```
----------------------------------------------------------------------
PC電話クライアントソースリスト
  iptel.h

#define BLKSIZE 160
#define RATE 8000

#define MAXHASH 20
#define CACHE 5
#define MYNUMBER "99900000000001"
#define GKADDRESS "203.178.141.176"

                        /* 使用するオーディオデバイスによって変更 */
#define AUDIO_DEVICE "/dev/audio0"
#define FULLDUPLEX 1
#define HALFDUPLEX 0
#define ON 1
#define OFF 0
                        /* 使用するポート番号によって変更 */
#define MYLISTENPORT 7000
#define MYDATAPORT 7001
#define GKLISTENPORT 16000
#define DATA 0
#define CTL 1
#define IDLE    1       /* 待機         */
#define CALLED  2       /* 呼ばれている */
#define CALLING 3       /* 呼んでいる   */
#define TALKING 4       /* 通話中       */
#define GW_YOUTALKING 5  /* 通話相手が話し中  */
#define GW_CALLING    6  /* 自局に着信あり    */
#define GW_NOTHING    7  /* 相手が電話に出ない*/
#define GW_SUCCEED    8  /* 通話に成功        */
#define GW_ERROR      9  /* エラー発生        */

#define PT_GSM 3        /* PAYLOAD TYPE GSM */

#define MD_CTX MD5_CTX
#define MDInit MD5Init
#define MDUpdate MD5Update
#define MDFinal MD5Final

#define REQ_MSG     0x0001
#define ACCEPT_MSG  0x0002
#define REFUSE_MSG  0x0003
#define FIN_MSG     0x0004
#define GKREQ_MSG   0x0005
#define GKREP_MSG   0x0006
#define GKUPDATE_MSG 0x0007
#define GKUPDATEREP_MSG 0x0008

char menu[] =
"    d number : dial \n"
"    a        : answer incoming call\n"
"    s        : stop call or refuse call\n"
"    q        : quit\n"
"    ?        : show status\n"
;

struct iptel_status {
  u_long  your_ipaddr;     /* 通話相手のIPアドレス network order */
  u_short your_listen_port;/* 通話相手の待ち受けポート host order*/
  u_short your_data_port;  /* 通話相手の音声パケットポート host order*/
  u_short my_listen_port;  /* 自分の待ち受けポート host order*/
  u_short my_data_port;    /* 自分の音声パケットポート host order*/
  char my_number[32];      /* 自分のIP電話番号 */
  char your_number[32];    /* 通話相手の電話番号 */
```



```
  u_long gk_ipaddr;        /* gatekeeperのIPアドレス */
  u_short gk_port;         /* gatekeeperのポート */
  u_long play_fmt;         /* 再生フォーマット */
  u_long rec_fmt;          /* 録音フォーマット */

  int listen_sock;         /* 待ち受け用ソケット(TCP) */
  int ctl_sock;            /* 制御パケット用ソケット(TCP) */
  int data_sock;           /* 音声パケット用ソケット(UDP) */
  int tmp_sock;            /* 待ち受け用ソケット for catch phone(TCP) */

  int tty_fd;              /* コマンド入力FD */
  int audio_fd;            /* オーディオデバイス FD  */

  unsigned long nowtime;   /* RATE 1カウントにつき1カウントが増えるカウンター
*/
  unsigned long my_ssrc;   /* RTP Syncronization src */
  unsigned long delta;

  int suppress;            /* 無音圧縮フラグ */
  int duplex;              /* 全二重フラグ   */
  int status;              /* 通話状態フラグ */
};

struct msg_header{
  u_short msg_id;         /* message ID      */
  u_short msg_len;        /* message_lendgth */
  char my_number[32];     /* IP telephone number */
};

struct req_msg{
  char your_number[32];  /* IP telephone number */
};

struct refuse_msg{
  u_short reply_type;    /* refuse type */
};

struct gkreq_msg{
  char your_number[32];  /* IP telephone number */
};

struct gkrep_msg{
  u_short rep_type;      /* reply type 0x0001=FOUND 0x0002=NOTFOUND*/
  u_long  ipaddr;        /* client(or gk) ipaddr */
  u_short listen_port;   /* client(or gk) listen_port */
  u_short data_port;     /* client(or gk) data_port */
  char your_number[32];  /* dialing telephone number */
};

struct gkupdate_msg{
  u_short listen_port;   /* my listen_port */
  u_short data_port;     /* my data_port */
};

struct gkupdaterep_msg{
  u_short rep_type;      /* reply type */
};

struct rtphdr {          /* RTP ヘッダ */
#if _BIG_ENDIAN
  unsigned int v:2;
  unsigned int p:1;
  unsigned int x:1;
  unsigned int cc:4;
  unsigned int m:1;
  unsigned int pt:7;
```

```
128    unsigned int seq:16;
129  #else
130    unsigned int seq:16;
131    unsigned int m:1;
132    unsigned int pt:7;
133    unsigned int v:2;
134    unsigned int p:1;
135    unsigned int x:1;
136    unsigned int cc:4;
137  #endif
138    unsigned long timestamp;
139    unsigned long ssrc;
140    gsm_frame data;
141    char padding;
142  };
143
144  struct snd_queue {
145    unsigned long timestamp;
146    gsm_frame data;
147  };
148
149
150
151
152
153
154
```

iptel.c

```
  1  /* IP telephone  1998.*/
  2
  3  #include <stdio.h>
  4  #include <unistd.h>
  5  #include <fcntl.h>
  6  #include <machine/soundcard.h>
  7  #include <netinet/in.h>
  8  #include <netdb.h>
  9  #include <sys/types.h>
 10  #include <sys/socket.h>
 11  #include <sys/ioctl.h>
 12  #include <sys/time.h>    /* gettimeofday() */
 13  #include <sys/utsname.h> /* uname() */
 14  #include <sys/errno.h>
 15  #include <md5.h>
 16  #include <time.h>        /* clock() */
 17  #include "gsm.h"
 18  #include "iptel.h"
 19
 20  struct iptel_status iptel;  /* IP電話のパラメータ */
 21  struct snd_queue queue[MAXHASH];
 22  unsigned long start_time;
 23  int queue_num;
 24  int errno;
 25
 26  int readn(int fd, char *ptr, int nbytes);
 27  int writen(int fd, char *ptr, int nbytes);
 28  static u_long md_32(char *string, int length);
 29  unsigned int random32(int type);
 30  int audio_open();
 31  void audio_read();
 32  void audio_write();
 33  void net_read();
 34  void trans_sockaddr(struct sockaddr_in *net , u_long addr , u_short port);
 35  void make_listen_sock();
 36  int connect_sock(u_long ipaddr , u_short port);
```



```
 37  void close_sock(int *sock);
 38  void print_menu();
 39  void send_msg(int msg_id);
 40  void recv_msg(char *buf);
 41  void cmd_line(char *buf);
 42
 43  int readn(int fd, char *ptr, int nbytes)
 44  {
 45          int nleft,n;
 46          nleft = nbytes;
 47          while(nleft > 0){
 48                  n = read(fd,ptr,nleft);
 49                  if(n < 0){
 50                    return(n);
 51                  }
 52                  else if(n == 0){
 53                    break;
 54                  }
 55                  nleft -= n;
 56                  ptr += n;
 57          }
 58          return(nbytes - nleft);
 59  }
 60
 61  int writen(int fd, char *ptr, int nbytes)
 62  {
 63          int nleft,nwritten;
 64          nleft = nbytes;
 65          while(nleft > 0){
 66                  nwritten = write(fd,ptr,nleft);
 67                  if (nwritten < 0){
 68                    return (nwritten);
 69                  }
 70                  else if(nwritten == 0){
 71                    break;
 72                  }
 73                  nleft -= nwritten;
 74                  ptr += nwritten;
 75          }
 76          return(nbytes - nleft);
 77  }
 78
 79  static u_long md_32(char *string, int length)
 80  {
 81    MD_CTX context;
 82    union {
 83      char   c[16];
 84      u_long x[4];
 85    } digest;
 86    u_long r;
 87    int i;
 88
 89    MDInit (&context);
 90    MDUpdate (&context, string, length);
 91    MDFinal ((unsigned char *)&digest, &context);
 92    r = 0;
 93    for (i = 0; i < 3; i++) {
 94      r ^= digest.x[i];
 95    }
 96    return r;
 97  }
 98
 99  unsigned int random32(int type)
100  {
101    struct {
102      int     type;
```

```
    struct  timeval tv;
    clock_t cpu;
    pid_t   pid;
    u_long  hid;
    uid_t   uid;
    gid_t   gid;
    struct  utsname name;
  } s;

  gettimeofday(&s.tv, 0);
  uname(&s.name);
  s.type = type;
  s.cpu  = clock();
  s.pid  = getpid();
  s.hid  = gethostid();
  s.uid  = getuid();
  s.gid  = getgid();

  return md_32((char *)&s, sizeof(s));
}                               /* random32 */

/*------------------------------------------------------------
 * int audio_open()
 *          オーディオデバイスのオープン
 *          オーディオデバイスの初期化
 *          オーディオデバイスのFDを返す
 *------------------------------------------------------------*/
int audio_open()
{
  int audio_fd;               /* オーディオデバイスのFD */
  int i=-1;
  u_long fmt,play_fmt,rec_fmt;
  snd_chan_param sndparam;    /* オーディオの設定 */
  snd_capabilities soundcaps; /* オーディオのサポート機能 */
  struct snd_size sndsize;    /* オーディオバッファサイズ */

  /* オーディオデバイス オープン */
  audio_fd = open(AUDIO_DEVICE,O_RDWR);
  if (audio_fd < 0) {
    fprintf(stderr,"can not open audio_device %s\n",AUDIO_DEVICE);
    exit(-1);
  }

  i=ioctl(audio_fd,AIOGCAP, &soundcaps);
  fmt=soundcaps.formats;

  /* 使用できるサウンドフォーマットを調べる */
  switch(fmt & (AFMT_FULLDUPLEX | AFMT_WEIRD)) {
  case AFMT_FULLDUPLEX:
    /* if sound card supports FULL-DUPLEX operation */
    if ( fmt & AFMT_S16_LE )
      play_fmt=rec_fmt=AFMT_S16_LE;
    else {
      fprintf(stderr,"no supported sound formats\n");
      close(audio_fd);
      exit(-1);
    }
    iptel.duplex=FULLDUPLEX;
    break;
  case AFMT_FULLDUPLEX | AFMT_WEIRD:
    /* if sound card supports FULL-DUPLEX operation but weird hardware */
    if (fmt & AFMT_S16_LE){
      rec_fmt=AFMT_S16_LE;
      play_fmt=AFMT_U8;
    }
    else {
```



```
      fprintf(stderr,"no supported sound formats\n");
      close(audio_fd);
      exit(-1);
    }
    iptel.duplex=FULLDUPLEX;
    break;
  default:
    /* if sound card can't support FULL-DUPLEX operation */
    printf("Half Duplex Mode\n");
    if ( fmt & AFMT_U8 ){
      printf("unsigned 8bits format \n");
      rec_fmt=AFMT_U8;
      play_fmt=AFMT_U8;
    }
    iptel.duplex=HALFDUPLEX;
    break;
  }

  /* オーディオフォーマットの設定 */
  switch(iptel.duplex){
  case FULLDUPLEX:
    sndparam.rec_format=rec_fmt;
    sndparam.play_format=play_fmt;
    sndparam.play_rate=sndparam.rec_rate=RATE;
    ioctl(audio_fd,AIOSFMT,&sndparam);
    sndsize.play_size= (sndparam.play_format == AFMT_S16_LE) ? 2*BLKSIZE : BL
KSIZE;
    sndsize.rec_size=  (sndparam.rec_format  == AFMT_S16_LE) ? 2*BLKSIZE : BL
KSIZE;
    i = ioctl(audio_fd,AIOSSIZE,&sndsize);

    iptel.play_fmt=play_fmt;
    iptel.rec_fmt=rec_fmt;
    break;
  case HALFDUPLEX:
    sndparam.rec_format=rec_fmt;
    sndparam.play_format=play_fmt;
    sndparam.play_rate=sndparam.rec_rate=RATE;
    ioctl(audio_fd,AIOSFMT,&sndparam);
    sndsize.play_size= (sndparam.play_format == AFMT_S16_LE) ? 2*BLKSIZE : BL
KSIZE;
    sndsize.rec_size=  (sndparam.rec_format  == AFMT_S16_LE) ? 2*BLKSIZE : BL
KSIZE;
    i = ioctl(audio_fd,AIOSSIZE,&sndsize);

    iptel.play_fmt=play_fmt;
    iptel.rec_fmt=rec_fmt;

/*    sndparam.play_format=play_fmt;
    sndparam.play_rate=RATE;
    ioctl(audio_fd,AIOSFMT,&sndparam);
    sndsize.play_size=BLKSIZE;
    i=ioctl(audio_fd,AIOSSIZE,&sndsize);*/


    break;
  }

  return audio_fd;
}

/*------------------------------------------------------------
 * void audio_read()
 *          オーディオデバイスから音声を取り込む
 *          音声をGSMエンコードする
 *          エンコードしたデータをネットワークに投げる
```

```
 *-------------------------------------------------------*/
void audio_read()
{
  static struct rtphdr *rtp=NULL;
  struct sockaddr_in you;
  double energy = 0.0;
  double avg ;
  short buf[BLKSIZE];    /* 16bits用録音バッファ */
  u_char ubuf[BLKSIZE];  /*  8bits用録音バッファ */
  int len;               /* サンプルのバイト数   */
  int i;

  /* RTPパケットの初期化 */
  if (!rtp ){
    rtp = (struct rtphdr *)malloc(sizeof(*rtp));
    rtp->v = 2;
    rtp->p = 0;
    rtp->x = 0;
    rtp->cc = 0;
    rtp->m = 1;
    rtp->pt = PT_GSM;
    rtp->seq = htons(0x1234);
    rtp->timestamp = htonl(iptel.nowtime);
    rtp->padding=0;
  }

  /* オーディオ録音 */
  switch(iptel.rec_fmt){
  case AFMT_S16_LE:
    /* BLKSIZE個のサンプルを読み込む。16bitsなので、2倍読み込む */
    len = read(iptel.audio_fd , buf , 2 * BLKSIZE );
    if (len != 2*BLKSIZE)
      fprintf(stderr , "can't fill rec_buf. \n");
    break;
  case AFMT_U8:
    /* BLKSIZE個のサンプルを読み込む。 8bitsなので、そのまま */
    len = read(iptel.audio_fd , ubuf , BLKSIZE );
    if (len != BLKSIZE)
      fprintf(stderr , "can't fill rec_buf. \n");
    /* ただし、u_char -> signed short に変換する */
    for (i=0 ; i < len ; i++){
      buf[i]=(ubuf[i] - 128 ) << 8 ;
    }
    break;
  default:
    fprintf(stderr,"no support format.\n");
    exit(-1);
  }

  /* 無音圧縮 from  tel.c
   * compute energy, useful for silence suppression...
   * Use a highpass filter to remove the DC component.
