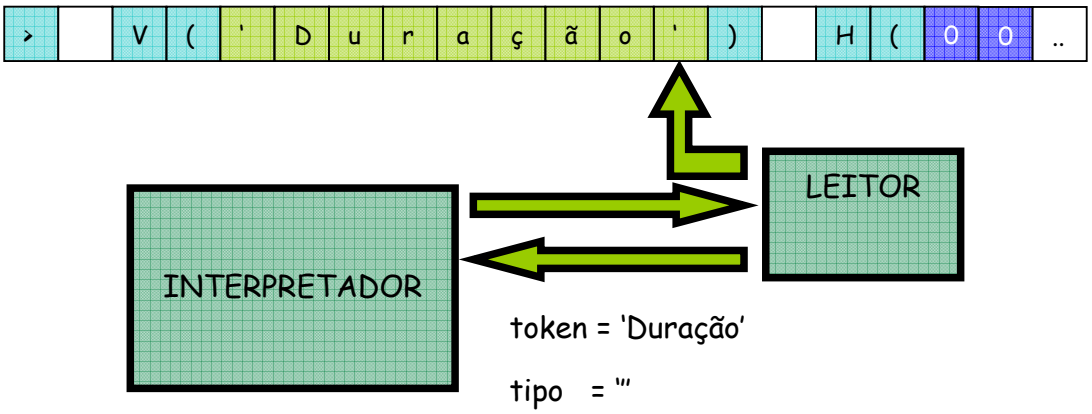

Figura 4 - Fluxo interpretador e leitor

O *token* lido pode ser acessado através do atributo *Leitor::_token*.

Levando-se em consideração que não deve existir nenhum erro léxico, sintático ou semântico na expressão, o leitor identifica os *tokens* em apenas três categorias: literal, número e caractere. O leitor identifica um *token* como literal se começar com apóstrofo, contiver qualquer caractere exceto apóstrofo e terminar com um apóstrofo. Um *token* identificado como número deve conter caracteres ASCII de '0' a '9'. Qualquer outro caractere será identificado caractere.

Se o *token* lido for caractere, o identificador retornado será o caractere ''' (apóstrofo), se for número retornará o caractere '1' e nos outros casos retornará o próprio caractere lido. Em vista disso, chamadas sucessivas ao leitor, com a expressão "> V('Duração') H(000003)", teriam o seguinte retorno:

1. Leitor::_token: '>'; identificador: '>'
2. Leitor::_token: 'V'; identificador: 'V'
3. Leitor::_token: '('; identificador: '('
4. Leitor::_token: 'Duração'; identificador: ''
5. Leitor::_token: ')'; identificador: ')'
6. Leitor::_token: 'H'; identificador: 'H'

7. Leitor::_token: '('; identificador: ')'
8. Leitor::_token: '000003'; identificador: '1'
9. Leitor::_token: ')'; identificador: ')'

De acordo com Stroustrup (2000), com o retorno do identificador sendo definido como *char* e o uso de ponteiros para métodos de um classe é possível criar um *array* no qual o índice é o identificador e o elemento é um ponteiro para o método a ser executado.

```
...
identificadorParaMetodo['=']=&Interpretador::empilheOperador;
..
```

Dessa forma, com o identificador retornado pelo leitor tem-se acesso direto à função a ser executada pelo interpretador. O interpretador interrompe o processo no caso em que o *token* retornado pelo leitor não for conhecido.

3.2.4 Executor

O executor é quem efetivamente realiza as operações relacionais, operações lógicas, funções lógicas e de manipulação de campos. Algumas explicações serão dadas utilizando-se da ICLang para que fiquem mais claras.

O Executor possui três objetos com função importante: `RegistroDeOperadores`, `RegistroDeOperandos` e `RegistroLogico`.

No `RegistroDeOperandos` são armazenados os operandos das funções relacionais e lógicas. Toda vez que este registro tiver dois operandos eles são retirados do registro e sobre eles é realizada a operação retirada do `RegistroDeOperadores`. O resultado desta operação será armazenado no `RegistroLogico`.

O RegistroLogico armazena os valores lógicos. Se a quantidade de elementos contidos no RegistroLogico for igual a um, verifica-se no RegistroDeOperadores se o operador é igual a Se ou Não. No caso do operador Se, o valor lógico é utilizado para decidir se deve dar um pulo para a expressão do Senão. Com o operador Não o valor é retirado, aplicado o operador de negação e o valor novamente incluído no RegistroLogico. Por outro lado toda vez que a quantidade de operadores for igual a dois, é realizada sobre eles a operação de retirada do RegistroDeOperadores.

As operações relacionais realizadas sobre os operandos dependem do tipo e cabe ao executor realizar as operações de conversão, uma vez que os campos vêm no formato *char*. Assim, operandos do tipo TEXTO, são comparados como um *array* de *char*. Campos do tipo NUMÉRICO, DATA e HORA são convertidos para número e então comparados.

Os *tokens* empilhados no RegistrodeOperadores são: Se, Não, E, Ou, <, <=, =, !=, =>, >. Os *tokens* empilhados no RegistroDeOperandos: constantes do tipo DATA, HORA, NUMÉRICO e TEXTO, e retorno das funções de manipulação de campos (SubCampo, ValorDoCampo, RestoDoCampo, TamanhoDoCampo). No RegistroLogico são empilhados os valores VERDADEIRO (1) ou FALSO (0).

As tabelas explicitadas nas funções são carregadas na memória de forma ordenada, permitindo assim a pesquisa binária.

3.2.5 Forma de integração

A integração é feita através de uma biblioteca onde está o interpretador com a chamada da seguinte função:

```
bool Interpretador::valideCDRPelaExpressao(char *_expressao,int *_erro, Campo[] _cdr);
```

Onde:

_expressão: é a expressão criada em ICLang e convertida em Notação Polonesa.
_erro: é o código de erro retornado, caso o método retorne falso.
_cdr: *array* das informações que formam um CDR, definida abaixo.

```
typedef struct Campo {
  char *valor; // Valor contido no campo
  char *nome; // Nome do campo
  char *tipo; // Tipo do campo
  int tamanho; // Tamanho do campo
} ;
```

Um exemplo de um array de Campo:

```
{ {'2373162','Assinante A', 'Numerico', 7}, ..., {'000123', 'Duração', 'Hora', '6'}, ..}
```

Para que o interpretador tenha acesso ao Banco de Dados, é necessário que seja repassado na sua inicialização um objeto que permita esta funcionalidade.

3.2.6 Conclusão

A utilização de código intermediário permite a independência do código em ICLang. O código intermediário possibilita, por exemplo, que as palavras reservadas da ICLang possam ser traduzidas para outra língua, como inglês ou espanhol, mantendo-se o mesmo código intermediário.

O uso de apenas um caractere para representar uma função ou operação reduz o tempo varredura do código em busca de *tokens*.

A disposição do código em notação polonesa permite com que o interpretador não se preocupe com a precedência de operações, uma vez que a ordem já é dada pela disposição dos elementos.

A divisão de tarefas fazendo com que o Leitor busque os *tokens*, o Interpretador reconheça os *tokens* e busque os parâmetros e os repasse para que o Executor os execute, facilita a manutenção e entendimento.

Reconhecimento dos *tokens* utilizando-se de um *array* onde o índice é o token e o elemento é função do interpretador a ser executada é uma solução que aumenta a eficiência do interpretador.

Um ponto que merece destaque é que o código intermediário é interpretado novamente a cada novo CDR. Todo o trabalho de leitura, reconhecimento e execução é refeito a cada CDR, fazendo com que a eficiência do interpretador esteja abaixo do que poderia.

3.3 Eficiência no Interpretador e Modificações na ICLang

O processo de avaliação se um CDR deve ou não ser filtrado, resume-se basicamente em receber como entrada uma expressão e um CDR, avaliando-o segundo as ações contidas na expressão. Baseando-se nessa avaliação é retornada uma saída indicando se o CDR cumpre ou não o especificado na expressão. Para cada CDR o interpretador passa por todo o processo de ler, reconhecer e buscar os parâmetros dos *tokens* contidos na expressão. Este pode ser um preço muito caro, tendo em vista que uma mesma expressão é sempre aplicada sobre um conjunto de CDRs cuja a quantidade ultrapassa a casa dos milhares por dia.

3.3.1 Roteiro de Execução

Tendo em vista que a mesma expressão é avaliada a cada novo CDR, foi criada a idéia de roteiro de execução. O roteiro de execução é o conjunto de todas as operações realizadas pelo executor. Todas as funções de reconhecimento de *token*, busca de parâmetros, são feitas apenas uma vez e guardadas então seqüencialmente, como um conjunto de instruções a serem executadas.

