UNIVERSIDADE DO ESTADO DO AMAZONAS - UEA

ESCOLA SUPERIOR DE TECNOLOGIA

ENGENHARIA DE COMPUTAÇÃO

EMILIANO CARLOS DE MORAES FIRMINO

WEBSOCKET PARA APLICAÇÕES WEB EM ERLANG

Manaus

2011

EMILIANO CARLOS DE MORAES FIRMINO

WEBSOCKET PARA APLICAÇÕES WEB EM ERLANG

Trabalho de Conclusão de Curso apresentado à banca avaliadora do Curso de Engenharia de Computação, da Escola Superior de Tecnologia, da Universidade do Estado do Amazonas, como pré-requisito para obtenção do título de Engenheiro de Computação.

Orientador: Prof. M. Sc. Jucimar Maia da Silva Júnior

Manaus

2011

Universidade do Estado do Amazonas - UEA
Escola Superior de Tecnologia - EST

Reitor:
José Aldemir de Oliveira
Vice-Reitora:
Marly Guimarães Fernandes Costa
Diretor da Escola Superior de Tecnologia:
Mário Augusto Bessa de Figueirêdo
Coordenador do Curso de Engenharia de Computação:
Danielle Gordiano Valente
Coordenador da Disciplina Projeto Final:
Raimundo Correa de Oliveira

Banca Avaliadora composta por: *Data da Defesa: 16/12/2011.*
Prof. M.Sc. Jucimar Maia da Silva Júnior *(Orientador)*
Prof. M.Sc. Raimundo Correa de Oliveira
Prof. M.Sc. Jorge Luiz Silva Barros

CIP - Catalogação na Publicação

F525w FIRMINO, Emiliano

Websocket para aplicações Web em Erlang / Emiliano Carlos de Moraes Firmino; [orientado por] Prof. MSc. Jucimar Maia da Silva Júnior - Manaus: UEA, 2011.

60 p.: il.; 30cm

Inclui Bibliografia

Trabalho de Conclusão de Curso (Graduação em Engenharia de Computação). Universidade do Estado do Amazonas, 2011.

CDU: _____

EMILIANO CARLOS DE MORAES FIRMINO

WEBSOCKET PARA APLICAÇÕES WEB EM ERLANG

Trabalho de Conclusão de Curso apresentado à banca avaliadora do Curso de Engenharia de Computação, da Escola Superior de Tecnologia, da Universidade do Estado do Amazonas, como pré-requisito para obtenção do título de Engenheiro de Computação.

Aprovado em: 16/12/2011

BANCA EXAMINADORA

Prof. Jucimar Maia da Silva Júnior, Mestre
UNIVERSIDADE DO ESTADO DO AMAZONAS

Prof. Raimundo Correa de Oliveira, Mestre.
UNIVERSIDADE DO ESTADO DO AMAZONAS

Prof. Jorge Luiz Silva Barros, Mestre.
UNIVERSIDADE DO ESTADO DO AMAZONAS

Agradecimentos

Agradeço a minha família por cuidar de mim durante todos esses anos.
Ao mestre Jucimar Júnior por me orientar no desenvolvimento deste projeto, ao Lanier Santos por preparar o modelo LaTeX que permitiu escrever está monografia.
A meus amigos Bruno Mendes, Clarice Santos, João Guilherme, George Nunes, Victor Kaleb, Yasmine Souza e a todos que me acompanharam nesses 5 anos de muita luta.

Resumo

Este trabalho descreve a implementação em Erlang do *internet-draft WebSocket* Hixie 76 do HTML5. A descrição do protocolo, o processo de modelagem, definição da *Application Programming Interface* e os resultados obtidos com os experimentos que demonstram o uso da API. A implementação foi estruturada em módulos que permitisse a máxima reutilização de código por outros desenvolvedores. O sistema foi estruturado em processo com interfaces de passagem de mensagens bem definidas que podem ser extendidas para uso em outros protocolos semelhantes ao *WebSocket*.

Palavras Chave: WebSocket, Protocolo de redes, HTML, HTTP, Erlang, Middleware

Abstract

This document describes a Erlang implementation of internet draft WebSocket Hixie 76 of HTML5. The Protocol description, solution design, application programming interface definition, experiments and results gather using the solution. The program was organized in modules which could permit others developers and research reuse the code created. The run-time execution was organized on lightweight process that uses a message passing interface API that could be use to implement similar protocols.

Key-words: WebSocket, Network Protocol, HTML, HTTP, Erlang, Middleware.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Códigos

Capítulo 1

Introdução

O surgimento de aplicações web interativas a partir da segunda década de 2000 foi permitido pela popularização de uma técnica de comunicação na web que se tornou conhecida como AJAX (*Asynchronous JavaScript and XML*).

O AJAX permite que um documento na web possa a qualquer momento requerer novas informações do servidor utilizando JavaScript para modificar seu conteúdo sem precisar recarregar todo o documento. A informação é formatada *de facto* usando XML ou JSON, as requisições usam o HTTP ou HTTPS. Entre as diversas aplicações que pode-se citar tem o Google Maps, o Facebook e o Twitter.

Porém o AJAX possui problemas. Ele não foi projetado para aplicações interativas e de comunicação bidirecional. Para obter atualizações constantes do servidor foi necessário o desenvolvimento de diversas técnicas que sobrecarregam o servidor com múltiplas conexões.

Uma solução para o problema é o *WebSocket*. O *WebSocket* é uma nova *Application Programming Interface* (API) e protocolo de comunicação entre cliente e servidor web. Ele faz parte da específicação do HTML5 e seu protocolo é desenvolvido pelo *Internet Engineering Task Force* (IETF).

O HTML5 é uma linguagem e um conjunto de APIs utilizadas no desenvolvimento de páginas web que começou a ser desenvolvido 2004 pelo *World Wide Web Consortium* (W3C) e *Web Hypertext Application Technology Working Group* (WHATWG). O HTML5 promete revolucionar o desenvolvimento na web e foi adotada para o desenvolvimento de aplicações para celulares, *tablets*, *smart tv* e *desktop*.

O *WebSocket* disponibiliza uma conexão entre cliente e servidor bidirecional, em conformidade com a política de segurança da web, permite reduzir o consumo de rede, reduzir a latência na comunicação e simplificar o gerenciamento das conexões.

A linguagem de programação do servidor é muito importante para garantir a robustez, escalabilidade e integridade da aplicação. Uma boa linguagem facilita o desenvolvimento, esclarece o funcionamento da aplicação e restringi os efeitos colaterais que possam ocorrer. A linguagem que foi escolhida para desenvolver o projeto foi o Erlang.

Erlang é uma linguagem funcional de código aberto criada no laboratório de Ciência da Computação da Ericsson para o desenvolvimento de aplicações de telecomunicação. Aplicações caracterizadas por um alto número de transações simultanêas e alta disponibilidade.

1.1 Objetivos Gerais

Implementar uma biblioteca *WebSocket* em Erlang que permita desenvolver aplicações capazes de se comunicar diretamente com aplicações em HTML5.

1.1.1 Objetivos Específicos

Implementação da versão do protocolo *WebSocket Draft-Hixie* 76. Suporte a comunicação segura usando *Transport Layer Security* (TLS). Desenvolver uma API para codificação e decodificação de JSON em Erlang. Construir aplicações protótipo para demonstrar a capacidade da biblioteca.

1.2 Justificativa

Aplicações web interativas usando AJAX utilizam tecnologias complexas que consomem maior banda de rede, recursos dos servidores. O *WebSocket* permite resolver estes problemas sem o uso de *plugins* proprietários.

O *WebSocket* é uma nova tecnologia com possibilidade de ajudar no desenvolvimentos aplicações interativas que de antes só poderiam ser feitas usando *plugins*. Mas para isso é necessário uma linguagem de servidor capaz de desenvolver aplicações para milhares de

usuários simultâneos, escalável, robusta e comprovada na prática. Erlang atende esses requisitos.

A biblioteca em Erlang vai permitir o desenvolvimento de aplicações web interativas com menor complexidade e acessível a grande quantidade de usuários.

1.3 Metodologia

Leitura e análise da especificação do protocolo *WebSocket* da IETF. Definir os requisitos da biblioteca e da API seguindo a API descrita pelo W3C e socket similares do Erlang.

Desenvolvimento incremental da biblioteca. Cada versão deverá passar por um conjunto de testes para validar o protocolo. Ao final de cada etapa foi desenvolvido uma aplicação-protótipo para validar os recursos usando Orientação a Objeto em JavaScript no cliente e servidor em Erlang.

Ambiente de desenvolvimento na plataforma Linux, editor VIM, gerenciamento de versão usando GIT, Profile e Debug do servidor usando as ferramentos do Erlang, Debug do cliente usando Firebug do Firefox 4.

1.4 Organização da Monografia

O capítulo 2 é uma breve introdução à linguagem Erlang com objetivo de apresentar suas principais características, tipos de dados e outros detalhes significativos que influenciaram o trabalho.

O capítulo 3 aborda o protocolo *WebSocket* e seu funcionamento.

O capítulo 4 descreve a modelagem da solução do protocolo *WebSocket* em Erlang. Os requisitos, divisão em módulos e organização em processos.

O capítulo 5 detalha de implementação em Erlang. A API produzida e detalhes dos principais módulos.

O capítulo 6 são apresentadas os programas desenvolvidos usando o módulo *gen_ws*. Primeiro um cliente de *echo* capaz de se conectar com servidor de *WebSocket* de terceiro. Segundo um o servidor de chat simples. Por último uma aplicação composta de um servidor

erlang, cliente HTML5 e comunicação usando o formato JSON no qual uma tela de desenho (*canvas*) é compartilhada que foi chamado MultiDot.

O capítulo 7 apresenta as conclusões obtidas e os trabalhos futuros. Seguido por três apêndices contendo o código fonte produzido durante o trabalho.

Capítulo 2

Linguagem Erlang

Erlang foi criada pela Ericsson para desenvolvimento de sistemas de telecomunicação. Sistemas caracterizados por alta disponibilidade, tolerante a falhas, grande quantidade de requisições simultâneas. Linguagem de código fonte aberto e com suporte a várias plataformas.

Um programa Erlang é constituído de processos que executam em uma máquina virtual. Os processos do Erlang são leves e fáceis de criar. A máquina virtual permite que milhares executem de modo concorrente sem depender do gerenciamento do sistema operacional.

Um programa escrito em Erlang é organizado em módulos que contém um conjunto de funções que são visivéis ou não para outros módulos. As funções manipulam estruturas, processos, entrada e saída de informações.

As principais características da linguagem são:

1. Linguagem Funcional: Permite modularizar e abstrair a manipulação de informações pelo uso de funções que descrevem vagamente um modo de determinar o valor de saída a partir dos parâmetro de entrada por meio do cálculo λ, através de regras que fazem a correspodência entre entrada e saída [Rabhi and Lapalme1999].

2. Código compilado: Programas em Erlang utilizam uma representação intermediária chamada *byte code* das intruções a serem executadas na máquina virtual. O código fonte compilado permite bom desempenho e portabilidade entre plataformas por meio da máquina virtual.

3. Orientada a concorrência: Elementos em execução são processos, isolados entre si, comunicam-se por passagem de mensagem, limitam propagação de erros, fáceis de criar ou destruir, leves no consumo de memória o que permitem milhares deles serem criados e executados de modo concorrente.

