```
1  /// Skeleten implemented by Camille Rasmussen and Jessie Delacenserie
2  /// CS 3500 Fall 2014
3
4  using System;
5  using System.Collections.Generic;
6  using System.Linq;
7  using System.Net.Sockets;
8  using System.Text;
9  using System.Threading;
10 using System.Threading.Tasks;
11
12 namespace CustomNetworking
13 {
14     /// <summary>
15     /// A StringSocket is a wrapper around a Socket.  It provides methods that
16     /// asynchronously read lines of text (strings terminated by newlines) and
17     /// write strings. (As opposed to Sockets, which read and write raw bytes.)
18     ///
19     /// StringSockets are thread safe.  This means that two or more threads may
20     /// invoke methods on a shared StringSocket without restriction.  The
21     /// StringSocket takes care of the synchonization.
22     ///
23     /// Each StringSocket contains a Socket object that is provided by the client.
24     /// A StringSocket will work properly only if the client refrains from calling
25     /// the contained Socket's read and write methods.
26     ///
27     /// If we have an open Socket s, we can create a StringSocket by doing
28     ///
29     ///    StringSocket ss = new StringSocket(s, new UTF8Encoding());
30     ///
31     /// We can write a string to the StringSocket by doing
32     ///
33     ///    ss.BeginSend("Hello world", callback, payload);
34     ///
35     /// where callback is a SendCallback (see below) and payload is an arbitrary object.
36     /// This is a non-blocking, asynchronous operation.  When the StringSocket has
37     /// successfully written the string to the underlying Socket, or failed in the
38     /// attempt, it invokes the callback.  The parameters to the callback are a
39     /// (possibly null) Exception and the payload.  If the Exception is non-null, it is
40     /// the Exception that caused the send attempt to fail.
41     ///
42     /// We can read a string from the StringSocket by doing
43     ///
44     ///     ss.BeginReceive(callback, payload)
45     ///
46     /// where callback is a ReceiveCallback (see below) and payload is an arbitrary object.
47     /// This is non-blocking, asynchronous operation.  When the StringSocket has read a
48     /// string of text terminated by a newline character from the underlying Socket, or
49     /// failed in the attempt, it invokes the callback.  The parameters to the callback are
50     /// a (possibly null) string, a (possibly null) Exception, and the payload.  Either the
51     /// string or the Exception will be non-null, but nor both.  If the string is non-null,
52     /// it is the requested string (with the newline removed).  If the Exception is non-null,
53     /// it is the Exception that caused the send attempt to fail.
54     /// </summary>
55
56     public class StringSocket
57     {
58         // the underlying socket
59         private Socket socket;
60         // the encoder sent over by the user
61         private Encoding encoding;
62
63         // delegates for both send and receiving callbacks
64         public delegate void SendCallback(Exception e, object payload);
65         public delegate void ReceiveCallback(String s, Exception e, object payload);
66
```

```
67          // queue member vairables to hold current requests and received messages
68          private Queue<Tuple<String, SendCallback, object>> toSend;
69          private Queue<String> receivedMessages;
70          private Queue<Tuple<ReceiveCallback, object>> receiveRequests;
71
72          // booleans to control spinning of threads
73          private bool currentlySending;
74          private bool sendSpin;
75          private bool receiveSpin;
76
77          // strings to hold messages outgoing and incoming
78          private string outgoingMessage;
79          private string incomingMessage;
80
81          // lock object
82          private readonly object sendSync = new object();
83
84          // two threads for sending and receiving
85          private Thread sendingThread;
86          private Thread receivingThread;
87
88          // buffer for BeginReceive
89          private byte[] buffer;
90
91          /// <summary>
92          /// Creates a StringSocket from a regular Socket, which should already be connected.
93          /// The read and write methods of the regular Socket must not be called after the
94          /// LineSocket is created.  Otherwise, the StringSocket will not behave properly.
95          /// The encoding to use to convert between raw bytes and strings is also provided.