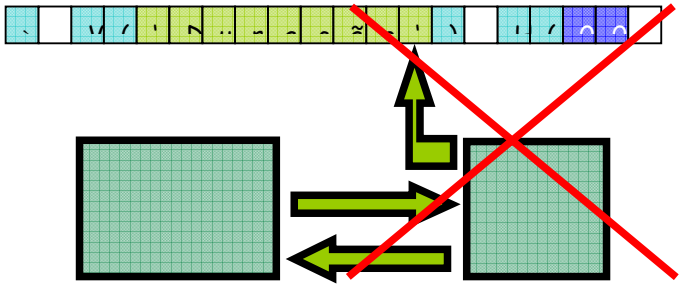

Figura 5 - Retirada da leitura da expressão a cada CDR

Assim, a interpretação é divida em duas partes. A primeira em que o primeiro CDR, juntamente com a expressão a ser interpretada são passados ao interpretador que percorre a expressão gerando um conjunto de passos necessários para interpretar aquela expressão. A partir deste momento, tem-se um roteiro que deve ser usado toda vez que se quiser filtrar um CDR. Exemplificando:

Em ICLang: `ValorDoCampo('Duração') > Hora(000003)`

No código fonte do interpretador:

```
> V ('Duração') Hora(000003)
```

Na Figura 6, pode-se verificar que existe muita interação entre o interpretador e o leitor na leitura dos *tokens* e busca de parâmetros.

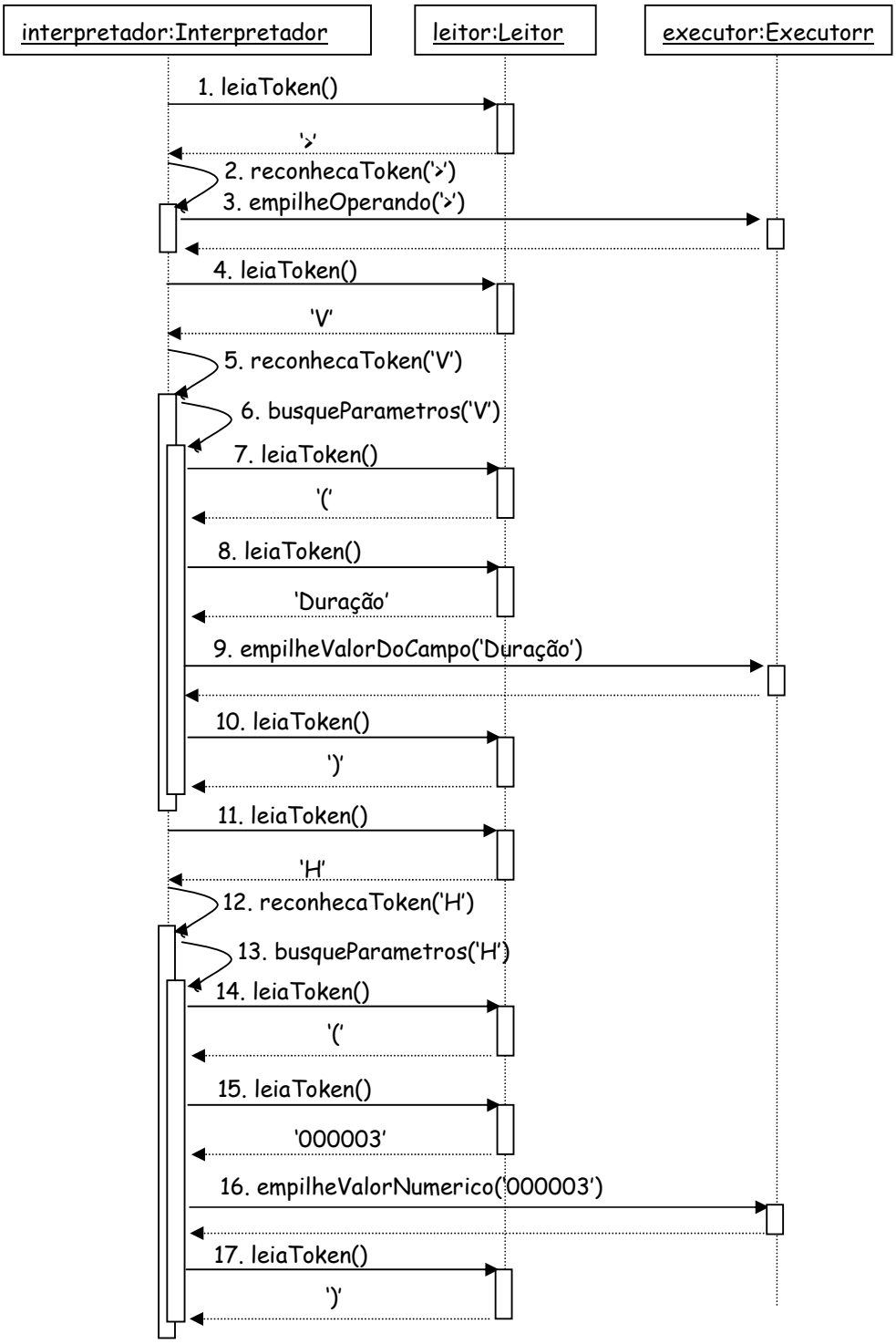

Figura 6 - Diagrama de seqüência de interpretação

O mecanismo do roteiro de execução consiste em buscar todos os *tokens* juntamente com seus parâmetros mapeando todas as chamadas de métodos do executor e seus respectivos parâmetros. Dessa maneira, tem-se uma lista das chamadas que devem ser feitas ao executor juntamente com os seus parâmetros.

No exemplo o seguinte roteiro de execução seria gerado:

Método do executor	Parâmetro
1. empilheOperando()	'>'
2. empilheValorDoCampo()	'Duração'
3. empilheValorNumerico()	'000003'

Tabela 2: Roteiro de execução gerado

A existência de um CDR para a criação do roteiro de execução faz-se necessária para que sejam realizadas a verificação de tipos e a existência de campos.

3.3.2 Forma de Integração

Para cada nova expressão, juntamente com o primeiro CDR.

```
int retorneRoteiroExecucao(RoteiroDeExecucao **roteiro,char *expressao,CDR * _cdr);
```

Para cada avaliação de um determinado CDR.

```
int valideCDRPeloRoteiroDeExecucao(RoteiroDeExecucao *roteiro, int *erro, CDR *cdr);
```

3.3.3 Modificações na ICLang

3.3.3.1 Constante Falso

A linguagem na sua primeira versão levava em consideração que as expressões estavam associadas aos campos que compõem um CDR. Por isso, quando a avaliação de uma expressão retornava FALSO, a mensagem de erro era implicitamente associada ao campo que continha a expressão.

Isso se mostrou indesejável em algumas situações, principalmente nos casos em que deveria ser feita uma alteração na ordem de avaliação das expressões. Como a ordem de avaliação das expressões era a ordem de definição dos campos que compõem um CDR, a necessidade de precedência na aplicação de um filtro poderia associá-lo a um campo que não possui nenhuma relação com o filtro especificado.

Baseado nesse fato, a linguagem foi modificada levando em consideração que uma expressão não estaria mais associada a um campo. As expressões que antes eram associadas aos campos que formavam um filtro para um determinado sistema, agora são um conjunto de expressões associadas diretamente ao sistema para o qual se pretende fazer a filtragem.

Com base na versão anterior a Tabela 3 mostra como é definido um filtro. Como não existe mais uma associação a um determinado campo, quando um erro é encontrado, é necessário que se explicite o campo a qual este erro está associado. Dessa forma, foi modificada a sintaxe da constante FALSO adicionando-se além do erro, o campo a qual o erro está associado e uma ação. Essa ação permite ao usuário definir o que deve ser feito com o CDR.

As possíveis ações são:

- Gerar erro e parar - não executa mais nenhuma expressão para o CDR em questão, retorna o campo e a mensagem e termina o processamento do CDR.
- Gerar erro e continuar - retorna o campo e a mensagem, mas continua processando o CDR, executando as próximas expressões.

- Finalizar - não executa mais nenhuma expressão para o CDR em questão, retorna VERDADEIRO, terminando o processamento do CDR.

Filtro para o Sistema de Faturamento			
Nome	Tipo	Posição	Conjunto Associado ao Sistema de Faturamento
Assinante A	Texto	1 –18	Se Pertence(ValorDoCampo('Fim de Seleçao'), Intervalo(1,3) Entao VERDADEIRO Else FALSO//'Fim de Seleção Inválido', 'Fim de Seleção', 'Gerar Erro e Parar'// Se (ValorDoCampo('Duracao') > Hora(000003)) Entao VERDADEIRO Else FALSO//'Duração inferior a 3 segundos','Duração', 'Gerar Erro e Continuar' //
Assinante B	Texto	19-36	
Data de Início	Data	37-44	
Fim de seleção	Numérico	45-46	
Duração	Hora	47-52	

Tabela 3 - Conjunto de expressões associadas ao sistema

Com isso é possível fazer com que se apliquem todos os filtros mesmo que a condição FALSO seja atingida ou se determinada condição for satisfeita não realizar mais nenhum filtro.

3.3.3.2 Função Lógica ExecuteRotinaExterna

A ICLang não possui operações matemáticas, nem mecanismos de definição de funções. Visando eficiência e uma forma de proporcionar aos usuários avançados maior liberdade, uma nova função pode fazer chamadas a funções de uma biblioteca compartilhada. Para tanto é necessário passar o nome da lib.so e o nome da função.

