

1 eipushd.h

```
1 #define LIMIT_NORMAL 2
2 #define LIMIT_PARANOID 9
3 #define TIME_NORMAL 1000
4 #define TIME_PARANOID 10000
5
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <sys/types.h>
9 #include <sys/stat.h>
10 #include <iostream>
11 #include <fstream>
12 using namespace std;
13
14
15 /*****
16 FUNCTION log_this(int port, string adresse, string fehler)
17
18 input: current port, current IP, message
19 output: int - fil size in bytes
20 *****/
21 void log_this(int port, string adresse, string fehler)
22 {
23     time_t Zeitstempel;
24     tm *nun;
25     ofstream log ("logfile.txt", ofstream::binary | ios::app);
26     Zeitstempel = time(0);
27     nun = localtime(&Zeitstempel);
28     log << nun->tm_mday << '.' << nun->tm_mon+1 << '.' << nun->tm_year+1900 << " - " << nun->tm_hour << ':' << nun->tm_min << "
29     on "<<adresse.c_str()<<":<<port<<" - "<<fehler.c_str()<<endl;
30     log.close();
31 };
32
33 /*****
34 FUNCTION get_file_size(char *filename)
35
36 input: filename
37 output: int - fil size in bytes
38 *****/
39 /*int get_file_size(char *filename)
```

```

40 {
41     struct stat buf;
42     stat(filename, &buf);
43     int size = buf.st_size;
44     return size;
45 };*/
46
47 int get_file_size(const char* sFileName)
48 {
49     std::ifstream f;
50     f.open(sFileName, std::ios_base::binary | std::ios_base::in);
51     if (!f.good() || f.eof() || !f.is_open()) { return 0; }
52     f.seekg(0, std::ios_base::beg);
53     std::ifstream::pos_type begin_pos = f.tellg();
54     f.seekg(0, std::ios_base::end);
55     return static_cast<int>(f.tellg() - begin_pos);
56 }
57
58 /*****
59 FUNCTION get_file_segment_count(char *filename)
60
61 input: filename
62
63 output: int - number of available segments, (all 247 + the last in appropriate size)
64 *****/
65 int get_file_segment_count(char *filename)
66 {
67     // struct stat buf;
68     int pakete;
69     // stat(filename, &buf);
70     // int size = buf.st_size;
71     int size = get_file_size(filename);
72     pakete = size / 247;
73     if(pakete*247 != size){pakete++;};
74     return pakete;
75 };
76
77 /*****
78 FUNCTION *get_file_segment(int count, char *filename)
79
80 input: segment number, filename
81

```

```

82 output: pointer to char with segment data
83 *****
84 char *get_file_segment(int count, char *filename)
85 {
86     int readBufferSize, size;
87     char *readBuffer;
88
89
90     size = get_file_size(filename);
91     if((size - (count-1)*247) < 247)
92     {
93         readBufferSize = size%247;
94     }
95     else
96     {
97         readBufferSize = 247;
98     }
99
100     ifstream seg_file;
101     seg_file.open (filename, ios::binary );
102     seg_file.seekg ((247*(count-1)), ios::beg);
103
104     if (!(readBuffer = (char*)malloc(readBufferSize*sizeof(char)))) {
105         abort();
106     }
107     //cout<<readBufferSize<<"\n";
108     seg_file.read (readBuffer,readBufferSize);
109     //cout.write (readBuffer,readBufferSize);
110     seg_file.close();
111
112     return readBuffer;
113
114 };
115
116 *****
117 FUNCTION write_to_file(int count, int size, char *writeBuffer, char *filename)
118
119 input: sement number, file size, writebuffer, filename
120
121 output: 0 for success
122 *****
123

```

```

124
125 int write_to_file(int count, int size, char *writeBuffer, char *filename) {
126     int writeBufferSize, position;
127
128     ofstream set_file_segment (filename, ofstream::binary | ios::app);
129
130     if((size - (count-1)*247) < 247) {
131         writeBufferSize = size%247;
132     } else {
133         writeBufferSize = 247;
134     }
135     position = (count - 1) * 247;
136     set_file_segment.seekp(position);
137     set_file_segment.write (writeBuffer, writeBufferSize);
138
139     free(writeBuffer);
140
141     set_file_segment.close();
142     return 0;
143 };
144
145 /*****
146 CLASS MyTimer
147
148     provides custom timers in milliseconds
149
150     Constructor: MyTimer()
151
152     void MyTimer.start(void): start timer, set zero
153     double MyTimer.stop(void): stop timer, return time
154
155 *****/
156 class MyTimer {
157 private:
158     int merker, merker2;
159
160 public:
161     void reset(void) {
162         merker = time(0);
163     }
164
165     int status(void){

```

```

166         merker2 = time(0);
167         return(merker2 - merker);
168     }
169
170     MyTimer() {
171         reset();
172     };
173 };
174
175 /*****
176 CLASS STrans
177
178     Keeps all values during transaction
179
180     Constructor: STrans(int mode)
181
182         mode:    0 - normal timing
183                1 - paranoid timing
184 *****/
185 class STrans {
186
187     private:
188         string current_file;           //current_file
189         bool seg_ack;                 //received, new send allowed
190         int seg_counter;              //current segment for sending
191         double ack_timer;             //time between last send and ack
192         int resend_counter;           //packet resent how often?
193         int resend_limit;             //max. number of allowed resends
194         int max_time;                 //max. timeout time for keepalive
195         bool connected;               //conection ok
196         bool file_info;              //file info received?
197
198     public:
199         char unique_id;               //id for client identification
200
201         STrans (bool mode) {
202             this->current_file = "";
203             this->seg_ack = 0;
204             this->seg_counter = 0;
205             //this->ack_timer = time_microseconds... bla;
206             this->resend_counter = 0;
207

```

```

208         if(!mode) {
209             this->resend_limit = LIMIT_NORMAL;           //choose timing mode
210             this->max_time = TIME_NORMAL;               //default value "normal"
211         } else {
212             this->resend_limit = LIMIT_PARANOID;         //default value "paranoid"
213             this->max_time = TIME_PARANOID;             //default value "paranoid"
214         }
215         this->connected = false;
216     };
217
218     ~STrans() {};
219
220 //connection
221     bool get_connection_status(void) { return this->connected; }
222     void set_connection_status(bool) { this->connected = true; }
223
224     string getStatusConnection(void) {
225         if (this->connected) {
226             return "connected";
227         }
228         return "disconnected";
229     }
230
231 //current file
232     string get_current_file(void) { return this->current_file; }
233     void set_current_file(string filename) { this->current_file = filename; }
234
235     string getCurrentFile(void) {
236         if (strlen(this->current_file.c_str()) != 0) {
237             return current_file;
238         }
239         return "none";
240     }
241
242 //resend
243     bool check_resend(void) {
244         if(this->resend_counter < this->resend_limit) return false;
245         else return true;
246     }
247
248     int get_resend_counter(void) { return this->resend_counter; }
249     void reset_resend_counter(void) { this->resend_counter = 0; }