4. Pacotes: são estruturas hierárquicas que delimitam um conjunto de módulos e outros pacotes.

5. Módulos: Delimitam um conjunto de funções que manipulam estruturas, listas, números e outros processos. As funções são visíveis fora do módulo quando declaradas explicitamente.

6. Funções: são rotinas que recebem parametros e sempre retornam um valor. Funções são definadas utilizando indução e definições de casos. A partir de casos mais simples que podem ser verificados são agrupados novos casos de modo a cobrir todas os possíveis casos. Algoritmos são definidos principalmente por recursão. Na função fatorial no código 2.0.1, declaração um caso base para o fatorial de zero, para os casos em que N é maior que zero usa-se recursão para determinar o valor do fatorial e para qualquer outro cenário um erro é gerado.

```
1 fatorial(0) ->
2         1;
3 fatorial(N) when N > 0 ->
4         N * fatorial(N-1);
5 fatorial(_) ->
6         error.
```

Código 2.0.1: *Função fatorial em Erlang.*

7. Expressões Nativas: Erlang possui expressões para decisão, atribuição, operações matemáticas, além de um conjunto próprio para receber e enviar mensagens para outros processos.

8. Variáveis: Diferente de linguagens imperativas, variaveis em Erlang são de atribuição única. Não podem ser reutilizadas em um escopo o que inviabiliza a implementação de laços de repetição. As varíaveis em Erlang são fracamente tipadas, não é necessário

indicar que tipo de dado ela irá receber, porém para ser utilizada requerem que sejam inicializadas com um valor.

9. *Patter Match*: Poderosa funcionalidade que permite escrever programas concisos e legíveis [Cesarini and Thompson2009], permite:

   (a) Atribuir valores a variáveis, na forma *Pattern = Expression.*

   (b) Controlar fluxo de execução de programas.

   (c) Extrair valores de dados compostos.

2.1 Tipos de dados

São classificados em dois grupos principais os tipos primitivos e os compostos.

Os tipos primitivos são inteiro, ponto flutuante, átomos , referências, binários, pid (*Process Identifier*), ports e *Funs* (Funções anônimas). Os números são representados como inteiros de precisão infinita ou ponto flutuante.

Átomos representam idéias e conceitos. São utilizados para comparação de igualdade e clarificam conceitos abstratos como *verdadeiro*, *falso*, *cor* e *vazio*.

Referências são apelidos registrados para um processo o que permite a qualquer um enviar mensagens para o processo sem conhecer seu identificador de processo.

Identificadores de processo ou *Process Identifier* (pid) é um identificador único à um processo que permite a este poder receber e enviar mensagens de e para outros processos.

Binários são cadeias de *bytes* que representam dados em memória. São muito utilizados para manipular sequencias de *bytes*, facilita a interpretação e formatação de arquivos binários ou de mensagens transmitidas pela rede na qual a posição e valores dos bits possuem um significado específico definido em um protocolo.

Ports são pontos de comunicação com o exterior entre o processo e seu ambiente.

fun são funções declaradas em tempo de execução.

Tipos compostos são aglomerações dados de modo para formar um elemento mais complexo. São dois tipos principais listas e tuplas. a lista é uma estrutura sequencial de elementos, enquanto que a tupla é um bloco de elementos assessíveis usando *pattern match.*

Para facilitar o entedimento da linguagem, as funcionalidades da linguagem serão explicadas no código 2.1.1.

```
1  Erlang R14B04 (erts-5.8.5) [smp:2:2] [rq:2] [async-threads:0]
2
3  Eshell V.5.8.5 (abort with ^G)
4  1>100.
5  100
6
7  2>16#ff.
8  255
9
10 3>1000.0.
11 1.0e3
12
13 4>um_atomo.
14 um_atomo.
15
16 5><<"binario">>.
17 <<"binario">>
18
19 6>F = fun() -> funcao_anonima end.
20 #Fun<erl_eval.20.213243>
21
22 7>F().
23 funcao_anonima
24
25 8>[101,114,108,97,97,110,103].
26 "erlang"
27
28 9>{101,"carro", tuple}.
29 {101,"carro",tuple}
```

Código 2.1.1: *Exemplos de tipos de dados Erlang usando interpretador* Eshell.

Nas linha 4 e 7 ocorre a declaração do tipo inteiro. Inteiro em Erlang possui precisão infinita o que permite representar qualquer valor inteiro existente. Erlang também permite declarar um número binários, hexadecimais, octais entre outras representações a partir da declaração explicita de sua base como prefixo e separado por `#`. Como no segundo comando no qual é declarado o numero FF de base 16 é interpretado como 255 em base decimal.

Na linha 10 é declarado um número real, ou conhecido por ponto flutuante, no qual é possível representar números fracionários.

Na linha 13 é declarado um tipo de dado chamado atómo, átomos começam com letra minúscula ou entre aspas simples. Um atómo é um identificador unico que representa uma idéia ou conceito. Muito utilizado para representar idéias abstratas como sucesso, falha, nomes, verdadeiro, falso.

Na linha 16 é declarado um tipo binário, diferente de outras linguagens Erlang possui esse tipo de dado específico para representar um fluxo de *bytes*, além de suportar diversas

operações de manipulação e conversão para esse formato.

Na linha 19 ocorre a declaração dinâmica de um função usando a sintaxe *fun*, os parametros dentro do parentes, escopo da função e a palavra reservada *end*. A função é armazenada na varíavel F que após atribuída não se pode modificar. Assim F pode ser usado para chamar a função com exemplificado na linha 22.

Na linha 25 é declarado uma lista. Listas são implementadas usando lista encadeada. Não permitem acesso aleatórioores, apenas sequencial usando manipulação de lista. Existe uma sintaxe própria para geração de listas e são muito utilizadas para processar fluxos de informação. A linguagem Erlang não possui um tipo exclusivo para representar string, por isso é utilizada um a lista de numéros que lembra o array de caracteres de linguagem de baixo nível com o C.

Na linha 28 é a declarado uma tupla. Tuplas são estruturas com número definido de elementos que permite agrupar um ou mais elementos.

Comandos em Erlang utilizam uma sintaxe que utiliza os caracteres ".", "," e ";" para separar os comandos e delimitar funções. a virgula é usada para indicar que existem mais instruções a executar. O ponto e vírgula é utilizado para delimitar o fim de um bloco de instruções porém indicando que existe alternativas na execução muito usado quando define múltiplos comportamentos de uma função. O ponto final indica o fim de um bloco de instruções.

2.1.1 Listas

Listas em Erlang são estruturas na qual itens de qualquer tipo são armazenados. O acesso a esses elementos é sequencial. São listas encadeadas de elementos. O acesso aos elementos ocorre por operações de lista ou por *pattern match*.

Listas são utilizadas para processamento de fluxo (*stream*) como *strings* e vetores, onde a mesma operação é realizada em todos os elementos e não se conhece o número total de elementos a *priori*. Exemplos das principais operações em lista são descritas no código 2.1.2.

- Lista vazia: `[]`
- Lista com um item: `[item]`

```
1  match() ->
2      [First, Second, Third] = [e1, "e2", []].
3      % First = e1
4      % Second = "e2"
5      % Third = [] ou ""
6
7  head(List) ->
8      [Head|_DiscardTail] = List,
9      Head. % Retorna a cabeca da lista
10
11 add_head(List, Item) ->
12     Head = Item,
13     [Item|List]. % Adicionar ao inicio da lista
14
15 rm_head(List) ->
16     [_DiscardHead|Tail] = List, % Remove cabeca,
17     Tail. % Retorna o resto da lista
18
19 concat(List1, List2) ->
20     List1 ++ List2.
21
22 iterate([Head|Tail]) -> % Exemplo de processamento em Lista:
23     execute(Head),      % Para cada item da lista executa uma funcao
24     iterate(Tail);
25 iterate([]) ->
26     ok.                 % Ate a lista ser finalizada
```

Código 2.1.2: *Principais operações em listas.*

- Lista com dois itens: `[item1,item2]`

2.1.2 Tuplas

Tuplas são semelhantes a listas porém com número definido de itens, itens podem ser qualquer tipo de dado Erlang, similares a *struct* da linguagem C. Possibilitam agrupar uma informação de maneira estruturada, permite acesso a elemento de modo rápido e eficiente; utilizadas para representar estrutura de dados. São delimitadas por chaves `{}` com cada elemento separado por vírgula. O código 2.1.3 exemplifica o uso de tupla para agrupar elementos e uso do *pattern match* para extraílos.

- Tupla vazia: `{}`
- Tupla com um item: `{item}`
- Tupla com dois itens: `{item1,item2}`

```
1  Tuple = {e1, "e2", []},
2  {Item1, Item2, Item3} = Tuple. %Match de Tuplas
3
4  % Equivale:
5  % Item1 = e1
6  % Item = "e2"
7  % Item = []
```

Código 2.1.3: *Exemplo de tupla em Erlang.*

2.2 Módulo em Erlang

Programas em Erlang são definidos usando módulos de modo equivalente a classes em linguagens orientadas a objeto. Os módulos em Erlang são arquivos de texto com a extensão *.erl*.

```
1  -module(example).
2  -author('emiliano').
3  -vsn(1).
4
5  -export([soma/1, soma/2]).
6
7  soma(V1, V2) ->
8      V1 + V2.
9
10 soma([H|T]) when is_number(H) ->
11     [H|T] = List,
12     H + soma(T);
13 soma([H|T]) ->
14     soma(T);
15 soma([]) ->
16     0.
```

Código 2.2.1: *Exemplo de um módulo em Erlang*

Todo módulo em Erlang deve conter a diretiva **-module** no qual é definido o nome do módulo. Possuem o mesmo nome que o arquivo. O módulo *example* é armazenado no arquivo *example.erl*, a compilação gera o arquivo *example.beam*.

Um módulo apresenta um interface na qual funções podem ser utilizadas por outros módulos usando a diretiva export na qual são declaradas a função e o aríete (número de paramêtros) dela. Funções de aríete diferente são consideradas distintas em Erlang. O código 2.2.1 descreve um tipico módulo Erlang, consite em um cabeçalho contento a definição das diretivas seguidas por definição de funções.

Cada função é discriminada pelo seu nome, um átomo, e pelo seu número de paramêtros conhecido como aríete.

Uma função é como um contrato que define que para determinado tipo de parâmetro será retornado tal valor. A função soma de um parametros aceita apenas listas como parametro. Esta garante retorna zero para uma lista vazia ou a soma dos números que estão contidos nessa lista. Como não existe laços de repetição em linguagens funcionais é utilizado o chamadas recursivas.

2.2.1 *Built In Functions*

São funções que são nativas do Erlang e não precisam declarar o módulo de origem da rotina quando utilizadas. Além disso podem ser utilizadas em *guardas* nas funções, case, receive, if e outras estrutura de decisão para filtrar a chamada a rotina.

2.3 Processos

Processos em Erlang são unidades de execução de instruções que estão isoladas entre si e comunicam–se por passagem de mensagem.

