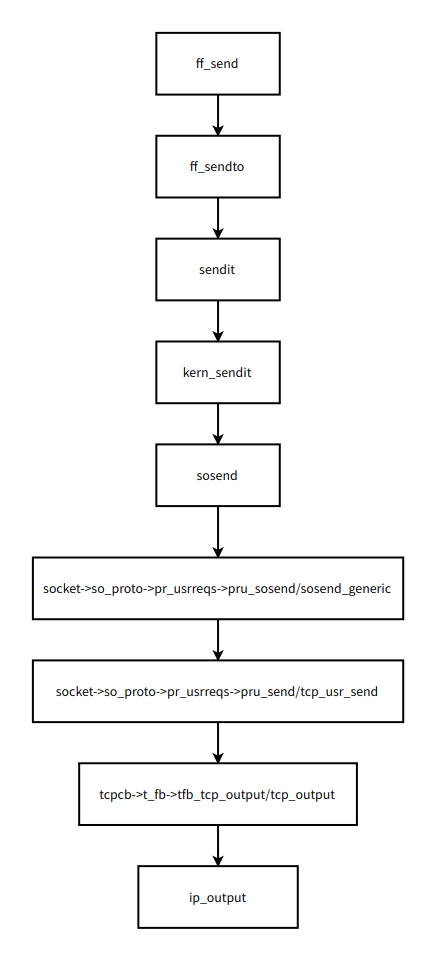
**ff\_send**

**发包流程**



**ff\_send**

|  |
| --- |
| C ssize\_t ff\_send(int s, const void \*buf, size\_t len, int flags) {  return (ff\_sendto(s, buf, len, flags, NULL, 0)); } |

**ff\_sendto**

|  |
| --- |
| C ssize\_t ff\_sendto(int s, const void \*buf, size\_t len, int flags,  const struct linux\_sockaddr \*to, socklen\_t tolen) {  struct msghdr msg;  struct iovec aiov;  int rc;   struct sockaddr\_storage bsdaddr;  struct sockaddr \*pf = (struct sockaddr \*)&bsdaddr;   if (to) {  linux2freebsd\_sockaddr(to, tolen, pf);  } else {  pf = NULL;  }   msg.msg\_name = pf;  msg.msg\_namelen = tolen;  msg.msg\_iov = &aiov;  msg.msg\_iovlen = 1;  msg.msg\_control = 0;  aiov.iov\_base = \_\_DECONST(void \*, buf);  aiov.iov\_len = len;  if ((rc = sendit(curthread, s, &msg, flags)))  goto kern\_fail;   rc = curthread->td\_retval[0];   return (rc); kern\_fail:  ff\_os\_errno(rc);  return (-1); } |

1.把socket信息和buf传入到msg(struct msghdr).

2.调用sendit 使用线程curthread 发送msg信息

**sendit**

|  |
| --- |
| C int sendit(struct thread \*td, int s, struct msghdr \*mp, int flags) {  struct mbuf \*control;  struct sockaddr \*to;  int error;  #ifdef CAPABILITY\_MODE  if (IN\_CAPABILITY\_MODE(td) && (mp->msg\_name != NULL))  return (ECAPMODE); #endif   if (mp->msg\_name != NULL) {  error = getsockaddr(&to, mp->msg\_name, mp->msg\_namelen);  if (error != 0) {  to = NULL;  goto bad;  }  mp->msg\_name = to;  } else {  to = NULL;  }   if (mp->msg\_control) {  if (mp->msg\_controllen < sizeof(struct cmsghdr) #ifdef COMPAT\_OLDSOCK  && (mp->msg\_flags != MSG\_COMPAT ||  !SV\_PROC\_FLAG(td->td\_proc, SV\_AOUT)) #endif  ) {  error = EINVAL;  goto bad;  }  error = sockargs(&control, mp->msg\_control,  mp->msg\_controllen, MT\_CONTROL);  if (error != 0)  goto bad; #ifdef COMPAT\_OLDSOCK  if (mp->msg\_flags == MSG\_COMPAT &&  SV\_PROC\_FLAG(td->td\_proc, SV\_AOUT)) {  struct cmsghdr \*cm;   M\_PREPEND(control, sizeof(\*cm), M\_WAITOK);  cm = mtod(control, struct cmsghdr \*);  cm->cmsg\_len = control->m\_len;  cm->cmsg\_level = SOL\_SOCKET;  cm->cmsg\_type = SCM\_RIGHTS;  } #endif  } else {  control = NULL;  }   error = kern\_sendit(td, s, mp, flags, control, UIO\_USERSPACE);  bad:  free(to, M\_SONAME);  return (error); } |