```

```

250         void inc_resend_counter(void) { this->resend_counter++; }
251         void set_resend_limit(int limit) { this->resend_limit = limit; }
252
253         //timeout
254         void set_max_time(int time) { this->max_time = time; }
255
256         //file info
257         bool get_file_info_status(void) { return this->file_info; }
258         void set_file_info_status(void) { this->file_info = true; }
259         void reset_file_info_status(void) { this->file_info = false; }
260
261         //seg counter
262         int get_seg_counter() { return this->seg_counter; }
263         void reset_seg_counter() { this->seg_counter = 0; }
264         void inc_seg_counter() { this->seg_counter++; }
265
266         //seg ack
267         bool get_seg_ack(void) { return this->seg_ack; }
268         void set_seg_ack(bool) { this->seg_ack = true; }
269
270     };
271
272     //struct - data type for single packet
273     typedef struct data_packet {
274         unsigned int nutzdatenlaenge;
275         unsigned int paketkennung;
276         unsigned long paketnummer;
277         unsigned long paketlaenge;
278     } Packet;        // Ganz zum Anfang!
279
280
281
282     /*****
283     FUNCTION dissect_buffer()
284
285     completely dissects a received buffer and validates packet structure
286     according to protocol and writes values to given struct members:
287
288     012 - packet number
289     345 - overall packet length including payload
290     67 - packet id (function)
291     8x - payload

```

```

292
293     this function CAN handle \0 in payload...
294
295     input:
296         adress to new packet struct
297         adress to buffer
298         buffer length
299
300     output:
301         none
302 *****/
303 unsigned char *dissect_buffer(Packet *packet, const unsigned char *buffer, size_t bufferSize) {
304
305 //header always 8 byte
306     const size_t headerSize = 8;
307
308 /* DEBUG
309     printf("bufferSize: %lu\n", bufferSize);
310 */
311
312 //packet too short
313     if( bufferSize < headerSize + 1 ) {
314         // cout << "Broken packet (< headerSize + 1) received. Someone wants to hack us :)" << endl;
315     }
316
317 //payload? how much?
318     size_t payloadSize = bufferSize - headerSize;
319
320 /* DEBUG
321     printf("payloadSize: %lu\n", payloadSize);
322 */
323     //new buffer for paylod
324     unsigned char *payLoadBuffer;
325
326     if (!(payLoadBuffer = (unsigned char*)malloc(payloadSize*sizeof(char)))) {
327         abort();
328     }
329
330 //packet too long? (more than 247 bytes, null byte from string attacks will be cut here)
331     if( payloadSize > sizeof( payLoadBuffer ) ) { // no space for \0 here
332         //send packet error, kill connection?
333         // cout << "Broken packet (> headerSize + maxPayload ) received. Someone wants to hack us :)" << endl;

```

```

334     }
335
336     /* DEBUG dissect hex parts of header into int
337     printf("Parsing header\n");
338
339     cout << "PN: " << (unsigned int)buffer[0] << " " << (unsigned int)buffer[1] << " " << (unsigned int)buffer[2] << endl;
340 */
341     packet->paketnummer = ((unsigned int)buffer[0] << 16 ) + ((unsigned int)buffer[1] << 8 ) + (unsigned int)buffer[2];
342     packet->paketlaenge = (((unsigned int) buffer[3] << 16 ) + ((unsigned int) buffer[4] << 8 ) + (unsigned int) buffer[5])
343     ;
344     packet->paketkennung = ((unsigned int) buffer[6] << 8 ) + (unsigned int) buffer[7];
345
346     cout << "-----" << endl;
347
348     printf("nummer: %10d 0x%06x laenge: %10d 0x%06x kennung: %10d 0x%04x\n",
349     packet->paketnummer, packet->paketnummer,
350     packet->paketlaenge, packet->paketlaenge,
351     packet->paketkennung, packet->paketkennung
352     );
353
354     //typically happens when zero byte is inside payload
355     if( packet->paketlaenge != bufferSize ) {
356         //throw error, but better don't quit server... write to log...
357     //     cout << "Broken packet (> headerSize + maxPayload ) received. Someone wants to hack us :)" << endl;
358     }
359
360     //set payload buffer size
361     packet->nutzdatenlaenge = payloadSize;
362
363
364     //copy payload to struct buffer
365     memcpy(payloadBuffer, buffer + headerSize, payloadSize);
366
367     /*DEBUG: print payload
368     int i;
369     for(i=0; i < bufferSize ;++i) {
370         unsigned char c = buffer[i];
371         printf("buffer [%2d]: %3u 0x%02x ('%c')\n", i, c, c, c);
372     }
373 */
374     return payloadBuffer;

```

```

375 }
376
377
378
379 /*****
380 FUNCTION make_packet
381
382 input: kennung, nutzdaten (chars)
383
384 output: char
385 *****/
386 //kaputt
387 unsigned char *make_packet(Packet *packet, int packetId, long packetNumber, const unsigned char *payload, size_t payloadSize) {
388     const size_t headerSize = 8;
389     const size_t packetLength = payloadSize+headerSize;
390     unsigned char *sendBuffer;
391
392     if (!(sendBuffer = (unsigned char*)malloc(packetLength*sizeof(char)))) {
393         abort();
394     }
395
396     packet->paketlaenge = packetLength;
397
398     //packet nummer
399     sendBuffer[0] = (packetNumber >> 16) & 0xFF;
400     sendBuffer[1] = (packetNumber >> 8) & 0xFF;
401     sendBuffer[2] = packetNumber & 0xFF;
402
403     /* DEBUG
404     cout << std::hex << std::showbase;
405     cout << "PN: " << (unsigned int)sendBuffer[0] << " " << (unsigned int)sendBuffer[1] << " " << (unsigned int)sendBuffer
         [2] << endl;
406 */
407     //packet length
408     sendBuffer[3] = (packetLength >> 16) & 0xFF;
409     sendBuffer[4] = (packetLength >> 8) & 0xFF;
410     sendBuffer[5] = packetLength & 0xFF;
411
412     //packet ID
413     sendBuffer[6] = (packetId >> 8) & 0xFF;
414     sendBuffer[7] = packetId & 0xFF;
415

```

```

416     memcpy(sendBuffer+8, payload, payloadSize);
417
418     /* DEBUG
419     cout << "=BEGIN= CLIENT - make_packet() =====" << endl;
420     cout << "Paketnummer: " << packetNumber << endl;
421     cout << "Paketlaenge: " << packetLength << endl;
422     cout << "Paketkennung: " << packetId << endl;
423     cout << "Paketbuffer: " << *sendBuffer << endl;
424
425     int i;
426     for(i=0; i < packetLength ;++i) {
427         unsigned char c = sendBuffer[i];
428         printf("sendBuffer [%2d]: %5u 0x%02x ('%c')\n", i, c, c, c);
429     }
430     cout << "=END= CLIENT - make_packet() =====" << endl;
431 */
432     return sendBuffer;
433 }

```

2 eipushd.cpp

```

1  #include <iostream>
2  #include <cstdlib> // für atoi()
3  #include <vector>
4  #include <string>
5  #include <time.h>
6  using namespace std;
7  using std::string;
8
9  #include "eipushd.h"
10 #include "SimpleSocket.h"
11
12 #define USAGE "Usage: ei-pushd <ServerPort>"
13 const int MAX_BUFFER = 255;
14 bool final_fail = false;
15
16 /* PACKET DEFINITIONS
17
18 FF - ACK
19 .00 - NACK