No código 2.3.1 é apresentado um exemplo de criação de processo e passagem de mensagens. A função *soma* do módulo *process* que daqui em diante será referenciada como *process:soma* implementa a rotina de cálculo usando o módulo do código 2.2.1.

```
1  -module(process).
2  -author('emiliano').
3  -vsn(1).
4
5  -export([soma/1]).
6
7  soma(List) ->
8      Me = self(), % Pid que identifica o processo atual
9
10     %Cria um processo para realizar a soma dos elementos da lista
11     Pid = spawn(process, process_soma, [List, Me]),
12
13     receive
14         {Pid, Result} -> % Aguarda resposta do processo
15                 Result
16     end,
17     Result. % Valor de Retorno
18
19 process_soma(List, Owner) ->
20     Result = example:soma(List),
21     Owner ! {self(), Result}. % Envia a tupla ao processo Owner
```

Código 2.3.1: *Uso de processos em Erlang.*

Porém é utilizado um outro processo para executar o cálculo. Esse processo é criado usando a função nativa, *Built in funciton (BIF)*, *spawn* na qual é indicado que a função

process:process_soma(L, I) deve ser executada em um novo processo. O parametro L é a lista de valores e I é o Identificador de Processo para onde enviar o resultado.

A BIF *receive* aguarda receber a mensagem de resposta do processo. A mensagem de resposta é enviada usando o operador **!** na linha 21, representa a operação de envio de mensagem para o identificador do processo *Owner*.

Capítulo 3

Protocolo WebSocket

O protocolo *WebSocket* é desenvolvido pela *Internet Engineering Task Force* (IETF). A especificação utilizada no projeto é em conformidade com o *draft-hixie-thewebsocketprotocol-76* [Hickson2010b] publicado em 6 de maio de 2010 e editado por Ian Hickson do Google[1].

O *draft-hixie-thewebsocketprotocol* foi o primeiro *draft* do protocolo *WebSocket* publicado em janeiro de 2009 [Hickson2009], revisado periódicamente e modificado de acordo com os resultados das discussões no grupo de trabalho. Foram 76 versões no período de janeiro de 2009 até maio de 2010 quando um novo draft foi criado.

O *draft-hybi-thewebsocketprotocol* é o segundo *draft* do protocolo *WebSocket* utilizou como ponto de partida o *draft-hixie-thewebsocketprotocol-76* no qual foi renomeado para *draft-ietf-hybi-thewebsocketprotocol-00* e publicado em 23 de maio de 2010 em um novo grupo de trabalho chamado *Hypertext-Bidirectional* (Hybi) e editado por Ian Fette do Google.

Em outubro de 2011, o *draft-ietf-hybi-thewebsocketprotocol* se encontrava em sua 17ªversão, publicada em 30 de setembro de 2011. Esta versão é incompatível com o *draft-hixie-thewebsocketprotocol-76*.

3.1 Descrição do Protocolo

O protocolo *WebSocket* permite uma comunicação bidirecional entre um navegador e um servidor de Internet sobre o modelo de segurança baseado na origem, o protocolo consiste:

1. Início da conexão

[1]Google: http://www.google.com/

2. Negociação entre cliente e servidor chamada *handshake*.

3. Troca assíncrona e bidirecional de mensagens.

4. Encerramento da conexão.

3.1.1 Iniciando Conexão

O *WebSocket* é um protocolo pertecente a camada de aplicação sobre o modelo *TCP/IP*, representado na figura 3.1.

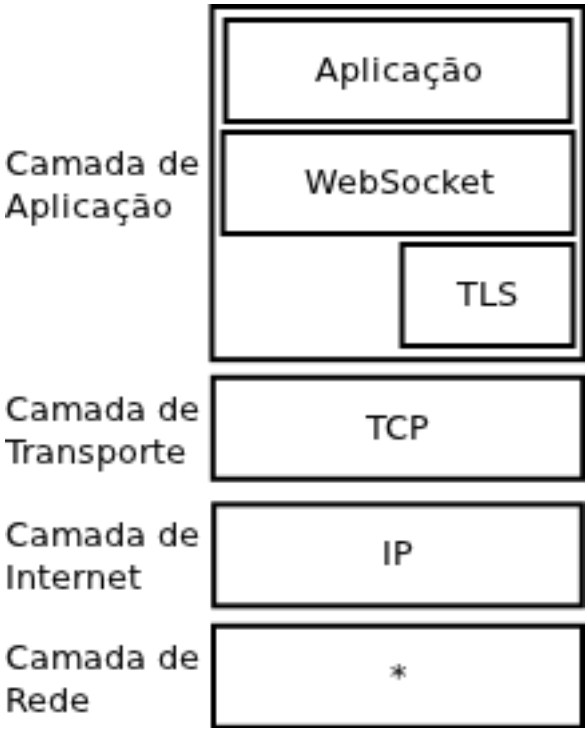

Figura 3.1: Pilha de protocolo.

O protocolo possui dois modos de comunicação. O modo seguro utiliza o *Transport Layer Security* (TLS) e o modo não seguro utiliza o *Transmission Control Protocol* (TCP). Um servidor de aplicação *WebSocket* é identificado na Internet por meio de um *Uniform Resource Locator* (URL).

$$scheme://domain[:port]absolute\_path$$

- **scheme**: Determina o tipo de protocolo e aplicação a ser utilizado. No *WebSocket* são suportados dois *schemes*.

1. *ws*: Conexão *WebSocket* sem criptografia usando TCP.
2. *wss*: Conexão *WebSocket* com criptografia usando TLS.

- **domain**: Identifica o computador servidor do *WebSocket* na Internet. Pode ser um endereço IP ou um nome de domínio a ser traduzido por um *Domain Name Server* (DNS).
- **port** (opcional):
  É um número de 16bits sem sinal, zero a 65.535, que identifica uma conexão em um computador que está ligado a Internet.
  Quando omitido o valor padrão é 80 para *ws* ou 443 para *wss*. Mesmos valores utilizados no protocolo HTTP para permitir compatilidade com servidores web.
- **absolute_path**: Identifica o caminho no servidor para aplicação. Obrigatório o caminho completo para o recurso.

3.1.2 *Handshake*

O processo de *handshake* é iniciado pelo cliente ao enviar um cabeçalho de requisição HTTP que contém um desafio. O desafio é um problema que só um servidor que implemente a versão correta do protocolo pode responder. O servidor deve analisar o cabeçalho, solucionar o desafio e responder a requisição para estabelecer a conexão *WebSocket*. O código 3.1.1 mostra o exemplo de cabeçalho de enviado pelo cliente.

Iniciando *Handshake*

```
1  GET /demo HTTP/1.1
2  Host: example.com
3  Origin: http://example.com
4  Connection: Upgrade
5  Upgrade: WebSocket
6  Sec-WebSocket-Key1: 18x 6]8vM;54 *(5:  {   U1]8  z [  8
7  Sec-WebSocket-Key2: 1_ tx7X d  <  nw  334J702) 7]o}‘ 0
8  Sec-WebSocket-Protocol: sample
9
10 ^Tm[K T2u
```

Código 3.1.1: *Exemplo de cabeçalho enviado pelo cliente.*

A linha 1 é chamada de *request line*, descrita na seção 5.1 do *RFC 2616* [Group1999]. É composta de três elementos. O método, o *Path* e a versão do HTTP. No WebSocket

o método é obrigatoriamente **GET** e a versão do HTTP/1.1. O *Path* é utilizada para identificar unicamente uma aplicação em um servidor com múltiplas aplicações.

As linhas 2 a 8 são chamadas de *message-header* e são descrita na seção 4.2 da *RFC 2616* [Group1999]. Cada *message-header* é uma associação entre nome e valor separado por ": ". Servem para definir parâmetros na negociação do *handshake* WebSocket e a ordem dos *message-headers* é aleatória.

A linha 2 é nome do domínio onde a aplicação *WebSocket* está hospedada. É possivel um servidor de Internet possuir múltiplos nomes de domínio apontando para ele. Cada domínio pode receber um tratamento diferente pelo servidor como se cada um estivesse hospedado em uma máquina diferentes. É um campo obrigatório.

A linha 3 indica qual foi a página web onde está hospedada aplicação que utiliza o *WebSocket*. Permite que o servidor discrime e recuse conexões de página que não considere confiáveis. É um campo obrigatório.

As linhas 4 e 5 permita a compatibilidade com o protocolo HTTP. A linha 4 informa ao servidor que o cliente está propondo um mudança no protocolo usado na conexão. A linha 5 informa que a mudança é para o protocolo *WebSocket*. Caso o servidor não suporte, a proposta será recusada. São campos obrigatórios.

As linhas 6 e 7 são duas partes das três que compoẽm o desafio ao servidor. Esse desafio garante que o servidor processou o cabeçalho antes de responder. A terceira parte do desafio se encontra logo após fim do cabeąlho. Mais gerado a resposta do desafio no próximo capítulo. São campos obrigatórios.

A linha 8 pode conter um ou mais subprotocolos. Subprotocolos são protocolos de comunicação utilizados na camada superior da aplicação que utiliza o *WebSocket* por exemplo um chat que possua protocolos diferentes como *ICQ* e *IRC* usando um mesmo servidor. É um campo opcional.

A linha 9 é uma linha que delimitar o fim do cabeçalho, Ela é seguida por 8 *bytes* aleatórios que compõem o ultimo elemento do desafio proposto ao servidor. É obrigatória.

Os *message-header* diferentes dos citados acima são ignorados na negociação do protocolo.

Resolvendo Desafio

Para solucionar o desafio é necessário obter três informações. Sec-WebSocket-Key1, Sec-WebSocket-Key2 e os últimos 8 *bytes* da requisição.

Cada Sec-WebSocket-Key contém um número cifrado. Para descifrar são necessárias duas informações o valor numerico da concatenação dos algarismo presentes no Sec-WebSocket-Key e o número de espaços existentes. Cada etapa é apresentada na figura 3.2.

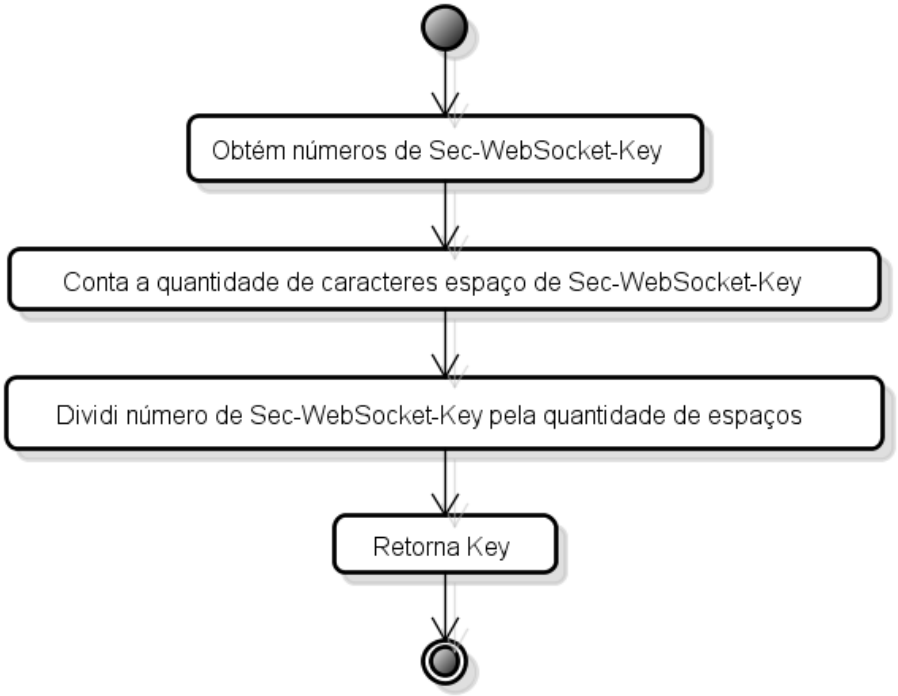

Figura 3.2: Algoritmo de decifragem da *Sec-WebSocket-Key*.