1.CAPABILITY\_MODE安全机制及套接字检查，结果传入套接字变量to

2.辅助数据msg\_control检查

3.调用kern\_sendit发送数据

**kern\_sendit**

|  |
| --- |
| C int kern\_sendit(struct thread \*td, int s, struct msghdr \*mp, int flags,  struct mbuf \*control, enum uio\_seg segflg) {  struct file \*fp;  struct uio auio;  struct iovec \*iov;  struct socket \*so;  cap\_rights\_t \*rights; #ifdef KTRACE  struct uio \*ktruio = NULL; #endif  ssize\_t len;  int i, error;   AUDIT\_ARG\_FD(s);  rights = &cap\_send\_rights;  if (mp->msg\_name != NULL) {  AUDIT\_ARG\_SOCKADDR(td, AT\_FDCWD, mp->msg\_name);  rights = &cap\_send\_connect\_rights;  }  error = getsock\_cap(td, s, rights, &fp, NULL, NULL);  if (error != 0) {  m\_freem(control);  return (error);  }  so = (struct socket \*)fp->f\_data;  #ifdef KTRACE  if (mp->msg\_name != NULL && KTRPOINT(td, KTR\_STRUCT))  ktrsockaddr(mp->msg\_name); #endif #ifdef MAC  if (mp->msg\_name != NULL) {  error = mac\_socket\_check\_connect(td->td\_ucred, so,  mp->msg\_name);  if (error != 0) {  m\_freem(control);  goto bad;  }  }  error = mac\_socket\_check\_send(td->td\_ucred, so);  if (error != 0) {  m\_freem(control);  goto bad;  } #endif   auio.uio\_iov = mp->msg\_iov;  auio.uio\_iovcnt = mp->msg\_iovlen;  auio.uio\_segflg = segflg;  auio.uio\_rw = UIO\_WRITE;  auio.uio\_td = td;  auio.uio\_offset = 0; /\* XXX \*/  auio.uio\_resid = 0;  iov = mp->msg\_iov;  for (i = 0; i < mp->msg\_iovlen; i++, iov++) {  if ((auio.uio\_resid += iov->iov\_len) < 0) {  error = EINVAL;  m\_freem(control);  goto bad;  }  } #ifdef KTRACE  if (KTRPOINT(td, KTR\_GENIO))  ktruio = cloneuio(&auio); #endif  len = auio.uio\_resid;  error = sosend(so, mp->msg\_name, &auio, 0, control, flags, td);  if (error != 0) {  if (auio.uio\_resid != len && (error == ERESTART ||  error == EINTR || error == EWOULDBLOCK))  error = 0;  /\* Generation of SIGPIPE can be controlled per socket \*/  if (error == EPIPE && !(so->so\_options & SO\_NOSIGPIPE) &&  !(flags & MSG\_NOSIGNAL)) {  PROC\_LOCK(td->td\_proc);  tdsignal(td, SIGPIPE);  PROC\_UNLOCK(td->td\_proc);  }  }  if (error == 0)  td->td\_retval[0] = len - auio.uio\_resid; #ifdef KTRACE  if (ktruio != NULL) {  ktruio->uio\_resid = td->td\_retval[0];  ktrgenio(s, UIO\_WRITE, ktruio, error);  } #endif bad:  fdrop(fp, td);  return (error); } |