96          /// </summary>
97          public StringSocket(Socket s, Encoding e)
98          {
99              // initialize all variables
100             socket = s;
101             encoding = e;
102             currentlySending = false;
103             toSend = new Queue<Tuple<string, SendCallback, object>>();
104             receivedMessages = new Queue<String>();
105             receiveRequests = new Queue<Tuple<ReceiveCallback, object>>();
106             outgoingMessage = "";
107             incomingMessage = "";
108
109             // begin the spinning of threads to constantly check if messages need to go out
110             // or if messages are available to receive
111             sendSpin = true;
112             receiveSpin = true;
113             sendingThread = new Thread(StartSendMessage);
114             sendingThread.Start();
115             receivingThread = new Thread(SpinReceiveThread);
116             receivingThread.Start();
117         }
118
119         /// <summary>
120         /// We can write a string to a StringSocket ss by doing
121         ///
122         ///    ss.BeginSend("Hello world", callback, payload);
123         ///
124         /// where callback is a SendCallback (see below) and payload is an arbitrary object.
125         /// This is a non-blocking, asynchronous operation.  When the StringSocket has
126         /// successfully written the string to the underlying Socket, or failed in the
127         /// attempt, it invokes the callback.  The parameters to the callback are a
128         /// (possibly null) Exception and the payload.  If the Exception is non-null, it is
129         /// the Exception that caused the send attempt to fail.
130         ///
131         /// This method is non-blocking.  This means that it does not wait until the string
132         /// has been sent before returning.  Instead, it arranges for the string to be sent
```

```
133         /// and then returns.  When the send is completed (at some time in the future), the
134         /// callback is called on another thread.
135         ///
136         /// This method is thread safe.  This means that multiple threads can call BeginSend
137         /// on a shared socket without worrying around synchronization.  The implementation of
138         /// BeginSend must take care of synchronization instead.  On a given StringSocket, each
139         /// string arriving via a BeginSend method call must be sent (in its entirety) before
140         /// a later arriving string can be sent.
141         /// </summary>
142         public void BeginSend(String s, SendCallback callback, object payload)
143         {
144             // tuple object to hold this request
145             Tuple<String, SendCallback, object> queueTuple;
146             if (s.Length != 0)
147             {
148                 queueTuple = Tuple.Create(s, callback, payload);
149                 // add it to the queue
150                 toSend.Enqueue(queueTuple);
151             }
152         }
153 
154         /// <summary>
155         /// A helper method to send a message if one isn't currently being sent
156         /// Also invokes the sendCallback
157         /// </summary>
158         private void StartSendMessage()
159         {
160             Tuple<String, SendCallback, object> tupleToSend;
161             // Spin as long as program is active
162             while (sendSpin)
163             {
164                 // if there is at least one message to send and nothing is currently being sent
165                 if (toSend.Count != 0)
166                 {
167                     if (!currentlySending)
168                     {
169                         // dequeue and send the next available message
170                         tupleToSend = toSend.Dequeue();
171                         outgoingMessage = tupleToSend.Item1;
172 
173                         SendMessage(outgoingMessage);
174 
175                         // invoke callback on its own thread
176                         new Thread( () => tupleToSend.Item2.Invoke(null, tupleToSend.Item3)).Start();
177                     }
178 
179                 }
180             }
181         }
182 
183         /// <summary>
184         /// Sends a string to the socket
185         /// </summary>
186         private void SendMessage(String message)
187         {
188             // Lets see what thread we are
189             int managedThreadId = Thread.CurrentThread.ManagedThreadId;
190 
191             // Get exclusive access to send mechanism
192             lock (sendSync)
193             {
194                 // If there's not a send ongoing, start one.
195                 if (!currentlySending)
196                 {
197                     currentlySending = true;
198                     SendBytes();
```

```
199                 }
200             }
201         }
202 
203 
204         /// <summary>
205         /// Attempts to send the entire outgoing string.