Exemplo:

```
ExecuteRotinaExterna('libUsuario.so', 'calculaDuracao');
```

No exemplo acima, existe uma biblioteca compartilhada chamada libUsuario.so com uma função chamada calculaDuracao, que poderia ser uma função que calcula o tempo de duração[3] de uma chamada. Essa função deve, assim como uma expressão, retornar um valor VERDADEIRO ou FALSO juntamente com a mensagem de erro, o campo e ação a ser executada.

Foram adicionadas também as seguintes funções de coersão de tipo, TextoParaNumero e NumeroParaTexto. Ambas as funções realizam a troca de tipo e recebem como parâmetro alguma das funções de manipulação de dados.

Exemplos:

```
NumeroParaTexto('FimDeSeleção') = '01'
Pertence(TextoParaNumero(SubCampo('Assinante A',2, 2)), Conjunto(Intervalo(47,49))
```

3.3.4 Forma de Integração

Para o uso de rotinas externas é necessário na definição das funções o uso do seguinte padrão, na criação das funções:

```
bool nomeDaFuncao(
    int *__codErro,
    int *__index,
    int *__acao,
    Campos *__campos,  const int __numeroDeCampos)
```

3.3.5 Teste de Desempenho

Para avaliar as possíveis diferenças de desempenho inseridas com a definição do roteiro de execução e o uso de bibliotecas compartilhadas foram realizados alguns testes. A máquina utilizada foi uma IBM pSeries 610 Model 6C1 com 1 processador de 450MHz 64 bits, 1GB de memória e sistema operacional AIX 5L. Foram feitos 21 testes com 500.000 chamadas para cada uma das 4 expressões diferentes, juntamente com CDR padrão.

A complexidade do ponto de vista do interpretador é dada pelo número de tokens, a quantidade de parâmetros requeridos e pelas funções executadas. A expressão de menor complexidade caracteriza-se pela constante lógica VERDADEIRO, que transformada para a linguagem intermediária possui apenas uma *token*. A segunda expressão utilizada foi a constante FALSO com um código de erro, código do campo e código da ação.

Nas outras duas expressões foram acrescidas operadores relacionais lógicos e Se aninhados.

1. VERDADEIRO

2. FALSO//'Erro teste', 'teste9', 'Gerar Erro e Parar'//

3. Se ValorDoCampo('teste9') = 1234567 Então
   VERDADEIRO
   Senão
   FALSO//'Erro teste', 'teste9', 'Gerar Erro e Parar'//

```
4.  Se  ValorDoCampo('teste9') = 1234567 Então
        VERDADEIRO
    Senão
        Se  ValorDoCampo('teste9') = 1234567 Então
        VERDADEIRO
    Senão
        FALSO//'Erro teste', 'teste9', 'Gerar Erro e Parar'//
```

Os tempos de execução foram tomados, utilizando-se do comando time e, para a criação dos gráficos, do tempo total que o processo demorou para executar 500.000 vezes.

A Figura 7 mostra o gráfico onde são apresentados os valores do tempo de execução com e sem o roteiro de execução. Verifica-se, analisando o gráfico, que com o roteiro de execução o tempo para avaliar as expressões cai pela metade com o aumento da complexidade.

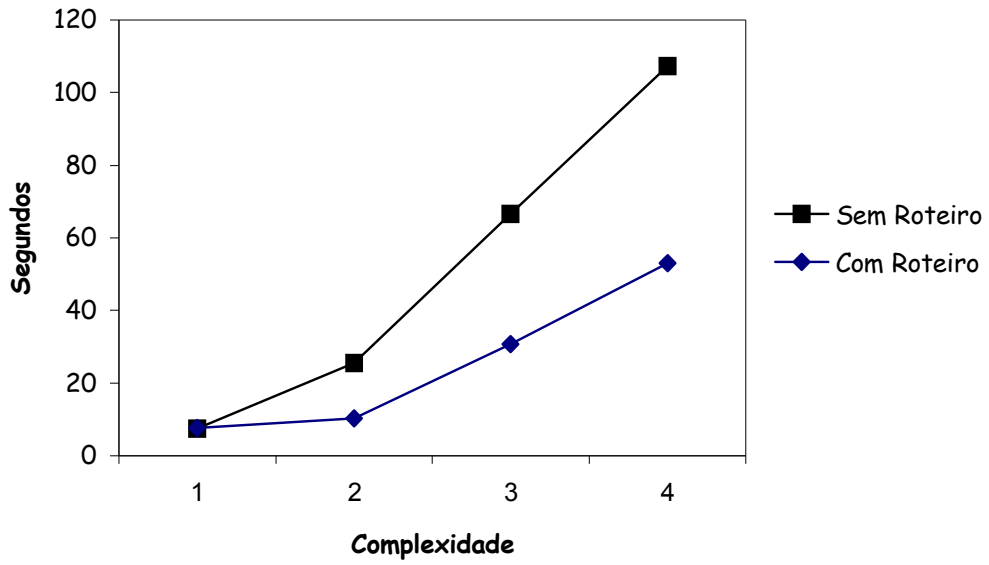

Figura 7 – Tempos de interpretação com e sem roteiro de execução

Para tornar mais evidente, essa diferença é apresentada na Figura 8 novamente, com os mesmos tempos. A diferença é que os tempos de interpretação sem o roteiro de execução são plotados de acordo com o eixo Y da esquerda. Já os tempos de interpretação com o roteiro de execução são plotados tendo como base os valores do eixo Y da direita. Pode-se notar que os valores do eixo Y da direita são a metade dos seus correspondentes da esquerda e que os pontos de plotagem seguindo estes valores quase se sobrepõem, indicando que a utilização do roteiro de execução é 100% mais eficiente em relação à interpretação sem a sua utilização.

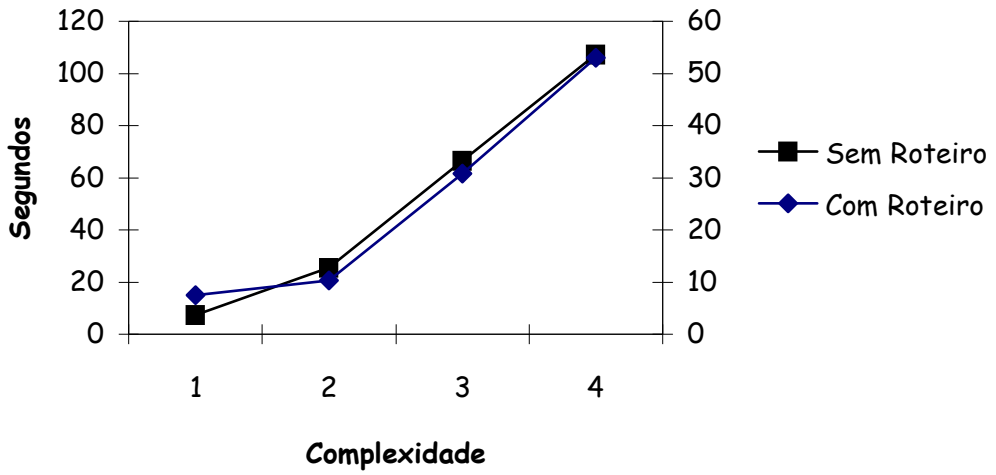

Figura 8 - Tempos de interpretação com e sem roteiro de execução

Com a introdução nessa etapa de chamadas de funções contidas em bibliotecas compartilhadas, o próximo teste foi comparar o tempo da interpretação utilizando o roteiro de execução com a interpretação utilizando o roteiro de execução, mas chamando apenas a função `ExecuteRotinaExterna`. Para isso, as expressões foram traduzidas para o código equivalente em C++. Foram geradas quatro funções e criada uma biblioteca compartilhada com elas.

Os valores, na Figura 9, mostram as diferenças entre a interpretação, utilizando o roteiro de execução e a interpretação também com o roteiro de execução, mas chamando somente as funções definidas em C++ contidas na biblioteca compartilhada criada. Quanto maior a complexidade, maior é a diferença entre os tempos de execução. Isto porque em C++ os tipos, as formas de comparação, já são definidos em tempo de compilação. Já com a utilização do interpretador, no momento da execução é que os tipos são verificados e as transformações necessárias são feitas.

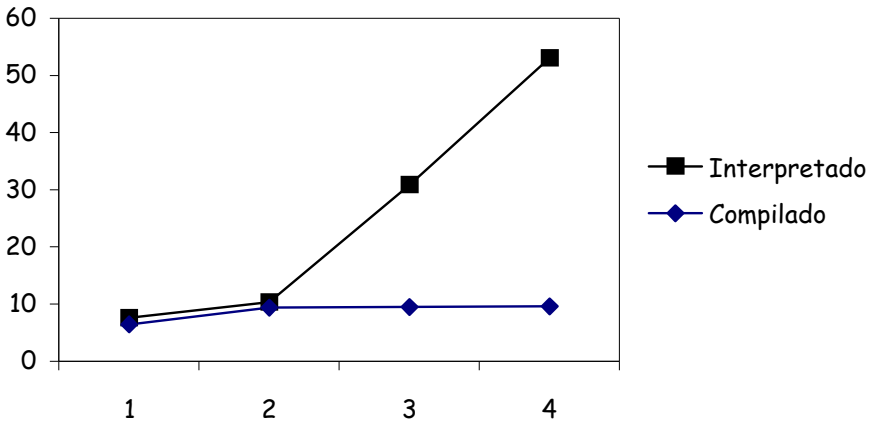

Figura 9 - Tempos do código interpretado e compilado

3.3.6 Conclusão

Os novos parâmetros que acompanham a constante FALSO permitem mais flexibilidade e controle sobre o conjunto de expressões que devem ser avaliadas. Pode-se agora mudar com facilidade a ordem em que as expressões são avaliadas.