1.定义权限指针right,通过 getsock\_cap() 函数获取套接字 s 的权限，并将权限保存到 rights 指向的指针所指向的变量中。

2.msg检查，并将相关的数据传入到auio(struct uio )中

3.调用sosend发送auio消息

**sosend**

|  |
| --- |
| C int sosend(struct socket \*so, struct sockaddr \*addr, struct uio \*uio,  struct mbuf \*top, struct mbuf \*control, int flags, struct thread \*td) {  int error;   CURVNET\_SET(so->so\_vnet);  if (!SOLISTENING(so))  error = so->so\_proto->pr\_usrreqs->pru\_sosend(so, addr, uio,  top, control, flags, td);  else {  m\_freem(top);  m\_freem(control);  error = ENOTCONN;  }  CURVNET\_RESTORE();  return (error); } |

1.网络环境设置，CURVNET\_SET() 宏将当前线程的虚拟网络环境设置为套接字 so 所在的网络环境。

2.判断套接字是否处于监听状态。如果不是，则调用协议栈中的 pru\_sosend() 函数进行实际的发送操作，并将返回值保存到 error 变量中。其中，addr 表示目标地址，uio 表示发送数据的缓冲区，top 表示发送的数据包，control 表示消息控制数据，flags 表示发送标志，td 表示当前线程的进程描述符。