Divide-se o valor obtido na concatenação pelo número de espaços para obter o valor decifrado. O valor decifrado é um número inteiro.

Utilizando o valor do Sec-WebSocket-Key1 no código 3.1.1, "`18x 6]8vM;54 *(5: { U1]8 z [ 8`" , como exemplo, concatenação dos algarismo obtemos o valor 1868545188, o número de espaços é 12. O valor decifrado é obtido dividindo 1868545188 por 12. Obtemos o valor 155712099.

Obtido os números decifrados de Sec-WebSocket-Key1 e Sec-WebSocket-Key2 e os úl-

timos oito *bytes* da requisição são necessários para gerar a solução. os 4 primeiros *bytes* é o valor decifrado da chave 1 como inteiro de 32bits, os 4 seguintes da chave 2 no mesmo formato da chave 1 e os últimos 8 são os mesmos obtidos no fim da requisição. Aplica-se o algoritmo MD5 nesses 16 *bytes* para gerar um novo binário de 16 *bytes* que é a solução ao desafio proposto na requisição. A figura 3.3 representa as etapas para solucionar o desafio, o algoritmo em Erlang está localizado no módulo *hixie 76* do pacote *wslib*.

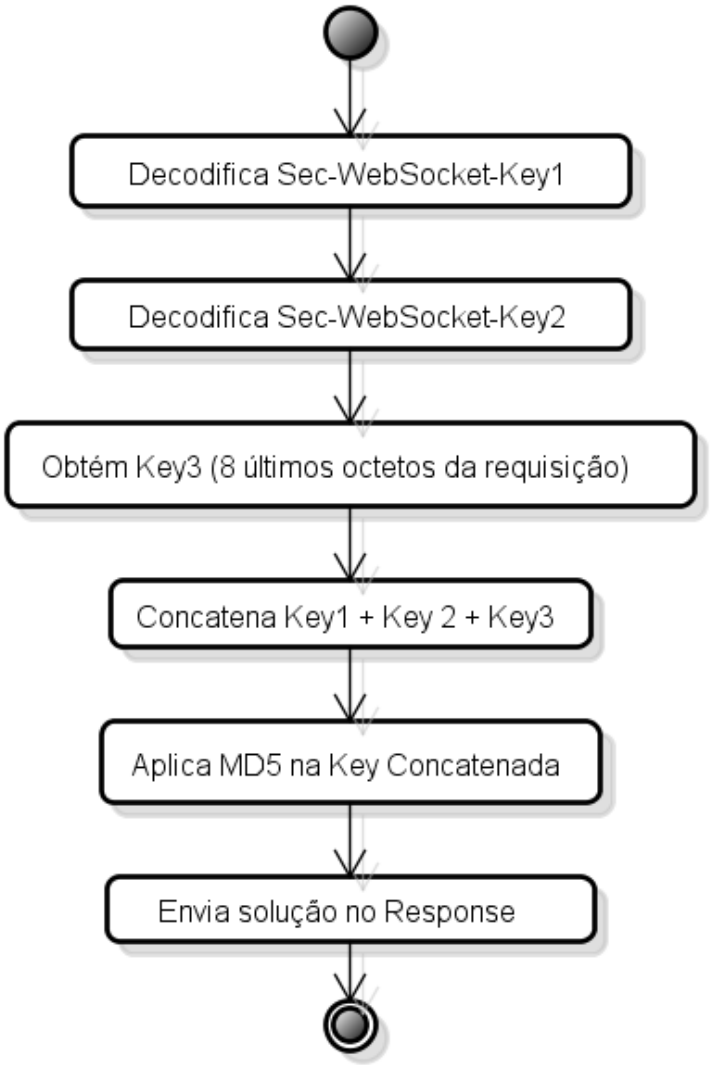

Figura 3.3: Algoritmo que soluciona o desafio.

Aplicando no exemplo do código 3.1.1. As chave são 8293092023, 1733470270 e

`^Tm[K T2u` em representação hexadecimal são EE-4E-8A-B7, 67-52-A8-3E, 54-6D-5B-4B-20-54-32-75. O desafio é a concatenação das três chaves, EE-4E-8A-B7-67-52-A8-3E-54-6D-5B-4B-20-54-32-75, que possui 16 *bytes* de comprimento. A solução é obtida aplicando o algoritmo MD5 no desafio. aplicado no exemplo é obtido o valor `fQJ,fN/4F4!~K~MH` em ASCII,

Finalizando *Handshake*

```
1  HTTP/1.1 101 WebSocket Protocol Handshake
2  Upgrade: WebSocket
3  Connection: Upgrade
4  Sec-WebSocket-Location: ws://example.com/demo
5  Sec-WebSocket-Origin:   http://example.com
6  Sec-WebSocket-Protocol: sample
7
8  fQJ,fN/4F4!~K~MH
```

Código 3.1.2: *Exemplo de cabeçalho enviado pelo servidor*

A linha 1 é chamada de *status-line*, descrita na seção 6.1 da *RFC 2616* [Group1999]. É composta de três informações. Versão do HTTP, *status code* e *reason phrase*.
O *status code* é um número de três digitos que indica o resultado da requisião. No caso do *WebSocket* o valor esperado é 101 indicando sucesso na mudança do protocolo.
O *reason phrase* é uma pequena descrição textual que obrigatóriamente deverá ser "WebSocket Protocol Handshake".

A linha 2, 3 e o código 101 na *status-line* confirmam que o servidor concordou em migrar para conexão *WebSocket*.

A linha 4 é a *WebSocket* URL utilizada para gerar a requisição ao servidor.

A linha 5 é a URL da página de origem da requisição *WebSocket*.

A linha 6 somente existirá caso o campo Sec-WebSocket-Protocol tenha sido enviado na requisição. O campo contém um único subprotocolo dentre os suportados pela aplicação cliente.

A linha 7 é uma linha em branco que indica o fim do cabeçalho, seguida por 16 *bytes* que é a solução ao desafio proposto na requisição.

Os valores de Sec-WebSocket-* e a resposta ao desafio permitem ao cliente validar que a resposta do servidor foi gerada seguindo os paramêtros enviados na requisição. Caso diferir dos valores esperados a conexão é encerrada.

3.1.3 Troca de Mensagens

Após o *handshake* o *WebSocket* está pronto para o uso. Ambas as partes da conexão podem enviar e receber mensagens. As mensagens são de dois tipos: texto codificado em utf8 ou binário. Sendo que o modo binário não está disponível para uso no *draft-hixie76*.

As mensagens são transmitidas através da rede usando frames que delimitam onde começa e termina.

3.1.4 Encerrando Conexão

A conexão é encerrada (*graceful close*) quando é enviado por uma das partes da conexão uma mensagem binária vazia que é usada para sinalizar o encerramento da conexão. Ambas as partes devem em seguida encerrar a conexão.

3.2 Considerações do Protocolo

O protocolo *WebSocket* pretende manter a compatibilidade com os servidores de Internet. Para permitir que estes possam o utilizar apenas utilizando *plugins* como os utilizados na geração dinâmica de documentos como PHP ou ASP.

Os *proxies* que servem para restringir a capacidade de acesso de uma rede a Internet costumam bloquear a maioria das portas com excessão da utilizada no HTTP e HTTPS. O uso dessas portas pelo *WebSocket* vai permitir o protocolo ser acessivel navegadores *Web* que estejam atrás de um *proxy*, *proxies* costuman bloquear outras portas com excessão das portas utilizadas no HTTP.

O *WebSocket* utiliza quadros delimitados por marcadores para enviar mensagens, portanto não é possivél acessar a mensagem enquanto o quadro não for completado. Segundo Ian Hickson esse modo de transmitir as mensagens foi escolhida em detrimento de outros porque tem menor possibilidade de os usuários e desenvolvedores da API de inserir falhas de segurança, dificulta a subversão do protocolo, além de que no ambiente do navegador de Internet é necessário discretizar o evento de chegada de mensagem para que o JavaScript possa tratá-lo [Hickson2010a].

Capítulo 4

Modelagem do Projeto

A partir dos documentos de expecificação do *draft-hixie* e *draft-hybi* foram identificados os casos de uso, a sequência de operações para cada caso de uso, os objetos manipulados o que permitiu identificar os módulos do sistema.

4.1 Modelagem dos Casos de Uso

O *WebSocket* é um tipo de *Socket* de rede orientado a conexão. O Socket fornece a capacidade de enviar e receber informações entre duas aplicações através de uma rede como a Internet.

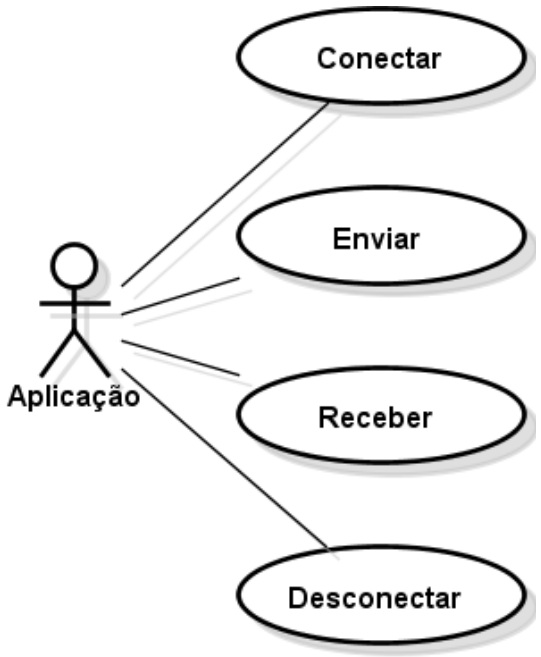

Figura 4.1: Caso de uso do *WebSocket*.

Uma aplicação *WebSocket* possui quatro casos de uso principais descritos na figura 4.1.

- Conecta, estabelece uma conexão entre cliente e servidor.
- Envia mensagem para a outra parte.
- Recebe mensagem da outra parte.
- Desconecta, encerra uma conexão.

4.2 Protocolo

O *WebSocket* é um protocolo cliente-servidor no qual o cliente inicia a conexão, seguida por uma negociação chamada *handshake* na qual é estabelecida uma conexão onde ambos podem trocar mensagens. Como descrito na figura 4.2.