A introdução de retorno de uma ação a ser feita também permite que o usuário tenha mais controle sobre o que ele quer fazer no CDR, podendo parar o processamento daquele CDR naquele momento, continuar, mas sinalizar o erro ou ainda não executar mais nenhuma avaliação sobre o CDR.

Com a nova função lógica `ExecuteRotinaExterna` é permitido ao usuário construir suas próprias funções de filtro e chamá-las quando bem entender.

As funções de coersão de tipo fazem com que o usuário tenha mais controle sobre os dados permitindo o uso de operadores relacionais entre campos e constantes de tipos diferentes.

3.4 Eficiência, Distribuição e Segurança

Os testes verificaram que a eficiência do código compilado ainda é muito superior ao do código interpretado. A eficiência é um ponto de grande importância levando-se em consideração que o processo de geração de CDRs pelas centrais bilhetadoras, e conseqüentemente a geração de arquivos que devem ser filtrados, é um processo contínuo. Por outro lado todas as características disponíveis na ICLang fornecem ao usuário uma série de facilidades que não podem ser ignoradas.

O principal objetivo dessa etapa é propor um mecanismo que permita a união da eficiência das linguagens compiladas à redigibilidade e legibilidade das linguagens criadas para fins específicos, como no caso da ICLang.

A primeira vista, uma solução é criar um tradutor que transforme a expressão especificada em ICLang para código fonte C++ correspondente. Dessa forma o usuário teria as facilidades disponíveis em ICLang e a eficiência de um código compilado.

O ambiente de criação da expressão e o tradutor são partes do módulo denominado Cliente. A partir código C++ disponibilizado pelo módulo Cliente é criada uma biblioteca compartilhada. O módulo responsável pela criação dessa biblioteca é chamado de Servidor de Compilação.

Nesse ponto, o código, primeiramente descrito em ICLang, está pronto para ser executado. Para tanto, é necessário um novo módulo responsável por executar uma determinada função contida numa biblioteca compartilhada. Esse servidor é chamado de Servidor de Execução.

Na figura 10 tem-se uma visão geral de todo o processo.

No ponto 1 a expressão feita em ICLang é passado para o tradutor. O tradutor então cria um novo arquivo, no ponto 2, agora em código fonte C++. Este arquivo é repassado para o Servidor de Compilação em 3 para que ele gere uma biblioteca compartilhada e a armazene (em 4). O cliente então ou um outro processo invoca, em 5, a execução de filtros, presentes nas bibliotecas compartilhas, sobre um conjunto de CDRs. O Servidor de Execução então carrega a biblioteca dinamicamente e aplica as funções de filtros sobre os CDRs (7) , retornando então o resultado.

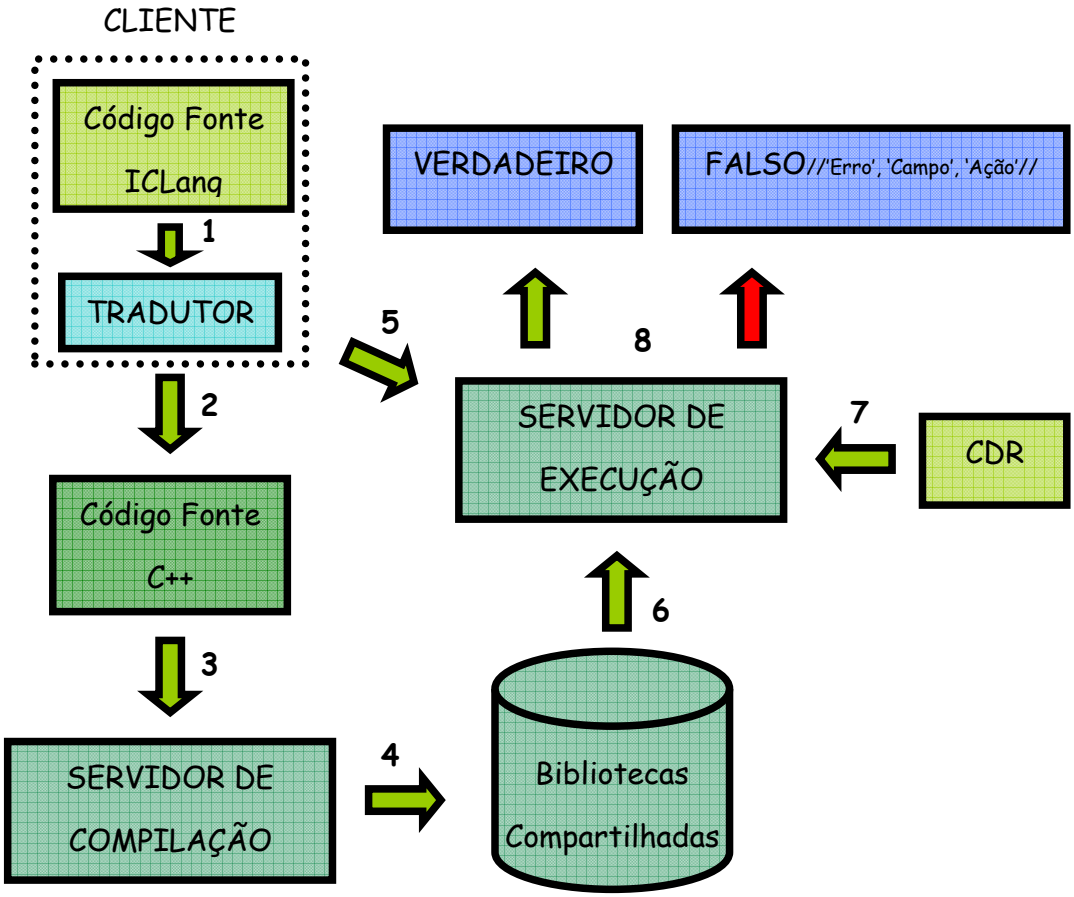

Figura 10 - Processo de execução

A comunicação entre os módulos Cliente, Servidor de Compilação e Servidor de Execução é feita utilizando-se de CORBA. Isto permite que o módulo Cliente, com recursos de interface com o usuário, seja implementado numa linguagem diferente dos Servidores de Compilação e Execução. O CORBA também permite que os módulos estejam distribuídos dentro da rede.

Seguindo as definições apresentadas até o momento, os módulos criados estão distribuídos e cooperando entre si, tanto na criação de novas funções de filtragem, como na aplicação destes filtros sobre os arquivos de CDRs.

Observando-se mais cuidadosamente os pontos 3, 5 e 6, pode-se perceber que são pontos críticos dentro do processo. Em 3 porque o Servidor de Compilação não tem como saber se o módulo Cliente, que está enviando um código C++ para ser compilado, é confiável. De forma semelhante em 5 e 6, se é seguro executar a função pedida pelo o usuário e se ele realmente tem autoridade para pedir que tal função seja executada sobre o conjunto de CDRs.

Para resolver este problema, é necessária a criação do Servidor de Segurança que será responsável por garantir que somente usuários autorizados, através do módulo Cliente, tenham acesso aos Servidores de Compilação e Execução.

Agora somente usuários autorizados podem repassar códigos em C++ para o Servidor de Compilação, como também pedir que um conjunto de filtros sejam aplicados sobre os CDRs para o Servidor de Execução.

Apesar do Servidor de Compilação ter garantias de que o usuário que está lhe enviando um arquivo com código C++ ser um usuário autorizado para tal, nada lhe garante que o conteúdo do arquivo seja realmente a conversão de um filtro para a linguagem C++. Sem mecanismos para verificação desse código, qualquer arquivo C++ sem erros na sua codificação geraria uma biblioteca compartilhada que mais tarde poderia ser executada pelo Servidor de Execução. Um usuário mal-intencionado poderia criar um filtro que por exemplo apagasse toda a massa de dados no Servidor de Execução. É necessário que se tenha garantias que o código que será executado é um seguro.

Para prover tal funcionalidade, além do próprio Servidor de Compilação, é necessário a modificação no módulo Cliente. Essa funcionalidade será implementada, utilizando-se de características presente em XML.

Dessa forma o módulo cliente não traduzirá o código em ICLang para um arquivo de código fonte C++, mas sim para um arquivo no formato XML.