```

```

20 .A0 - INIT_REQ - unique_id
21 .A1 - FAIL
22 .A2 - INIT_ACK - unique_id + paranoia level.
23 .B0 - FILE_REQ - B1
24 .B1 - FILE_INFO
25 .B2 - FILE_ERR
26 .B3 - FILE_INFO_OK
27 .B4 - SEG_ERR
28 .B6 - SEG_DATA
29
30 */
31
32
33 int main(void) {
34
35     final_fail = false;
36
37     unsigned short echoServPort = 6881;           // Genutzer Server-Port
38
39     try {
40
41         UDPSocket sock(echoServPort);           // Socket-Objekt generieren
42         int msgSize;
43         unsigned char echoBuffer[MAX_BUFFER];   // Puffer für Echo-String
44         unsigned char sendBuffer[MAX_BUFFER];
45
46         log_this(echoServPort, "127.0.0.1", "SRV: Socket created. Waiting for connections.");
47
48         string sourceAddress;                   // Adresse der Quelle des empfangenen
49         Datagramms                               // Port der Datagramm-Quelle
50         unsigned short sourcePort;
51
52         /** fresh init *****/
53         STrans *myConnection = NULL;
54         Packet MyPacket;                       //new packet struct
55         MyTimer *keepAlive;                   //new timer for keep alive checks
56         keepAlive = new MyTimer();
57
58
59
60 /** infinite working loop *****/

```

```

61     while (1) {          //check for packets and take appropriate action
62
63         //new connection object if non-existent
64         if (myConnection == NULL) {
65             myConnection = new STrans(1);          //pranoid mode - long timeouts
66         } else {
67
68             cout << "Connection status: " << myConnection->get_connection_status() << endl;
69             cout << "Resend counter:      " << myConnection->get_resend_counter() << endl;
70         }
71
72         //check for keep alive timer, only if connected of course
73         if(myConnection->get_connection_status()) {
74             //didn't get anything for 10sec and longer? kick client.
75             if(10 < keepAlive->status()) {
76                 log_this(echoServPort,"127.0.0.1","SRV: Timeout. Resetting connection.");
77                 final_fail = true;
78             }
79         }
80
81         //get Msg
82         msgSize = sock.recvFrom(echoBuffer, MAX_BUFFER, sourceAddress, sourcePort);
83
84         //dissect buffer, payload to recvBuffer
85         unsigned char *recvBuffer = dissect_buffer(&MyPacket, echoBuffer, msgSize);
86
87         switch (MyPacket.paketkennung) {
88
89             /*** 0xA0 - INIT_REQ *****/
90             case 0xA0:
91
92                 //got something, reset keepalive timer
93                 keepAlive->reset();
94
95                 log_this(sourcePort,sourceAddress,"SRV: INIT REQ from client received.");
96                 cout << "Init REQ von Client " << sourceAddress << ":" << sourcePort << " erhalten." <<
97                     endl;
98                 //not yet connected, resend limit NOT reached
99                 if(!myConnection->get_connection_status() && !myConnection->check_resend()) {
100                     //send A2

```

```

101         unsigned char *packetBuffer = make_packet(&MyPacket,0xA2,0x00,echoBuffer,
102             msgSize);
103         if (packetBuffer)
104             sock.sendTo(packetBuffer, MyPacket.paketlaenge, sourceAddress,
105                 sourcePort);
106         free(packetBuffer);
107
108         log_this(0,"127.0.0.1","SRV: Sent INIT ACK to client.");
109
110         myConnection->inc_resend_counter();
111
112     } else {
113         if(!myConnection->check_resend()){
114
115             unsigned char *packetBuffer = make_packet(&MyPacket,0xA2,0x00,
116                 echoBuffer,msgSize);
117             if (packetBuffer)
118                 sock.sendTo(packetBuffer, MyPacket.paketlaenge, sourceAddress,
119                     sourcePort);
120             free(packetBuffer);
121
122             myConnection->inc_resend_counter();
123
124         } else {
125
126             log_this(0,"127.0.0.1","SRV: RESEND LIMIT reached. Bailing out.");
127
128             //connection failed
129             //send A1
130             final_fail = true;
131
132         }
133     }
134     break;
135
136     /*** 0xA1 - FAIL *****/
137
138     case 0xA1:
139
140         //got something, reset keepalive timer
141         keepAlive->reset();

```

```

139
140         log_this(sourcePort,sourceAddress,"SRV: FAIL from client received.");
141         cout << "Could not recognize packet." << endl;
142         final_fail = true;
143         break;
144
145     /** 0xA2 - INIT_ACK *****/
146         case 0xA2:
147
148             //got something, reset keepalive timer
149             keepAlive->reset();
150
151             log_this(sourcePort,sourceAddress,"SRV: INIT ACK from client received.");
152             cout << "Init ACK von Client " << sourceAddress <<":" << sourcePort << " erhalten." <<
153                 endl;
154             myConnection->set_connection_status(true);
155
156             //now we are finally connected, restart keepalive timer
157             keepAlive->reset();
158
159             myConnection->reset_resend_counter();
160             break;
161
162     /** 0xB0 - FILE_REQ *****/
163         case 0xB0:
164
165             //got something, reset keepalive timer
166             keepAlive->reset();
167
168             log_this(sourcePort,sourceAddress,"SRV: FILE REQ from client received.");
169
170             if (myConnection->get_connection_status()) {
171
172
173                 string filename = myConnection->getCurrentFile();
174                 /** the cast doesn't work !!
175                 if (0/* < get_file_segment_count((unsigned char)(filename.c_str()))*/) {
176                     //send B1 - fileinfo
177                     //sock.sendTo(echoBuffer, msgSize, sourceAddress, sourcePort);
178                 } else {
179

```

```

180                                     //send file error
181                                     //send B2
182                                     //sock.sendTo(echoBuffer, msgSize, sourceAddress, sourcePort);
183                                 }
184
185         } else {
186
187             //connection failed
188             //send A1
189             final_fail = true;
190         }
191
192         break;
193
194     /*** 0xB3 - FILE_INFO_OK *****/
195
196     case 0xB3:
197
198         //got something, reset keepalive timer
199         keepAlive->reset();
200
201         log_this(sourcePort, sourceAddress, "SRV: FILE INFO OK from client received.");
202
203         if (myConnection->get_connection_status()) {
204             if (!myConnection->get_resend_counter()) {
205                 if (myConnection->get_seg_ack()) {
206                     myConnection->inc_seg_counter();
207                     myConnection->reset_resend_counter();
208                 }
209             } else {
210                 //connection failed
211                 //send A1
212                 final_fail = true;
213             }
214         } else {
215
216             //connection failed
217             //send A1
218             final_fail = true;
219         }
220     }
221