206         /// </summary>
207         private void SendBytes()
208         {
209             // if there is no outgoingMessage left to send, stop sending
210             if (outgoingMessage == "")
211             {
212                 currentlySending = false;
213             }
214             // otherwise send next piece of message to socket
215             else
216             {
217                 byte[] outgoingBuffer = encoding.GetBytes(outgoingMessage);
218                 outgoingMessage = "";
219                 socket.BeginSend(outgoingBuffer, 0, outgoingBuffer.Length,
220                                  SocketFlags.None, MessageSent, outgoingBuffer);
221             }
222         }
223 
224 
225         /// <summary>
226         /// Called when a message has been successfully sent
227         /// </summary>
228         private void MessageSent(IAsyncResult result)
229         {
230             // Find out how many bytes were actually sent
231             int bytes = socket.EndSend(result);
232 
233             // Get exclusive access to send mechanism
234             lock (sendSync)
235             {
236                 // Get the bytes that we attempted to send
237                 byte[] outgoingBuffer = (byte[])result.AsyncState;
238 
239                 // Prepend the unsent bytes and try sending again.
240                 if (bytes != 0)
241                 {
242                     outgoingMessage = encoding.GetString(outgoingBuffer, bytes,
243                                                          outgoingBuffer.Length - bytes) + outgoingMessage;
244                     SendBytes();
245                 }
246             }
247 
248         }
249 
250         /// <summary>
251         ///
252         /// <para>
253         /// We can read a string from the StringSocket by doing
254         /// </para>
255         ///
256         /// <para>
257         ///     ss.BeginReceive(callback, payload)
258         /// </para>
259         ///
260         /// <para>
261         /// where callback is a ReceiveCallback (see below) and payload is an arbitrary object.
262         /// This is non-blocking, asynchronous operation.  When the StringSocket has read a
263         /// string of text terminated by a newline character from the underlying Socket, or
264         /// failed in the attempt, it invokes the callback.  The parameters to the callback are
```

```
265         /// a (possibly null) string, a (possibly null) Exception, and the payload.  Either the
266         /// string or the Exception will be non-null, but nor both.  If the string is non-null,
267         /// it is the requested string (with the newline removed).  If the Exception is non-null,
268         /// it is the Exception that caused the send attempt to fail.
269         /// </para>
270         ///
271         /// <para>
272         /// This method is non-blocking.  This means that it does not wait until a line of text
273         /// has been received before returning.  Instead, it arranges for a line to be received
274         /// and then returns.  When the line is actually received (at some time in the future), the
275         /// callback is called on another thread.
276         /// </para>
277         ///
278         /// <para>
279         /// This method is thread safe.  This means that multiple threads can call BeginReceive
280         /// on a shared socket without worrying around synchronization.  The implementation of
281         /// BeginReceive must take care of synchronization instead.  On a given StringSocket, each
282         /// arriving line of text must be passed to callbacks in the order in which the corresponding
283         /// BeginReceive call arrived.
284         /// </para>
285         ///
286         /// <para>
287         /// Note that it is possible for there to be incoming bytes arriving at the underlying Socket
288         /// even when there are no pending callbacks.  StringSocket implementations should refrain
289         /// from buffering an unbounded number of incoming bytes beyond what is required to service
290         /// the pending callbacks.
291         /// </para>
292         ///
293         /// <param name="callback"> The function to call upon receiving the data</param>
294         /// <param name="payload">
295         /// The payload is "remembered" so that when the callback is invoked, it can be associated
296         /// with a specific Begin Receiver....