O Servidor de Compilação, utilizando-se de recursos presentes em XML pode verificar se o arquivo é bem formato, ou seja, está dentro das especificações XML. Além da verificação básica de sintaxe, é possível a criação de regras que determinam como o arquivo deve ser formado, num processo de definição formal chamado de modelagem de documento. Isto pode ser feito através das Definições de Tipo de Documento (DTD).

Uma vez que o Servidor de Compilação tem a capacidade de avaliar se o arquivo em XML corresponde a uma especificação de um filtro em ICLang ele pode utilizar de outra ferramenta presente em XML, para converter esse arquivo para o código fonte C++ definitivo e então gerar a biblioteca compartilhada. A conversão do arquivo é feita usando XSLT. A XSLT é projetada especificamente para realizar transformações.

Serão discutidos a seguir cada módulo em particular.

3.4.1 Aplicação Cliente

A suas principais funções são permitir às aplicações, acesso aos Servidores de Segurança, Execução e Compilação e tradução da expressão, escrita em ICLang, para XML, para posterior transferência para o Servidor de Compilação.

O acesso ao Servidor de Segurança é essencial, uma vez que é através dele que serão conseguidas as referências aos servidores de Compilação e Execução. Este acesso ao servidor de Segurança se dará através do serviço definido no CORBA como *Naming Service.* Através desse serviço é possível fazer um mapeamento do nome dado ao Servidor de Segurança, para a sua referência de objeto e conseguir com ele a referência dos Servidores de Compilação e Execução.

Com a referência do Servidor de Execução a Aplicação Cliente está apta a fazer requisições de execução de determinada função pré-definida sobre um conjunto de CDRs. De forma equivalente, a referência do Servidor de Compilação, permite que aplicação envie para ele as especificações criadas pelo usuário e transformadas em

arquivos XML. Essa necessidade de tradução para XML faz com que a interface possa permitir que uma determinada expressão possa ser implementada da forma mais amigável possível. Os exemplos podem ser elementos gráficos que combinados representem uma expressão ou o uso de editores com *code completation*.

Segundo Henning (1999) e Siegel (2000) a passagem dessa especificação, já transformada em XML, para o Servidor de Compilação é feita via CORBA, utilizando-se, por exemplo, de uma seqüência de octetos.

O módulo Cliente é totalmente dependente do Servidor de Compilação, uma vez que é ele que tem que entender suas especificações e prover os mecanismos necessários para sua transformação para código fonte.

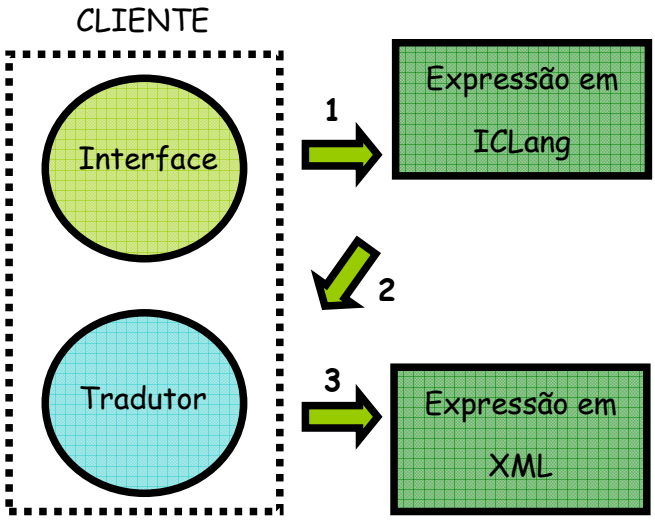

Figura 11 - Aplicação Cliente

A figura 11 mostra os passos necessários para criar o arquivo necessário em XML. Utilizando a interface, o usuário cria a expressão em ICLang. A interface deve prover mecanismos que permitam a correta criação dessa expressão. Uma vez pronta a

expressão, ela é repassada para o tradutor responsável por criar uma nova expressão no formato XML.

3.4.2 Servidor de Execução

O Servidor de Execução é o módulo responsável por executar as requisições feitas pelo usuário. Ele utiliza o Servidor de Segurança verificando as permissões do usuário para a execução de uma determinada requisição. Possui mecanismos que permitem buscar o código a ser executado em outros locais na rede.

Existem duas formas básicas de utilização do Servidor de Execução. A primeira seria utilizar o Servidor de Execução independente da aplicação cliente.

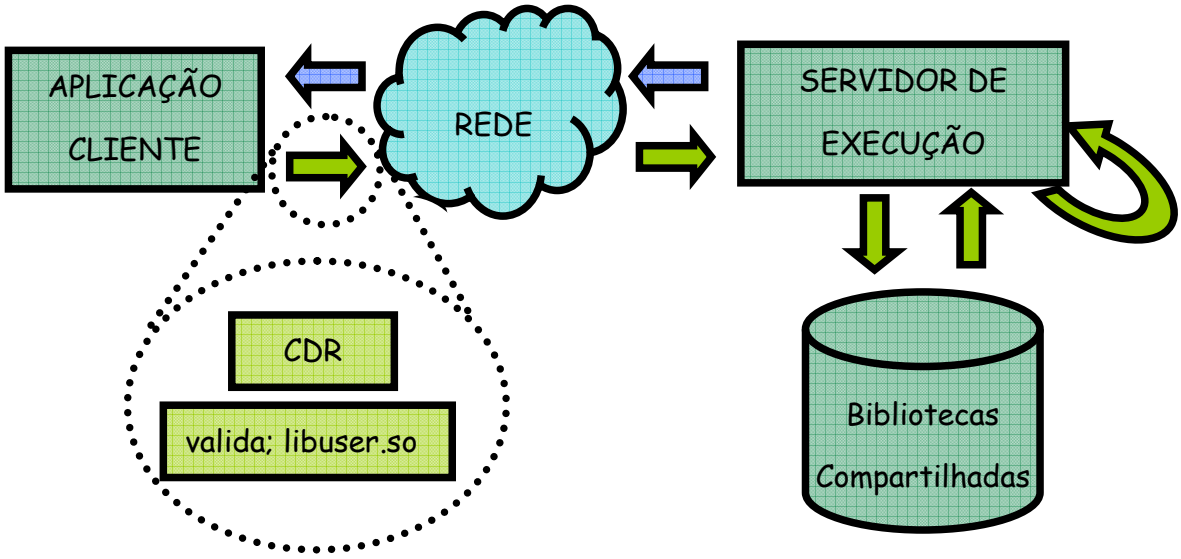

Figura 12 - Servidor de Execução independente

Nessa modalidade é necessário executar o Servidor de Execução antes de utilizar a aplicação Cliente. As requisições da aplicação cliente seriam repassadas, via CORBA, para o Servidor de Execução, que poderia estar na mesma máquina ou em outro ponto

qualquer da rede. Dessa forma, aplicações que exigem velocidade poderiam ser executadas numa máquina com maior poder de processamento.

Na segunda forma, o Servidor de Execução é parte integrante da Aplicação Cliente, habilitando-a a executar o código em qualquer lugar que esteja. Dessa forma, novas funcionalidades criadas, por exemplo, por um gerente, poderiam ser executadas por agentes.

Figura 13 – Integração Servidor de Execução e Aplicação Cliente

O Servidor de Execução provê os mecanismos necessários a execução de funções. A sua interação com o Sistema de Segurança permite a verificação de permissões. Mecanismos de atualização e busca de funções também são funcionalidades previstas. Através de mecanismos de verificação de atualizações o Servidor pode buscar novas bibliotecas, até mesmo por demanda.

O Servidor de Execução poderá rodar em sistemas operacionais que possuam mecanismos de carga e execução dinâmica.

Como o Servidor de Execução é capaz de executar qualquer código que esteja no formato por ele esperado, pode ser uma fonte vulnerável a ataques. É interessante prover algum mecanismo que autentique o código a ser executado como seguro.

A primeira vista, é desejável que todas as funcionalidades fornecidas pelo Servidor de Execução sejam controladas pelo Servidor de Segurança, mas não são descartadas aplicações que não possuam um controle assim tão rígido.

3.4.3 Servidor de Segurança

O Servidor de Segurança é um módulo de suma importância, uma vez que é o responsável pela exclusão de usuários não autorizados ao sistema. Ele serve aos diversos módulos, fornecendo as referências, para acesso do módulo Cliente, ao Servidor de Compilação e Execução. Fornece todas as permissões necessárias para que o Servidor de Execução execute as funções e para que o cliente crie especificações.

A autenticação do usuário que estiver utilizando a aplicação é feita mediante a utilização de usuário/senha.

Assim, ao utilizar a aplicação cliente é necessário que o usuário entre com sua identificação, que verifica se aquele usuário pode ou não criar especificações. É o servidor de Segurança que autentica o acesso a funções no qual o usuário possa executar. Por motivos de auditoria, o Servidor de Segurança fornece um *log* de todas as funções executadas no sistema.