```

```

222         break;
223
224 /*** 0xB4 - SEG_ACK *****/
225
226         case 0xB4:
227
228             //got something, reset keepalive timer
229             keepAlive->reset();
230
231             log_this(sourcePort,sourceAddress,"SRV: SEG ACK from client received.");
232
233             if (myConnection->get_connection_status()) {
234                 myConnection->inc_seg_counter();
235                 //pretty quick? set timeouts to normal
236                 if (myConnection->get_resend_counter() < LIMIT_NORMAL) {
237                     myConnection->set_resend_limit(LIMIT_NORMAL);
238                 }
239
240                 myConnection->reset_resend_counter();
241             } else {
242
243                 //connection failed
244                 //send A1
245                 final_fail = true;
246             }
247
248             break;
249
250 /*** 0xB5 - SEG_ERR *****/
251         case 0xB5:
252
253             //got something, reset keepalive timer
254             keepAlive->reset();
255
256             log_this(sourcePort,sourceAddress,"SRV: SEG ERROR from client received.");
257
258             if (myConnection->get_connection_status()) {
259                 if (!myConnection->get_resend_counter()) {
260                     myConnection->inc_resend_counter();
261                 } else {
262                     //connection failed
263

```

```

264                                     //send A1
265                                     final_fail = true;
266                                     }
267
268     } else {
269
270         //connection failed
271         //send A1
272         final_fail = true;
273     }
274
275
276     break;
277
278 /*** 0x00 - unknown command *****/
279     default:
280
281         //got something, reset keepalive timer
282         keepAlive->reset();
283
284         log_this(sourcePort,sourceAddress,"SRV: UNKNOW PACKET TYPE. Ignoring.");
285         //protocoll error - unknown command, broken packet
286         //send 00
287
288         sock.sendTo(make_packet(&MyPacket,0x00,1,echoBuffer,sizeof(echoBuffer)), MyPacket.
                paketlaenge, sourceAddress, sourcePort);
289
290         cout << "Wrong ID, ignoring packet." << endl;
291
292
293     } //switch()
294
295 /*** GAME OVER - fresh restart *****/
296     if (final_fail) {
297
298         //sending 0xA1
299
300         unsigned char *packetBuffer = make_packet(&MyPacket,0xA1,0x00,echoBuffer,msgSize);
301         if (packetBuffer)
302             sock.sendTo(packetBuffer, MyPacket.paketlaenge, sourceAddress, sourcePort);
303         free(packetBuffer);
304         myConnection->set_connection_status(false);

```

```

305         myConnection->reset_resend_counter();
306
307         //Destroy objects
308         delete myConnection;
309         myConnection = NULL;
310     }
311
312
313     /** DATA ROCKET LAUNCHER *****/
314
315     //used to automatically send datapackages according to seg_count
316
317     /*if ( myConnection != NULL && myConnection->get_seg_counter() {
318
319         log_this(0,"127.0.0.1","SRV: SENDING DATA SEGMENT to client.");
320
321         //max. to last file-segment
322         if(myConnection->get_seg_counter() <= get_file_segment_count(myConnection->
323             getCurrentFile()) {
324
325             //send segment
326
327
328             unsigned char *packetBuffer = make_packet(MyPacket,0xA1,0x00,
329                 get_file_segment(myConnection->get_seg_counter()),
330                 myConnection->getCurrentFile()),
331                 sizeof(get_file_segment(myConnection->get_seg_counter()),
332                     myConnection->getCurrentFile
333                         ());
334
335             if (packetBuffer)
336                 sock.sendTo(packetBuffer, MyPacket.paketlaenge, sourceAddress,
337                     sourcePort);
338                 free(packetBuffer);
339
340             myConnection->reset_resend_counter();
341             myConnection->reset_seg_counter();
342             myConnection->reset_file_info_status();
343         }

```

```

344
345                                     //sent out new data packet, still waiting for ACK for THIS packet
346                                     //will send out again if not received within timeout limits
347                                     myConnection->set_seg_ack(false);
348                                     }*/
349
350                                 } //while(1)
351
352 /** catch standard exceptions *****/
353
354     //why does this class NOT output what has been given as argument?
355
356     } catch (SocketException &e) { // Exception-Behandlung
357         cout << e.what() << endl;           // Fehlerausgabe
358         //exit(1);                          // Programmabbruch
359
360     }
361
362     return 0;
363 }

```

3 eipush.cpp

```

1  #include <iostream>
2  #include <cstdlib> // für atoi()
3  #include <vector>
4  #include <string>
5  #include <time.h>
6  using namespace std;
7  using std::string;
8
9  #include "eipushd.h"
10 #include "SimpleSocket.h"
11
12
13 const int MAX_BUFFER = 255;
14 bool final_fail = false;
15
16 /* PACKET DEFINITIONS
17

```

```

18 FF - ACK
19 .00 - NACK
20 .A0 - INIT_REQ - unique_id
21 .A1 - FAIL
22 .A2 - INIT_ACK - unique_id + paranoia level.
23 .B0 - FILE_REQ - B1
24 .B1 - FILE_INFO
25 .B2 - FILE_ERR
26 .B3 - FILE_INFO_OK
27 .B4 - SEG_ERR
28 .B6 - SEG_DATA
29
30 */
31
32
33 int main(void) {
34
35
36     /** fresh init *****/
37     STrans *myConnection = NULL;
38     Packet MyPacket; //new packet struct
39
40
41     try {
42     while(1) {
43
44         final_fail = false;
45
46         log_this(0, "127.0.0.1", "Starting Client.");
47
48     /** setup menu *****/
49
50
51         int end = 0;
52         int selection = 0;
53         char buff [25];
54         char *end_ptr;
55         long long_var;
56         bool error;
57         unsigned int targetPort = 0;
58         bool connected = false;
59         string targetAddress, str2send;

```

```

60         do{
61             cout << "*****" << endl;
62             cout << "*" << endl;
63             cout << "* 1 : Server Adresse/Port aenden" << endl;
64             cout << "*" << endl;
65             cout << "* 2 : Verbinden" << endl;
66             cout << "*" << endl;
67             cout << "* 3 : Programm beenden" << endl;
68             cout << "*" << endl;
69             cout << "*****" << endl << endl;
70             // if(connected == true) { printf ("Verbunden zu %s, auf targetPort %i\n",targetAddress.c_str(),targetPort
// ); }
//
71             else { cout << "Nicht verbunden!" << endl; }
72             cout << "eipush> ";
73             cin >> selection;
74             cout << endl;
75
76             switch(selection) {
77                 case 1:
78                     cout << "Neue Server Adresse? ";
79                     cin >> targetAddress;
80                     do{
81                         cout << "Neuer Server targetPort (1-65535)? ";
82                         do{
83                             cin >> buff;
84                             long_var = strtol(buff, &end_ptr, 0);
85                             if (end_ptr == buff) { cout << endl << "Falsche Eingabe!" << endl;
86                                 error = true;}
87                             else {error = false; targetPort = long_var;}
88                             }while(error == true);
89                         }while((targetPort < 1) || (targetPort > 65535));
90                         connected=true;
91                         break;
92
93                 case 2:
94                     log_this(0,"127.0.0.1","CLT: Connection request by user.");
95                     end = 1;
96                     break;
97
98                 case 3:
99                     cout << "goodbye.";