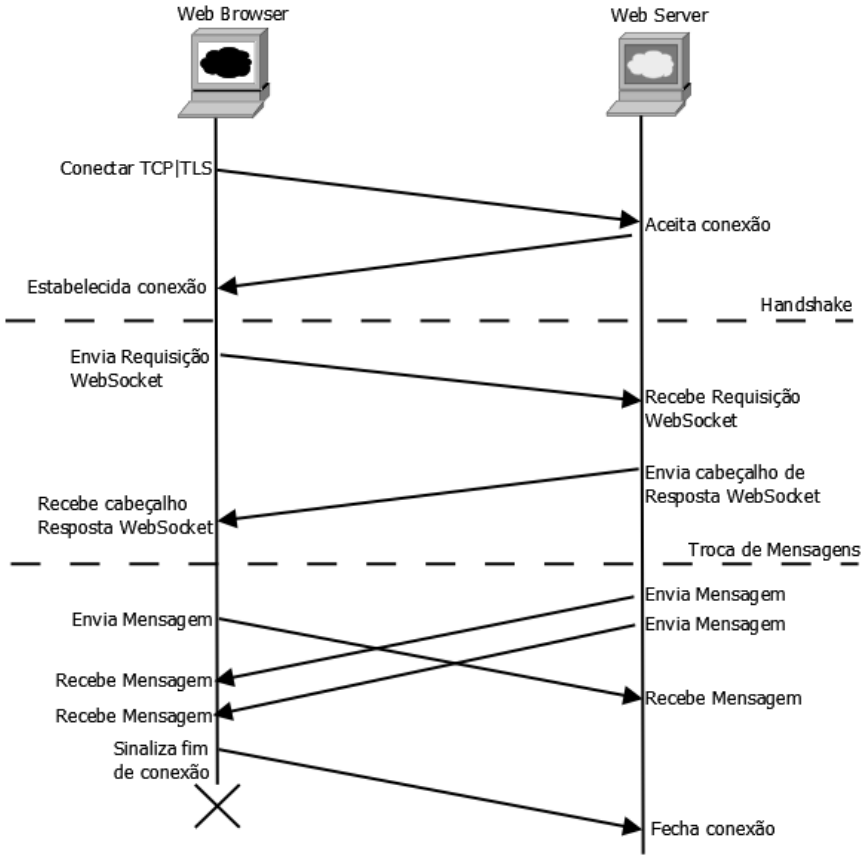

Figura 4.2: Fluxo de mensagens do protocolo *WebSocket*.

O cliente estabelece uma conexão TCP ou TLS, em seguida envia um requisição ao servidor para estabelecer a conexão *WebSocket*, aguarda a resposta, valida os parâmetros da resposta para estabelecer a conexão.

O servidor aguarda novas conexões, quando uma nova conexão é criada o servidor aguarda a requisição do cliente, cria uma resposta seguindo os parâmetros definidos na requisição e envia a resposta.

O cliente e o servidor podem enviar e receber mensagens entre si, para isso o protocolo utiliza *quadros*. Os quadros permitem que as mensagens possam ter qualquer tamanho variável e o protocolo gerencie o fluxo de dados da rede.

O cliente ou o servidor podem encerrar unilateralmente a conexão. A parte que vai encerrar a conexão envia um sinal para a outra parte e encerra a conexão, a outra parte recebe o sinal e também encerra a conexão.

4.3 Máquina de estados

A representação usando máquina de estados nas figura 4.3 e 4.4 permitem compreender melhor o ciclo de vida da conexão WebSocket. O comportamento é definido em função de eventos que produzem mudança nesse comportamento conhecido como estado. O inicio é um circulo preenchido e o fim um com anel ao redor.

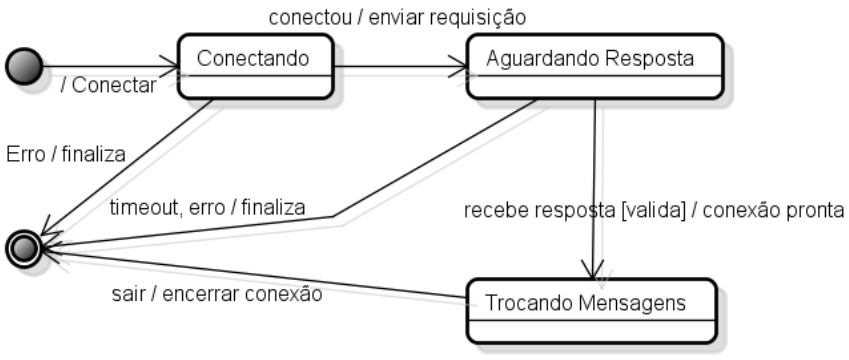

Figura 4.3: Máquina de estados do Cliente.

O cliente precisa se conectar ao servidor, negociar a conexão atravéz de um *handshake* composto de uma requisição e resposta. Passam a trocar mensagens e encerrar quando satisfeito.

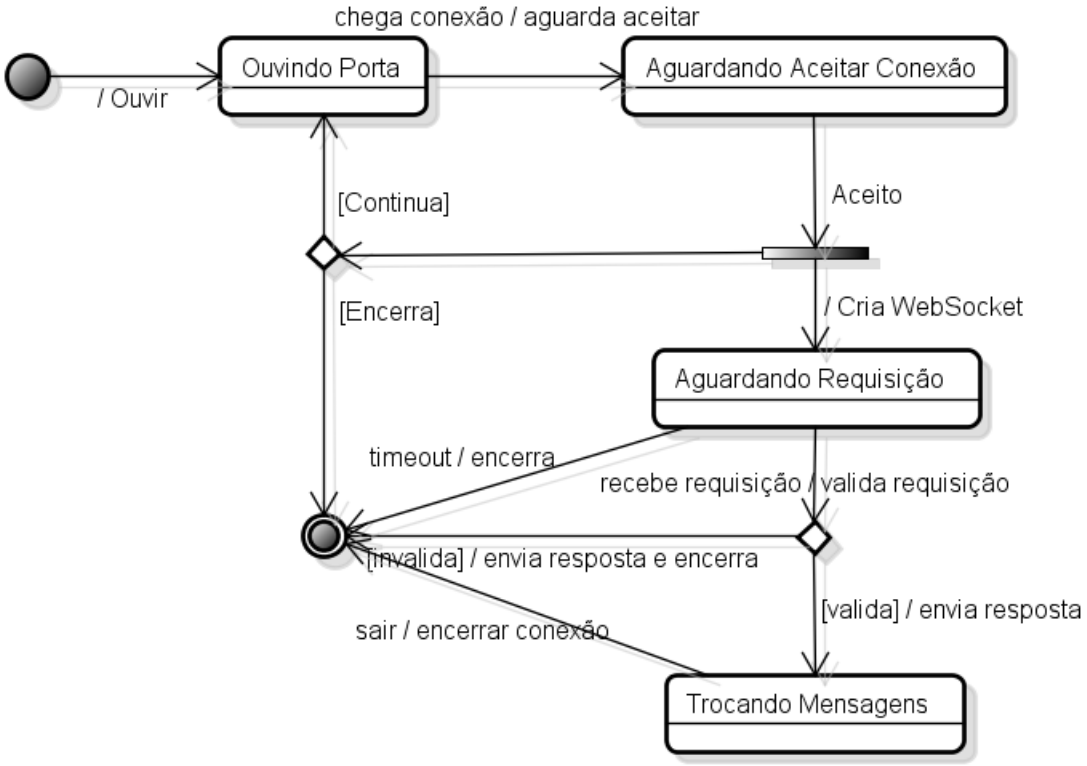

Figura 4.4: Máquina de estados do Servidor.

O servidor precisa ouvir uma porta para aceitar as conexões que chegam. Cada conexão aceita cria um websocket próprio que permite trocar mensagens com o cliente até esta conexão ser encerrada. O servidor para de aceitar novas conexões quando parar de ouvir a porta.

4.4 Organização em Módulos

Na linguagem Erlang não existem classes ou objetos, o programa é organizado em módulos. Cada módulo delimita um conjunto de operações e funcionalidade referentes a um tipo abstrato de dado como lista, pilha, árvore; domínio do problema como ordenação, busca; ou sistema. O *WebSocket* foi organizado em vários módulo e pacotes com objetivo de delimitar claramente as responsabilidade de cada um.

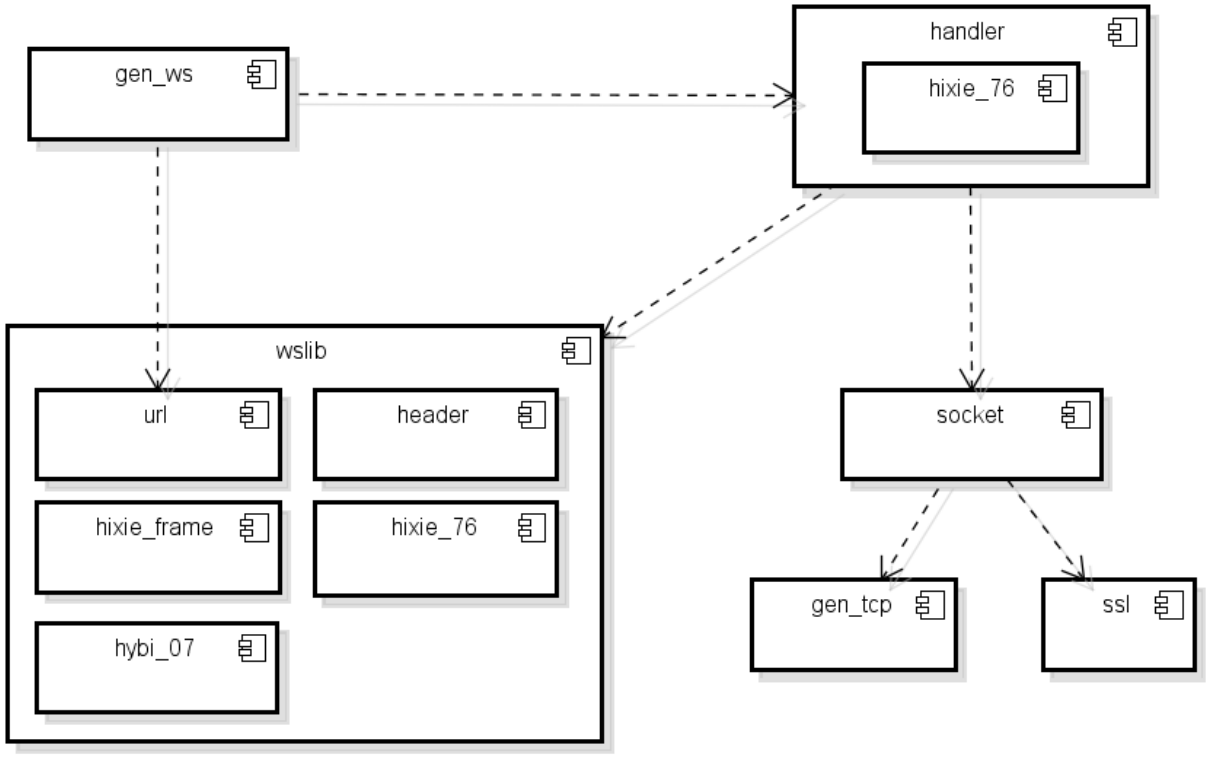

Figura 4.5: Organização em Módulos.

- *gen_ws*: módulo principal onde é definida a API que controla o *WebSocket*. Essa API internamente implementa um protocolo de comunicação entre o processos. Permite usar a mesma API para controlar diferentes implementações do protocolo.
- *socket*: módulo é um adaptador que padroniza a interface dos módulos da biblioteca padrão *gen_tcp* e *ssl*. Permite estabelecer de forma transparente o modo protegido do *WebSocket*.

- *handler*: pacote no qual são agrupados os módulos que implementam os processos que gerenciam o *socket*. São definida as rotinas de tratamento de eventos, enquanto os algoritmos são armazenados na *wslib*.

- *wslib*: pacote *WebSocket Library* que contém módulos específicos para componentes do protocolo e rotinas específicas de um versão do protocolo que podem ser reaproveitadas em outros programas ou módulos.

O do pacote *wslib*:

- *header*: algoritmos que tratam o cabeçalhos do *WebSocket*.

- *hixie_frame*: algoritmos que tratam o processamento de quadros do *draft hixie*.