   */
  for (i = 0 , energy = 0.0 , avg = 0.0 ; i < BLKSIZE ; i++) {
    int a ;
    static int dc = 0 ;
#define BITS 8
    dc +=  buf[i] - (dc >> BITS) ;
    a = ( buf[i] - (dc>>BITS) ) ;
    if (a < -32767) buf[i] = -32767 ;
    else if (a > 32767) buf[i] = 32767 ;
    else buf[i] = a ;
    avg += a ;
    energy += a*a ;
  }
  energy /= BLKSIZE ;
```



```
  avg /= BLKSIZE ;

  /* RTPヘッダの作成 */
  rtp->ssrc = iptel.my_ssrc;
  rtp->timestamp = htonl(iptel.nowtime);
  rtp->seq  = htons(1 + ntohs(rtp->seq));

  if (iptel.suppress == 0 || energy > 5e4 ){
    /* GSMエンコーディング */
    gsm handle;
    handle = gsm_create();
    if (handle == 0) fprintf(stderr,"gsm_create error\n");
    gsm_encode(handle , buf , rtp->data );

    /* パケットを指定した相手に送る */
    if ( iptel.your_data_port != 0 ){
      trans_sockaddr(&you , iptel.your_ipaddr , iptel.your_data_port);
      len=sendto(iptel.data_sock , rtp , sizeof(*rtp) , 0 ,
               (struct sockaddr *)&you,sizeof(you));
      rtp->m = 0;
    }

    gsm_destroy(handle);
  }
}

/*-------------------------------------------------------
 * void net_read()
 *          ネットワークからデータを読み込む
 *          GSMデータをキューに貯める
 *-------------------------------------------------------*/
void net_read()
{
  static unsigned short start_seq;
  struct sockaddr_in from;
  struct rtphdr *rtp = NULL ;
  int buf_num=0;
  int fromlen;
  int len;
  int i;

  rtp = (struct rtphdr *) malloc (sizeof(*rtp)) ;

  /* パケットをソケットから受け取る */
  fromlen=sizeof(from);
  bzero(&from,sizeof(from));
  len=recvfrom(iptel.data_sock , rtp , sizeof(*rtp) , 0 ,
               (struct sockaddr *)&from,&fromlen);

  /* 通話相手以外から来たデータは捨てる */
  if ( iptel.your_ipaddr != from.sin_addr.s_addr) return ;
  /* 待ち受け時は相手のUDPポートが何番かわらないのでそれを上書き */
  iptel.your_data_port = ntohs(from.sin_port);

  /* 最初のパケットの場合は差分時間を算出する */
  if ( rtp->m==1 || iptel.delta==0 ){
    iptel.delta=iptel.nowtime-ntohl(rtp->timestamp);
    start_seq=ntohs(rtp->seq);
  }
  /* 到着時に再生可能時間が過ぎている場合 */
  if (ntohl(rtp->timestamp) + iptel.delta <  iptel.nowtime - CACHE*BLKSIZE  )
{
    free(rtp);
    return;
  }

```

```
   /* 正しい大きさ(33bytes)読んだかどうかをチェックする */
   /* (gsm_decodeの時点で33bytesないという可能性をなくす) */
   if (len!=sizeof(*rtp))
     fprintf(stderr,"can't read all of sock data len=%d\n",len);

   /* サウンドキューに蓄える (ハッシュ方式) */
   buf_num = ( ntohs(rtp->seq) - start_seq ) % MAXHASH ;
   queue[ buf_num ].timestamp=ntohl(rtp->timestamp);
   bcopy(rtp->data , queue[ buf_num ].data , sizeof(gsm_frame));

   free(rtp);
   return;
 }

 void audio_write()
 {
   short  buf[BLKSIZE];
   u_char ubuf[BLKSIZE];
   gsm handle;
   int len;
   int i;

   /* 最初のバッファリング */
   if ( start_time + CACHE*BLKSIZE > iptel.nowtime ) {
     fprintf(stderr,"buffering... %d\n",queue_num);
     return;
   }
   /* キューにバッファが存在していれば再生する */
   if (queue[queue_num].timestamp!=0 ){
   /* GSMデコーディング */
     handle = gsm_create();
     if (handle == 0 ) fprintf(stderr,"gsm_create error\n");
     gsm_decode(handle , queue[queue_num].data , buf );
     /* オーディオ再生 */
     switch(iptel.play_fmt){
     case AFMT_S16_LE:
       /* play_formatがsigned 16bits ならそのまま出力 */
       len = write(iptel.audio_fd , buf , 2*BLKSIZE);
       if (len != 2*BLKSIZE )
         fprintf(stderr , "can't play all of buffer. S16_LE\n");
       break;
     case AFMT_U8:
       /* play_formatがunsigned 8bits なら変換して出力 */
       for (i=0 ;  i < BLKSIZE ; i++){
         ubuf[i] = (u_char)(buf[i] >> 8 ) ^ 0x80 ;
       }
       len = write(iptel.audio_fd , ubuf , BLKSIZE);
       if (len != BLKSIZE )
         fprintf(stderr , "%d %d can't play all of buffer.  %d U8\n",iptel.aud
io_fd,len,errno);
       break;
     default:
       fprintf(stderr," no support format.\n");
       exit(-1);
     }

     gsm_destroy(handle);
     /* キューをクリアする */
     queue[queue_num].timestamp=0;
   }
   else {
   }

   queue_num++;
   if (queue_num==MAXHASH) queue_num=0;
   return;
```



```
 }

 /*-------------------------------------------------------------
  * void trans_sockaddr()
  *          与えられたIPアドレスとポートから、sockaddr_in構造体を返す
  *          ipaddrはNetwork Byte Order.PortはHost byte orderとする
  *-------------------------------------------------------------*/
 void trans_sockaddr(struct sockaddr_in *net , u_long ipaddr , u_short port)
 {
   bzero(net,sizeof(*net));
   net->sin_family = AF_INET;
   net->sin_port = htons(port);
   net->sin_addr.s_addr = ipaddr;
 }

 /*-------------------------------------------------------------
  * void make_listen_sock()
  *          iptel.my_listen_portに待ち受けソケットを作る。
  *-------------------------------------------------------------*/
 void make_listen_sock(){
   char hostname[41];
   struct sockaddr_in net;
   struct hostent *host;

   /* make socket */
   iptel.listen_sock=socket(AF_INET,SOCK_STREAM,0);
   if (iptel.listen_sock < 0 ){
     fprintf(stderr,"can't make sockets\n");
     exit(-1);
   }
   /* bind listen_socket */
   /* INADDR_ANYで待ち受けソケットをbindする */
   trans_sockaddr(&net , INADDR_ANY , iptel.my_listen_port);
   printf("Listening %s port %d\n",inet_ntoa(net.sin_addr),iptel.my_listen_por
t);
   if ( bind(iptel.listen_sock,(struct sockaddr *)&net , sizeof(net)) < 0 ) {
     fprintf(stderr,"can't bind listen_sock\n");
     exit(-1);
   }
   /* 最大1つの接続を待ち受け */
   listen(iptel.listen_sock,1);
 }

 /*-------------------------------------------------------------
  * void make_data_sock()
  *          iptel.my_data_portに音声データの送受信ソケットを作る。
  *-------------------------------------------------------------*/
 void make_data_sock()
 {
   struct sockaddr_in net;

   /* make socket */
   iptel.data_sock=socket(AF_INET,SOCK_DGRAM, 0);
   if ( iptel.data_sock < 0 ){
     fprintf(stderr,"can't make socket\n");
     exit(-1);
   }
   /* connect data_socket */
   bzero(&net,sizeof(net));
   trans_sockaddr(&net , INADDR_ANY , iptel.my_data_port);
   if (bind(iptel.data_sock , (struct sockaddr *)&net , sizeof(net)) < 0 ) {
     fprintf(stderr,"can't bind data_sock\n");
     exit(-1);
   }
   printf("binded data_sock \n");
 }
```

```
492
493 /*-------------------------------------------------------------
494  * int connect_sock(int insock)
495  *          通話相手への接続要求を出す。
496  *-------------------------------------------------------------*/
497 int connect_sock(u_long ipaddr , u_short port)
498 {
499   int sock;
500   struct sockaddr_in net;
501
502   /* make socket */
503   sock=socket(AF_INET,SOCK_STREAM,0);
504   if ( sock < 0 ){
505     fprintf(stderr,"can't make socket\n");
506     exit(-1);
507   }
508   /* connect ctl_socket */
509   bzero(&net,sizeof(net));
510   trans_sockaddr(&net , ipaddr , port);
511   if (connect(sock ,  (struct sockaddr *)&net , sizeof(net)) < 0 ) {
512     fprintf(stderr,"can't connect ctl_sock\n");
513     return -1;
514   }
515   printf("  connected sock to %s port%d\n",inet_ntoa(ipaddr),port);
516   return sock;
517 }
518
519 void close_sock(int *sock)
520 {
521   close(*sock);
522   *sock=-1;
523 }
524
525 void recv_msg(char *buf)
526 {
527   int n;
528   char content[32];
529   struct msg_header *p = NULL;
530   struct req_msg    *r = NULL;
531   struct refuse_msg *q = NULL;
532
533   p = (struct msg_header *)buf;
534   switch(ntohs(p->msg_id)){
535     /* 通話要求メッセージの場合 */
536   case REQ_MSG:
537     n=readn(iptel.tmp_sock , content, ntohs(p->msg_len));
538     r = (struct req_msg *)content;
539     printf("recv CALLING to %s from %s\n",r->your_number,p->my_number);
540     break;
541     /* 通話許可メッセージの場合 */
542   case ACCEPT_MSG:
543     printf("recv ACCEPT_MSG from %s. Start!\n",p->my_number);
544     iptel.status=TALKING; /* 通話開始 */
545     if(iptel.audio_fd<0) iptel.audio_fd=audio_open();
546     start_time=iptel.nowtime;
547     queue_num=0;
548     iptel.delta=0;
549     break;
550     /* 通話拒否メッセージの場合 */
551   case REFUSE_MSG:
552     n=readn(iptel.ctl_sock , content, ntohs(p->msg_len));
553     q = (struct refuse_msg *)content;
554     printf("recv REFUSE_MSG from %s\n",p->my_number);
555     switch(ntohs(q->reply_type)){
556     case TALKING:
557       printf("  He is talking now. So He refused connection from you.\n");
```



```
558       break;
559     case CALLING:
560       printf("  He is calling now. So He refused connection from you.\n");
561       break;
562     case CALLED:
563       printf("  He is called  now. So He refused connection from you.\n");
564       break;
565     case IDLE:
566       printf("  He don't want to talk with you. \n");
567       break;
568     case GW_YOUTALKING :
569       printf("  He is talking to someone. So GW refused connection. \n");
570       break;
571     case GW_CALLING :
572       printf("  GW is just called from someone. So GW refused connection. \n"
);
573       break;
574     case GW_NOTHING :
575       printf("  He didn't hang up . So GW disconnected. \n");
576       break;
577     case GW_ERROR:
578       printf("  Some error happend at GW.So GW disconnected.\n");
579       break;
580     }
581     close_sock(&iptel.ctl_sock);
582     iptel.status=IDLE;
583     break;
584     /* 通話終了メッセージの場合 */
585   case FIN_MSG:
586     printf("recv FIN_MSG from %s.\n",p->my_number);
587     iptel.status=IDLE;
588     close_sock(&iptel.tmp_sock);
589     close_sock(&iptel.ctl_sock);
590     close_sock(&iptel.audio_fd);
591     break;
592   }
593 }
594
595 /* ネゴシエーションのためにメッセージを送る */
596
597 void send_msg(int msg_id)
598 {
599   int n;
600   struct msg_header *p = NULL;
601   struct req_msg    *r = NULL;
602   struct refuse_msg *q = NULL;
603
604   switch(msg_id){
605   case REQ_MSG:
606     printf("send REQ_MSG\n");
607     p = (struct msg_header *)malloc(sizeof(*p));
608     p->msg_id=htons(REQ_MSG);
609     p->msg_len=htons(sizeof(*r));
610     strcpy(p->my_number,iptel.my_number);
611     r = (struct req_msg *)malloc(sizeof(*r));
612     strcpy(r->your_number,iptel.your_number);
613     n=writen(iptel.ctl_sock, (char *)p , sizeof(*p));
614     n=writen(iptel.ctl_sock, (char *)r , sizeof(*r));
615     free(p);
616     free(r);
617     break;
618   case ACCEPT_MSG:
619     printf("send ACCEPT_MSG\n");
620     p = (struct msg_header *)malloc(sizeof(*p));
621     p->msg_id=htons(ACCEPT_MSG);
622     p->msg_len=htons(0);
```

```
623     strcpy(p->my_number,iptel.my_number);
624     n=writen(iptel.ctl_sock, (char *)p , sizeof(*p));
625     free(p);
626     break;
627   case REFUSE_MSG:
628     printf("send REFUSE_MSG\n");
629     p = (struct msg_header *)malloc(sizeof(*p)+sizeof(*q));
630     p->msg_id=htons(REFUSE_MSG);
631     p->msg_len=htons(sizeof(*q));
632     strcpy(p->my_number,iptel.my_number);
633     q = (struct refuse_msg *)malloc(sizeof(*q));
634     /* 通話拒否メッセージ */
635     switch(iptel.status){
636     case TALKING:
637       q->reply_type=htons(TALKING);
638       printf("I'm talking now. So I refused connection. \n");
639       break;
640     case CALLING:
641       q->reply_type=htons(CALLING);
642       printf("I'm calling now. So I refused connection. \n");
643       break;
644     case CALLED:
645       q->reply_type=htons(CALLED);
646       printf("I'm called . So I refused connection.\n");
647       break;
648     case IDLE:
649       q->reply_type=htons(IDLE);
650       printf("I don't want to accept. So I refused connection.\n");
651       break;
652     }
653     n=writen(iptel.tmp_sock, (char *)p , sizeof(*p));
654     n=writen(iptel.tmp_sock, (char *)q , sizeof(*q));
655     free(p);
656     free(q);
657     break;
658
659   case FIN_MSG:
660     /* 通話終了メッセージ */
661     printf("send FIN_MSG\n");
662     p = (struct msg_header *)malloc(sizeof(*p));
663     p->msg_id=htons(FIN_MSG);
664     p->msg_len=htons(0);
665     strcpy(p->my_number,iptel.my_number);
666     n=writen(iptel.ctl_sock, (char *)p , sizeof(*p));
667     free(p);
668     break;
669   }
670 }
671
672 void print_menu()
673 {
674     fprintf(stderr,"%s",menu);
675     fprintf(stderr,">");
676 }
677
678 void cmd_line(char *buf)
679 {
680   char ipaddr[256];
681
682   switch(buf[0]){
683   case 'd':   /* dialing */
684     /* 通話可能状態ならダイアルする */
685     if ( iptel.status == IDLE ){
686       /* 電話番号をコピー */
687       strcpy(iptel.your_number,buf+1);
688       printf("  dialing to %s\n",iptel.your_number);
```



```
689
690       /* GKにアドレスを聞く */
691       if (ask_gk(GKREQ_MSG)==1){
692
693         /* GKが何らかのアドレスを返してくれた場合 */
694         /* IPアドレスが存在すれば通話要求をする */
695         if (iptel.your_ipaddr!=0){
696           printf("  dialing to %s... (port %d & %d)\n",inet_ntoa(iptel.your_i
paddr),
697                  iptel.your_listen_port,iptel.your_data_port);
698           /* 相手に接続する */
699           iptel.status=CALLING;
700           iptel.ctl_sock =
701             connect_sock(iptel.your_ipaddr , iptel.your_listen_port);
