Conexão Socket na MIDP

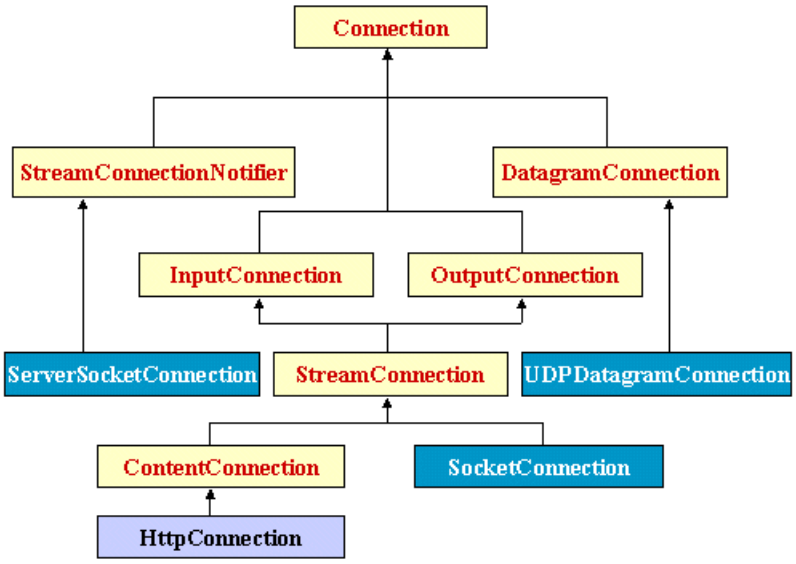
A comunicação entre diferentes sistemas é comum na programação, e esta regra não desaparece na construção de aplicativos para pequenos dispositivos e, consequentemente, com o Java ME (MIDP). Além das comunicação corriqueiras, como o HTTP (Hypertext Transfer Protocol), SMS (Short Message Service) e etc, a biblioteca da MIDP fornece novas formas de comunicações na sua nova versão, a 2.0, como a *SocketConnection*.

Como o objetivo deste texto não é dissecar completamente a comunicação socket, descrevo o mesmo com a seguinte descrição. Socket é um ponto de comunicação, e é composto de um IP e uma porta [GURJÃO, Edmar Candeia]. Também não é objetivo deste texto se posicionar sobre os pontos positivos e negativos das várias formas de comunicação. O foco principal aqui é mostrar como efetuar uma comunicação socket usando Java ME (MIDP) para se comunicar com um servidor.

Inicialmente é apresentado a estrutura de comunicação da MIDP, posteriormente é mostrada a *SocketConnection* e seus principais métodos, e por fim, será construído um pequeno exemplo onde o aplicativo MIDP acessa um servidor através de socket e recebe uma mensagem de boas vindas, que é definida conforme a hora atual do servidor.

GCF

A Generic Connection Framework é um framework criado para uso exclusivo no Java ME, isso porque as classes que são usadas nos pacotes *java.io.** e *java.net.** para conexões em Java SE, oferecem um conjunto muito grande de classes e interfaces, sendo assim, seu uso em pequenos dispositivos é inconcebível. Portanto, a GFC foi criada para oferecer um conjunto diminuído mas totalmente funcional para prover a maioria das formas de comunicação existenten atualmente. A figura abaixo mostra sua estrutura a partir da MIDP 2.0, visto que, as classes *javax.microedition.io.SocketConnection, javax.microedition.io.ServerSocketConnection e javax.microedition.io.UDPDatagramConnection* não estão disponíveis na MIDP 1.0.



Generic Connection Framework. Fonte: MAHMOUD, Qusay H.

Uma características marcante e inteligente da GCF é a forma de criar uma conexão, sendo que para todas os tipos usa-se a mesma estrutura de código, ou seja, independe se seu código vai gerar uma conexão HTTP ou socket, a codificação seguirá a mesma lógica e as mesmas classes. Veja a Listagem 1. O ponto crucial é o uso da classe *Connector* e seu método *open()*, passando o protoclo, seu

endereço e os possíveis parâmetros.

```
...

try {
      Connector.open("protocol:address;parameters");
} catch (ConnectionNotFoundExeption e) {
      ..
}

...
```

Listagem 1 – Estrutura para abrir uma conexão.

Para exemplificar os conceitos apresentados anteriormente veja a Listagem 2. Neste trecho de código estão sendo criados três tipos de conexão, primeiramente uma conexão HTTP, posteriormente uma conexão socket e por fim, uma conexão por datagramas. Em todas elas usamos o mesmo método *open()* da classe *Conector*. Um conceito importante é apresentado por MAHMOUD, para este autor, o método *open()* é uma fábrica para novas conexões, você não usa o operador *new* para instanciar uma conexão. Quando um tipo específico de conexão não é implementado pelo dispositivo, a tentativa de sua criação cria uma exceção *ConnectionNotFoundException.*

```
...

Connector.open("http://www.javafree.com.br")
Connector.open("socket://www.javafree.com.br:5500")
Connector.open("datagram://www.javafree.com.br:5500")

...
```

Listagem 2 – Criação de três tipos de conexão.