- *hybi_07*: algortimos utilizados no *draft hybi* 7.

- *url*: algoritmos que tratam *uniform resource locator*.

O pacote *handler*

- *hixie_76*: Define o processo que gerencia a conexão e o tratamento de mensagens para o protocolo *draft hixie 76*.

4.5 Organização em Processos

A execução do *WebSocket* foi organizada em processos. Os processos são isolados entre si, podem executar em qualquer local, não compartilham informações apenas trocam mensagens. Foram organizados em três processos como mostrado na figura 4.6, os circulos representam processos e as setas direção das mensagens; *Owner*, *WebSocket Handler* e *WebSocket Receiver*.

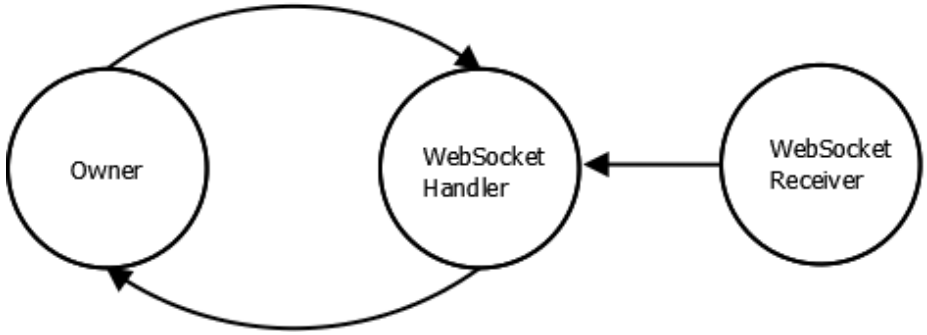

Figura 4.6: Divisão em Processos.

- *Owner*: processo que utiliza serviços do *WebSocket*, se comunicam utilizando a API do módulo *gen_ws* com o processo *WebSocket Handler*. O processo original que utilizou a API para criar um *WebSocket*, pode ser modificado usando a função *controlling_process*.

- *WebSocket Handler* processo de controle que é responsável por atender as necessidades do processo *Owner* ou armazena as mensagens recebidas pelo *WebSocket Receiver*.

- *WebSocket Receiver*: é o processo responsável por organizar os *bytes* recebido da rede em quadros que são convertido mensagem e enviadas ao processo *WebSocket Handler* para ser armazenada até o processo *Owner* retirar.

Capítulo 5

WebSocket em Erlang

O módulo principal da implementação *WebSocket* é o módulo *gen_ws*. Sua API é descrita na tabela 5.1. Contém um conjunto mínimo de operações que permite utilizar o *WebSocket* para prover serviços e estabelecer conexões.

Nome	**Descrição**
connect	Conecta a um servidor *WebSocket* endereçado por uma URL.
listen	Associa-se a uma porta no máquina para estabelecer conexões *WebSocket*.
accept	Aguarda uma conexão e aceita assim que estabelecida, realiza *handshake*, gerando um *WebSocket*.
recv	Aguarda e recebe de uma mensagem enviada pela outra parte.
send	Transmite um mensagem a outra parte.
close	Encerra a conexão, impedindo envio e recebimento de novas mensagens.
getinfo	Obtém os parâmetros de configuração, cabeçalhos de requisição e resposta.
controlling_process	Modifica o processo destino ao qual são transmitidas as mensagens no modo ativo, por padrão o processo é aquele que realizou a chamada ao *connect* ou *accept*.

Tabela 5.1: Descrição da *Application Programming Interface*.

Um servidor baseado em *WebSocket* utiliza a função *listen* e *accept* para estabelecer uma porta que aguarda conexões externas. Para cada nova conexão *WebSocket* estabelecida as funções *send* e *recv* permitem a comunicação. A função *close* encerra a conexão.

Um cliente baseado em *WebSocket* utiliza a função *connect* para estabelecer a conexão com um servidor. As funções *send* e *recv* permite a comunicação. *close* encerra a conexão.

A API é baseada na comunicação entre processos, o processo que cria um *WebSocket* recebe de modo automático a denominação de proprietário (*Owner*) do *WebSocket*. O *Owner* receberá todas as mensagens assíncronas transmitidas pelo *WebSocket*. Quando se

deseja mudar o processo *Owner* utiliza-se a função *controlling_process*, apenas o processo *Owner* pode determinar um novo processo *Owner*.

5.1 Módulo WebSocket Generator

WebSocket são representados por tuplas de três elementos contendo o tipo do *WebSocket*, a versão do protocolo e por último o processo que controla a conexão.

{websocket|websocket_listen, DraftVersion, HandlerPid}
DraftVersion = draft_hixie76 | atom()
HandlerPid = pid()

O módulo principal serve de *wrapper* para chamar os processos responsáveis por gerenciar o *WebSocket*.

5.1.1 *Connect*

A função *connect* , descrito no código 5.1.1, permite iniciar uma conexão *WebSocket*. Recebe dois parâmetros, o primeiro é a url como por exemplo "ws://echo.websocket.org" que identifica o modo do *socket*, nome do host e porta de acesso. O segundo parâmetro é opcional, nele são definidas as opções de configuração do *socket*. Quando executado com sucesso retorna um *WebSocket*, caso contrário retorna um código de erro.

```
gen_ws:connect(Url) -> {ok, WebSocket}|{error, Reason}
gen_ws:connect(Url, Options) -> {ok, WebSocket}|{error, Reason}

Url = WebSocketUrl = string()
Options = [{active, true|false}, {origin, Origin},
{timeout, Timeout}, {subprotocol, SubprotocolList}]
Origin = HttpUrl = string()
Timeout = number()|infinity
SubProtocolList = [string()] | []
```

Código 5.1.1: Definição da função *connect* módulo gen_ws.

Os parâmetros de configurações do parâmetro *Options* são:

- {active, true|false}: quando *true* define o modo ativo no qual qualquer mensagem recebida no *WebSocket* deve ser enviada ao processo *Owner*. Quando *false* o processo armazena as mensagens recebidas, as mensagens são retiradas usando *recv*.

- {origin, Origin}: configura a origem da conexão na requisição. Por padrão esse valor é "127.0.0.1".
- {subprotocol, SubprotocolList}: define a lista de subprotocolos suportados pela aplicação que são listadas no *Web-Socket-Protocol* o padrão é [].
- {timeout, Timeout}: determina o tempo máximo que se deve aguardar para estabelecer a conexão o padrão é {timeout, infinity}, quando o tempo é ultrapassado a tentativa de conexão é abortada.

Quando é estabelecido com sucesso a conexão é retornado a tupla {ok, WebSocket} com o *WebSocket* com *WebSocket* pronto para uso. Quando ocorre um erro a tupla {error, Reason} é retornada.

5.1.2 *Listen* e *Accept*

A função *listen* e *accepts*, descritos no código 5.1.2, são responsáveis por associar o *WebSocket* a uma o porta e estabelecer uma conexão respectivamente. O servidor utiliza essa porta para aguardar a rede sinalizar novas conexões *WebSocket*. As conexões que chegam ficam aguardando serem processadas.

O primeiro parâmetro de *listen* é a porta a ser associar, o segundo parâmetro é opcional, serve para definir configurações dos *WebSocket*. Quando executado com sucesso retorna um *WebSocket* do tipo *listen*, caso contrário retornará um código de erro.

O processamento é feito pela função *accept*, a função retira um *socket* da fila de espera e realiza o *handshake*, se ocorre com sucesso um *WebSocket* é retornado, caso exista algum problema um erro é retornado. Quando não existe nenhum *socket* na fila de espera o *accept* bloqueia a chamada até que um *WebSocket* seja processado, o paramêtro Timeout permite limitar o tempo de bloqueio do função. Se o valor de Timeout for ultrapassado é retornado {error, timeout}.

```
gen_ws:listen(Port) -> {ok, WebSocketListen}|{error, Reason}
gen_ws:listen(Port, Options) -> {ok, WebSocketListen}|{error, Reason}
Port = 0..65535
Options = [Key, Value]

gen_ws:accept(WebSocketListen) -> {ok, WebSocket} | {error, Reason}
gen_ws:accept(WebSocketListen, Timeout) -> {ok, WebSocket} | {error, Reason}
Timeout = number() | infinity
```

Código 5.1.2: Definição da funções *listen* e *accept* módulo gen_ws.

5.1.3 *Send* e *Recv*

Os funçãos *recv* é uma contração da palavra *receive*. Esse função permite receber uma mensagem armazenada no *WebSocket*. Sua contra parte é a função *send* que permite enviar uma mensagem pelo *WebSocket*. A definição da *Application programming interface* é descrita no código 5.1.3.

O *recv* é bloqueante, caso não exista uma mensagem disponível a função aguardará até que uma seja depositada. O *WebSocket* armazenará mensagens quando configurado para {active, false}, uma mensagem de erro é gerada caso contrário. O modo {active, true} envia mensagens direto para o processo que criou o *WebSocket* ou para um outro processo definido por meio de *controlling_process*.

O código 5.1.3 descreve a API de *send* e *recv*. Ambos podem ser configurados com tempo limite para forçar interrupção e retornam o átomo *ok* ou um código de erro.

```
gen_ws:recv(WebSocket) -> {ok, Data}|{error, Reason}
Data = string() | binary()

gen_ws:recv(WebSocket, Timeout) -> {ok, Data}|{error, Reason}
Timeout = number() | infinity
Data = string() | binary()

gen_ws:send(WebSocket, Data) -> ok |{error, Reason}
Data = string() | binary()

gen_ws:send(WebSocket, Data, Timeout) -> ok | {error, Reason}
Timeout = number() | infinity
Data = string() | binary()
```

Código 5.1.3: Definição das funções *recv* e *send* módulo gen_ws.

5.1.4 *Close*, *Controlling Process* e *Info*

A função *close* encerra a conexão. O processo responsável por gerenciar a conexão continua a existir e gera um erro nas chamadas *send* ou *recv* quando utilizadas identificando que a conexão foi encerrada. O átomo *ok* sempre é retornado, mesmo que a conexão tenha

sido encerrada antes. O código 5.1.4 descreve a *Application Programming Interface* de close e as funções *controlling_process* e *getinfo*.

```
gen_ws:close(WebSocketType, Timeout) -> ok
Timeout = number() | infinity

gen_ws:controlling_process(WebSocketType, NewOwner) -> ok | {error, Reason}
NewOwner = pid()
Reason = atom()

gen_ws:getinfo(WebSocketType) -> [{Key, Value}]
```

Código 5.1.4: Definição da função *close*, *controlling_process* e *getinfo* módulo gen_ws.

5.1.5 Mensagens Assíncronas

O processo responsável pelo *WebSocket* quando configurado para {active, true} enviará uma mensagem assíncrona para o processo a qual pertence a conexão *WebSocket*. São três mensagens, uma para o recebimento de uma mensagem, o segundo para erros que ocorram no *WebSocket* e o terceiro para informar do encerramento da conexão.

- Recebimento de Mensagem:
  {*ws*, WebSocket, {*text*|*binary*, Message}}
- Error no *WebSocket*:
  {*ws_error*, WebSocket, Reason}
- Fim de conexão:
  {*ws_closed*, WebSocket}

A mensagem de encerramento de conexão *sempre* será enviada, mesmo quando configurado para {active, false}. As mensagens assíncronas do modo {active, true} permitem criar um servidor *WebSocket* orientado a tratamento de eventos de rede utilizando o *BIF* receive para tratar cada tipo de mensagem e definir para cada tipo de mensagem uma rotina adequada.