702           if (iptel.ctl_sock!=-1){
703             /* 回線的につながる */
704             printf("  connected!\n");
705             send_msg(REQ_MSG);
706             printf("  waiting for reply...\n");
707           }
708           else {
709             print_menu();
710             iptel.status=IDLE;
711           }
712         }
713
714         /* IPアドレスが存在しない場合 */
715         /* (相手がネットワーク上にいないorGWが存在しない場合) */
716         else {
717           printf("IPTEL.NO = %s is not on the network\n",iptel.your_number);
718         }
719       }
720       else {
721         fprintf(stderr,"cannot connect to gk.\n");
722       }
723     }
724     break;
725   case 'u':
726     /* DNSエントリーの更新 */
727     ask_gk(GKUPDATE_MSG);
728     break;
729   case 'a':   /* accept */
730     if (iptel.status==CALLED){
731       iptel.ctl_sock=iptel.tmp_sock;
732       iptel.tmp_sock=-1;
733       send_msg(ACCEPT_MSG); /* 了解メッセージを送信 */
734       iptel.status=TALKING; /* 通話開始 */
735       if(iptel.audio_fd<0) iptel.audio_fd=audio_open();
736       start_time=iptel.nowtime;
737       queue_num=0;
738       iptel.delta=0;
739     }
740     else {
741       print_menu();
742     }
743     break;
744   case 's':   /* stop   */
745     if (iptel.status==TALKING || iptel.status==CALLING ){
746       send_msg(FIN_MSG);
747       close_sock(&iptel.ctl_sock);
748       close_sock(&iptel.audio_fd);
749       iptel.status=IDLE;
750     }
751     else if (iptel.status==CALLED){
752       iptel.status=IDLE;
753       send_msg(REFUSE_MSG);
```

```
754       close_sock(&iptel.tmp_sock);
755       close_sock(&iptel.ctl_sock);
756       close_sock(&iptel.audio_fd);
757     }
758     else {
759       print_menu();
760     }
761     break;
762   case 'q':   /* quit   */
763     if (iptel.status==TALKING){
764       send_msg(FIN_MSG);
765       close_sock(&iptel.audio_fd);
766       iptel.status=IDLE;
767     }
768     /* DNSエントリー削除 */
769     iptel.my_listen_port=0;
770     iptel.my_data_port=0;
771     ask_gk(GKUPDATE_MSG);
772     close_sock(&iptel.ctl_sock);
773     close_sock(&iptel.listen_sock);
774     iptel.status=IDLE;
775     iptel.tty_fd=-1;
776     fprintf(stdout,"byebye..\n");
777     break;
778   case '?':
779     fprintf(stderr,"status = %d\n",iptel.status);
780     fprintf(stderr,"listen_sock %d\n",iptel.listen_sock);
781     fprintf(stderr,"tmp_sock %d\n",iptel.tmp_sock);
782     fprintf(stderr,"ctl_sock %d\n",iptel.ctl_sock);
783     fprintf(stderr,"data_sock %d\n",iptel.data_sock);
784     fprintf(stderr,"iptel.delta %d\n",iptel.delta);
785     break;
786   default:
787     print_menu();
788     break;
789   }
790
791 }
792
793 int ask_gk(int msg_id)
794 {
795   int gk_sock,n;
796   char buf[256];
797   struct msg_header *p = NULL;
798   struct gkreq_msg  *q = NULL;
799   struct gkrep_msg  *r = NULL;
800   struct gkupdate_msg *s = NULL;
801   struct gkupdaterep_msg *t=NULL;
802
803   /* gatekeeperへTCPでコネクションを張る */
804   if ( (gk_sock=connect_sock(iptel.gk_ipaddr,iptel.gk_port)) == -1 ){
805     fprintf(stderr,"cannot connect gatekeeper %s\n",inet_ntoa(iptel.gk_ipaddr
));
806     return -1;
807   }
808   switch(msg_id){
809   case GKREQ_MSG:
810     /* アドレスをGKに聞く */
811     p = (struct msg_header *)malloc(sizeof(*p));
812     p->msg_id=htons(GKREQ_MSG);
813     p->msg_len=htons(sizeof(*q));
814     strcpy(p->my_number,iptel.my_number);
815     q = (struct gkreq_msg *)malloc(sizeof(*q));
816     strcpy(q->your_number,iptel.your_number);
817     if ( (n=writen(gk_sock, (char *)p , sizeof(*p))) < 0 ){
818       return -1;
```



```
819     }
820     if ( (n=writen(gk_sock, (char *)q , sizeof(*q))) < 0 ){
821       fprintf(stderr,"cannot send to gatekeeper %s\n",inet_ntoa(iptel.gk_ipad
dr));
822       return -1;
823     }
824     printf("waiting for gk_rep\n");
825     free(p);
826     free(q);
827     break;
828   case GKUPDATE_MSG:
829     /* DNS更新メッセージを出す(ポート番号を知らせる) */
830     /* メッセージヘッダ */
831     p = (struct msg_header *)malloc(sizeof(*p));
832     p->msg_id=htons(GKUPDATE_MSG);
833     p->msg_len=htons(sizeof(*s));
834     strcpy(p->my_number,iptel.my_number);
835     /* メッセージ本体 */
836     s = (struct gkupdate_msg *)malloc(sizeof(*s));
837     s->listen_port=htons(iptel.my_listen_port);
838     s->data_port=htons(iptel.my_data_port);
839
840     if ( (n=writen(gk_sock, (char *)p , sizeof(*p))) < 0 ){
841       return -1;
842     }
843     if ( (n=writen(gk_sock, (char *)s , sizeof(*s))) < 0 ){
844       fprintf(stderr,"cannot send to gatekeeper %s\n",inet_ntoa(iptel.gk_ipad
dr));
845       return -1;
846     }
847     printf("waiting for gkupdate_rep\n");
848     free(p);
849     free(s);
850     break;
851   }
852
853   /* GKから返答をもらう */
854   if ( (n=readn(gk_sock,buf,sizeof(struct msg_header))) < 0 ) {
855     fprintf(stderr,"cannot recv from gatekeeper %s\n",inet_ntoa(iptel.gk_ipad
dr));
856     return -1;
857   }
858   p = (struct msg_header *)buf;
859   switch(ntohs(p->msg_id)){
860   case GKREP_MSG:
861     /* アドレスリプライの場合 */
862     printf("recv gk_rep\n");
863     /* メッセージ本体を読む */
864     if ( ( n=readn(gk_sock,buf,sizeof(struct gkrep_msg)) ) < 0 ){
865       fprintf(stderr,"cannot recv from gatekeeper %s\n",inet_ntoa(iptel.gk_ip
addr));
866       return -1;
867     }
868     r = (struct gkrep_msg *)buf;
869     /* アドレスが存在した場合 */
870     if (ntohs(r->rep_type) == 0x0001 ){
871       iptel.your_ipaddr=r->ipaddr;
872       iptel.your_listen_port=ntohs(r->listen_port);
873       iptel.your_data_port=ntohs(r->data_port);
874       strcpy(iptel.your_number,r->your_number);
875     }
876     /* アドレスが存在しなかった場合 */
877     else {
878       iptel.your_ipaddr=htonl(0);
879       iptel.your_listen_port=ntohs(0);
880       iptel.your_data_port=ntohs(0);
```

```
   881      }
   882      break;
   883    case GKUPDATEREP_MSG:
   884      /* DNSの更新メッセージの場合 */
   885      printf("recv gkupdate_rep\n");
   886      /* メッセージ本体を読む */
   887      if ( ( n=readn(gk_sock,buf,sizeof(struct gkupdaterep_msg)) ) < 0 ){
   888        fprintf(stderr,"cannot recv from gatekeeper %s\n",inet_ntoa(iptel.gk_ip
addr));
   889        return -1;
   890      }
   891      t = (struct gkupdaterep_msg *)buf;
   892      if (ntohs(t->rep_type) == 0x0001 ){
   893        printf("update succeeded.\n");
   894      }
   895      else {
   896        printf("update unsucceeded.\n");
   897      }
   898      break;
   899    }
   900
   901    /* GKとのコネクションを切る */
   902    close_sock(&gk_sock);
   903    return 1;
   904  }
   905
   906  /*---------------------------------------
   907   * void main
   908   *
   909   *
   910   *---------------------------------------*/
   911  void main(int argc, char *argv[])
   912  {
   913    int i,n,m,j,size,fromlen;
   914    char filename[255];
   915    char ipaddr[255];
   916    char buf[1024];
   917    fd_set readfds,fds;
   918    struct sockaddr_in net;
   919    struct timeval tv;
   920    queue_num=0;
   921    iptel.suppress=1;
   922    iptel.my_listen_port=MYLISTENPORT;
   923    iptel.my_data_port=MYDATAPORT;
   924    iptel.status=IDLE;
   925    /* 自分の電話番号 */
   926    strcpy(iptel.my_number,MYNUMBER);
   927    iptel.my_ssrc=random32(0);
   928    iptel.delta=0;
   929    iptel.tty_fd=0;
   930    iptel.listen_sock=-1;
   931    iptel.ctl_sock   =-1;
   932    iptel.data_sock  =-1;
   933    iptel.tmp_sock   =-1;
   934    iptel.audio_fd   =-1;
   935
   936    strcpy(ipaddr,GKADDRESS);
   937    inet_aton(ipaddr,&iptel.gk_ipaddr);
   938    iptel.gk_port=GKLISTENPORT;
   939    printf("gatekeeper=%s port%d\n",inet_ntoa(iptel.gk_ipaddr),iptel.gk_port);
   940
   941    /* 待ち受けソケットを作る */
   942    make_listen_sock();
   943    make_data_sock();
   944    print_menu();
   945
```



```
   946    while(!iptel.tty_fd){
   947      fprintf(stderr,"\r>");
   948      n=0;
   949      FD_ZERO(&readfds);
   950      if (iptel.tty_fd>=0){
   951        FD_SET(iptel.tty_fd,&readfds);        /* 標準入力を監視対象 */
   952        if (iptel.tty_fd > n ) n = iptel.tty_fd;
   953      }
   954      if (iptel.listen_sock>=0){
   955        FD_SET(iptel.listen_sock,&readfds);   /* 待ち受けソケットを監視対象 */
   956        if (iptel.listen_sock > n ) n = iptel.listen_sock;
   957      }
   958      if (iptel.tmp_sock>=0){
   959        FD_SET(iptel.tmp_sock,&readfds);   /* TMPソケットを監視対象 */
   960        if (iptel.tmp_sock > n ) n = iptel.tmp_sock;
   961      }
   962      if (iptel.ctl_sock>=0){
   963        FD_SET(iptel.ctl_sock,&readfds);      /* 制御ソケットを監視対象 */
   964        if (iptel.ctl_sock > n ) n = iptel.ctl_sock;
   965      }
   966      if (iptel.data_sock>=0){
   967        FD_SET(iptel.data_sock,&readfds);      /* 音声データソケットを監視対象
*/
   968        if (iptel.data_sock > n ) n = iptel.data_sock;
   969      }
   970      if (iptel.audio_fd>=0){
   971        FD_SET(iptel.audio_fd,&readfds);      /* オーディオデバイスを監視対象 *
/
   972        if (iptel.audio_fd > n ) n = iptel.audio_fd;
   973      }
   974      n++;
   975
   976      m=select(n,&readfds,NULL,NULL,NULL);
   977      gettimeofday(&tv, NULL);
   978      iptel.nowtime = tv.tv_sec * RATE + (tv.tv_usec*RATE/1000000);
   979
   980      /* 標準入力の場合 */
   981      if (iptel.tty_fd>=0 && FD_ISSET(iptel.tty_fd,&readfds) ){
   982        scanf("%s",buf);
   983        cmd_line(buf);
   984      }
   985
   986      /* 待ち受けソケットの場合 */
   987      if (iptel.listen_sock >= 0 && FD_ISSET(iptel.listen_sock,&readfds) ){
   988        /* 待ち受けソケットに来た通話をとりあえずacceptする */
   989        fromlen=sizeof(net);
   990        iptel.tmp_sock=
   991          accept(iptel.listen_sock,(struct sockaddr *)&net,&fromlen);
   992        if (iptel.tmp_sock < 0) fprintf(stderr,"error:can't accept \n");
   993
   994        i=readn(iptel.tmp_sock,buf,sizeof(struct msg_header));
   995        recv_msg(buf);
   996        /* 誰からの通話か表示する */
   997        /* 通話中、呼出中、呼び出され中なら即効拒否メッセージを送る */
   998        if (iptel.status!=IDLE ) {
   999          send_msg(REFUSE_MSG);
  1000          close_sock(&iptel.tmp_sock);
  1001        }
  1002        else {
  1003          iptel.status=CALLED;
  1004          iptel.your_ipaddr=net.sin_addr.s_addr;
  1005        }
  1006      }
  1007      /* テンポラリソケットの場合 */
  1008      if (iptel.tmp_sock >= 0 && FD_ISSET(iptel.tmp_sock,&readfds) ) {
  1009        i=readn(iptel.tmp_sock,buf,sizeof(struct msg_header));
```

```
1010        recv_msg(buf);
1011      }
1012      /* 制御ソケットの場合 */
1013      if (iptel.ctl_sock >= 0 && FD_ISSET(iptel.ctl_sock,&readfds) ) {
1014        i=readn(iptel.ctl_sock,buf,sizeof(struct msg_header));
1015        recv_msg(buf);
1016      }
1017      /* 音声データソケットの場合 */
1018      if (iptel.data_sock >= 0 && FD_ISSET(iptel.data_sock,&readfds) ) {
1019        /* networkからデータを読み、キューにためる */
1020        net_read();
1021      }
1022      /* オーディオデバイスの場合 */
1023      if (iptel.audio_fd >= 0 && FD_ISSET(iptel.audio_fd,&readfds) ) {
1024        if (iptel.duplex==FULLDUPLEX){
1025          audio_write();
1026          audio_read();
1027        }
1028        else if (iptel.duplex==HALFDUPLEX){
1029          audio_write();
1030        }
1031      }
1032
1033    }
1034    close_sock(&iptel.tmp_sock);
1035    close_sock(&iptel.data_sock);
1036    close_sock(&iptel.audio_fd);
1037  }
1038
```

ゲートウェイソースリスト

ipgw.h

```
   1  #define COUNTRYCODE "81" /* GWの所在する国番号 */
   2  #define DPREFIX "0:"       /* 外線発信のためのprefix */
   3  #define BLKSIZE 160
   4  #define RATE 8000
   5  #define MAXHASH 20
   6  #define CACHE 5
   7  #define GKADDRESS "203.178.141.176"
   8                          /* 使用するオーディオデバイスによって変更 */
   9  #define AUDIO_DEVICE "/dev/audio1"
  10                           /* 使用するserial portよって変更 */
  11  #define PNC_DEVICE "/dev/ttyd0"
  12  #define BAUDRATE B19200
  13  #define FULLDUPLEX 1
  14  #define HALFDUPLEX 0
  15  #define ON 1
  16  #define OFF 0
  17                          /* 使用するポート番号によって変更 */
  18  #define MYLISTENPORT 8000
  19  #define MYDATAPORT 8001
  20  #define GKLISTENPORT 16000
  21  #define DATA 0
  22  #define CTL 1
  23  #define IDLE    1       /* 待機         */
  24  #define CALLED  2       /* 呼ばれている */
  25  #define CALLING 3       /* 呼んでいる   */
  26  #define TALKING 4       /* 通話中       */
  27  #define GW_YOUTALKING 5  /* 通話相手が話し中  */
  28  #define GW_CALLING    6  /* 自局に着信あり    */
  29  #define GW_NOTHING    7  /* 相手が電話に出ない*/
  30  #define GW_SUCCEED    8  /* 通話に成功        */
  31  #define GW_ERROR      9  /* エラー発生        */
  32
```



```
  33  #define PT_GSM 3         /* PAYLOAD TYPE GSM */
  34
  35  #define MD_CTX MD5_CTX
  36  #define MDInit MD5Init
  37  #define MDUpdate MD5Update
  38  #define MDFinal MD5Final
  39