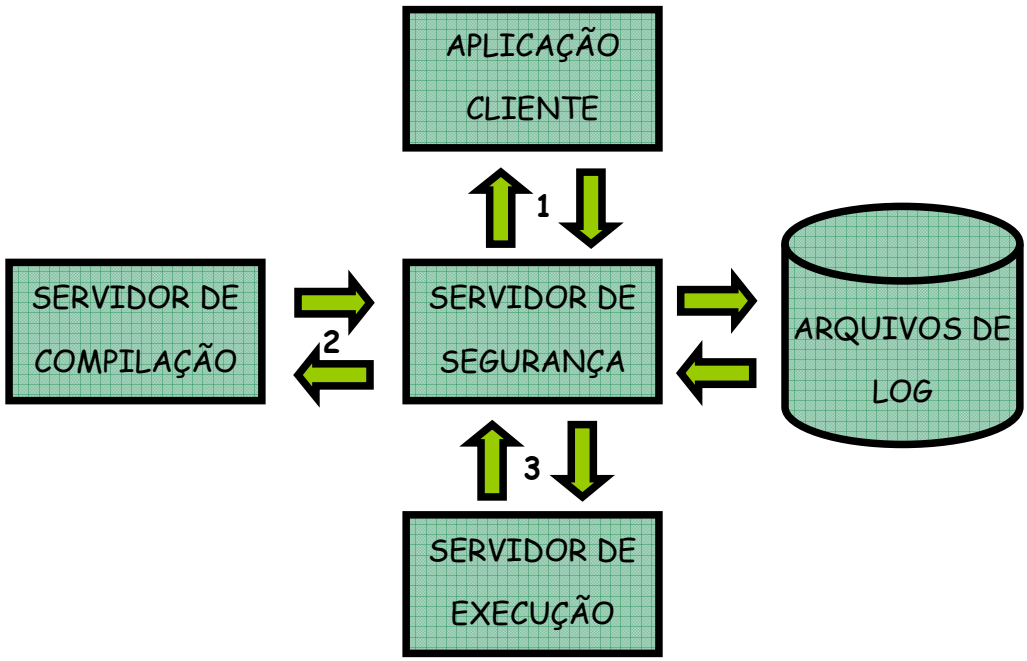

Figura 14 - Interação do Servidor de Segurança

Quando a aplicação Cliente é executada, através do *Naming Service* do CORBA obtém a referência ao Servidor de Segurança. A aplicação Cliente então registra junto ao Servidor mediante a usuário/senha, requisitando as referências aos outros servidores.

O Servidor de Compilação quando executado, também através do *Naming Service* do CORBA obtém a referência ao Servidor de Segurança. É por meio dela que ele verifica se usuário/senha está autorizado a gerar especificações que serão transformadas em bibliotecas compartilhadas e registra estas mesmas bibliotecas depois da sua geração.

Da forma descrita acima, o Servidor de Execução obtém a referência do Servidor de Segurança, registra, verifica se pode executar sobre o arquivo de CDRs, a função requisitada pelo usuário/senha, além de verificar se alguma atualização está disponível.

O Sistema de Segurança deve ser robusto, seguro e com resposta rápida. Para tanto, devido a sua complexidade, o Servidor de Segurança utilizado será o criado

através do *framework* desenvolvido por Kátyra (2002), sendo fornecida apenas a interface para sua integração.

Todos os outros módulos são dependentes do Servidor de Segurança, uma vez, ocorrendo uma falha ao se criar uma instância, nenhum dos módulos poderá executar. Já o Servidor de Segurança é um módulo totalmente independente.

3.4.4 Servidor de Compilação

O módulo Servidor de Compilação é o responsável em traduzir as especificações do usuário, para código que possa ser executado.

A solução proposta parte do princípio que estas especificações chegam ao servidor de compilação no formato XML, enviados pela aplicação Cliente. O Servidor de Compilação então verifica, junto ao Servidor de Segurança se o usuário tem permissão para geração de bibliotecas compartilhadas. No caso afirmativo o servidor validará a especificação de duas formas. A primeira é verificar se o arquivo está realmente no formato XML. A segunda é verificar se o arquivo enviado possui realmente especificações para geração de expressões. Isto é feito através da aplicação de uma DTD[4]. Isto permite verificar se clientes maliciosos não estão tentando gerar um código que eventualmente possa causar alguma vulnerabilidade ao sistema ou danificá-lo de alguma forma.

De acordo com Beggs e Boone (2001) e Egashira e Kiriha (2000) uma vez que o arquivo XML seja válido e que esteja definido corretamente apenas com funções válidas, o Servidor aplicará sobre os dados uma transformação utilizando XSLT. É essa transformação que, lendo o arquivo em XML, transformará em código fonte C++ pronto para ser transformado em uma biblioteca compartilhada. O Servidor então registra a nova biblioteca junto ao Servidor de Segurança.

---

[4] Ver Revisão de Literatura

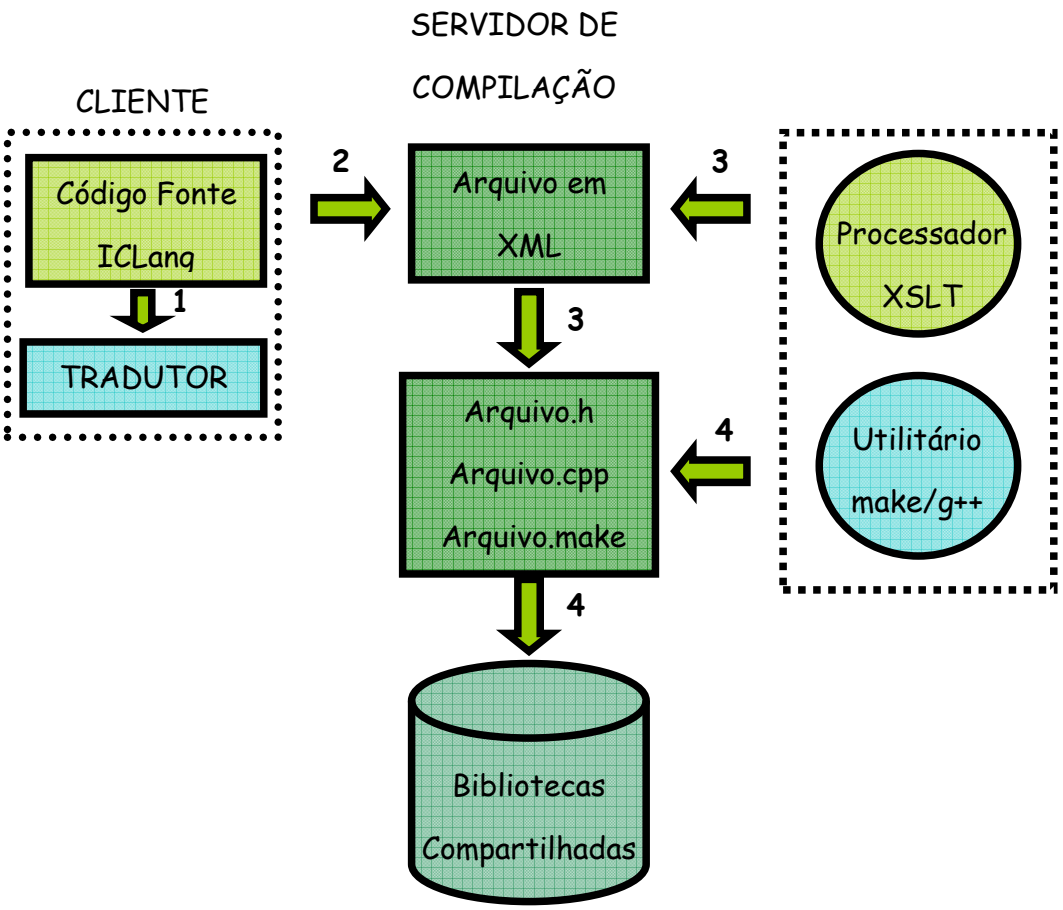

Figura 15 - Geração de uma biblioteca compartilhada

3.4.5 Aplicações de teste

O principal objetivo das aplicações implementadas é testar e validar as principais idéias contidas nesta dissertação.

Para validar o módulo Cliente, foi criado um protótipo contendo um analisador léxico, sintático para ICLang. O analisador semântico não chegou a ser implementado.

A geração do analisador sintático foi feita do gerador chamado GAS.

A aplicação foi implementada em Delphi, utilizando um componente para edição de texto chamado RAHLEditor. A aplicação possue também funcionalidade, para geração de marcação XML, para alguns casos.