```

```

100         log_this(0,"127.0.0.1","CLT: Stopping client.");
101         exit(1);
102
103         default:
104             cout << "Falsche Eingabe!" << endl;
105     }
106     if(end) break;
107
108 }while(selection != 5);
109
110
111
112 /** DEBUG - setup *****/
113
114     UDPSocket sock; // Socket-Objekt generieren
115     int msgSize; // Größe der empfangenen Nachricht
116     unsigned char echoBuffer[MAX_BUFFER]; // Puffer für Echo-String
117     unsigned char sendBuffer[MAX_BUFFER];
118
119     log_this(0,"127.0.0.1","CLT: Socket created.");
120
121     //DEBUG testing
122     unsigned char testsendung[] = "hallo welt";
123     //INIT request... connect to bla...
124     unsigned char *packetBuffer = make_packet(&MyPacket,160,0xfffff,testsendung,sizeof(testsendung));
125     if (packetBuffer)
126         sock.sendTo(packetBuffer, MyPacket.paketlaenge, targetAddress, targetPort);
127     free(packetBuffer);
128
129     log_this(targetPort,"127.0.0.1","CLT: INIT REQ sent to server.");
130
131     /** infinite working loop *****/
132     while (1) { //check for packets and take appropriate action
133
134
135         //new connection object if non-existent
136         if (myConnection == NULL) {
137             myConnection = new STrans(1); //pranoid mode - long timeouts
138         }
139
140     /** transfer menu *****/
141

```

```

142
143     if (myConnection->get_connection_status()) {
144
145     int auswahl = 0;
146     while (auswahl != -1 && myConnection->getCurrentFile() == "none"){
147         cout << "*****" << endl;
148         cout << "*" << endl;
149         cout << "* 1 : Datei anfordern" << endl;
150         cout << "*" << endl;
151         cout << "* 2 : Beenden" << endl;
152         cout << "*" << endl;
153         cout << "*****" << endl << endl;
154         string the_file;
155
156
157         cout << "eipush> ";
158         cin >> auswahl;
159         if(auswahl==1)
160         {
161             cout << "eipush> ";
162             cin >> the_file;
163             myConnection->set_current_file(the_file);
164             log_this(0,"127.0.0.1","CLT: Received FILE REQ from user.");
165
166         }
167         if(auswahl==2)
168         {
169             exit(1);
170             log_this(0,"127.0.0.1","CLT: Stopping client.");
171         }
172
173     }
174 }
175
176
177 cout << "      Verbindungstatus: " << myConnection->getStatusConnection() <<endl;
178 cout << "      Aktuelle Datei: " << myConnection->getCurrentFile() << endl;
179 cout << "Aktuelle Geschwindigkeit: " << endl;
180
181 msgSize = sock.recv(echoBuffer, MAX_BUFFER);
182
183 dissect_buffer(&MyPacket, echoBuffer, msgSize); //dissect packet into struct

```

```

184
185     switch (MyPacket.paketkennung) {
186
187
188         /** 0xA1 - FAIL *****/
189     case 0xA1:
190         cout << "FAIL von Server " << targetAddress <<":" << targetPort << " erhalten." << endl
191         ;
192         log_this(targetPort,targetAddress,"CLT: FAIL from server.");
193         final_fail = true;
194         break;
195
196         /** 0xA2 - INIT_ACK *****/
197     case 0xA2:
198         cout << "Init ACK von Server " << targetAddress <<":" << targetPort << " erhalten." <<
199         endl;
200         log_this(targetPort,targetAddress,"CLT: INIT ACK from server.");
201         if(!myConnection->get_connection_status()) {
202             //send back only if not connected, yet
203             log_this(0,"127.0.0.1","CLT: INIT ACK resent to server.");
204             unsigned char *packetBuffer = make_packet(&MyPacket,0xA2,0x00,echoBuffer,msgSize);
205             if (packetBuffer)
206                 sock.sendTo(packetBuffer, MyPacket.paketlaenge, targetAddress, targetPort);
207             free(packetBuffer);
208
209         }
210
211         myConnection->set_connection_status(true);
212         myConnection->reset_resend_counter();
213         log_this(0,"127.0.0.1","CLT: CONNECT SUCCESSFUL.");
214
215         break;
216
217         /** 0xB3 - FILE_INFO_OK *****/
218     case 0xB3:
219         log_this(targetPort,targetAddress,"CLT: FILE INFO OK received.");
220         if (myConnection->get_connection_status()) {
221             if (!myConnection->get_resend_counter()) {
222                 if (myConnection->get_seg_ack()) {
223                     myConnection->inc_seg_counter();
224                     myConnection->reset_resend_counter();

```

```

224         }
225     } else {
226
227         log_this(0,"127.0.0.1","CLT: RESEND LIMIT reached. Bailing out.");
228         //connection failed
229         //send A1
230         final_fail = true;
231
232     }
233
234 } else {
235     log_this(0,"127.0.0.1","CLT: NOT CONNECTED, no need for FILE INFO OK. Starting
236         over.");
237
238     //connection failed
239     //send A1
240
241     final_fail = true;
242 }
243 break;
244
245 /** 0xB4 - SEG_ACK *****/
246 case 0xB4:
247     log_this(targetPort,targetAddress,"CLT: SEG ACK received. Ready to send next segment.")
248     ;
249     if (myConnection->get_connection_status()) {
250         myConnection->inc_seg_counter();
251         //pretty quick? set timeouts to normal
252         if (myConnection->get_resend_counter() < LIMIT_NORMAL) {
253             myConnection->set_resend_limit(LIMIT_NORMAL);
254         }
255         myConnection->reset_resend_counter();
256     } else {
257         log_this(0,"127.0.0.1","CLT: NOT CONNECTED, no need for SEG ACK. Starting over.
258             ");
259         //connection failed
260         //send A1
261
262         final_fail = true;
263     }

```

```

263
264         break;
265
266         /** 0xB5 - SEG_ERR *****/
267     case 0xB5:
268         log_this(targetPort, targetAddress, "CLT: SEG ERR received.");
269         if (myConnection->get_connection_status()) {
270             if (!myConnection->get_resend_counter()) {
271                 //send again
272                 //log_this(0, "127.0.0.1", "RESEND LIMIT reached. Bailing out.");
273                 myConnection->inc_resend_counter();
274
275             } else {
276                 log_this(0, "127.0.0.1", "CLT: RESEND LIMIT reached. Bailing out.");
277                 //connection failed
278                 //send A1
279                 final_fail = true;
280
281             }
282
283         } else {
284             log_this(0, "127.0.0.1", "CLT: NOT CONNECTED, no need for SEG ERR. Starting over.");
285             //connection failed
286             //send A1
287
288             final_fail = true;
289         }
290
291         break;
292
293         /** 0x00 - unknown command *****/
294     default:
295         log_this(0, "127.0.0.1", "CLT: UNKNOWN PACKET TYPE. Ignoring.");
296         //protocoll error - unknown command, broken packet
297         //send 00
298
299         //         int z = sizeof(make_packet(0,1,echoBuffer));
300         //                                     cout << "testlaenge" << z << endl;
301
302         cout << echoBuffer << endl;
303