  40  #define REQ_MSG      0x0001
  41  #define ACCEPT_MSG   0x0002
  42  #define REFUSE_MSG   0x0003
  43  #define FIN_MSG      0x0004
  44  #define GKREQ_MSG   0x0005
  45  #define GKREP_MSG   0x0006
  46
  47  char menu[] =
  48  "    s        : stop call or refuse call\n"
  49  "    q        : quit\n"
  50  "    ?        : show status\n"
  51  ;
  52
  53  struct ipgw_status {
  54    u_long  your_ipaddr;     /* 通話相手のIPアドレス network order */
  55    u_short your_listen_port;/* 通話相手の待ち受けポート host order*/
  56    u_short your_data_port;  /* 通話相手の音声パケットポート host order*/
  57    u_short my_listen_port;  /* 自分の待ち受けポート host order*/
  58    u_short my_data_port;    /* 自分の音声パケットポート host order*/
  59    char my_number[32];      /* 自分のIP電話番号 */
  60    char your_number[32];    /* 通話相手の電話番号 */
  61    u_long gk_ipaddr;        /* gatekeeperのIPアドレス */
  62    u_short gk_port;         /* gatekeeperのポート */
  63    u_long play_fmt;         /* 再生フォーマット */
  64    u_long rec_fmt;          /* 録音フォーマット */
  65
  66    int listen_sock;         /* 待ち受け用ソケット(TCP) */
  67    int ctl_sock;            /* 制御パケット用ソケット(TCP) */
  68    int data_sock;           /* 音声パケット用ソケット(UDP) */
  69    int tmp_sock;            /* 待ち受け用ソケット for catch phone(TCP) */
  70
  71    int tty_fd;              /* コマンド入力FD */
  72    int audio_fd;            /* オーディオデバイス FD  */
  73    int pnc_fd;              /* PNC3400デバイス FD */
  74    unsigned long nowtime;   /* RATE1カウントにつき1カウントが増えるカウンター*
/
  75    unsigned long my_ssrc;   /* RTP Syncronization src */
  76    unsigned long delta;
  77
  78    int suppress;            /* 無音圧縮フラグ */
  79    int duplex;              /* 全二重フラグ   */
  80    int status;              /* 通話状態フラグ */
  81    char ccode[3];           /* 国番号 */
  82    char dialprefix[8];      /* 外線発信番号 */
  83  };
  84
  85  struct msg_header{
  86    u_short msg_id;         /* message ID      */
  87    u_short msg_len;        /* message_lendgth */
  88    char my_number[32];     /* IP telephone number */
  89  };
  90
  91  struct req_msg{
  92    char your_number[32];   /* IP telephone number */
  93  };
  94
  95  struct refuse_msg{
  96    u_short reply_type;    /* refuse type */
  97  };
```

```
 98  struct gkreq_msg{
 99    char your_number[32];  /* IP telephone number */
100  };
101
102  struct gkrep_msg{
103    u_short rep_type;      /* reply type 0x0001=FOUND 0x0002=NOTFOUND*/
104    u_long  ipaddr;        /* client(or gk) ipaddr */
105    u_short listen_port;   /* client(or gk) listen_port */
106    u_short data_port;     /* client(or gk) data_port */
107    char your_number[32];  /* dialing telephone number */
108  };
109
110  struct rtphdr {          /* RTP ヘッダ */
111  #if _BIG_ENDIAN
112    unsigned int v:2;
113    unsigned int p:1;
114    unsigned int x:1;
115    unsigned int cc:4;
116    unsigned int m:1;
117    unsigned int pt:7;
118    unsigned int seq:16;
119  #else
120    unsigned int seq:16;
121    unsigned int m:1;
122    unsigned int pt:7;
123    unsigned int v:2;
124    unsigned int p:1;
125    unsigned int x:1;
126    unsigned int cc:4;
127  #endif
128    unsigned long timestamp;
129    unsigned long ssrc;
130    gsm_frame data;
131    char padding;
132  };
133
134  struct snd_queue {
135    unsigned long timestamp;
136    gsm_frame data;
137  };
138
139
140
141
142
143
144
```

ipgw.c

```
  1  /* IP telephone Gateway  1998.*/
  2
  3  #include <stdio.h>
  4  #include <unistd.h>
  5  #include <fcntl.h>
  6  #include <sys/ioctl.h>
  7  #include <sys/ioctl_compat.h>
  8  #include <machine/soundcard.h>
  9  #include <netinet/in.h>
 10  #include <netdb.h>
 11  #include <sys/types.h>
 12  #include <sys/socket.h>
 13  #include <sys/time.h>    /* gettimeofday() */
 14  #include <sys/utsname.h> /* uname() */
 15  #include <sys/errno.h>
 16  #include <md5.h>
```



```
17  #include <time.h>        /* clock() */
18  #include <termios.h>     /* terminal device control */
19  #include <sys/ttydev.h>
20  #include <sys/stat.h>
21  #include <time.h>
22  #include "gsm.h"
23  #include "ipgw.h"
24
25  struct ipgw_status ipgw;  /* IP電話のパラメータ */
26  struct snd_queue queue[MAXHASH];
27  unsigned long start_time;
28  int queue_num;
29  int errno;
30
31  int readn(int fd, char *ptr, int nbytes);
32  int writen(int fd, char *ptr, int nbytes);
33  static u_long md_32(char *string, int length);
34  unsigned int random32(int type);
35  int pnc_open();
36  void read_pnc(char *msg);
37  void pnc_init();
38  int pnc_dial(char *number);
39  int pnc_stat();
40  void detect_lineconnect();
41  void pnc_onhook();
42  int ask_gk(int msg_id);
43  int audio_open();
44  void audio_read();
45  void audio_write();
46  void net_read();
47  void trans_sockaddr(struct sockaddr_in *net , u_long addr , u_short port);
48  void make_listen_sock();
49  int connect_sock(u_long ipaddr , u_short port);
50  void close_sock(int *sock);
51  void print_menu();
52  void send_msg(int msg_id);
53  void recv_msg(char *buf);
54  void cmd_line(char *buf);
55
56  int readn(int fd, char *ptr, int nbytes)
57  {
58          int nleft,n;
59          nleft = nbytes;
60          while(nleft > 0){
61                  n = read(fd,ptr,nleft);
62                  if(n < 0){
63                    return(n);
64                  }
65                  else if(n == 0){
66                    break;
67                  }
68                  nleft -= n;
69                  ptr += n;
70          }
71          return(nbytes - nleft);
72  }
73
74  int writen(int fd, char *ptr, int nbytes)
75  {
76          int nleft,nwritten;
77          nleft = nbytes;
78          while(nleft > 0){
79                  nwritten = write(fd,ptr,nleft);
80                  if (nwritten < 0){
81                    return (nwritten);
82                  }
```

```
 83                 else if(nwritten == 0){
 84                   break;
 85                 }
 86                 nleft -= nwritten;
 87                 ptr += nwritten;
 88         }
 89         return(nbytes - nleft);
 90 }
 91
 92 static u_long md_32(char *string, int length)
 93 {
 94   MD_CTX context;
 95   union {
 96     char   c[16];
 97     u_long x[4];
 98   } digest;
 99   u_long r;
100   int i;
101
102   MDInit (&context);
103   MDUpdate (&context, string, length);
104   MDFinal ((unsigned char *)&digest, &context);
105   r = 0;
106   for (i = 0; i < 3; i++) {
107     r ^= digest.x[i];
108   }
109   return r;
110 }
111
112 unsigned int random32(int type)
113 {
114   struct {
115     int     type;
116     struct  timeval tv;
117     clock_t cpu;
118     pid_t   pid;
119     u_long  hid;
120     uid_t   uid;
121     gid_t   gid;
122     struct  utsname name;
123   } s;
124
125   gettimeofday(&s.tv, 0);
126   uname(&s.name);
127   s.type = type;
128   s.cpu  = clock();
129   s.pid  = getpid();
130   s.hid  = gethostid();
131   s.uid  = getuid();
132   s.gid  = getgid();
133
134   return md_32((char *)&s, sizeof(s));
135 }                               /* random32 */
136
137 /*-------------------------------------------------------------
138  * int pnc_open()
139  *          PNC3400のオープン
140  *          PNCデバイスの初期化
141  *          PNCデバイスのFDを返す
142  *-------------------------------------------------------------*/
143 int pnc_open()
144 {
145   int pnc_fd;
146   struct termios pnc_interface;
147
148   /* PNC3400のオープン */
```



```
149   pnc_fd = open(PNC_DEVICE, O_RDWR | O_NOCTTY | O_NDELAY);
150   if (pnc_fd < 0 ){ /* Could not open the port */
151     fprintf(stderr, "can not open PNC %s\n",PNC_DEVICE);
152     exit(-1);
153   }
154
155   fprintf(stdout,"open PNC3400\n");
156
157   /* PNC3400が接続されているインタフェースの情報を得る */
158
159   fcntl(pnc_fd, F_SETFL, FNDELAY);  /* Configure port reading */
160   /* Get the current options for the port */
161   tcgetattr(pnc_fd, &pnc_interface);
162   cfsetispeed(&pnc_interface, BAUDRATE);    /* Set the baud rates to 9600 */
163   cfsetospeed(&pnc_interface, BAUDRATE);
164
165   /* Enable the receiver and set local mode */
166   pnc_interface.c_cflag |= (CLOCAL | CREAD);
167   /* Mask the character size to 8 bits, no parity*/
168   pnc_interface.c_cflag &= ~PARENB;
169   pnc_interface.c_cflag &= ~CSTOPB;
170   pnc_interface.c_cflag &= ~CSIZE;
171   pnc_interface.c_cflag |=  CS8;    /* Select 8 data bits*/
172   pnc_interface.c_cflag |= CRTSCTS; /* enable hardware flow control */
173
174   /* Enable data to be processed as raw input */
175   pnc_interface.c_lflag &= ~(ICANON | ECHO | ISIG);
176
177   /* Set the new options for the port */
178   tcsetattr(pnc_fd, TCSANOW, &pnc_interface);
179
180   return pnc_fd;
181 }
182
183 /*-------------------------------------------------------------
184  * void read_pnc()
185  *          PNCからインディケーションを読み取り、結果を返す
186  *-------------------------------------------------------------*/
187 void read_pnc(char *msg)
188 {
189   char chout;
190   while(chout!='\r'){
191     if (read(ipgw.pnc_fd,&chout,sizeof(chout))>0){
192       *msg=chout;
193       msg++;
194     }
195   }
196   *msg=(char )NULL;
197 }
198
199 /*-------------------------------------------------------------
200  * void pnc_init()
201  *          PNCの初期化を行なう
202  *-------------------------------------------------------------*/
203 void pnc_init()
204 {
205   char command[5];
206
207   sleep(2);
208   printf("initialize pnc\n");
209   printf("  3 Connect incoming call --> ");
210   write(ipgw.pnc_fd,"CIC3\r",5);
211   read_pnc(command);
212   fprintf(stdout,"%s\n",command);
213
214   printf("  speaker off --> ");
```

```
  write(ipgw.pnc_fd,"SPK0\r",5);
  read_pnc(command);
  fprintf(stdout,"%s\n",command);
}


/*-------------------------------------------------------
 * int pnc_dial(char *number)
 *          PNCに対して与えられた番号に発信する
 *          インディケーションを読み、その結果を返す
 *-------------------------------------------------------*/
int pnc_dial(char *number)
{
  char command[35];

  /* GWのCOUNTRYCODEに従って、ダイアルアウト文字列を作り、ダイアルする */
  if (strncmp(number,ipgw.ccode,strlen(ipgw.ccode))==0){
    strcpy(command,"CRN");
    strcat(command,ipgw.dialprefix);
    strcat(command,"0");
    strcat(command,number+strlen(ipgw.ccode));
    strcat(command,"\r");
    write(ipgw.pnc_fd,command,strlen(command));
  }
  /* CRNコマンドのインディケーションを読む */
  read_pnc(command);
  if (strncmp(command,"CFINO",5)==0){         /* 接続に成功 */
    return GW_SUCCEED;
  }
  else if (strncmp(command,"CFIET",5)==0){    /* 通話相手が話し中 */
    return GW_YOUTALKING;
  }
  else if (strncmp(command,"CFICB",5)==0){    /* GWに着信あり */
    return GW_CALLING;
  }
                                             /* 通話相手が電話をとらなかった
*/
  else if (strncmp(command,"CFIAB",5)==0 || strncmp(command,"CFIRT",5)== 0 ){
    return GW_NOTHING;
  }
  else {                                     /* それ以外のエラー */
    return GW_ERROR;
  }
}

/*-------------------------------------------------------
 * int pnc_stat()
 *          PNCの状態(発信可能か)を調べ、その結果を返す
 *-------------------------------------------------------*/
int pnc_stat()
{
  char command[16];

  /* STATUS REPORT */
  write(ipgw.pnc_fd,"STS\r",4);
  read_pnc(command);
  printf("pnc_stat = %s\n",command);

  /* GWが非発信中および非通話中の場合 */
  if (command[3]=='0' && command[4]=='0'){
    return IDLE;
  }
  /* GWが発信中の場合 */
  else if (command[3]=='1'){
    return CALLING;
  }
```



```
  /* GWが通話中の場合 */
  else if (command[4]=='1'){
    return TALKING;
  }
}

/*-------------------------------------------------------
 * int detect_lineconnect()
 *          回線切断を読みとるためのコマンドを送信する
 *-------------------------------------------------------*/
void detect_lineconnect()
{
  char command[6];

  write(ipgw.pnc_fd,"DLD43\r",6);
  read_pnc(command);
  printf("line detect = %s\n",command);
}

/*-------------------------------------------------------
 * void pnc_cmd(char *buf)
 *          PNCから割り込みで返されるインディケーションを読み、処理を行なう
 *-------------------------------------------------------*/
void pnc_cmd(char *buf)
{
  if (strncmp(buf,"KIL",3)==0){
    send_msg(FIN_MSG);
    close_sock(&ipgw.ctl_sock);
    close_sock(&ipgw.audio_fd);
    ipgw.status=IDLE;
  }
  else if(strncmp(buf,"INC",3)==0){
    int cnt=0;
    char command[6];
    char number[32];

    write(ipgw.pnc_fd,"PTR1\r",5);
    read_pnc(command);
    printf("take DTMP->%s\n",command);
    /* #が押されるまで待つ */
    while (command[3]!='#'){
      read_pnc(command);
      putchar(command[3]);
      number[cnt++]=command[3];
    }
    number[cnt-1]=(char)NULL;
    strcpy(ipgw.your_number,number);
    printf("  dialing to %s\n",ipgw.your_number);

    /* GKにアドレスを聞く */
    if (ask_gk(GKREQ_MSG)==1){

      /* GKが何らかのアドレスを返してくれた場合 */
      /* IPアドレスが存在すれば通話要求をする */
      if (ipgw.your_ipaddr!=0){
        printf("  dialing to %s... (port %d & %d)\n",inet_ntoa(ipgw.your_ipad
dr),