297         /// </param>
298         ///
299         /// <example>
300         ///   Here is how you might use this code:
301         ///   <code>
302         ///                     client = new TcpClient("localhost", port);
303         ///                     Socket       clientSocket = client.Client;
304         ///                     StringSocket receiveSocket = new StringSocket(clientSocket, new
    UTF8Encoding());
305         ///                     receiveSocket.BeginReceive(CompletedReceive1, 1);
306         ///
307         ///   </code>
308         /// </example>
309         /// </summary>
310         ///
311         ///
312
313         public void BeginReceive(ReceiveCallback callback, object payload)
314         {
315             // add request to queue
316             receiveRequests.Enqueue(new Tuple<ReceiveCallback, object>(callback, payload));
317             // begin receiving bytes from the socket
318             buffer = new byte[1024];
319             socket.BeginReceive(buffer, 0, buffer.Length,
320                                 SocketFlags.None, MessageReceived, buffer);
321         }
322
323         /// <summary>
324         /// This method constantly spins a thread looking for new incoming messages
325         /// from the socket. When it receives a message, the incoming byte array is
326         /// converted to a string and put on the receivedMessages queue.
327         /// It is controlled by a boolean which will be set to false to end spinning
328         /// when the close() method is called by the user.
329         /// </summary>
```

```
330         private void SpinReceiveThread()
331         {
332             // create variables
333             Tuple<ReceiveCallback, object> request;
334             string message;
335             // continue thread spin loop until close() is called
336             while (receiveSpin)
337             {
338                 // if there is a request to process and a message to be returned
339                 //      dequeue these and invoke the callback on a new thread
340                 if (receiveRequests.Count != 0 && receivedMessages.Count != 0)
341                 {
342                     request = receiveRequests.Dequeue();
343                     message = receivedMessages.Dequeue();
344                     new Thread( () => request.Item1.Invoke(message, null, request.Item2)).Start();
345 
346                 }
347             }
348         }
349 
350         /// <summary>
351         /// Called when some data has been received.
352         /// </summary>
353         private void MessageReceived(IAsyncResult result)
354         {
355             // Lets see what thread we are
356             int managedThreadId = Thread.CurrentThread.ManagedThreadId;
357 
358             // Get the buffer to which the data was written.
359             byte[] buffer = (byte[])(result.AsyncState);
360 
361             // Figure out how many bytes have come in
362             int bytes = socket.EndReceive(result);
363 
364             // If no bytes were received, it means the client closed its side of the socket.
365             // Report that to the console and close our socket.
366             if (bytes == 0)
367             {
368                 return;
369             }
370             // Otherwise, decode and display the incoming bytes.  Then request more bytes.
371             else
372             {
373                 // Convert the bytes into a string
374                 incomingMessage += encoding.GetString(buffer, 0, bytes);
375 
376                 // Echo any complete lines, converted to upper case
377                 int index;
378                 while ((index = incomingMessage.IndexOf('\n')) >= 0)
379                 {
380                     String line = incomingMessage.Substring(0, index);
381                     receivedMessages.Enqueue(line);
382                     incomingMessage = incomingMessage.Substring(index + 1);
383                 }
384 
385                 // Ask for some more data
386                 socket.BeginReceive(buffer, 0, buffer.Length,
387                     SocketFlags.None, MessageReceived, buffer);
388             }
389         }
390 
391         /// <summary>
392         /// Calling the close method will close the String Socket (and the underlying
393         /// standard socket).  The close method  should make sure all
394         ///
395         /// Note: ideally the close method should make sure all pending data is sent
```

```
396         ///
397         /// Note: closing the socket should discard any remaining messages and
398         ///       disable receiving new messages
399         ///
400         /// Note: Make sure to shutdown the socket before closing it.
401         ///
402         /// Note: the socket should not be used after closing.
403         /// </summary>
404         public void Close()
405         {
406             // send any remaining bytes
407             SendBytes();
408 
409             // stop spinning the threads and clear the queues
410             receiveSpin = false;
411             receivedMessages.Clear();
412             sendSpin = false;
413             toSend.Clear();
414 
415             // shutdown and close the socket
416             socket.Shutdown(SocketShutdown.Both);
417             socket.Close();
418         }
419     }
420 }
421 
```