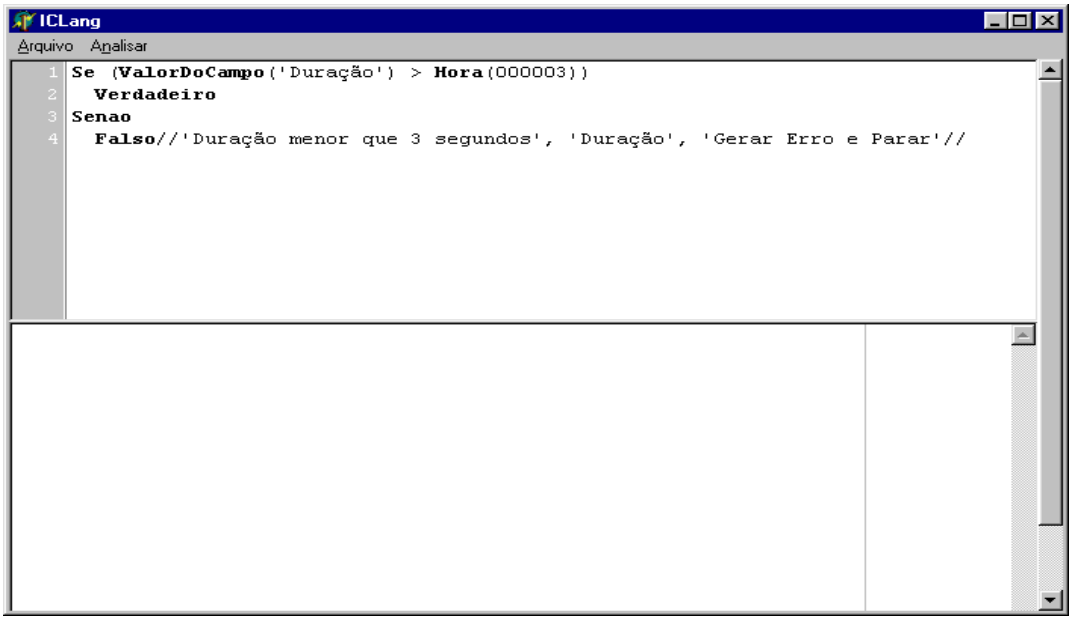

Figura 16 - Analisador léxico e semântico da ICLang

Para o módulo de Servidor de Compilação foram testadas as transferências de arquivos via CORBA, utilizando-se de uma seqüência de octetos. Para isso, foi implementando um servidor do qual podem-se pedir requisições de transferência de determinados arquivos presentes no sistema operacional do servidor.

A aplicação cliente foi escrita em Delphi usando a implementação CORBA chamada DORB. O servidor foi escrito em C++ utilizando MICO.

A aplicação Cliente precisa do IOR do servidor para poder fazer a requisição.

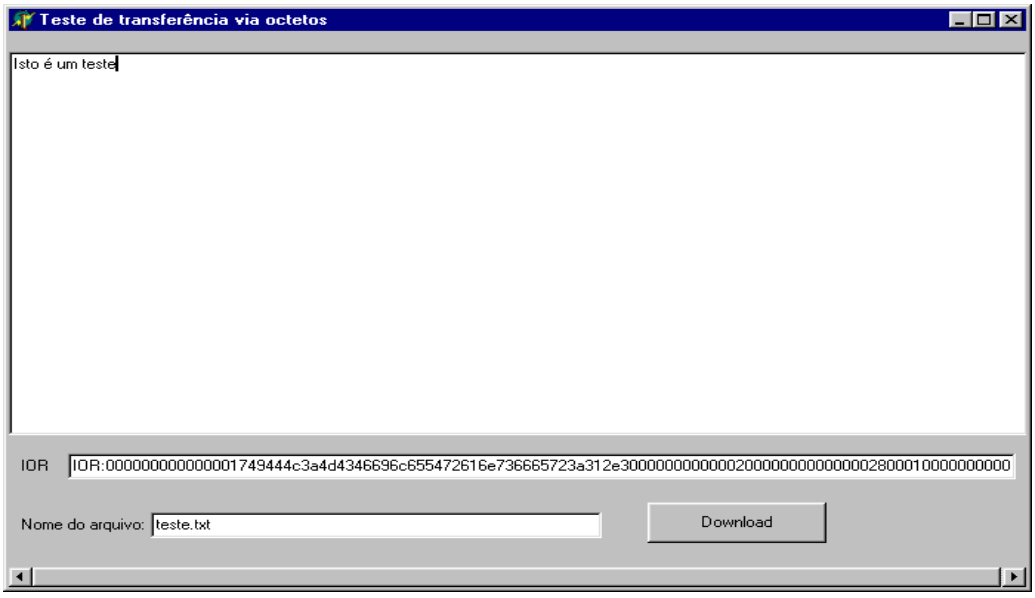

Figura 17 - Teste com transferência de arquivos

A definição IDL do servidor:

```
interface MCFileTransfer {
typedef sequence<octet> ByteSequence;
    void downLoad (in string fileName, out ByteSequence data);
    void upLoad (in string fileName, out ByteSequence data); };
```

Foram feitos testes de integração do Servidor de Segurança com clientes escritos utilizando-se de MICO C++ e DORB, todos sem nenhum problema.

4 CONCLUSÕES

4.1 Revisão das Motivações e Objetivos

O objetivo estabelecido neste trabalho foi a proposição de um mecanismo que permita a criação das funções de filtragem de dados, oferecendo as facilidades de uma linguagem ou forma visual voltadas ao conhecimento dos usuários, associado à eficiência de um código compilado.

Os filtros são executados sobre arquivos advindos de centrais bilhetadoras que contém um conjunto de bilhetes. Esse mesmo arquivo pode alimentar diferentes sistemas dentro de uma prestadora, mas geralmente contendo apenas informações úteis ao sistema em questão. Desse modo, é indesejável à área de faturamento receber bilhetes que não são faturáveis, provocando um processamento desnecessário.

Foi prevista a criação de uma nova linguagem para que o usuário tenha facilidade na criação de seus filtros. Um interpretador capaz de aplicar os filtros também foi objetivo desse trabalho.

Por fim, esperava-se que com a experiência adquira e estudos em CORBA e ferramentas XML fosse possível propor um mecanismo não somente voltado para as operadoras de STFC, mas para todas as aplicações que recebem um fluxo contínuo de dados.

4.2 Visão geral do trabalho

É apresentada na introdução, uma breve visão histórica sobre o desenrolar de alguns fatos relevantes associados ao Serviço Telefônico Fixo Comutado (STFC) até a

sua privatização. Tal conhecimento é a base para o entendimento da situação atual, com suas necessidades, problemas e direcionamentos.

São abordados na revisão da literatura alguns tópicos como definições importantes em CORBA, bibliotecas compartilhadas, XML e geração dos arquivos de CDRs nas centrais bilhetadoras.

O trabalho então foi divido em etapas a serem cumpridas.

Na primeira etapa é apresentada a definição de uma nova linguagem de programação, chamada ICLang. Esta linguagem é voltada para a criação de filtros e transformações sobre dados, tendo como pontos fundamentais a legibilidade e a redigibilidade.

A segunda etapa define o interpretador da linguagem. O interpretador não utiliza diretamente a ICLang como código fonte, mas sim uma linguagem intermediária. Esta linguaguem intermediária é apresentada informalmente, assim como alguns detalhes referentes à implementação do interpretador.

Com o objetivo de tornar o interpretador mais eficiente e a linguagem criada mais funcional, são apresentados na terceira etapa, uma forma de melhoria na eficiência e introduzidos novas funcionalidades na ICLang.

Em busca de uma abrangência maior, é apresentado na quarta fase um mecanismos para aumentar ainda mais a eficiência da filtragem bem como a sua distribuição e introdução de mecanismos de segurança. Nessa fase as definições escritas em ICLang são traduzidas no módulo Cliente para uma linguagem XML e enviadas ao Servidor de Compilação via CORBA. O Servidor de Compilação utilizando ferramentas de XML verifica não só a validade do arquivo recebido, mas se ele é realmente a tradução de uma expressão em ICLang. Isto é feito utilizando modelagem de documento através de uma DTD que descreve a estrutura de um documento de forma declarativa. O arquivo depois de verificado é traduzido para código fonte C++, através da utilização de XLST. O código é compilado e é criada então a biblioteca compartilhada.

A partir daí as funções estão aptas a serem executadas pelo Servidor de Execução.

Todas as operações são certificadas através do Servidor de Segurança

4.3 Contribuições e escopo do trabalho

Para este trabalho foram utilizadas apenas ferramentas de domínio público.

A implementação do interpretador e a utilização de bibliotecas dinâmicas foram realizadas em sistema UNIX.

Os filtros serão aplicados sobre o fluxo de dados gerados pelas chamadas telefônicas feitas pelos usuários da prestadora de STFC, disponibilizadas na forma de arquivos, mas o mecanismo proposto na última etapa adequa-se a uma série de aplicações que manipulam fluxo contínuos de dados como gerência de redes, sistemas de intrusão.

As propostas e os estudos feitos em C++ para melhoria de eficiência, como a utilização de ponteiros para métodos e a criação do roteiro de execução, também são uma das contribuições.

4.4 Perspectivas futuras

O Servidor de Execução, apesar da certificação dada pelo Servidor de Segurança não tem como verificar se o código que vai ser executado é um código seguro. É interessante que o Servidor de Compilação crie um certificado da biblioteca compartilhada criada, registrando junto ao Servidor de Segurança para que mais tarde o Servidor de Execução possa verificar sua validade.

Criação do conjunto de classes que darão suporte à geração de código em mapeando as funções de ICLang para C++.