```

```

304         //sock.sendTo(make_packet(&MyPacket,0,1,echoBuffer), MyPacket.paketlaenge,
305         targetAddress, targetPort);
306         cout << "Wrong ID, ignoring packet." << endl;
307
308     } //switch()
309
310
311     /** FILE REQUEST *****/
312
313     if (myConnection->get_connection_status() && myConnection->getCurrentFile() != "none") {
314
315         string request = myConnection->getCurrentFile();
316         int length = request.length();
317         unsigned char filerequest[length];
318         for(int i=0; i<length;i++) {
319             filerequest[i] = request.at(i);
320         }
321
322         unsigned char *packetBuffer = make_packet(&MyPacket,0xB0,0xffffffff,filerequest,sizeof(
323             filerequest));
324         if (packetBuffer)
325             sock.sendTo(packetBuffer, MyPacket.paketlaenge, targetAddress, targetPort);
326         free(packetBuffer);
327         log_this(0,"127.0.0.1","CLT: Sent FILE REQ to Server.");
328         cout << "Requesting file " << request << " from Server" << endl;
329     }
330
331     /** GAME OVER - fresh restart *****/
332     if (final_fail) {
333
334         //sending 0xA1
335
336         unsigned char *packetBuffer = make_packet(&MyPacket,0xA1,0x00,echoBuffer,msgSize);
337         if (packetBuffer)
338             sock.sendTo(packetBuffer, MyPacket.paketlaenge, targetAddress, targetPort);
339         free(packetBuffer);
340         myConnection->set_connection_status(false);
341         myConnection->reset_resend_counter();
342
343         //Destroy objects

```

```

344         delete myConnection;
345         myConnection = NULL;
346         break;
347     }
348
349
350     } //while(1)
351
352 }
353     /** catch standard exceptions *****/
354
355 } catch (SocketException &e) { // Exception-Behandlung
356     cout << e.what() << endl; // Fehlerausgabe
357
358                                     //exit(1);
359                                     // Programmabbruch
360 }
361
return 0;
}

```

4 SimpleSocket.cpp

```

1 // *****
2 // * vereinfachte TCP/IP Übertragung mit Windows *
3 // * Quelltextdatei: SimpleSocket.cpp *
4 // *****
5 //
6 //     Kompiliert auch unter diversen *NIX Varianten mit GCC ab V3
7 //     Portiert quick&dirty - Martin Henning / Michael Dirkska 02/07
8 //
9
10 #include "SimpleSocket.h"
11
12 #ifdef __APPLE__
13
14 #include <sys/types.h>
15 #include <sys/socket.h>
16 #include <netinet/in.h>
17 #include <arpa/inet.h>
18 #include <netdb.h>

```

```

19 #define WSAEAFNOSUPPORT 0
20
21 #else
22
23 #include <winsock.h>
24
25 typedef int socklen_t;
26
27 #endif
28
29 typedef char raw_type;
30 #include <errno.h>
31 using namespace std;
32 static bool initialized = false;
33
34
35 // SocketException Klasse
36
37 SocketException::SocketException(const string &message, bool inclSysMsg) throw() : userMessage(message) {
38     if (inclSysMsg) {
39         userMessage.append(": ");
40         userMessage.append(strerror(errno));
41     }
42 }
43
44 SocketException::~SocketException() throw() {}
45
46 const char *SocketException::what() const throw() {return userMessage.c_str();}
47
48
49 // globale Funktion zum Einfügen der IP-Adresse und des Ports in Adresstruktur
50
51 static void fillAddr(const string &address, unsigned short port, sockaddr_in &addr) {
52     memset(&addr, 0, sizeof(addr)); // Zero out address structure
53     addr.sin_family = AF_INET; // Internet address
54     hostent *host; // Resolve name
55     if ((host = gethostbyname(address.c_str())) == NULL) {
56         throw SocketException("Failed to resolve name (gethostbyname())");
57     }
58     addr.sin_addr.s_addr = *((unsigned long *) host->h_addr_list[0]);
59     addr.sin_port = htons(port); // Assign port in network byte order
60 }

```

```

61
62
63 // Socket Klasse
64
65 Socket::Socket(int type, int protocol) throw(SocketException) {
66
67     if (!initialized) {
68
69 #ifndef __APPLE__
70
71         WORD wVersionRequested;
72         WSADATA wsaData;
73         wVersionRequested = MAKEWORD(2, 0);           // Request WinSock v2.0
74         if (WSAStartup(wVersionRequested, &wsaData) != 0) { // Load WinSock DLL
75             throw SocketException("Unable to load WinSock DLL");
76         }
77
78 #endif
79
80         initialized = true;
81     }
82     if ((sockDesc = socket(PF_INET, type, protocol)) < 0) {
83         throw SocketException("Socket creation failed (socket())", true);
84     }
85 }
86
87 Socket::Socket(int sockDesc) {
88     this->sockDesc = sockDesc;
89 }
90
91 Socket::~Socket() {
92
93 #ifdef __APPLE__
94     close(sockDesc);
95 #else
96     ::closesocket(sockDesc);
97 #endif
98     sockDesc = -1;
99 }
100
101 string Socket::getLocalAddress() throw(SocketException) {
102     sockaddr_in addr;

```

```

103 unsigned int addr_len = sizeof(addr);
104 if (getsockname(sockDesc, (sockaddr *) &addr, (socklen_t *) &addr_len) < 0) {
105     throw SocketException("Fetch of local address failed (getsockname())", true);
106 }
107 return inet_ntoa(addr.sin_addr);
108 }
109
110 unsigned short Socket::getLocalPort() throw(SocketException) {
111     sockaddr_in addr;
112     unsigned int addr_len = sizeof(addr);
113     if (getsockname(sockDesc, (sockaddr *) &addr, (socklen_t *) &addr_len) < 0) {
114         throw SocketException("Fetch of local port failed (getsockname())", true);
115     }
116     return ntohs(addr.sin_port);
117 }
118
119 void Socket::setLocalPort(unsigned short localPort) throw(SocketException) {
120     // Socket an Port binden
121     sockaddr_in localAddr;
122     memset(&localAddr, 0, sizeof(localAddr));
123     localAddr.sin_family = AF_INET;
124     localAddr.sin_addr.s_addr = htonl(INADDR_ANY);
125     localAddr.sin_port = htons(localPort);
126     if (bind(sockDesc, (sockaddr *) &localAddr, sizeof(sockaddr_in)) < 0) {
127         throw SocketException("Set of local port failed (bind())", true);
128     }
129 }
130
131 void Socket::setLocalAddressAndPort(const string &localAddress, unsigned short localPort) throw(SocketException) {
132     // Host-Adresse ermitteln
133     sockaddr_in localAddr;
134     fillAddr(localAddress, localPort, localAddr);
135     if (bind(sockDesc, (sockaddr *) &localAddr, sizeof(sockaddr_in)) < 0) {
136         throw SocketException("Set of local address and port failed (bind())", true);
137     }
138 }
139
140 void Socket::cleanUp() throw(SocketException) {
141
142 #ifndef __APPLE__
143
144     if (WSACleanup() != 0) {