                ipgw.your_listen_port,ipgw.your_data_port);
        /* 相手に接続する */
        ipgw.status=CALLING;
        ipgw.ctl_sock =
          connect_sock(ipgw.your_ipaddr , ipgw.your_listen_port);
        if (ipgw.ctl_sock!=-1){
          /* 回線的につながる */
          printf("  connected!\n");
          send_msg(REQ_MSG);
```

```
345           printf("  waiting for reply...\n");
346         }
347         else {
348           print_menu();
349           ipgw.status=IDLE;
350           pnc_onhook();
351         }
352       }
353
354       /* IPアドレスが存在しない場合 */
355       /* (相手がネットワーク上にいないorGWが存在しない場合) */
356       else {
357         printf("IPGW.NO = %s is not on the network\n",ipgw.your_number);
358         pnc_onhook();
359       }
360     }
361     else {
362       fprintf(stderr,"cannot connect to gk.\n");
363       pnc_onhook();
364     }
365   }
366 }
367
368 /*-------------------------------------------------------------
369  * void pnc_onhook()
370  *          回線切断を行なう
371  *-------------------------------------------------------------*/
372 void pnc_onhook()
373 {
374   char command[5];
375
376   write(ipgw.pnc_fd,"ONH\r",4);
377   read_pnc(command);
378   printf("onhook = %s\n",command);
379
380 }
381
382 int ask_gk(int msg_id)
383 {
384   int gk_sock,n;
385   char buf[256];
386   struct msg_header *p = NULL;
387   struct gkreq_msg  *q = NULL;
388   struct gkrep_msg  *r = NULL;
389   struct gkupdate_msg *s = NULL;
390   struct gkupdaterep_msg *t=NULL;
391
392   /* gatekeeperへTCPでコネクションを張る */
393   if ( (gk_sock=connect_sock(ipgw.gk_ipaddr,ipgw.gk_port)) == -1 ){
394     fprintf(stderr,"cannot connect gatekeeper %s\n",inet_ntoa(ipgw.gk_ipaddr)
);
395     return -1;
396   }
397   switch(msg_id){
398   case GKREQ_MSG:
399     /* アドレスをGKに聞く */
400     p = (struct msg_header *)malloc(sizeof(*p));
401     p->msg_id=htons(GKREQ_MSG);
402     p->msg_len=htons(sizeof(*q));
403     strcpy(p->my_number,ipgw.my_number);
404     q = (struct gkreq_msg *)malloc(sizeof(*q));
405     strcpy(q->your_number,ipgw.your_number);
406     if ( (n=writen(gk_sock, (char *)p , sizeof(*p))) < 0 ){
407       return -1;
408     }
409     if ( (n=writen(gk_sock, (char *)q , sizeof(*q))) < 0 ){
```



```
410       fprintf(stderr,"cannot send to gatekeeper %s\n",inet_ntoa(ipgw.gk_ipadd
r));
411       return -1;
412     }
413     printf("waiting for gk_rep\n");
414     free(p);
415     free(q);
416     break;
417   }
418
419   /* GKから返答をもらう */
420   if ( (n=readn(gk_sock,buf,sizeof(struct msg_header))) < 0 ) {
421     fprintf(stderr,"cannot recv from gatekeeper %s\n",inet_ntoa(ipgw.gk_ipadd
r));
422     return -1;
423   }
424   p = (struct msg_header *)buf;
425   switch(ntohs(p->msg_id)){
426   case GKREP_MSG:
427     /* アドレスリプライの場合 */
428     printf("recv gk_rep\n");
429     /* メッセージ本体を読む */
430     if ( ( n=readn(gk_sock,buf,sizeof(struct gkrep_msg)) ) < 0 ){
431       fprintf(stderr,"cannot recv from gatekeeper %s\n",inet_ntoa(ipgw.gk_ipa
ddr));
432       return -1;
433     }
434     r = (struct gkrep_msg *)buf;
435     /* アドレスが存在した場合 */
436     if (ntohs(r->rep_type) == 0x0001 ){
437       ipgw.your_ipaddr=r->ipaddr;
438       ipgw.your_listen_port=ntohs(r->listen_port);
439       ipgw.your_data_port=ntohs(r->data_port);
440       strcpy(ipgw.your_number,r->your_number);
441     }
442     /* アドレスが存在しなかった場合 */
443     else {
444       ipgw.your_ipaddr=htonl(0);
445       ipgw.your_listen_port=ntohs(0);
446       ipgw.your_data_port=ntohs(0);
447     }
448     break;
449   }
450
451   /* GKとのコネクションを切る */
452   close_sock(&gk_sock);
453   return 1;
454 }
455
456 /*-------------------------------------------------------------
457  * int audio_open()
458  *          オーディオデバイスのオープン
459  *          オーディオデバイスの初期化
460  *          オーディオデバイスのFDを返す
461  *-------------------------------------------------------------*/
462 int audio_open()
463 {
464   int audio_fd;               /* オーディオデバイスのFD */
465   int i=-1;
466   u_long fmt,play_fmt,rec_fmt;
467   snd_chan_param sndparam;     /* オーディオの設定 */
468   snd_capabilities soundcaps;  /* オーディオのサポート機能 */
469   struct snd_size sndsize;     /* オーディオバッファサイズ */
470
471   /* オーディオデバイス オープン */
472   audio_fd = open(AUDIO_DEVICE,O_RDWR);
```

```
473    if (audio_fd < 0) {
474      fprintf(stderr,"can not open audio_device %s\n",AUDIO_DEVICE);
475      exit(-1);
476    }
477
478    i=ioctl(audio_fd,AIOGCAP, &soundcaps);
479    fmt=soundcaps.formats;
480
481    /* 使用できるサウンドフォーマットを調べる */
482    switch(fmt & (AFMT_FULLDUPLEX | AFMT_WEIRD)) {
483    case AFMT_FULLDUPLEX:
484      /* if sound card supports FULL-DUPLEX operation */
485      if ( fmt & AFMT_S16_LE )
486        play_fmt=rec_fmt=AFMT_S16_LE;
487      else {
488        fprintf(stderr,"no supported sound formats\n");
489        close(audio_fd);
490        exit(-1);
491      }
492      ipgw.duplex=FULLDUPLEX;
493      break;
494    case AFMT_FULLDUPLEX | AFMT_WEIRD:
495      /* if sound card supports FULL-DUPLEX operation but weird hardware */
496      if (fmt & AFMT_S16_LE){
497        rec_fmt=AFMT_S16_LE;
498        play_fmt=AFMT_U8;
499      }
500      else {
501        fprintf(stderr,"no supported sound formats\n");
502        close(audio_fd);
503        exit(-1);
504      }
505      ipgw.duplex=FULLDUPLEX;
506      break;
507    default:
508      /* if sound card can't support FULL-DUPLEX operation */
509      printf("Half Duplex Mode\n");
510      if ( fmt & AFMT_U8 ){
511        printf("unsigned 8bits format \n");
512        rec_fmt=AFMT_U8;
513        play_fmt=AFMT_U8;
514      }
515      ipgw.duplex=HALFDUPLEX;
516      break;
517    }
518
519    /* オーディオフォーマットの設定 */
520    switch(ipgw.duplex){
521    case FULLDUPLEX:
522      sndparam.rec_format=rec_fmt;
523      sndparam.play_format=play_fmt;
524      sndparam.play_rate=sndparam.rec_rate=RATE;
525      ioctl(audio_fd,AIOSFMT,&sndparam);
526      sndsize.play_size= (sndparam.play_format == AFMT_S16_LE) ? 2*BLKSIZE : BL
KSIZE;
527      sndsize.rec_size=  (sndparam.rec_format  == AFMT_S16_LE) ? 2*BLKSIZE : BL
KSIZE;
528      i = ioctl(audio_fd,AIOSSIZE,&sndsize);
529
530      ipgw.play_fmt=play_fmt;
531      ipgw.rec_fmt=rec_fmt;
532      break;
533    case HALFDUPLEX:
534      sndparam.play_format=play_fmt;
535      sndparam.play_rate=RATE;
536      ioctl(audio_fd,AIOSFMT,&sndparam);
```



```
537      sndsize.play_size=BLKSIZE;
538      i=ioctl(audio_fd,AIOSSIZE,&sndsize);
539      break;
540    }
541
542    return audio_fd;
543  }
544
545  /*-------------------------------------------------------------
546   * void audio_read()
547   *          オーディオデバイスから音声を取り込む
548   *          音声をGSMエンコードする
549   *          エンコードしたデータをネットワークに投げる
550   *-------------------------------------------------------------*/
551  void audio_read()
552  {
553    static struct rtphdr *rtp=NULL;
554    struct sockaddr_in you;
555    double energy = 0.0;
556    double avg ;
557    short buf[BLKSIZE];    /* 16bits用録音バッファ */
558    u_char ubuf[BLKSIZE];  /*  8bits用録音バッファ */
559    int len;               /* サンプルのバイト数   */
560    int i;
561
562    /* RTPパケットの初期化 */
563    if (!rtp ){
564      rtp = (struct rtphdr *)malloc(sizeof(*rtp));
565      rtp->v = 2;
566      rtp->p = 0;
567      rtp->x = 0;
568      rtp->cc = 0;
569      rtp->m = 1;
570      rtp->pt = PT_GSM;
571      rtp->seq = htons(0x1234);
572      rtp->timestamp = htonl(ipgw.nowtime);
573      rtp->padding=0;
574    }
575
576    /* オーディオ録音 */
577    switch(ipgw.rec_fmt){
578    case AFMT_S16_LE:
579      /* BLKSIZE個のサンプルを読み込む。16bitsなので、2倍読み込む */
580      len = read(ipgw.audio_fd , buf , 2 * BLKSIZE );
581      if (len != 2*BLKSIZE)
582        fprintf(stderr , "can't fill rec_buf. \n");
583      break;
584    case AFMT_U8:
585      /* BLKSIZE個のサンプルを読み込む。 8bitsなので、そのまま */
586      len = read(ipgw.audio_fd , ubuf , BLKSIZE );
587      if (len != BLKSIZE)
588        fprintf(stderr , "can't fill rec_buf. \n");
589      /* ただし、u_char -> signed short に変換する */
590      for (i=0 ; i < len ; i++){
591        buf[i]=(ubuf[i] - 128 ) << 8 ;
592      }
593      break;
594    default:
595      fprintf(stderr,"no support format.\n");
596      exit(-1);
597    }
598
599    /* 無音圧縮 from  tel.c
600     * compute energy, useful for silence suppression...
601     * Use a highpass filter to remove the DC component.
602     */
```

```
603    for (i = 0 , energy = 0.0 , avg = 0.0 ; i < BLKSIZE ; i++) {
604      int a ;
605      static int dc = 0 ;
606  #define BITS 8
607      dc +=  buf[i] - (dc >> BITS) ;
608      a = ( buf[i] - (dc>>BITS) ) ;
609      if (a < -32767) buf[i] = -32767 ;
610      else if (a > 32767) buf[i] = 32767 ;
611      else buf[i] = a ;
612      avg += a ;
613      energy += a*a ;
614    }
615    energy /= BLKSIZE ;
616    avg /= BLKSIZE ;
617
618    /* RTPヘッダの作成 */
619    rtp->ssrc = ipgw.my_ssrc;
620    rtp->timestamp = htonl(ipgw.nowtime);
621    rtp->seq  = htons(1 + ntohs(rtp->seq));
622
623    if (ipgw.suppress == 0 || energy > 5e4 ){
624      /* GSMエンコーディング */
625      gsm handle;
626      handle = gsm_create();
627      if (handle == 0) fprintf(stderr,"gsm_create error\n");
628      gsm_encode(handle , buf , rtp->data );
629
630      /* パケットを指定した相手に送る */
631      if ( ipgw.your_data_port != 0 ){
632        trans_sockaddr(&you , ipgw.your_ipaddr , ipgw.your_data_port);
633        len=sendto(ipgw.data_sock , rtp , sizeof(*rtp) , 0 ,
634                (struct sockaddr *)&you,sizeof(you));
635
636        rtp->m = 0;
637      }
638
639      gsm_destroy(handle);
640    }
641  }
642
643  /*-------------------------------------------------------------
644   * void net_read()
645   *          ネットワークからデータを読み込む
646   *          GSMデータをキューに貯める
647   *-------------------------------------------------------------*/
648  void net_read()
649  {
650    static unsigned short start_seq;
651    struct sockaddr_in from;
652    struct rtphdr *rtp = NULL ;
653    int buf_num=0;
654    int fromlen;
655    int len;
656    int i;
657
658    rtp = (struct rtphdr *) malloc (sizeof(*rtp)) ;
659
660    /* パケットをソケットから受け取る */
661    fromlen=sizeof(from);
662    bzero(&from,sizeof(from));
663    len=recvfrom(ipgw.data_sock , rtp , sizeof(*rtp) , 0 ,
664                (struct sockaddr *)&from,&fromlen);
665
666    /* 通話相手以外から来たデータは捨てる */
667    if ( ipgw.your_ipaddr != from.sin_addr.s_addr) return ;
668    /* 待ち受け時は相手のUDPポートが何番かわらないのでそれを上書き */
```



```
669    ipgw.your_data_port = ntohs(from.sin_port);
670
671    /* 最初のパケットの場合は差分時間を算出する */
672    if ( rtp->m==1 || ipgw.delta==0 ){
673      ipgw.delta=ipgw.nowtime-ntohl(rtp->timestamp);
674      start_seq=ntohs(rtp->seq);
675    }
676    /* 到着時に再生可能時間が過ぎている場合 */
677    if (ntohl(rtp->timestamp) + ipgw.delta <  ipgw.nowtime - CACHE*BLKSIZE  ){
678      free(rtp);
679      return;
680    }
681
682    /* 正しい大きさ(33bytes)読んだかどうかをチェックする */
683    /* (gsm_decodeの時点で33bytesないという可能性をなくす) */
684    if (len!=sizeof(*rtp))
685      fprintf(stderr,"can't read all of sock data len=%d\n",len);
686
687    /* サウンドキューに蓄える (ハッシュ方式) */
688    buf_num = ( ntohs(rtp->seq) - start_seq ) % MAXHASH ;
689    /*fprintf(stderr,"en hash %d\n",buf_num);*/
690    queue[ buf_num ].timestamp=ntohl(rtp->timestamp);
691    bcopy(rtp->data , queue[ buf_num ].data , sizeof(gsm_frame));
692
693    free(rtp);
694    return;
695  }
696
697  void audio_write()
698  {
699    short  buf[BLKSIZE];
700    u_char ubuf[BLKSIZE];
701    gsm handle;
702    int len;
703    int i;
704
705    /* 最初のバッファリング */
706    if ( start_time + CACHE*BLKSIZE > ipgw.nowtime ) {
707      fprintf(stderr,"buffering... %d\n",queue_num);
708      return;
709    }
710    /* キューにバッファが存在していれば再生する */
711    if (queue[queue_num].timestamp!=0 ){
712    /* GSMデコーディング */
713      handle = gsm_create();
714      if (handle == 0 ) fprintf(stderr,"gsm_create error\n");
715      gsm_decode(handle , queue[queue_num].data , buf );
716      /*printf("%d\n",queue_num);*/
717      /* オーディオ再生 */
718      switch(ipgw.play_fmt){
719      case AFMT_S16_LE:
720        /* play_formatがsigned 16bits ならそのまま出力 */
721        len = write(ipgw.audio_fd , buf , 2*BLKSIZE);
722        if (len != 2*BLKSIZE )
723          fprintf(stderr , "can't play all of buffer. S16_LE\n");
724        break;
725      case AFMT_U8:
726        /* play_formatがunsigned 8bits なら変換して出力 */
727        for (i=0 ;  i < BLKSIZE ; i++){
728          ubuf[i] = (u_char)(buf[i] >> 8 ) ^ 0x80 ;
729        }
730        len = write(ipgw.audio_fd , ubuf , BLKSIZE);
731        if (len != BLKSIZE )
732          fprintf(stderr , "%d %d can't play all of buffer.  %d U8\n",ipgw.audi
o_fd,len,errno);
733        break;
```

```
734      default:
735        fprintf(stderr," no support format.\n");