5.2 *Biblioteca* WebSocket (*wslib*)

Foram definidos estrutura de dados específicas para representar cada tipo de dados utilizado pelas funções da API *WebSocket* e organizados no pacote *wslib*.

5.2.1 *Módulo WebSocket Uniform Resource Locator*

O módulo *url* define o tratamento da *URL* utilizada para identificar um servidor *WebSocket*. Possui apenas duas funções:

1. parse: Converte a URL para a tupla {Mode, Domain, Port, Path}.
   (a) *Mode*: o valor é o átomo *normal* ou o átomo *secure* para os *schemes ws* e *wss* respectivamente.
   (b) *Domain*: o nome da máquina na rede, pode ser um nome de domínio ou um IP.
   (c) *Port*: um numero inteiro que indica qual porta se conectar.
   (d) *Path*: o caminho até a aplicação na máquina de destino.
2. to_string: Converte a tupla ao formato URL.

Exemplos:

- "ws://echo.websocket.org/" → {normal, "echo.websocket.org", 80, "/"}
- "wss://echo.websocket.org/" → {secure, "echo.websocket.org", 80, "/"}
- "ws://app.games.com:240/west/checker" → {normal, "app.games.com", 240, "/west/-checker"}
- "wss://192.168.10.15:241/east/go" → {normal, "192.168.10.15", 241, "/east/go"}

5.2.2 Módulo *WebSocket Header*

O módulo *header* trata os cabeçalhos *handshake* para isso utiliza uma estrutura de dados, uma tupla de elementos em série que permitem converter do formato *string* para o formato em Erlang, identificar a qual protocolo pertence o cabeçalho e recuperar um parâmetro do cabeçalho.

o primeiro elemento é um átomo que identifica se o cabeçalho é uma requisição (request) ou responsta (response). O segundo elemento é uma lista ordenada com os paramêtros do cabeçalho em tuplas de dois elementos no formato *nome* e *valor*. Um exemplo dessa estrutura está no códigos 5.2.1 e 5.2.2 contém a representação dos cabeçalhos dos códigos 3.1.1 e 3.1.2.

No *request* os dois primeiros paramêtros são *method* e *path*, no response são *status* e *reason*, parâmetros fora do padrão são identificados com *undefined* seguido pelo conteúdo da linha. A exemplo do código 5.2.1.

{*request*|*response*|undefined, [{Name, Value},...] }
Name = *method*|*path*|*status*|*undefined*|string(); Value = string()

```
{request,[{method,"GET"},
          {path,"/demo"},
          {"Host","example.com"},
          {"Origin","http://example.com"},
          {"Connection","Upgrade"},
          {"Upgrade","WebSocket"},
          {"Sec-WebSocket-Key1","18x 6]8vM;54 *(5:  {   U1]8  z [  "},
          {"Sec-WebSocket-Key2","1_ tx7X d  <  nw  334J702) 7]o}‘ 0"},
          {undefined,[]},
          {undefined,"^Tm[K T2u"}]},
```

Código 5.2.1: Formato em Erlang do cabeçalho descrito no código 3.1.1.

```
{response,[{status,"101"},
          {reason,"/demo"},
          {"Upgrade","WebSocket"},
          {"Connection","Upgrade"},
          {"Sec-WebSocket-Location","ws://example.com/demo"},
          {"Sec-WebSocket-Origin","http://example.com"},
          {"Sec-WebSocket-Protocol","sample"},
          {undefined,[]},
          {undefined,"fQJ,fN/4F4!~K~MH"}]},
```

Código 5.2.2: Formato em Erlang do cabeçalho descrito no código 3.1.2.

5.2.3 Quadro *WebSocket Hixie*

Os quadros são representados por tuplas contendo o identificado do tipo de quadro, *text* ou *binary*, seguido pelo conteúdo.

{*text*|*binary*, Message}

5.2.4 *WebSocket Hixie 76*

Módulo que contém os algoritmos relacionados ao processamento do protocolo *WebSocket* Hixie 76. As funções disponibilizadas para uso são:

1. *gen_request*: Cria uma requisição a partir da URL, url de origem e subprotocolos no formato utilizado por wslib.header.

2. *gen_response*: Cria uma resposta a partir do cabeçalho de requisição no formato de wslib.header.

3. *make_trial*: Cria o desafio de requisição no formato `{Key1, Key2, Key3, Solution}`

4. *resolve_trial*: Soluciona um desafio a partir dos três parâmetros.

5. *frame*: Formata mensagem em um quadro.

6. *unframe*: Obtêm mensagem de um quadro.

Capítulo 6

Experimentos e Resultados

Foram desenvolvidas três aplicações distribuídas para verificar o *WebSocket* desenvolvido em Erlang.

1. Cliente *Echo*.
2. Servidor de *Chat*.
3. Aplicação distribuída *HTML5 Multidot*.

Cada aplicação permitiu validar uma característica do protocolo. Os testes foram realizas no laboratório de engenharia da computação da Escola Superior de Tecnologia da Universidade do Estado do Amazonas.

6.1 Ambiente de Teste

- Sistema Operacional
  - Ubuntu – 11.04 (natty)
  - GNU/Linux – 2.6.38-8-server x86_64
- Hardware
  - Memória – 4 Gb - DDR2
  - Disco Rígido – 250 Gb - Sata 2
  - Processador – Intel© Core™2 Duo CPU E7400 - 2.8 GHz

- Erlang
  - Erlang R14B03 - (64-bits)
  - Eshell Versão - 5.8.4

6.2 Cliente de *echo*

O *echo* é um aplicação que permite verificar se as mensagens estão sendo transmitidas e recebidas corretamente pelo servidor.

O cliente *echo* está conectados a um servidor. O servidor não modifica o conteúdo das mensagens quando as recebe, apenas as transmite de volta ao remetente como na figura 6.1. No experimento o cliente em Erlang se conecta ao servidor de *echo* disponibilizado pela *KAAZING*[1].

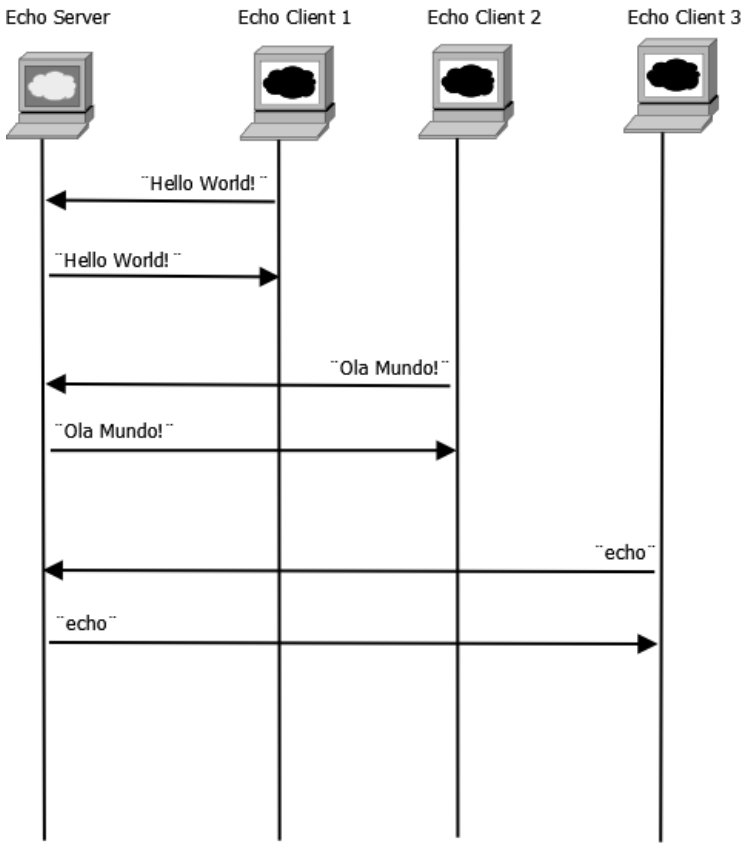

Figura 6.1: Aplicação de distribuída de *echo*.

[1]"ws://echo.websocket.org/"

6.2.1 Resultados

O módulo *echo_cli*, descrito no código 6.2.1, representa um cliente de *echo* usado diretamente pelo interpretador do erlang (Eshell). Um exemplo de uso do cliente é descrito no código 6.2.2.

```
1  -module(echo_cli).
2  -author("Emiliano Firmino").
3  -import(websocket).
4  -export([start/0,send/2,show_messages/0,close/1]).
5
6  start() ->
7      Url = "ws://echo.websocket.org/",
8      Origin = "http://websocket.org",
9      {ok,Server} = gen_ws:connect(Url,[{active,true},{origin,Origin}]),
10     Server.
11
12 send(Server,Message) ->
13    gen_ws:send(Server,Message).
14 show_messages() ->
15 receive
16    Any -> io:format("~w",[Any]),
17        show_messages()
18    after 1
19        ok
20 end.
21 close(Server) ->
22    gen_ws:close(Server).
```

Código 6.2.1: *Cliente* echo WebSocket

```
1  Eshell V5.8.5
2  Erlang R14B03 (erts-5.8.4) [smp:2:2] [rq:2] [async-threads:0]
3
4  Eshell V5.8.4  (abort with ^G)
5  1> EchoClient = echo_cli:start().
6  {websocket,draft_hixie76,<0.35.0>}
7  [{origin,"http://websocket.org"},
8   {request,[{method,"GET"},
9             {path,"/"},
10            {"Upgrade","WebSocket"},
11            {"Connection","Upgrade"},
12            {"Sec-WebSocket-Key2","1126PXgl7j 6  t4vfc828"},
13            {"Host","echo.websocket.org"},
14            {"Sec-WebSocket-Key1","1h ZKF6'-3;Ae9611-8y2sb3"},
15            {"Origin","http://websocket.org"},
16            {undefined,[]},
17            {undefined,[223,31,171,182,229,46,99,218]}]},
18   {host_addr,{192,168,0,187}},
19   {url,"ws://echo.websocket.org/"},
20   {peer_addr,{174,129,224,73}},
21   {peer_port,80},
22   {response,[{status,"101"},
23              {reason,"Web Socket Protocol Handshake"},
24              {"Upgrade","WebSocket"},
25              {"Connection","Upgrade"},
26              {"Sec-WebSocket-Origin","http://websocket.org"},
27              {"Sec-WebSocket-Location","ws://echo.websocket.org/"},
28              {"Server","Kaazing Gateway"},
29              {"Date","Fri, 25 Nov 2011 23:47:32 GMT"},
30              {"Access-Control-Allow-Origin","http://websocket.org"},
31              {"Access-Control-Allow-Credentials","true"},
32              {"Access-Control-Allow-Headers","content-type"},
33              {undefined,[]},
34              {undefined,[99,112,171,162,100,137,151,25,92,196,27,46,87,159,68,8]}]},
35   {active,true},
36   {host_port,56022},
37   {subprotocol,nil}]
38 2> echo_cli:send(EchoClient, "hello echo!").
39 ok
40 3> echo_cli:send(EchoClient, "websocket erlang").
41 ok
42 4> echo_cli:show_messages().
43 {ws, {websocket,draft_hixie76,<0.35.0>},"hello echo!"}
44 {ws, {websocket,draft_hixie76,<0.35.0>},"websocket erlang"}
45 ok
46 5> echo_cli:close(EchoClient).
47 ok
48 6> echo_cli:show_messages().
49 {ws_closed,{websocket,draft_hixie76,<0.35.0>}}
50 ok
51
```

Código 6.2.2: *Sessão do Cliente echo em Erlang.*

6.3 Servidor de *chat*

O *chat* permite demonstrar a capacidade do Erlang de suportar diversos usuários simultâneos. As mensagens de um usuário são compartilhadas com todos os usuários conectados aquele servidor como na figura 6.2.