```

```

145     throw SocketException("WSACleanup() failed");
146 }
147
148 #endif
149
150 }
151
152 unsigned short Socket::resolveService(const string &service, const string &protocol) {
153     struct servent *serv;           // Struktur mit Dienst-Information
154     if ((serv = getservbyname(service.c_str(), protocol.c_str())) == NULL)
155         return atoi(service.c_str()); // Service entspricht Port-Nummer
156     else
157         return ntohs(serv->s_port);  // Gefundener Port (network byte order)
158 }
159
160 // Klasse CommunicatingSocket
161
162 CommunicatingSocket::CommunicatingSocket(int type, int protocol) throw(SocketException) : Socket(type, protocol) {}
163
164 CommunicatingSocket::CommunicatingSocket(int newConnSD) : Socket(newConnSD) {}
165
166 void CommunicatingSocket::connect(const string &foreignAddress, unsigned short foreignPort) throw(SocketException) {
167     // Host-Adresse ermitteln
168     sockaddr_in destAddr;
169     fillAddr(foreignAddress, foreignPort, destAddr);
170     // Versuch eines Verbindungsaufbaus zum gegebenen Host
171     if (::connect(sockDesc, (sockaddr *) &destAddr, sizeof(destAddr)) < 0) {
172         throw SocketException("Connect failed (connect())", true);
173     }
174 }
175
176 void CommunicatingSocket::send(const void *buffer, int bufferLen) throw(SocketException) {
177     if (::send(sockDesc, (raw_type *) buffer, bufferLen, 0) < 0) {
178         throw SocketException("Send failed (send())", true);
179     }
180 }
181
182 int CommunicatingSocket::recv(void *buffer, int bufferLen) throw(SocketException) {
183     int rtn;
184     if ((rtn = ::recv(sockDesc, (raw_type *) buffer, bufferLen, 0)) < 0) {
185         throw SocketException("Received failed (recv())", true);
186     }

```

```

187     return rtn;
188 }
189
190 string CommunicatingSocket::getForeignAddress() throw(SocketException) {
191     sockaddr_in addr;
192     unsigned int addr_len = sizeof(addr);
193     if (getpeername(sockDesc, (sockaddr *) &addr, (socklen_t *) &addr_len) < 0) {
194         throw SocketException("Fetch of foreign address failed (getpeername())", true);
195     }
196     return inet_ntoa(addr.sin_addr);
197 }
198
199 unsigned short CommunicatingSocket::getForeignPort() throw(SocketException) {
200     sockaddr_in addr;
201     unsigned int addr_len = sizeof(addr);
202     if (getpeername(sockDesc, (sockaddr *) &addr, (socklen_t *) &addr_len) < 0) {
203         throw SocketException("Fetch of foreign port failed (getpeername())", true);
204     }
205     return ntohs(addr.sin_port);
206 }
207
208 // Code für Klasse TCP Socket
209
210 TCPSocket::TCPSocket() throw(SocketException) : CommunicatingSocket(SOCK_STREAM, IPPROTO_TCP) {}
211
212 TCPSocket::TCPSocket(const string &foreignAddress, unsigned short foreignPort) throw(SocketException) : CommunicatingSocket(
213     SOCK_STREAM, IPPROTO_TCP) {
214     connect(foreignAddress, foreignPort);
215 }
216
217 TCPSocket::TCPSocket(int newConnSD) : CommunicatingSocket(newConnSD) {}
218
219 // Code für Klasse TCP Server Socket
220
221 TCPServerSocket::TCPServerSocket(unsigned short localPort, int queueLen) throw(SocketException) : Socket(SOCK_STREAM,
222     IPPROTO_TCP) {
223     setLocalPort(localPort);
224     setListen(queueLen);
225 }
226
227 TCPServerSocket::TCPServerSocket(const string &localAddress, unsigned short localPort, int queueLen) throw(SocketException) :
228     Socket(SOCK_STREAM, IPPROTO_TCP) {

```

```

226     setLocalAddressAndPort(localAddress, localPort);
227     setListen(queueLen);
228 }
229
230 TCPSocket *TCPServerSocket::accept() throw(SocketException) {
231     int newConnSD;
232     if ((newConnSD = ::accept(sockDesc, NULL, 0)) < 0) {
233         throw SocketException("Accept failed (accept())", true);
234     }
235     return new TCPSocket(newConnSD);
236 }
237
238 void TCPServerSocket::setListen(int queueLen) throw(SocketException) {
239     if (listen(sockDesc, queueLen) < 0) {
240         throw SocketException("Set listening socket failed (listen())", true);
241     }
242 }
243
244
245 // Code für Klasse UDPsocket
246
247 UDPsocket::UDPsocket() throw(SocketException) : CommunicatingSocket(SOCK_DGRAM,
248     IPPROTO_UDP) {
249     setBroadcast();
250 }
251
252 UDPsocket::UDPsocket(unsigned short localPort) throw(SocketException) : CommunicatingSocket(SOCK_DGRAM, IPPROTO_UDP) {
253     setLocalPort(localPort);
254     setBroadcast();
255 }
256
257 UDPsocket::UDPsocket(const string &localAddress, unsigned short localPort) throw(SocketException) : CommunicatingSocket(
258     SOCK_DGRAM, IPPROTO_UDP) {
259     setLocalAddressAndPort(localAddress, localPort);
260     setBroadcast();
261 }
262 void UDPsocket::setBroadcast() {
263     // If this fails, we'll hear about it when we try to send. This will allow
264     // system that cannot broadcast to continue if they don't plan to broadcast
265     int broadcastPermission = 1;
266     setsockopt(sockDesc, SOL_SOCKET, SO_BROADCAST, (raw_type *) &broadcastPermission, sizeof(broadcastPermission));

```

```

267 }
268
269 void UDPSocket::disconnect() throw(SocketException) {
270     sockaddr_in nullAddr;
271     memset(&nullAddr, 0, sizeof(nullAddr));
272     nullAddr.sin_family = AF_UNSPEC;
273     // Verbindung abzubauen versuchen
274     if (::connect(sockDesc, (sockaddr *) &nullAddr, sizeof(nullAddr)) < 0) {
275         if (errno != WSAEAFNOSUPPORT) {
276             throw SocketException("Disconnect failed (disconnect())", true);
277         }
278     }
279 }
280
281 void UDPSocket::sendTo(const void *buffer, int bufferLen,
282     const string &foreignAddress, unsigned short foreignPort)
283     throw(SocketException) {
284     sockaddr_in destAddr;
285     fillAddr(foreignAddress, foreignPort, destAddr);
286     // Gesamtpuffer als einzelne Nachricht senden
287     if (sendto(sockDesc, (raw_type *) buffer, bufferLen, 0, (sockaddr *) &destAddr, sizeof(destAddr)) != bufferLen) {
288         throw SocketException("Send failed (sendTo())", true);
289     }
290 }
291
292 int UDPSocket::recvFrom(void *buffer, int bufferLen, string &sourceAddress, unsigned short &sourcePort) throw(SocketException)
293 {
294     sockaddr_in clntAddr;
295     socklen_t addrLen = sizeof(clntAddr);
296     int rtn;
297     if ((rtn = recvfrom(sockDesc, (raw_type *) buffer, bufferLen, 0, (sockaddr *) &clntAddr, (socklen_t *) &addrLen)) < 0) {
298         throw SocketException("reception failed (recvFrom())", true);
299     }
300     sourceAddress = inet_ntoa(clntAddr.sin_addr);
301     sourcePort = ntohs(clntAddr.sin_port);
302     return rtn;
303 }
304
305 void UDPSocket::setMulticastTTL(unsigned char multicastTTL) throw(SocketException) {
306     if (setsockopt(sockDesc, IPPROTO_IP, IP_MULTICAST_TTL,
307         (raw_type *) &multicastTTL, sizeof(multicastTTL)) < 0) {
308         throw SocketException("Multicast TTL set failed (setsockopt())", true);
309     }
310 }