736        exit(-1);
737      }
738
739      gsm_destroy(handle);
740      /* キューをクリアする */
741      queue[queue_num].timestamp=0;
742    }
743    else {
744      /*fprintf(stderr,"q_num=%d ts=%x nt=%x d=%d\n",
745           queue_num,queue[queue_num].timestamp,ipgw.nowtime,ipgw.delta);*/
746    }
747
748    queue_num++;
749    if (queue_num==MAXHASH) queue_num=0;
750    return;
751  }
752
753  /*-------------------------------------------------------------
754   * void trans_sockaddr()
755   *          与えられたIPアドレスとポートから、sockaddr_in構造体を返す
756   *          ipaddrはNetwork Byte Order.PortはHost byte orderとする
757   *-------------------------------------------------------------*/
758  void trans_sockaddr(struct sockaddr_in *net , u_long ipaddr , u_short port)
759  {
760    bzero(net,sizeof(*net));
761    net->sin_family = AF_INET;
762    net->sin_port = htons(port);
763    net->sin_addr.s_addr = ipaddr;
764  }
765
766  /*-------------------------------------------------------------
767   * void make_listen_sock()
768   *          ipgw.my_listen_portに待ち受けソケットを作る。
769   *-------------------------------------------------------------*/
770  void make_listen_sock(){
771    char hostname[41];
772    struct sockaddr_in net;
773    struct hostent *host;
774
775    /* make socket */
776    ipgw.listen_sock=socket(AF_INET,SOCK_STREAM,0);
777    if (ipgw.listen_sock < 0 ){
778      fprintf(stderr,"can't make sockets\n");
779      exit(-1);
780    }
781    /* bind listen_socket */
782    /* INADDR_ANYで待ち受けソケットをbindする */
783    trans_sockaddr(&net , INADDR_ANY , ipgw.my_listen_port);
784    printf("Listening %s port %d\n",inet_ntoa(net.sin_addr),ipgw.my_listen_port
);
785    if ( bind(ipgw.listen_sock,(struct sockaddr *)&net , sizeof(net)) < 0 ) {
786      fprintf(stderr,"can't bind listen_sock\n");
787      exit(-1);
788    }
789    /* 最大1つの接続を待ち受け */
790    listen(ipgw.listen_sock,1);
791  }
792
793  /*-------------------------------------------------------------
794   * void make_data_sock()
795   *          ipgw.my_data_portに音声データの送受信ソケットを作る。
796   *-------------------------------------------------------------*/
797  void make_data_sock()
798  {
```



```
799    struct sockaddr_in net;
800
801    /* make socket */
802    ipgw.data_sock=socket(AF_INET,SOCK_DGRAM, 0);
803    if ( ipgw.data_sock < 0 ){
804      fprintf(stderr,"can't make socket\n");
805      exit(-1);
806    }
807    /* connect data_socket */
808    bzero(&net,sizeof(net));
809    trans_sockaddr(&net , INADDR_ANY , ipgw.my_data_port);
810    if (bind(ipgw.data_sock , (struct sockaddr *)&net , sizeof(net)) < 0 ) {
811      fprintf(stderr,"can't bind data_sock\n");
812      exit(-1);
813    }
814    printf("binded data_sock \n");
815  }
816
817  /*-------------------------------------------------------------
818   * int connect_sock(int insock)
819   *          通話相手への接続要求を出す。
820   *-------------------------------------------------------------*/
821  int connect_sock(u_long ipaddr , u_short port)
822  {
823    int sock;
824    struct sockaddr_in net;
825
826    /* make socket */
827    sock=socket(AF_INET,SOCK_STREAM,0);
828    if ( sock < 0 ){
829      fprintf(stderr,"can't make socket\n");
830      exit(-1);
831    }
832    /* connect ctl_socket */
833    bzero(&net,sizeof(net));
834    trans_sockaddr(&net , ipaddr , port);
835    if (connect(sock ,  (struct sockaddr *)&net , sizeof(net)) < 0 ) {
836      fprintf(stderr,"can't connect ctl_sock\n");
837      return -1;
838    }
839    printf("  connected sock to %s port%d\n",inet_ntoa(ipaddr),port);
840    return sock;
841  }
842
843  void close_sock(int *sock)
844  {
845    close(*sock);
846    *sock=-1;
847  }
848
849  void recv_msg(char *buf)
850  {
851    int n;
852    struct msg_header *p = NULL;
853    struct req_msg    *r = NULL;
854    struct refuse_msg *q = NULL;
855    char content[33];
856
857    p = (struct msg_header *)buf;
858    /*fprintf(stdout,"msg_id=%x from %s len=%d\n",ntohs(p->msg_id),p->my_number
,ntohs(p->msg_len));*/
859    switch(ntohs(p->msg_id)){
860    case REQ_MSG:
861      n=readn(ipgw.tmp_sock , content , ntohs(p->msg_len));
862      r = (struct req_msg *)content;
863      printf("recv CALLING to %s from %s\n",r->your_number,p->my_number);
```

```
    strcpy(ipgw.your_number,r->your_number);
    fprintf(stdout,"recv REQ_MSG\n");
    break;
  case ACCEPT_MSG:
    printf("recv ACCEPT_MSG from %s. Start!\n",p->my_number);
    ipgw.status=TALKING; /* 通話開始 */
    start_time=ipgw.nowtime;
    queue_num=0;
    ipgw.delta=0;
    if(ipgw.audio_fd<0) ipgw.audio_fd=audio_open();
    break;
  case REFUSE_MSG:
    n=readn(ipgw.ctl_sock , content, ntohs(p->msg_len));
    q = (struct refuse_msg *)content;
    printf("recv REFUSE_MSG from %s\n",p->my_number);
    switch(ntohs(q->reply_type)){
    case TALKING:
      printf("  He is talking now. So He refused connection from you.\n");
      break;
    case CALLING:
      printf("  He is calling now. So He refused connection from you.\n");
      break;
    case CALLED:
      printf("  He is called  now. So He refused connection from you.\n");
      break;
    case IDLE:
      printf("  He don't want to talk with you. \n");
      break;
    }
    pnc_onhook();
    close_sock(&ipgw.ctl_sock);
    /*ipgw.ctl_sock=-1;*/
    ipgw.status=IDLE;
    break;
  case FIN_MSG:
    printf("recv FIN_MSG from %s.\n",p->my_number);
    ipgw.status=IDLE;
    pnc_onhook();
    close_sock(&ipgw.tmp_sock);
    close_sock(&ipgw.ctl_sock);
    close_sock(&ipgw.audio_fd);
    break;
  }
}

/* ネゴシエーションのためにメッセージを送る */

void send_msg(int msg_id)
{
  int n;
  struct msg_header *p = NULL;
  struct req_msg    *r = NULL;
  struct refuse_msg *q = NULL;

  switch(msg_id){
  case REQ_MSG:
    printf("send REQ_MSG\n");
    p = (struct msg_header *)malloc(sizeof(*p)+sizeof(*r));
    p->msg_id=htons(REQ_MSG);
    p->msg_len=htons(sizeof(*r));
    strcpy(p->my_number,ipgw.my_number);
    r = (struct req_msg *)malloc(sizeof(*r));
    strcpy(r->your_number,ipgw.your_number);

    n=writen(ipgw.ctl_sock, (char *)p , sizeof(*p));
    /*printf("send %dbytes\n",n);*/
```



```
    n=writen(ipgw.ctl_sock, (char *)r , sizeof(*r));
    /*printf("send %dbytes\n",n);*/
    free(p);
    free(r);
    break;
  case ACCEPT_MSG:
    printf("send ACCEPT_MSG\n");
    p = (struct msg_header *)malloc(sizeof(*p));
    p->msg_id=htons(ACCEPT_MSG);
    p->msg_len=htons(0);
    strcpy(p->my_number,ipgw.my_number);
    n=writen(ipgw.ctl_sock, (char *)p , sizeof(*p));
    /*printf("send %dbytes\n",n);*/
    free(p);
    break;
  case REFUSE_MSG:
    printf("send REFUSE_MSG\n");
    p = (struct msg_header *)malloc(sizeof(*p)+sizeof(*q));
    p->msg_id=htons(REFUSE_MSG);
    p->msg_len=htons(sizeof(*q));
    strcpy(p->my_number,ipgw.my_number);
    q = (struct refuse_msg *)malloc(sizeof(*q));
    /*printf("%x %x %d \n",p,q,ntohs(p->msg_len));*/

    /* 通話拒否メッセージ */
    switch(ipgw.status){
    case TALKING:
      q->reply_type=htons(TALKING);
      printf("GW is talking now. So GW refused connection. \n");
      break;
    case CALLING:
      q->reply_type=htons(CALLING);
      printf("GW is calling now. So GW refused connection. \n");
      break;
    case CALLED:
      q->reply_type=htons(CALLED);
      printf("GW called . So GW refused connection.\n");
      break;
    case IDLE:
      q->reply_type=htons(IDLE);
      printf("GW don't want to accept. So GW refused connection.\n");
      break;
    case GW_YOUTALKING :
      q->reply_type=htons(GW_YOUTALKING);
      printf("He is talking to someone. So GW refused connection. \n");
      break;
    case GW_CALLING :
      q->reply_type=htons(GW_CALLING);
      printf("GW is just called from someone. So GW refused connection. \n");
      break;
    case GW_NOTHING :
      q->reply_type=htons(GW_NOTHING);
      printf("He didn't hang up . So GW disconnected. \n");
      break;
    case GW_ERROR:
      q->reply_type=htons(GW_ERROR);
      printf("some error happend.So GW disconnected.\n");
      break;
    }
    n=writen(ipgw.tmp_sock, (char *)p , sizeof(*p));
    /*printf("send %dbytes\n",n);*/
    n=writen(ipgw.tmp_sock, (char *)q , sizeof(*q));
    /*printf("send %dbytes\n",n);*/
    free(p);
    free(q);
    break;
```

```
 996
 997    case FIN_MSG:
 998      printf("send FIN_MSG\n");
 999      /* 通話終了メッセージ */
1000      p = (struct msg_header *)malloc(sizeof(*p));
1001      p->msg_id=htons(FIN_MSG);
1002      p->msg_len=htons(0);
1003      strcpy(p->my_number,ipgw.my_number);
1004      n=writen(ipgw.ctl_sock, (char *)p , sizeof(*p));
1005      /*printf("send %dbytes\n",n);*/
1006      free(p);
1007      break;
1008    }
1009  }
1010
1011  void print_menu()
1012  {
1013      fprintf(stderr,"%s",menu);
1014      fprintf(stderr,">");
1015  }
1016
1017  void cmd_line(char *buf)
1018  {
1019    char ipaddr[256];
1020
1021    switch(buf[0]){
1022    case 's':   /* stop   */
1023      if (ipgw.status==TALKING || ipgw.status==CALLING ){
1024        send_msg(FIN_MSG);
1025        close_sock(&ipgw.ctl_sock);
1026        close_sock(&ipgw.audio_fd);
1027        ipgw.status=IDLE;
1028      }
1029      else if (ipgw.status==CALLED){
1030        ipgw.status=IDLE;
1031        send_msg(REFUSE_MSG);
1032        close_sock(&ipgw.tmp_sock);
1033        close_sock(&ipgw.ctl_sock);
1034        close_sock(&ipgw.audio_fd);
1035      }
1036      else {
1037        print_menu();
1038      }
1039      break;
1040    case 'q':   /* quit   */
1041      if (ipgw.status==TALKING){
1042        send_msg(FIN_MSG);
1043        close_sock(&ipgw.audio_fd);
1044        ipgw.status=IDLE;
1045      }
1046      close_sock(&ipgw.ctl_sock);
1047      close_sock(&ipgw.listen_sock);
1048      close_sock(&ipgw.pnc_fd);
1049      ipgw.status=IDLE;
1050      ipgw.tty_fd=-1;
1051      fprintf(stdout,"byebye..\n");
1052      break;
1053    case '?':
1054      fprintf(stderr,"status = %d\n",ipgw.status);
1055      fprintf(stderr,"listen_sock %d\n",ipgw.listen_sock);
1056      fprintf(stderr,"tmp_sock %d\n",ipgw.tmp_sock);
1057      fprintf(stderr,"ctl_sock %d\n",ipgw.ctl_sock);
1058      fprintf(stderr,"data_sock %d\n",ipgw.data_sock);
1059      fprintf(stderr,"ipgw.delta %d\n",ipgw.delta);
1060      fprintf(stderr,"pnc_fd = %d\n",ipgw.pnc_fd);
1061      break;
```



```
1062    default:
1063      print_menu();
1064      break;
1065    }
1066  }
1067
1068  /*---------------------------------------
1069   * void main
1070   *
1071   *
1072   *---------------------------------------*/
1073  void main(int argc, char *argv[])
1074  {
1075    int i,n,m,j,size,fromlen,status;
1076    char filename[255];
1077    char ipaddr[255];
1078    char buf[1024];
1079    fd_set readfds,fds;
1080    struct sockaddr_in net;
1081    struct timeval tv;
1082
1083    queue_num=0;
1084
1085    ipgw.suppress=1;
1086    ipgw.my_listen_port=MYLISTENPORT;
1087    ipgw.my_data_port=MYDATAPORT;
1088    ipgw.status=IDLE;
1089    strcpy(ipgw.my_number,"+81-466-49-1124");
1090    ipgw.my_ssrc=random32(0);
1091    ipgw.delta=0;
1092    ipgw.tty_fd=0;
1093    ipgw.listen_sock=-1;
1094    ipgw.ctl_sock   =-1;
1095    ipgw.data_sock  =-1;
1096    ipgw.tmp_sock   =-1;
1097    ipgw.audio_fd   =-1;
1098
1099    strcpy(ipgw.ccode,COUNTRYCODE);
1100    strcpy(ipgw.dialprefix,DPREFIX);
1101    strcpy(ipaddr,GKADDRESS);
1102    inet_aton(ipaddr,&ipgw.gk_ipaddr);
1103    ipgw.gk_port=GKLISTENPORT;
1104    printf("gatekeeper=%s port%d\n",inet_ntoa(ipgw.gk_ipaddr),ipgw.gk_port);
1105
1106    /* PNCデバイスをオープン 初期化 */
1107    ipgw.pnc_fd = pnc_open();
1108    pnc_init();
1109
1110    /* 待ち受けソケットを作る */
1111    make_listen_sock();
1112    make_data_sock();
1113    print_menu();
1114
1115    while(!ipgw.tty_fd){
1116      fprintf(stderr,"\r>");
1117      n=0;
1118      FD_ZERO(&readfds);
1119      if (ipgw.tty_fd>=0){
1120  /*      printf("tty_fd =%d\n",ipgw.tty_fd);*/
1121        FD_SET(ipgw.tty_fd,&readfds);        /* 標準入力を監視対象 */
1122        if (ipgw.tty_fd > n ) n = ipgw.tty_fd;
1123      }
1124      if (ipgw.listen_sock>=0){
1125  /*      printf("listen_sock =%d\n",ipgw.listen_sock);*/
1126        FD_SET(ipgw.listen_sock,&readfds);   /* 待ち受けソケットを監視対象 */
1127        if (ipgw.listen_sock > n ) n = ipgw.listen_sock;
```

```
1128      }
1129      if (ipgw.tmp_sock>=0){
1130  /*    printf("tmp_sock =%d\n",ipgw.tmp_sock);*/
1131        FD_SET(ipgw.tmp_sock,&readfds);   /* TMPソケットを監視対象 */
1132        if (ipgw.tmp_sock > n ) n = ipgw.tmp_sock;
1133      }
1134      if (ipgw.ctl_sock>=0){
1135  /*     printf("ctl_sock =%d\n",ipgw.ctl_sock);*/
1136        FD_SET(ipgw.ctl_sock,&readfds);      /* 制御ソケットを監視対象 */
1137        if (ipgw.ctl_sock > n ) n = ipgw.ctl_sock;
1138      }