O servidor é organizado em dois processos. O primeiro processo é o responsável por estabelecer conexões com os clientes enquanto. O segundo é controle de uma sala de *chat*, inclusão e remoção de usuários, retransmissão de mensagens.

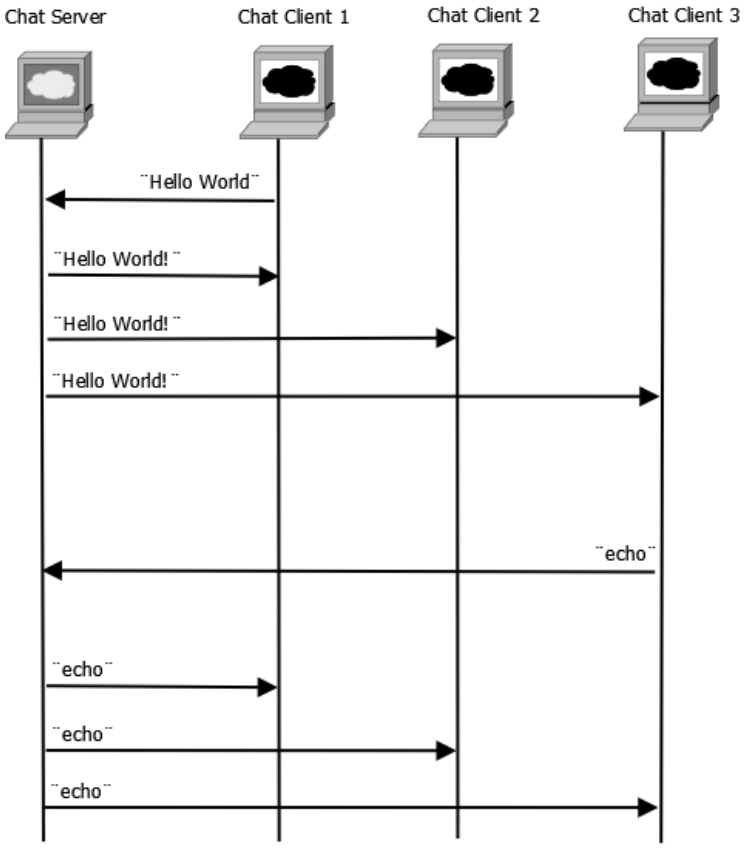

Figura 6.2: Aplicação de distribuída de *chat*.

6.3.1 Resultados

O módulo chat_srv implementa um servidor simples de chat. A porta padrão do servidor é a 80. O *echo_cli* ou o uso direto da classe *WebSocket* em JavaScript podem estabelecer uma conexão com o servidor.

O servidor é organizado em dois processos principais, *loop* e *chat*, em recursão.

O processo *loop* executa a função *chat_srv:loop*, aguardando e estabelecendo novas conexões como descrito no codigo 6.3.1, cada nova conexão é redirecionada o processo *chat*.

O processo *chat* executa a função *chat_srv:chat* em recursão contendo a lista de usuários e o tratamento dos respectivo eventos como visto no código 6.3.2. Os três eventos são:

1. Entrada usuário: adiciona um usuário a lista.

2. Mensagem: envia a mensagem enviada por um usuário a todos (*broadcast*) os usuários da lista.

3. Saída de usuário: remove o usuário da lista.

Não é definido um protocolo de comunicação, todas as mensagem são enviadas em *plain text* usando quadros UTF8.

```
1  loop(LS, ChatPid) ->
2      case gen_ws:accept(LS) of
3          {ok, S} ->
4              ChatPid ! {add, S},
5              gen_ws:controlling_process(S, ChatPid),
6              loop(LS, ChatPid);
7         {error, _IgnoreError} -> loop(LS, ChatPid)
8  end.
```

Código 6.3.1: *Função* chat_srv:loop

```
1  chat(UserList) ->
2  receive
3      {add, NewUser} ->
4          chat([NewUser|UserList]);
5      {ws_closed, ExitUser} ->
6          UpdatedList = remove_user(ExitUser, UserList),
7          chat(UpdatedList);
8      {ws, _, {text, Message}} ->
9          io:format("recebida mensagem: ~s~n",[Message]),
10         broadcast(Message, UserList),
11         chat(UserList);
12     _Ignore ->
13         chat(UserList)
14 end.
```

Código 6.3.2: *Função* chat_srv:chat

6.4 HTML5 *Multidot*

O *multidot* é uma tela (*canvas*) compartilhando entre vários computadores, um dos usuários insere pelo *mouse* um disco de cor aleatória na tela. Todos os computadores conectados ao serviço renderizam o disco em suas telas. Equivale ao chat porém com recursos visuais que podem ser explorados para desenvolver aplicativos ricos em detalhes gráficos como jogos de computador.

A aplicação é estruturada em camadas como descrito na figura 6.3. O *WebSocket* é o *middleware* comum que permite a comunicação entre cliente e servidor. A informação é codificada no formato JSON. O servidor é em Erlang, enquanto os clientes em JavaScript.

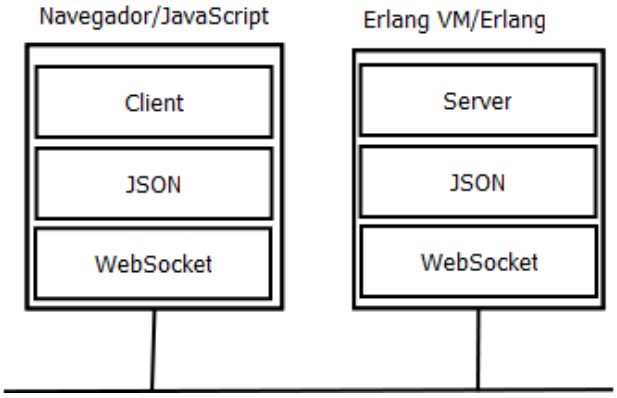

Figura 6.3: Estrutura de comunicação em camadas.

6.4.1 Resultados

O resultado foi uma aplicação em HTML5 apresentada na figura 6.4. O servidor o módulo *multidot_srv* e uma biblioteca de codificação e decodificação de JSON para Erlang.

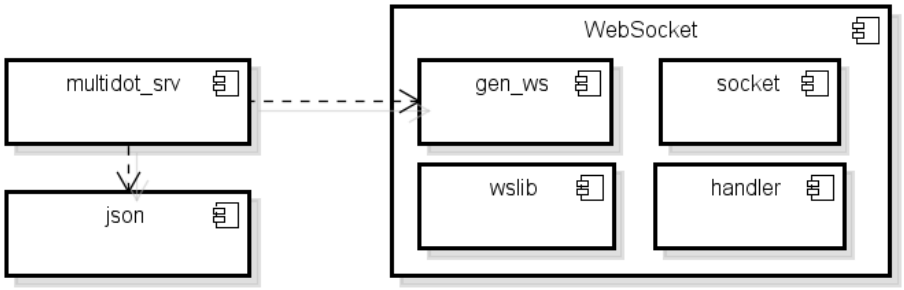

Figura 6.4: Organização em Módulos do Servidor.

O HTML5 Multidot é um documento Web como visto na figura 6.5. A aplicação se conecta via *WebSocket* a URL inserida na caixa de texto. Recebe e envia assincronamente mensagens JSON conténdo a posição da esfera no canvas e a cor.

Um servidor Erlang recebe as mensagens, converte do JSON para o formato do Erlang e registra escreve na tela, em seguida informa todos os clientes para renderizarem o disco na tela como a aplicação de *chat*.

Figura 6.5: Cliente HTML5 MultiDot.

Capítulo 7
Conclusão

O protocolo *WebSocket* é um protocolo simples que permite criar diversas aplicações ao combinar HTML5 e serviços na Web. A implementação dos vários recursos de um protocolo é uma tarefa exaustiva, requer atenção, planejamento e validação para evitar retrabalho.

Este trabalho obteve uma solução funcional do protocolo *draft hixie websocket 76* em Erlang com com suporte a modo seguro usando TLS e capaz de desenvolver clientes ou servidores em Erlang. Demonstrados em aplicações de demonstração.

A solução permite se conectar a um servidor por uma URL ou ser usada por um servidor de aplicação Erlang para provêr serviços pelo protocolo a qualquer cliente. A solução suporta conexões via TLS e foram criados um cliente *echo*, servidor de *chat* e o HTML5 MultiDot como aplicativos de demonstração.

7.1 Trabalhos Futuros

A solução obtida pode ser explorada para demonstrar os recursos nativos da linguagem Erlang de suporte a *cluster* de servidores de aplicações com milhares de usuários ou transações como serviços de mensagens e jogos em rede.

A modelagem em processos, padrão de API adotada e os algoritmos desenvolvidos nesse trabalho pode ser exploradas por outros pesquisadores para:

7.1.1 Protocolos Cliente-Servidor

Ponto inicial para implementação de protocolos como:

- FTP
- HTTP

7.1.2 Servidor

Os algoritmos de tratamento de cabeçalhos HTTP, JSON, *Socket* TCP e TLS, a arquitetura desenvolvida podem ser empregadas para:

- O desenvolvimento de um servidor HTTP(S)
- O desenvolvimento de WebService REST
- O desenvolvimento de AJAX Reverso (Comet)
- O desenvolvimento de suporte a versão final do *WebSocket*

7.1.3 Cliente

Os algoritmos e arquitetura podem ser empregados para:

- O desenvolvimento de um cliente HTTP(S)
- Cliente de serviços REST
- Cliente que suporte AJAX

7.1.4 Formato de Intercâmbio de Dados

- Desenvolvimento parse em XML
- Otimização do *parser* JSON
- Otimização do *parser* cabeçalho HTTP

7.1.5 Aplicações

- Avaliação de desempenho com milhares de usuários
- Exploração de jogos de rede baseados em HTML5

Referências Bibliográficas

[Cesarini and Thompson2009] Cesarini, F. and Thompson, S. (2009). *Erlang Programming*. O'Reilly Media, 1st edition.

[Group1999] Group, N. W. (1999). *RFC2616: Hypertext Transfer Protocol – HTTP/1.1*. Internet Engineering Task Force.

[Hickson2009] Hickson, I. (2009). The websocket protocol, draft-hixie-thewebsocketprotocol-00. `http://tools.ietf.org/html/draft-hixie-thewebsocketprotocol-00`.

[Hickson2010a] Hickson, I. (2010a). Framing and design philosophy. `http://answerpot.com/showthread.php?761364-Framing+and+design+philosophy/Page1`.

[Hickson2010b] Hickson, I. (2010b). The websocket protocol, draft-hixie-thewebsocketprotocol-76. `http://tools.ietf.org/html/draft-hixie-thewebsocketprotocol-76`.

[Rabhi and Lapalme1999] Rabhi, F. and Lapalme, G. (1999). *Algorithms: a functional programming approach*. Pearson Education Limited, 2nd edition.