Estudo de caso

Como descrito no início deste texto, o estudo de caso implementa um servidor que retorna uma mensagem de saudação a aplicação MIDP, *sendo Bom Dia, Boa Tarde* ou *Boa* Noite, dependendo da hora em que o serviço rodando no servidor foi acessado. A Listagem 3 traz o código completo.

```
/*
 * Server.java
 *
 * Created on 15 de Maio de 2007, 14:28
 */

package home.ping;

import java.io.*;
import java.net.*;
import java.util.*;

/**
 *
 * @author  ping
 */
public class Server {

    /** Creates a new instance of Server */
    public Server() {
        super();
    }
```

```
    public static void main(String[] main)
    {
        int server_port = 6500;

        ServerSocket server_socket = null;
        Socket socket_base = null;

        try
        {
            server_socket = new ServerSocket(server_port);

            do
            {
                socket_base = server_socket.accept();
                InetAddress inet_connected = socket_base.getInetAddress();

                System.out.println("Uma nova conexão foi estabelecida com:
"+inet_connected.getHostName()+" -> ip: "+inet_connected.getHostAddress()+" na
porta: "+socket_base.getPort());

                InputStream input = socket_base.getInputStream();
                int ch = input.read();
                StringBuffer leu = new StringBuffer();

                while (ch != -1)
                {
                    leu.append((char)ch);
                    ch = input.read();
                }

                OutputStream output = null;

                if (leu.toString().trim().equals("abbced"))
                {
                    Calendar cal = Calendar.getInstance();
                    cal.setTime(new Date());
                    int hora = cal.get(Calendar.HOUR_OF_DAY);

                    output = socket_base.getOutputStream();

                    if (hora >= 0 && hora < 12)
                        output.write("Olá.. Bom dia".getBytes());
                    else if (hora >= 12 && hora < 19)
                        output.write("Olá.. Bom tarde".getBytes());
                    else
                        output.write("Olá.. Bom noite".getBytes());
                }

                if (output != null)
                {
                    output.flush();
                    output.close();
                }
            }
            while (true);
        }
        catch (Exception e)
        {
            System.out.println("Exception: "+e);
        }
    }
}
```

Listagem 3 – Implementação do servidor

O começo significativo do código começa no trecho de código:

```
server_socket = new ServerSocket(server_port);
```

Neste momento definimos que um servidor de socket estará ativo na porta passado por parâmetro para o construtor da classe ServerSocket, no exemplo definimos uma variável *server_port.* Posteriormente, o código entra em um loop *do*-while, o trecho de código que segue é importante para o entendimento do restante do programa:

```
      socket_base = server_socket.accept();
      InetAddress inet_connected = socket_base.getInetAddress();

      System.out.println("Uma nova conexão foi estabelecida com:
"+inet_connected.getHostName()+" -> ip: "+inet_connected.getHostAddress()+" na
porta: "+socket_base.getPort());
```

Quando uma nova requisição de comunicação socket chega neste servidor, o método *accept()* da *classe* ServerSocket é chamado, permitindo a comunicação com o servidor. A seguir, recupera-se as informações referentes ao ponto que está se comunicando com o servidor, através da classe *InetAddress*. A linha seguinte somente mostra estas informações. Depois de ter aceitado que uma nova comunicação se efetue, o próximo passo é receber as informações que estão sendo passadas pela porta que foi definida na inicialização do *ServerSocket*.

```
      InputStream input = socket_base.getInputStream();
      int ch = input.read();
      StringBuffer leu = new StringBuffer();

      while (ch != -1)
      {
            leu.append((char)ch);
            ch = input.read();
      }

      OutputStream output = null;

      if (leu.toString().trim().equals("abbced"))
      {
```

Inicialmente cria-se um *InputStream* para começar a receber os dados oriundos da conexão. Em seguida criamos um laço *while* que lerá todos os bytes recebidos até chegar ao final do *InputStream,* quando encontrara o valor *-1.* Neste momento é verificado se o conjunto de dados recebidos forma a String *"abbced",* que foi apenas um protocolo criado para saber que quem está acessando o servidor é a aplicação MIDP que foi construída com o intuito específico de se comunicar com o servidor de socket criado. Passando pelo teste o que nos resta é enviar a saudação para a aplicação que estará residente no dispositivo móvel.

```
      Calendar cal = Calendar.getInstance();
      cal.setTime(new Date());
      int hora = cal.get(Calendar.HOUR_OF_DAY);

      output = socket_base.getOutputStream();

      if (hora >= 0 && hora < 12)
            output.write("Olá.. Bom dia".getBytes());
      else if (hora >= 12 && hora < 19)
            output.write("Olá.. Bom tarde".getBytes());
      else
            output.write("Olá.. Bom noite".getBytes());
```