1139      if (ipgw.data_sock>=0){
1140  /*     printf("data_sock =%d\n",ipgw.data_sock);*/
1141        FD_SET(ipgw.data_sock,&readfds);      /* 音声データソケットを監視対象 *
/
1142        if (ipgw.data_sock > n ) n = ipgw.data_sock;
1143      }
1144      if (ipgw.audio_fd>=0){
1145  /*     printf("audio_fd =%d\n",ipgw.audio_fd);*/
1146        FD_SET(ipgw.audio_fd,&readfds);      /* オーディオデバイスを監視対象 */
1147        if (ipgw.audio_fd > n ) n = ipgw.audio_fd;
1148      }
1149      if (ipgw.pnc_fd>=0){
1150  /*     printf("pnc_fd =%d\n",ipgw.pnc_fd); */
1151        FD_SET(ipgw.pnc_fd,&readfds);        /* PNCを監視対象 */
1152        if (ipgw.pnc_fd > n ) n = ipgw.pnc_fd;
1153      }
1154      n++;
1155
1156      /* printf("select\n");*/
1157      m=select(n,&readfds,NULL,NULL,NULL);
1158      gettimeofday(&tv, NULL);
1159      ipgw.nowtime = tv.tv_sec * RATE + (tv.tv_usec*RATE/1000000);
1160
1161      /* 標準入力の場合 */
1162      if (ipgw.tty_fd>=0 && FD_ISSET(ipgw.tty_fd,&readfds) ){
1163        scanf("%s",buf);
1164        cmd_line(buf);
1165      }
1166
1167      /* 待ち受けソケットの場合 */
1168      if (ipgw.listen_sock >= 0 && FD_ISSET(ipgw.listen_sock,&readfds) ){
1169        /* fprintf(stdout,"Somebody connect to me\n"); */
1170        /* 待ち受けソケットに来た通話をとりあえずacceptする */
1171        fromlen=sizeof(net);
1172        ipgw.tmp_sock=
1173          accept(ipgw.listen_sock,(struct sockaddr *)&net,&fromlen);
1174        if (ipgw.tmp_sock < 0) fprintf(stderr,"error:can't accept \n");
1175
1176        /* 待ち受けソケットからデータを読む */
1177        i=readn(ipgw.tmp_sock,buf,sizeof(struct msg_header));
1178        /*fprintf(stdout,"%d\n",i);*/
1179
1180        /* 誰からの通話か表示する */
1181        recv_msg(buf);
1182        /* PNC STATUSを調べ、通話中、呼出中、呼び出され中なら
1183           即効拒否メッセージを送る */
1184        if (pnc_stat()!=IDLE || ipgw.status != IDLE ){
1185          send_msg(REFUSE_MSG);
1186          close_sock(&ipgw.tmp_sock);
1187        }
1188        /* PNCが発信可能状態(IDLE)であればダイアル */
1189        else {
1190          ipgw.status=CALLED;
1191          ipgw.your_ipaddr=net.sin_addr.s_addr;
1192          fprintf(stderr,"dialing to %s\n",ipgw.your_number);
```



```
1193          i=pnc_dial(ipgw.your_number);
1194          /* dialコマンドのインディケーションを検出*/
1195          switch(i){
1196          case GW_SUCCEED:   /* 回線接続 */
1197            ipgw.ctl_sock=ipgw.tmp_sock;
1198            ipgw.tmp_sock=-1;
1199            send_msg(ACCEPT_MSG); /* 了解メッセージを送信 */
1200            ipgw.status=TALKING;  /* 通話開始 */
1201            start_time=ipgw.nowtime;
1202            queue_num=0;
1203            ipgw.delta=0;
1204            if(ipgw.audio_fd<0) ipgw.audio_fd=audio_open();
1205            detect_lineconnect();
1206            break;
1207          default:       /* それ以外の場合は理由をそえて、拒否メッセージを送信
*/
1208            ipgw.status=i;
1209            send_msg(REFUSE_MSG);
1210            close_sock(&ipgw.tmp_sock);
1211            ipgw.status=IDLE; /* アイドル状態に戻す */
1212            break;
1213          }
1214        }
1215      }
1216      /* テンポラリソケットの場合(呼ばれている時に、更に別の誰かに呼ばれた場合)
*/
1217      if (ipgw.tmp_sock >= 0 && FD_ISSET(ipgw.tmp_sock,&readfds) ) {
1218        i=readn(ipgw.tmp_sock,buf,sizeof(struct msg_header));
1219        recv_msg(buf);
1220      }
1221      /* 制御ソケットの場合 */
1222      if (ipgw.ctl_sock >= 0 && FD_ISSET(ipgw.ctl_sock,&readfds) ) {
1223        i=readn(ipgw.ctl_sock,buf,sizeof(struct msg_header));
1224        recv_msg(buf);
1225      }
1226      /* 音声データソケットの場合 */
1227      if (ipgw.data_sock >= 0 && FD_ISSET(ipgw.data_sock,&readfds) ) {
1228        net_read();
1229      }
1230      /* オーディオデバイスの場合 */
1231      if (ipgw.audio_fd >= 0 && FD_ISSET(ipgw.audio_fd,&readfds) ) {
1232        if (ipgw.duplex==FULLDUPLEX){
1233          audio_write();
1234          audio_read();
1235        }
1236      }
1237      if (ipgw.pnc_fd >= 0 && FD_ISSET(ipgw.pnc_fd,&readfds) ) {
1238        read_pnc(buf);
1239        pnc_cmd(buf);
1240      }
1241    }
1242    close_sock(&ipgw.tmp_sock);
1243    close_sock(&ipgw.data_sock);
1244    close_sock(&ipgw.audio_fd);
1245    close_sock(&ipgw.pnc_fd);
1246  }
1247
1248
```

Gatekeeperソースリスト

ipgk.h

```
   1  #define ON 1
   2  #define OFF 0
   3                          /* 使用するポート番号によって変更 */
```

```
 4  #define GKLISTENPORT 16000
 5  #define DATA 0
 6  #define CTL 1
 7  #define TTL 10000
 8
 9  #define REQ_MSG         0x0001
10  #define ACCEPT_MSG      0x0002
11  #define REFUSE_MSG      0x0003
12  #define FIN_MSG         0x0004
13  #define GKREQ_MSG       0x0005
14  #define GKREP_MSG       0x0006
15  #define GKUPDATE_MSG    0x0007
16  #define GKUPDATEREP_MSG 0x0008
17
18  #define TELDOMAIN "e164.int.mf.ps.sfc.wide.ad.jp"
19  #define GWDOMAIN "gw.e164.mf.ps.sfc.wide.ad.jp"
20
21  char menu[] =
22  "    l number : lookup iptel-number \n"
23  "    q        : quit\n"
24  "    ?        : show status\n"
25  ;
26
27  struct ipgk_status {
28    u_long gk_ipaddr;        /* gatekeeperのIPアドレス */
29    u_short gk_port;         /* gatekeeperのポート */
30    u_long client_ipaddr;    /* 接続してくるクライアントのIPアドレス */
31    char my_number[32];      /* 自分のIP電話番号 */
32    char your_number[32];    /* 通話相手の電話番号 */
33
34    int listen_sock;         /* 待ち受け用ソケット(TCP) */
35    int ctl_sock;            /* 制御パケット用ソケット(TCP) */
36    int data_sock;           /* 音声パケット用ソケット(UDP) */
37    int tmp_sock;            /* 待ち受け用ソケット for catch phone(TCP) */
38
39    int tty_fd;              /* コマンド入力FD */
40    int audio_fd;            /* オーディオデバイス FD  */
41
42  };
43
44  struct msg_header{
45    u_short msg_id;         /* message ID      */
46    u_short msg_len;        /* message_lendgth */
47    char my_number[32];     /* IP telephone number */
48  };
49
50  struct gkreq_msg{
51    char your_number[32];  /* IP telephone number */
52  };
53
54  struct gkrep_msg{
55    u_short rep_type;      /* reply type */
56    u_long  ipaddr;        /* client(or gk) ipaddr */
57    u_short listen_port;   /* client(or gk) listen_port */
58    u_short data_port;     /* client(or gk) data_port */
59    char your_number[32];  /* dialing telephone number */
60  };
61
62  struct gkupdate_msg{
63    u_short listen_port;   /* my listen_port */
64    u_short data_port;     /* my data_port */
65  };
66
67  struct gkupdaterep_msg{
68    u_short rep_type;      /* reply type */
69  };
```



```
70
71  struct Arecord{
72    unsigned long ipaddr;
73  };
74
75  static u_char* skip_dname(u_char* dname){
76    register u_char* ucp;
77    for(ucp = dname; *ucp && !(*ucp & INDIR_MASK); ucp += (*ucp)+1)
78      /* skip */;
79    if(*ucp & INDIR_MASK)
80      return ucp+2;
81    else
82      return ucp+1;
83  }
84
85
86
87
```

ipgk.c

```
 1  /* IP telephone gatekeeper 1999.*/
 2
 3  #include <unistd.h>
 4  #include <fcntl.h>
 5  #include <netinet/in.h>
 6  #include <netdb.h>
 7  #include <sys/types.h>
 8  #include <sys/socket.h>
 9  #include <sys/ioctl.h>
10  #include <sys/errno.h>
11  #include <sys/wait.h>
12  #include <stdlib.h>
13  #include <string.h>
14  #include <arpa/nameser.h>
15  #include <resolv.h>
16  #include <stdio.h>
17  #include "ipgk.h"
18
19  struct ipgk_status ipgk;  /* IP電話のパラメータ */
20  int errno;
21
22  int readn(int fd, char *ptr, int nbytes);
23  int writen(int fd, char *ptr, int nbytes);
24  void reaper();
25  void trans_sockaddr(struct sockaddr_in *net , u_long addr , u_short port);
26  void make_listen_sock();
27  int connect_sock(u_long ipaddr , u_short port);
28  void close_sock(int *sock);
29  void print_menu();
30  void send_msg(int msg_id);
31  int ask_dns(char *number ,  struct gkrep_msg *msg);
32  int update_dns(char *number , u_long ipaddr , struct gkupdate_msg *msg);
33  int gk_main(char *buf);
34  void cmd_line(char *buf);
35
36  int readn(int fd, char *ptr, int nbytes)
37  {
38          int nleft,n;
39          nleft = nbytes;
40          /*printf("nleft =%d\n",nleft);*/
41          while(nleft > 0){
42                  n = read(fd,ptr,nleft);
43                  /*printf("n =%d\n",n);*/
44                  if(n < 0){
45                    return(n);
```

```
46                }
47                else if(n == 0){
48                  break;
49                }
50                nleft -= n;
51                ptr += n;
52        }
53        return(nbytes - nleft);
54  }
55
56  int writen(int fd, char *ptr, int nbytes)
57  {
58        int nleft,nwritten;
59        nleft = nbytes;
60        while(nleft > 0){
61                nwritten = write(fd,ptr,nleft);
62                if (nwritten < 0){
63                  return (nwritten);
64                }
65                else if(nwritten == 0){
66                  break;
67                }
68                nleft -= nwritten;
69                ptr += nwritten;
70        }
71        return(nbytes - nleft);
72  }
73
74  /*-------------------------------------------------------------*/
75  /* void reaper()                                               */
76  /* ゾンビプロセスを消す                                        */
77  /*-------------------------------------------------------------*/
78  void reaper()
79  {
80    int status;
81    while (wait3(&status,WNOHANG,(struct rusage *)0) >= 0)
82      /* empty */;
83  }
84
85  /*-------------------------------------------------------------*/
86  /* char *convert(char *foo)                                    */
87  /* 電話番号を逆にしてドットで区切りサブドメインとしてDOMAINを  */
88  /* くっつける                                                  */
89  /*-------------------------------------------------------------*/
90  int convert(char *foo,u_char *out){
91    int len,i,j=0;
92
93    len=strlen(foo);
94    if (len<1) return -1;
95
96    /* まずは、ドットで区切る */
97    for (i=len-1;i>=0;i--){
98      out[j++]=foo[i];
99      out[j++]='.';
100   }
101   out[j++]=(char)NULL;
102
103   /* 先頭が999の場合はIP電話なので、TELDOMAINを最後に付ける */
104   if (foo[0]=='9' && foo[1]=='9' && foo[2]=='9' ) {
105     strcat(out,TELDOMAIN);
106   }
107   /* それ以外は、PSTN電話への電話なので、GWDOMAINを付ける */
108   else {
109     strcat(out,GWDOMAIN);
110   }
111   return 1;
```



```
112 }
113
114 /*-------------------------------------------------------------
115  * void trans_sockaddr()
116  *          与えられたIPアドレスとポートから、sockaddr_in構造体を返す
117  *          ipaddrはNetwork Byte Order.PortはHost byte orderとする
118  *-------------------------------------------------------------*/
119 void trans_sockaddr(struct sockaddr_in *net , u_long ipaddr , u_short port)
120 {
121   bzero(net,sizeof(*net));
122   net->sin_family = AF_INET;
123   net->sin_port = htons(port);
124   net->sin_addr.s_addr = ipaddr;
125 }
126
127 /*-------------------------------------------------------------
128  * void make_listen_sock()
129  *          ipgk.gk_portに待ち受けソケットを作る。
130  *-------------------------------------------------------------*/
131 void make_listen_sock(){
132   char hostname[41];
133   struct sockaddr_in net;
134   struct hostent *host;
135
136   /* make socket */
137   ipgk.listen_sock=socket(AF_INET,SOCK_STREAM,0);
138   if (ipgk.listen_sock < 0 ){
139     fprintf(stderr,"can't make sockets\n");
140     exit(-1);
141   }
142   /* bind listen_socket */
143   /* INADDR_ANYで待ち受けソケットをbindする */
144   trans_sockaddr(&net , INADDR_ANY , ipgk.gk_port);
145   printf("Listening %s port %d\n",inet_ntoa(net.sin_addr),ipgk.gk_port);
146   if ( bind(ipgk.listen_sock,(struct sockaddr *)&net , sizeof(net)) < 0 ) {
147     fprintf(stderr,"can't bind listen_sock\n");
148     exit(-1);
149   }
150   /* 最大1つの接続を待ち受け */
151   listen(ipgk.listen_sock,1);
152 }
153
154 void close_sock(int *sock)
155 {
156   close(*sock);
157   *sock=-1;
158 }
159
160 int gk_main(char *buf)
161 {
162   int n;
163   char content[32];      /* 汎用バッファ */
164   struct msg_header      *p = NULL;
165   struct gkreq_msg       *r = NULL;
166   struct gkrep_msg       *q = NULL;
167   struct gkupdate_msg    *s = NULL;
168   struct gkupdaterep_msg *t = NULL;
169
170   p = (struct msg_header *)buf;
171   printf("msg_id=%x from %s\n",ntohs(p->msg_id),p->my_number);
172   switch(ntohs(p->msg_id)){
173   case GKREQ_MSG:                 /* アドレス要求メッセージの場合 */
174     /* 要求メッセージを読む */
175     if ( (n=readn(ipgk.ctl_sock , content, ntohs(p->msg_len)) ) < 0 ){
176       fprintf(stderr,"cannot recv from client \n");
177       return -1;
```

```
178     }
179     /* メッセージバッファからGKREQ_MSG構造体へ */
180     r = (struct gkreq_msg *)content;
181     printf("recv GKREQ_MSG from %s\n",p->my_number);
182
183     /* リプライメッセージ用のメモリ確保 */
184     q = (struct gkrep_msg *)malloc(sizeof(*q));
185     /* DNSに聞く */
186     /* 対応するIPアドレスが見つからなかった場合 */
187     if ( ask_dns(r->your_number,q) == -1){
188       q->rep_type=htons(0x0002);
189       q->ipaddr=htonl(0);
190       q->listen_port=htons(0);
191       q->data_port  =htons(0);