```

```

308     }
309 }
310
311 void UDPSocket::joinGroup(const string &multicastGroup) throw(SocketException) {
312     struct ip_mreq multicastRequest;
313     multicastRequest.imr_multiaddr.s_addr = inet_addr(multicastGroup.c_str());
314     multicastRequest.imr_interface.s_addr = htonl(INADDR_ANY);
315     if (setsockopt(sockDesc, IPPROTO_IP, IP_ADD_MEMBERSHIP, (raw_type *) &multicastRequest, sizeof(multicastRequest)) < 0) {
316         throw SocketException("Multicast group join failed (setsockopt())", true);
317     }
318 }
319
320 void UDPSocket::leaveGroup(const string &multicastGroup) throw(SocketException) {
321     struct ip_mreq multicastRequest;
322     multicastRequest.imr_multiaddr.s_addr = inet_addr(multicastGroup.c_str());
323     multicastRequest.imr_interface.s_addr = htonl(INADDR_ANY);
324     if (setsockopt(sockDesc, IPPROTO_IP, IP_DROP_MEMBERSHIP, (raw_type *) &multicastRequest, sizeof(multicastRequest)) < 0) {
325         throw SocketException("Multicast group leave failed (setsockopt())", true);
326     }
327 }

```

5 SimpleSocket.h

```

1 // *****
2 //     * vereinfachte TCP/IP ?bertragung mit Windows *
3 // * Header-Datei: SimpleSocket.h *
4 // *****
5 #include <string>
6 #include <exception>
7 using namespace std;
8
9
10 // Exception-Klasse zur Meldung von Fehlerzust?nden
11 class SocketException : public exception {
12 private:
13     string userMessage; // Exception message
14 public:
15     SocketException(const string &message, bool inclSysMsg = false) throw();
16
17     ~SocketException() throw();

```

```

18
19  const char *what() const throw();
20 };
21
22 // Basis-Klasse f,r das Einrichten von Sockets
23 class Socket {
24 public:
25     ~Socket();
26
27     string getLocalAddress() throw(SocketException);           // liefert lokale IP-Adresse
28
29     unsigned short getLocalPort() throw(SocketException); // liefert lokalen Port
30
31     void setLocalPort(unsigned short localPort) throw(SocketException); // lokalen Port setzen
32
33     void setLocalAddressAndPort(const string &localAddress, unsigned short localPort = 0) throw(SocketException); // setzen von
        Port und Adresse
34
35     static void cleanUp() throw(SocketException); // L?schen der Speicherbelegung durch Sockets
36
37     // Generierung einer Portnummer abh?ngig von service und Protokoll
38     static unsigned short resolveService(const string &service, const string &protocol = "tcp");
39 private:
40     // Prevent the user from trying to use value semantics on this object
41     Socket(const Socket &sock);
42     void operator=(const Socket &sock);
43 protected:
44     int sockDesc;           // Socket descriptor
45     Socket(int type, int protocol) throw(SocketException);
46     Socket(int sockDesc);
47 };
48
49 // Abgeleitete Socket-Klasse mit den F?higkeiten "verbinden", "senden" und "empfangen"
50 class CommunicatingSocket : public Socket {
51 public:
52     void connect(const string &foreignAddress, unsigned short foreignPort) throw(SocketException); // Aufbau einer Verbindung
53
54     void send(const void *buffer, int bufferLen) throw(SocketException); // Senden von Daten
55
56     int recv(void *buffer, int bufferLen) throw(SocketException); // Empfangen von Daten
57
58     string getForeignAddress() throw(SocketException); // Ermittlung der Host IP-Adresse

```

```

59
60 unsigned short getForeignPort() throw(SocketException); // Ermittlung des Host Port
61 protected:
62 CommunicatingSocket(int type, int protocol) throw(SocketException);
63
64 CommunicatingSocket(int newConnSD);
65 };
66
67 // Einrichten eines TCP-Sockets f,r einen Klienten
68 class TCPSocket : public CommunicatingSocket {
69 public:
70 TCPSocket() throw(SocketException); // TCP-Socket ohne Verbindung
71
72 TCPSocket(const string &foreignAddress, unsigned short foreignPort) throw(SocketException);
73 private:
74 // Access for TCPServerSocket::accept() connection creation
75 friend class TCPServerSocket;
76 TCPSocket(int newConnSD);
77 };
78
79 // Einrichten eines TCP-Sockets f,r einen Server
80 class TCPServerSocket : public Socket {
81 public:
82 // Erzeugt einen TCP-Socket f,r einen Server
83 // Verbindungen auf dem angegebenen Port ,ber jede Schnittstelle werden akzeptiert
84 // Bei localPort=0 wird ein freier Port automatisch zugewiesen
85 // queueLen legt die maximale Anzahl der offenen Verbindungsaufforderungen fest
86 TCPServerSocket(unsigned short localPort, int queueLen = 5) throw(SocketException);
87
88 // Erzeugt einen TCP-Socket f,r einen Server
89 // Verbindungen auf dem angegebenen Port und der festgelegten Schnittstelle werden akzeptiert
90 // Bei localPort=0 wird ein freier Port automatisch zugewiesen
91 // queueLen legt die maximale Anzahl der offenen Verbindungsaufforderungen fest
92 TCPServerSocket(const string &localAddress, unsigned short localPort,
93 int queueLen = 5) throw(SocketException);
94
95 // blockiert bis zum Aufbau einer neuen Socket-Verbindung oder Fehlermeldung
96 TCPSocket *accept() throw(SocketException);
97 private:
98 void setListen(int queueLen) throw(SocketException);
99 };
100

```

```

101 // Einrichten eines UDP-Sockets
102 class UDPSocket : public CommunicatingSocket {
103 public:
104     UDPSocket() throw(SocketException); // UDP-Socket
105
106     UDPSocket(unsigned short localPort) throw(SocketException); // UDP-Socket mit Portnummer
107
108     UDPSocket(const string &localAddress, unsigned short localPort) throw(SocketException); // UDP-Socket mit Adresse und
        Portnummer
109
110     void disconnect() throw(SocketException); // Freigabe von Adresse und Port
111
112     // Senden eines Datagrammes mit vorgegebener Länge an spezifizierte Adresse und Port
113     void sendTo(const void *buffer, int bufferLen, const string &foreignAddress, unsigned short foreignPort) throw(SocketException
        );
114     // Empfangen eines Datagrammes mit vorgegebener Länge von spezifizierter Adresse und Port
115     int recvFrom(void *buffer, int bufferLen, string &sourceAddress,
        unsigned short &sourcePort) throw(SocketException);
116
117     // Setzen der Multicast TTL
118     void setMulticastTTL(unsigned char multicastTTL) throw(SocketException);
119     // Der spezifizierten Multicast-Gruppe beitreten
120     void joinGroup(const string &multicastGroup) throw(SocketException);
121     // Multicast-Gruppe verlassen
122     void leaveGroup(const string &multicastGroup) throw(SocketException);
123 private:
124     void setBroadcast();
125 };

```