Conforme Stearns (2002) o Servidor de Execução proposto também pode ser portado para JAVA utilizando a JNI (Java Native Interface). A JNI permite que o código Java executado numa Java Virtual Machine (JVM) possa executar bibliotecas escritas, por exemplo, em C++. Além disso, a Invocation API lhe permite embutir a JVM nas

aplicações. Com isso é possível aliar algumas facilidades do Java com a eficiência do código compilado.

É possível que com a utilização cada vez maior da linguagem Java, novas máquinas e mecanismos sejam criados, permitindo que a execução de um código Java seja próxima de um código compilado, tornando a utilização do Java aceitável.

REFERÊNCIAS

ANATEL. **Glossário Termos Técnicos de Telecomunicações**. Disponível em: <http://www.anatel.gov.br/AJUDA/GLOSSARIO/DEFAULT.ASP>. Acesso em: 21 mar. 2002.

ARMANINI, Kátyra Kowalski**. SeguraWeb – Framework RBAC para aplicações WEB**. 2002. 126f. Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Santa Catarina, Florianópolis.

BEGGS, Barry R. & BOONE, Joan. **Using XML and XSL for code generation**. Maio de 2001. Disponível em: <http://www.106.ibm.com/developerworks/xml/library/ibm-codegen/>. Acesso em: 15 jan. 2002.

COBB, Bradford et al. **AIX® Linking and Loading Mechanisms**. Disponível em: <http://www-1.ibm.com/servers/esdd/pdfs/aix_ll.pdf.>. Acesso em: 01 jun. 2002.

FENG, N., GANG, A., WHITE, T., and PAGUREK, B. Dynamic Evolution of Network Management Software by Software Hot-Swapping. In: **Proc. of the Seventh IFIP/IEEE International Symposium on Integrated Network Management**. Seattle. Maio 2001, pp. 63-76. Disponível em: <http://www.sce.carleton.ca/netmanage/publications.html>. Acesso em: 01 jun. 2002.

HENNING, Michi; VINOSKI, Steve. **Advanced CORBA Programming with C++.** USA: Addison-Wesley, 1999. ISBN 0-201-37927-9.

LU, Guido Carls Ying. **Validating Interaction Patterns of CORBA Based Network Management Systems.** NOMS 2000. Disponível em: <http://www.informatik.tu-darmstadt.de/VS/Publikationen/papers/noms2000.pdf>. Acesso em: 23 mar. 2002.

LUCA, Sandro D. **IDENTIFICAÇÃO DE PROBLEMAS NO FLUXO DE FATURAMENTO DAS OPERADORAS DE TELECOMUNICAÇÕES: UMA ABORDAGEM EMPREGANDO LÓGICA FUZZY E REGRAS DE PRODUÇÃO.** 2002. 108f. Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Santa Catarina, Florianópolis.

MICO is CORBA. MICO a *freely available* and *fully compliant* implementation of the CORBA standard. Disponível em: < http://www.mico.org>. Acesso em: 20 out. 2001.

OBJECT MANAGEMENT GROUP. **CORBA BASICS.** Disponível em: <http://www.omg.org/gettingstarted/corbafaq.htm>. Acesso em: 13 out. 2001.
OBJECT MANAGEMENT GROUP. **OMG IDL: Details .** Disponível em: <http://www.omg.org/gettingstarted/omg_idl.htm>. Acesso em: 15 out. 2001.

OBJECT MANAGEMENT GROUP. **ORB Basics.** Disponível em: <http://www.omg.org/gettingstarted/orb_basics.htm>. Acesso em: 15 out. 2001.

SIEGEL, Jon. **CORBA 3 Fundamentals and Programming.** USA: Wiley, 2000. 2 ed. ISBN 0-471-29518-3 **.**

STROUSTRUP, **Bjarne. The C++ Programming Language**. USA: Addison-Wesley. Special Edition, 2000. ISBN 0201700735. p. 418-420.

BIBLIOGRAFIA CONSULTADA

AGÊNCIA NACIONAL DE TELECOMUNICAÇÕES. **Aprova o Plano Geral de Metas de Qualidade para o Serviço Telefônico Fixo Comutado**. Resolução n. 30, de 29 de junho de 1998. Disponível em: <http://www.anatel.gov.br/Tools/frame.asp?link=/biblioteca/resolucao/1998/res_030_1998.pdf>. Acesso em: 22 jan. 2002.

DORB. DORB a *freely available* and *fully compliant* implementation of the CORBA standard to Delphi. Disponível em: < http://dorb.inec.ru/eng/>. Acesso em: 11 jun. 2002.

EGASHIRA, Tôru; KIRIHA, Yoshiak. **Management Middleware for Application Front-ends on Active Networks**. IEEE/IFIP Network Operations and Management Seminar (NOMS 2000), Hawaii, Abril.2000.

GEORGESCU, Cristian. **Code Generation Templates Using XML and XSL.** C/C++ Users Journal, v. 20, n. 1, 2002. Disponível em: <http://www.cuj.com/articles/2002/0201/0201toc.htm?topic=articles>. Acesso em: 22 fev. 2002.

GINGEL, Robert A. et al. **Shared Libraries in SunOS**. Proceedings of the USENIX Summer Conference, 1987. Disponível em: <http://citeseer.nj.nec.com/gingell87shared.html>. Acesso em: 07 set. 2002.

HJÁLMTÝSSON, Gísli; GRAY, Robert. **Dynamic C++.** USENIX Papers, 1998. Disponível em:<http://www.usenix.org/publications/library/proceedings/usenix98/full_papers/hjalmtysson/hjalmtysson_html/hjalmtysson.html>. Acesso em: 13 jan. 2002.

HO Wilson W.; OLSSON Ronald A. **An Approach to Genuine Dynamic Linking** 1991. Disponível em: <http://citeseer.nj.nec.com/ho91approach.html>. Acesso em: 07 set. 2002.

ISOTTON, Aaron. C++ dlopen mini HOWTO. Disponível em: <http://www.tldp.org/HOWTO/mini/C++-dlopen/>. Acesso em: 10 jan. de 2002.

LARMAN, Craig. **Utilizando UML e Padrões.** Tradução de Luiz Augusto Meirelles Salgado. São Paulo: Prentice-Hall, 1999. Título original: Applying UML and Patterns. ISBN 0-13-748880-7.

LI, H., et al. **System Designs for Adaptive, Distributed Network Monitoring and Control**. Disponível em: <www.isr.umd.edu/TechReports/CSHCN/2002/CSHCN_PhD_20022/CSHCN_PhD_2002-2.pdf.>. Acesso em: 23 mar. 2002.

M. .... **Advanced Event Filtering Approach For CORBA-Based Management Systems**. IEEE/IFIP Network Operations and Management Seminar (NOMS 2000), Hawaii, Abril.2000.

NISHIKI, Ken'ya; YOSHIDA, Kenichi; OOTA, Masataka**. Integrated Management Architecture based on CORBA.** IMT-2000 Base Transceiver Station Management System. IEEE/IFIP Network Operations and Management Seminar (NOMS 2000), Hawaii, Abril.2000.

NISHIKI, Ken'ya; YOSHIDA, Kenichi; OOTA, Masataka**. Integrated Management Architecture based on CORBA.** IMT-2000 Base Transceiver Station Management System. IEEE/IFIP Network Operations and Management Seminar (NOMS 2000), Hawaii, Abril.2000.

NORTON, James. **Dynamic Class Loading for C++ on Linux**. Linux Journal, 2002. Disponível em: <http://www.linuxjournal.com/categories.php?op=newindex&catid=173>. Acesso em: 10 jan. 2002.

ORFALI, Robert; HARKEY, Dan. **Client/Server Programing with JAVA e CORBA.** USA: Wiley, 1997. ISBN 0-471-16351-1.

ORR, Douglas B. et al.**Fast and Flexible Shared Libraries**.1993.USENIX Summer. Disponível em: <http://citeseer.nj.nec.com/orr93fast.html>. Acesso em: 07 set. 2002.

SILVA, Ricardo Pereira e. **Suporte ao desenvolvimento e uso de frameworks e componentes.** Tese (Doutorado em Ciência da Computação) – Universidade Federal do Rio Grande do Sul, Porto Alegre.

STEARNS, Beth. **Trail:** Java Native Interface. The Java Tutorial. Disponível em: <http://java.sun.com/docs/books/tutorial/native1.1/index.html>. Acesso em: 13 jan. 2002.

TAN, L., ESFANDIARI, B., PAGUREK, B. The SwapBox: A Test Container and a Framework for Hot-swappable JavaBeans. In: **Proc.of the WCOP (Workshop on Component-Oriented Programming)** workshop (at ECOOP 2001), Budapest, Hungria, Junho 19, 2001. Disponível em: <http://www.sce.carleton.ca/netmanage/papers/TanEtAlWCOP01.pdf>. Acesso em: 01 jun. 2002.

UNIVERSIDADE FEDERAL DE SANTA CATARINA. **NORMA ABNT PARA REFERÊNCIAS BIBLIOGRÁFICAS**. Disponível em: <http://bu.ufsc.br/framerefer.html>. Acesso em: 21 mar. 2002.

W3**.XML in 10 points**. <http://www.w3.org/XML/1999/XML-in-10-points.html>. Acesso em: 07 set. 2002.