192       bzero(q->your_number,sizeof(q->your_number));
193     }
194     /* 対応するIPアドレスが見つかった場合 */
195     else {
196       q->rep_type=htons(0x0001);
197     }
198
199     /* リプライメッセージ */
200     p->msg_id=htons(GKREP_MSG);
201     p->msg_len=htons(sizeof(*q));
202     strcpy(p->my_number,ipgk.my_number);
203
204     n=writen(ipgk.ctl_sock, (char *)p , sizeof(*p));
205     n=writen(ipgk.ctl_sock, (char *)q , sizeof(*q));
206     printf("send GKREP_MSG \n",p->my_number);
207     break;
208
209   case GKUPDATE_MSG:                        /* アドレス更新メッセージの場合 *
/
210     if ( ( n=readn(ipgk.ctl_sock , content, ntohs(p->msg_len)) ) < 0 ){
211       fprintf(stderr,"cannot recv from client \n");
212       return -1;
213     }
214     /* メッセージバッファからGKUPDATE_MSG構造体へ */
215     s = (struct gkupdate_msg *)content;
216     printf("recv GKUPDATE_MSG from %s\n",p->my_number);
217
218     /* リプライメッセージ用バッファを確保 */
219     t = (struct gkupdaterep_msg *)malloc(sizeof(*t));
220
221     /* my_numberに対応するArecord,TXTrecordを書き換える */
222     t->rep_type=htons(update_dns(p->my_number, ipgk.client_ipaddr , s));
223
224     /* リプライメッセージ */
225     p->msg_id=htons(GKUPDATEREP_MSG);
226     p->msg_len=htons(sizeof(*t));
227     strcpy(p->my_number,ipgk.my_number);
228     n=writen(ipgk.ctl_sock, (char *)p , sizeof(*p));
229     n=writen(ipgk.ctl_sock, (char *)t , sizeof(*t));
230     printf("send GKUPDATEREP_MSG \n",p->my_number);
231     break;
232   }
233   return 1;
234 }
235
236
237 int ask_dns(char *number , struct gkrep_msg *msg)
238 {
239   struct Arecord record;
240   HEADER* hdrp;
241   u_char msgbuf[1024];
242   u_char dname[256];
```



```
243   u_char dnamebuf[256];
244   register u_char* rp;
245   int i,j,len,flag;
246   u_short rtype, rclass, rdlen, preference;
247   u_long ttl;
248   char temp[6],isdn[32];
249   flag=1;
250
251   /* IP電話番号からドメイン名へ変換する */
252   if (convert(number,dname)!=1){
253     fprintf(stderr,"cannot convert domain dname\n");
254     return -1;
255   }
256   printf("  lookup (%s)\n",dname);
257   /* ISDNレコードが見つかる以外は、*/
258   /* そのままリクエストの番号をダイアル文字として返す*/
259   strcpy(msg->your_number,number);
260
261   if (number[0]=='9' && number[1]=='9' && number[2]=='9' ){
262     flag=0;
263     /* DNSにqueryを出し、ARecordを得る */
264     res_query(dname, C_IN, T_A, msgbuf, sizeof(msgbuf));
265     hdrp = (HEADER*)msgbuf;
266     rp = (u_char*)(hdrp+1);
267
268     /* DNSからの応答メッセージにエラーがないか調べる */
269     if(hdrp->tc || hdrp->rcode != NOERROR || ntohs(hdrp->qdcount) != 1){
270       fprintf(stderr,"cannot find A Record\n");
271       fprintf(stderr,"lookup ISDN Record\n");
272       /* ISDNレコードを調べにいく */
273       res_query(dname, C_IN, T_ISDN, msgbuf, sizeof(msgbuf));
274       hdrp = (HEADER*)msgbuf;
275       rp = (u_char*)(hdrp+1);
276
277       /* それでもなかったらやめる */
278       if(hdrp->tc || hdrp->rcode != NOERROR || ntohs(hdrp->qdcount) != 1){
279         fprintf(stderr,"tc=%d\n",hdrp->tc);
280         fprintf(stderr,"rcode=%d\n",hdrp->rcode);
281         fprintf(stderr,"qdcount=%d!!\n",hdrp->qdcount);
282         fprintf(stderr,"cannot lookup %s\n",dname);
283         return -1;
284       }
285       /* 応答メッセージのquestion部をスキップする */
286       rp = skip_dname(rp); /* skip QNAME */
287       rp+=4;               /* skip QTYPE and QCLASS */
288
289       /* answer部を読む */
290       rp = skip_dname(rp); /* skip NAME */
291       GETSHORT(rtype, rp); /* read rtype    */
292       GETSHORT(rclass, rp);/* read rclass   */
293       GETLONG(ttl, rp);    /* read ttl      */
294       GETSHORT(rdlen, rp); /* read rdlength(answer本体の長さ) */
295       rp++;                /* skip '\n' */
296       bzero(isdn,sizeof(isdn));
297       if(rtype == T_ISDN && rclass == C_IN){
298         bcopy(rp,&isdn,rdlen-1);
299       }
300       printf("  ISDN record = %s\n",isdn);
301
302       /* もしISDNレコードがあった場合は、返すダイアル文字を書き換える */
303       strcpy(msg->your_number,isdn);
304
305       /* MXレコードを調べるようにフラグをセット */
306       flag=1;
307       /* PSTN電話番号からドメイン名へ変換する */
308       if (convert(isdn,dname)!=1){
```

```
309         fprintf(stderr,"cannot convert domain dname\n");
310         return -1;
311       }
312       printf("  lookup (%s)\n",dname);
313     }
314     else {
315       /* 応答メッセージのquestion部をスキップする */
316       rp = skip_dname(rp); /* skip QNAME */
317       rp+=4;               /* skip QTYPE and QCLASS */
318
319       /* answer部を読む */
320       rp = skip_dname(rp); /* skip NAME */
321       GETSHORT(rtype, rp); /* read rtype    */
322       GETSHORT(rclass, rp);/* read rclass   */
323       GETLONG(ttl, rp);    /* read ttl      */
324       GETSHORT(rdlen, rp); /* read rdlength(answer本体の長さ) */
325
326       if(rtype == T_A && rclass == C_IN){
327         bcopy(rp,&msg->ipaddr,4);
328       }
329       printf("  A record = %s\n",inet_ntoa(msg->ipaddr));
330     }
331   }
332
333   /* GWのIPアドレスの場合 */
334   /* MXレコードを引く     */
335   if (flag==1){
336     res_query(dname, C_IN, T_MX, msgbuf, sizeof(msgbuf));
337     hdrp = (HEADER*)msgbuf;
338     rp = (u_char*)(hdrp+1);
339
340     /* DNSからの応答メッセージにエラーがないか調べる */
341     if(hdrp->tc || hdrp->rcode != NOERROR || ntohs(hdrp->qdcount) != 1){
342       fprintf(stderr,"tc=%d\n",hdrp->tc);
343       fprintf(stderr,"rcode=%d\n",hdrp->rcode);
344       fprintf(stderr,"qdcount=%d!!\n",hdrp->qdcount);
345       fprintf(stderr,"cannot lookup %s\n",dname);
346       return -1;
347     }
348
349     /* 応答メッセージのquestion部をスキップする */
350     rp = skip_dname(rp); /* skip QNAME */
351     rp+=4;               /* skip QTYPE and QCLASS */
352
353     /* answer部を読む */
354     rp = skip_dname(rp); /* skip NAME */
355     GETSHORT(rtype, rp); /* read rtype    */
356     GETSHORT(rclass, rp);/* read rclass   */
357     GETLONG(ttl, rp);    /* read ttl      */
358     GETSHORT(rdlen, rp); /* read rdlength(answer本体の長さ) */
359
360     GETSHORT(preference,rp);
361     len = dn_expand(msgbuf, msgbuf+sizeof(msgbuf), rp,
362                     dnamebuf, sizeof(dnamebuf));
363     rp += len;
364     /* GWのドメイン名をdnameにコピーする */
365     strcpy(dname, dnamebuf);
366     printf("  MX record = %s\n",dname);
367
368     /* skip NS record */
369     rp = skip_dname(rp);
370     rp += (2*2)+4;
371     GETSHORT(rdlen, rp);
372     rp+=rdlen;
373
374     /* lookup A record */
```



```
375     len = dn_expand(msgbuf, msgbuf+sizeof(msgbuf), rp,
376                     dnamebuf, sizeof(dnamebuf));
377     rp = skip_dname(rp);
378     GETSHORT(rtype, rp);
379     GETSHORT(rclass, rp);
380     GETLONG(ttl, rp);
381     GETSHORT(rdlen, rp);
382     if(rtype == T_A && rclass == C_IN){
383       bcopy(rp,&msg->ipaddr,4);
384     }
385     printf("  A record = %s\n",inet_ntoa(msg->ipaddr));
386   }
387
388   /* ポート番号の取得 */
389   /* DNSにqueryを出し、TXTRecordを得る */
390   res_query(dname, C_IN, T_TXT, msgbuf, sizeof(msgbuf));
391   hdrp = (HEADER*)msgbuf;
392   rp = (u_char*)(hdrp+1);
393
394   /* DNSからの応答メッセージにエラーがないか調べる */
395   if(hdrp->tc || hdrp->rcode != NOERROR || ntohs(hdrp->qdcount) != 1){
396     fprintf(stderr,"tc=%d\n",hdrp->tc);
397     fprintf(stderr,"rcode=%d\n",hdrp->rcode);
398     fprintf(stderr,"qdcount=%d!!\n",hdrp->qdcount);
399     return -1;
400   }
401   /* 応答メッセージのquestion部をスキップする */
402   rp = skip_dname(rp); /* skip QNAME */
403   rp+=4;               /* skip QTYPE and QCLASS */
404
405   /* answer部を読む */
406   rp = skip_dname(rp); /* skip NAME */
407   GETSHORT(rtype, rp); /* read rtype    */
408   GETSHORT(rclass, rp);/* read rclass   */
409   GETLONG(ttl, rp);    /* read ttl      */
410   GETSHORT(rdlen, rp); /* read rdlength(answer本体の長さ) */
411
412   rp++;                /* skip '\n' */
413
414   j=0;
415   /* TXTrecordから.とEで区切られた2つ数字(ポート番号)を取り出す */
416   if(rtype == T_TXT && rclass == C_IN){
417     for (i=0;i<rdlen;i++){
418       if ( *rp=='.' ) {
419         temp[j]=(char)NULL;
420         msg->listen_port=htons(atoi(temp));
421         rp++;
422         j=0;
423       }
424       else if (*rp=='E') {
425         temp[j]=(char)NULL;
426         msg->data_port=htons(atoi(temp));
427         rp++;
428         j=0;
429       }
430       temp[j++]=*rp;
431       rp++;
432     }
433   }
434   printf("  listen_port=%d data_port=%d\n",ntohs(msg->listen_port),ntohs(msg-
>data_port));
435 }
436
437 int update_dns(char *number , u_long ipaddr , struct gkupdate_msg *msg)
438 {
439   u_char dname[256];
```

```
   int pid,status;
   FILE *fp;
   char update_log[256];

   printf("%s addr=%s port=%d %d\n",number,inet_ntoa(ipaddr),
          ntohs(msg->listen_port),ntohs(msg->data_port));

   /* IP電話番号からドメイン名へ変換する */
   if (convert(number,dname)!=1){
     fprintf(stderr,"cannot convert domain dname\n");
     return -1;
   }
   printf("  update (%s)\n",dname);

   sprintf(update_log,"updatelog%s.log",number);

   /* scriptファイルのopen */
   if ((fp=fopen(update_log,"w"))==NULL){
     fprintf(stderr,"cannot open file\n");
     exit(-1);
   }

   /* scriptファイルの記述 */
   /* エントリーの削除 */
   fprintf(fp,"update: {delete} %s. IN A \n",dname);
   fprintf(fp,"update: {delete} %s. IN TXT \n",dname);
   /* クライアントの終了(listen_port=0)によるエントリー削除の場合は、deleteだ
けを行なう */
   /* そうでなければ、エントリーの追加を行なう */
   if (ntohs(msg->listen_port) != 0 ){
     fprintf(fp,"update: {add}    %s. %d IN A %s \n",dname,TTL,inet_ntoa(ipadd
r));
     fprintf(fp,"update: {add}    %s. %d IN TXT \"%d.%dE\n",
             dname,TTL,ntohs(msg->listen_port),ntohs(msg->data_port));
   }
   fprintf(fp,"  \n");
   fprintf(fp,"  \n");
   fclose(fp);

   if (( pid = fork()) < 0 ) {
     /* 正常にforkできなかった場合 */
     fprintf(stderr,"cannot fork(nsupdate)\n");
     return 0x0002;
   }

   /* nsupdateの実行 */
   if ( pid == 0 ){
     execlp("nsupdate","nsupdate",update_log,(char*)0);
   }
   /* nsupdateが終了するまで待機 */
   wait(&status);
   /* 正常復帰 */
   return 0x0001;
 }

 void print_menu()
 {
   fprintf(stderr,"%s",menu);
   fprintf(stderr,">");
 }

 void cmd_line(char *buf)
 {
   char ipaddr[256];
   char number[32];
   struct gkrep_msg *p = NULL;
```



```

   switch(buf[0]){
   case 'l':   /* lookup */
     p = (struct gkrep_msg *)malloc(sizeof(*p));
     strcpy(number,buf+1);
     ask_dns(number,p);
     break;
   case 'q':   /* quit   */
     close_sock(&ipgk.ctl_sock);
     close_sock(&ipgk.listen_sock);
     ipgk.tty_fd=-1;
     fprintf(stdout,"byebye..\n");
     break;
   case '?':   /* print status */
     fprintf(stderr,"listen_sock %d\n",ipgk.listen_sock);
     fprintf(stderr,"ctl_sock %d\n",ipgk.ctl_sock);
     break;
   default:
     print_menu();
     break;
   }

 }

 /*---------------------------------------
  * void main
  *
  *
  *---------------------------------------*/
 void main(int argc, char *argv[])
 {
   int i,n,m,fromlen,pid;
   char buf[1024];
   fd_set readfds,fds;
   struct sockaddr_in net;
   struct gkrep_msg msg;
   strcpy(ipgk.my_number,"I'm Gatekeeper");
   ipgk.gk_port=GKLISTENPORT;
   ipgk.tty_fd=0;
   ipgk.listen_sock=-1;
   ipgk.ctl_sock   =-1;

   /* 待ち受けソケットを作る */
   make_listen_sock();
   /* メニューを表示 */
   print_menu();

   while(!ipgk.tty_fd){
     fprintf(stderr,"\r>");
     n=0;
     FD_ZERO(&readfds);
     if (ipgk.tty_fd>=0){
       FD_SET(ipgk.tty_fd,&readfds);        /* 標準入力を監視対象 */
       if (ipgk.tty_fd > n ) n = ipgk.tty_fd;
     }
     if (ipgk.listen_sock>=0){
       FD_SET(ipgk.listen_sock,&readfds);   /* 待ち受けソケットを監視対象 */
       if (ipgk.listen_sock > n ) n = ipgk.listen_sock;
     }
     n++;

     m=select(n,&readfds,NULL,NULL,NULL);

     /* 標準入力の場合 */
     if (ipgk.tty_fd>=0 && FD_ISSET(ipgk.tty_fd,&readfds) ){
       scanf("%s",buf);
```

```
570       cmd_line(buf);
571     }
572
573     /* 待ち受けソケットの場合 */
574     if (ipgk.listen_sock >= 0 && FD_ISSET(ipgk.listen_sock,&readfds) ){
575       /* 並列処理のためforkする */
576       if (( pid = fork() ) <0 ){
577         fprintf(stderr,"cannot fork(gk_main)\n");
578         exit(1);
579       }
580       if ( pid == 0 ) {
581         /* 待ち受けソケットに来た通話をとりあえずacceptする */
582         fromlen=sizeof(net);
583         ipgk.ctl_sock=
584           accept(ipgk.listen_sock,(struct sockaddr *)&net,&fromlen);
585         if (ipgk.ctl_sock < 0) fprintf(stderr,"error:can't accept \n");
586         /* クライアントのIPアドレスを取得 */
587         ipgk.client_ipaddr=net.sin_addr.s_addr;
588         /* クライアントからのメッセージを取り込む */
589         i=readn(ipgk.ctl_sock,buf,sizeof(struct msg_header));
590         /* メッセージに従って処理を行なう */
591         gk_main(buf);
592         /* ソケットと閉じる */
593         close_sock(&ipgk.ctl_sock);
594         exit(1);
595       }
596
597     }
598     /* その時点でゾンビプロセスを消す */
599     reaper();
600   }
601   close_sock(&ipgk.ctl_sock);
602   close_sock(&ipgk.listen_sock);
603   sleep(3);
604   exit(1);
605 }
606
607
608
609
```