Da mesma forma que foi criado um *InputStream* para receber os dados, cria-se um *OutputStream* para retornar alguma informação pela conexão socket. O restante de código apenas pega a hora atual, verifica esta informação e escreve a informações correta na saída do nosso servidor, através do

método *write(byte[])*. O código responsável pela comunicação da MIDP ao nosso servidor é mostrado na Listagem 4.

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import javax.microedition.io.Connector;
import javax.microedition.io.SocketConnection;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

/**
 *
 * @author  ping
 * @version
 */
public class UsandoSocket extends MIDlet implements CommandListener, Runnable{
    private Display display;
    private Form fmMain;

    private Command cmConectar;

    private SocketConnection sc;
    private InputStream is;

    public void startApp() {
        display = Display.getDisplay(this);

        fmMain = new Form("Using Sockets:");

        cmConectar = new Command("Conectar", Command.ITEM, 1);
        fmMain.addCommand(cmConectar);
        fmMain.setCommandListener(this);

        display.setCurrent(fmMain);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    public void commandAction(Command c, Displayable d) {
        Thread t = new Thread(this);
        t.start();
    }

    public void run() {
        try {
            SocketConnection sc = (SocketConnection)
            Connector.open("socket://localhost:6500");

            OutputStream os = sc.openOutputStream();

            os.write("abbced".getBytes());
            os.flush();
            os.close();

            StringBuffer sb = new StringBuffer();
            is = sc.openInputStream();
            int c = is.read();
            while (c != -1) {
               sb.append((char)c);
               c = is.read();
            }
```

```
            fmMain.append(sb.toString());

        } catch(IOException e) {
             System.out.println("erro: "+e);
        } finally {
           try {
             if(is != null) {
               is.close();
             }
             if(sc != null) {
               sc.close();
             }
           } catch(IOException e) {
             e.printStackTrace();
           }
        }
    }
}
```

O código pertencente ao Java ME será ignorado, presupondo que o leitor tenha conhecimento. A parte realmente importante do código é listada abaixo:

```
        SocketConnection sc = (SocketConnection)
        Connector.open("socket://localhost:6500");

        OutputStream os = sc.openOutputStream();
        os.write("abbced".getBytes());
        os.flush();
        os.close();

        StringBuffer sb = new StringBuffer();
        is = sc.openInputStream();
        int c = is.read();
        while (c != -1) {
            sb.append((char)c);
            c = is.read();
        }

        fmMain.append(sb.toString());
```

NO trecho de código acima é criada uma conexão socket para um servidor local, acessando a porta 6500. Posteriormente, uma String contendo o nosso protocolo de verificação é enviado através da classe *OutputStream,* de maneira análoga, a classe *InputStream* é usada para receber os dados que o servidor retornou, inserindo esta informações na instância de nossa classe *Form.*

Conclusão

A conexão através de sockets foi adicionada na MIDP 2.0 e pode ser uma excelente alternativa em alguns casos, também, é importante que você tenha conhecimento desta possibilidade, adicionando novas possibilidades na construção de seus aplicativos Java ME. O objetivo deste texto foi mostrar de forma sucinta e rápida como criar, enviar e receber dados de uma aplicação Java ME, também mostrou a implementação no lado do servidor, não entrando em detalhes específicos da comunicação via socket. Esperamos que a síntese mostrada aqui seja útil para o leitor.

Discuta sobre este artigo: http://www.javafree.org/javabb/viewtopic.jbb?t=855636#110890

Ricardo da Silva Ogliari é formando em Ciência da Computação pela Universidade de Passo Fundo RS e mestrando como aluno especial da Universidade Federal do Rio Grande do Sul. Desenvolveu sistemas mobile para empresas do Rio Grande do Sul, atualmente trabalha como desenvolvedor de

aplicações direcionadas a telefones celulares na Kwead.com de São Paulo. Também, foi palestrante dos eventos JustJava 2005 e EJES (Encontro Java do Espírito Santo) 2005, além de ministrar alguns cursos sobre a tecnologia Java ME.

Referências Bibliográficas

GURJÃO, Edmar Candeia. Programação com Socket usando DEV C++. Universidade Federal de Campina Grande.

MAHMOUD, Qusay H. (2003). J2ME Low-Level Network Programming with MIDP 2.0.