如果套接字处于监听状态，则释放待发送的数据包 top 和消息控制数据 control，并将错误码设置为 ENOTCONN，表示套接字未连接。

3. CURVNET\_RESTORE() 宏将当前线程的虚拟网络环境恢复为之前的状态，并返回错误码。

**DEFAULT(pu->pru\_sosend, sosend\_generic);**

|  |
| --- |
| C int sosend\_generic(struct socket \*so, struct sockaddr \*addr, struct uio \*uio,  struct mbuf \*top, struct mbuf \*control, int flags, struct thread \*td) {  long space;  ssize\_t resid;  int clen = 0, error, dontroute;  int atomic = sosendallatonce(so) || top;  int pru\_flag; #ifdef KERN\_TLS  struct ktls\_session \*tls;  int tls\_enq\_cnt, tls\_pruflag;  uint8\_t tls\_rtype;   tls = NULL;  tls\_rtype = TLS\_RLTYPE\_APP; #endif  if (uio != NULL)  resid = uio->uio\_resid;  else if ((top->m\_flags & M\_PKTHDR) != 0)  resid = top->m\_pkthdr.len;  else  resid = m\_length(top, NULL);  /\*  \* In theory resid should be unsigned. However, space must be  \* signed, as it might be less than 0 if we over-committed, and we  \* must use a signed comparison of space and resid. On the other  \* hand, a negative resid causes us to loop sending 0-length  \* segments to the protocol.  \*  \* Also check to make sure that MSG\_EOR isn't used on SOCK\_STREAM  \* type sockets since that's an error.  \*/  if (resid < 0 || (so->so\_type == SOCK\_STREAM && (flags & MSG\_EOR))) {  error = EINVAL;  goto out;  }   dontroute =   (so->so\_proto->pr\_flags & PR\_ATOMIC);  if (td != NULL)  td->td\_ru.ru\_msgsnd++;  if (control != NULL)  clen = control->m\_len;   error = sblock(&so->so\_snd, SBLOCKWAIT(flags));  if (error)  goto out;  #ifdef KERN\_TLS  tls\_pruflag = 0;  tls = ktls\_hold(so->so\_snd.sb\_tls\_info);  if (tls != NULL) {  if (tls->mode == TCP\_TLS\_MODE\_SW)  tls\_pruflag = PRUS\_NOTREADY;   if (control != NULL) {  struct cmsghdr \*cm = mtod(control, struct cmsghdr \*);   if (clen >= sizeof(\*cm) &&  cm->cmsg\_type == TLS\_SET\_RECORD\_TYPE) {  tls\_rtype = \*((uint8\_t \*)CMSG\_DATA(cm));  clen = 0;  m\_freem(control);  control = NULL;  atomic = 1;  }  }  } #endif  restart:  do {  SOCKBUF\_LOCK(&so->so\_snd);  if (so->so\_snd.sb\_state & SBS\_CANTSENDMORE) {  SOCKBUF\_UNLOCK(&so->so\_snd);  error = EPIPE;  goto release;  }  if (so->so\_error) {  error = so->so\_error;  so->so\_error = 0;  SOCKBUF\_UNLOCK(&so->so\_snd);  goto release;  }  if ((so->so\_state & SS\_ISCONNECTED) == 0) {  /\*  \* `sendto' and `sendmsg' is allowed on a connection-  \* based socket if it supports implied connect.  \* Return ENOTCONN if not connected and no address is  \* supplied.  \*/  if ((so->so\_proto->pr\_flags & PR\_CONNREQUIRED) &&  (so->so\_proto->pr\_flags & PR\_IMPLOPCL) == 0) {  if ((so->so\_state & SS\_ISCONFIRMING) == 0 &&  !(resid == 0 && clen != 0)) {  SOCKBUF\_UNLOCK(&so->so\_snd);  error = ENOTCONN;  goto release;  }  } else if (addr == NULL) {  SOCKBUF\_UNLOCK(&so->so\_snd);  if (so->so\_proto->pr\_flags & PR\_CONNREQUIRED)  error = ENOTCONN;  else  error = EDESTADDRREQ;  goto release;  }  }  space = sbspace(&so->so\_snd);  if (flags & MSG\_OOB)  space += 1024;  if ((atomic && resid > so->so\_snd.sb\_hiwat) ||  clen > so->so\_snd.sb\_hiwat) {  SOCKBUF\_UNLOCK(&so->so\_snd);  error = EMSGSIZE;  goto release;  }  if (space < resid + clen &&  (atomic || space < so->so\_snd.sb\_lowat || space < clen)) {  if ((so->so\_state & SS\_NBIO) ||  (flags & (MSG\_NBIO | MSG\_DONTWAIT)) != 0) {  SOCKBUF\_UNLOCK(&so->so\_snd);  error = EWOULDBLOCK;  goto release;  }  error = sbwait(&so->so\_snd);  SOCKBUF\_UNLOCK(&so->so\_snd);  if (error)  goto release;  goto restart;  }  SOCKBUF\_UNLOCK(&so->so\_snd);  space -= clen;  do {  if (uio == NULL) {  resid = 0;  if (flags & MSG\_EOR)  top->m\_flags |= M\_EOR; #ifdef KERN\_TLS  if (tls != NULL) {  ktls\_frame(top, tls, &tls\_enq\_cnt,  tls\_rtype);  tls\_rtype = TLS\_RLTYPE\_APP;  } #endif  } else {  /\*  \* Copy the data from userland into a mbuf  \* chain. If resid is 0, which can happen  \* only if we have control to send, then  \* a single empty mbuf is returned. This  \* is a workaround to prevent protocol send  \* methods to panic.  \*/ #ifdef KERN\_TLS  if (tls != NULL) {  top = m\_uiotombuf(uio, M\_WAITOK, space,  tls->params.max\_frame\_len,  M\_EXTPG |  ((flags & MSG\_EOR) ? M\_EOR : 0));  if (top != NULL) {  ktls\_frame(top, tls,  &tls\_enq\_cnt, tls\_rtype);  }  tls\_rtype = TLS\_RLTYPE\_APP;  } else #endif  top = m\_uiotombuf(uio, M\_WAITOK, space,  (atomic ? max\_hdr : 0),  (atomic ? M\_PKTHDR : 0) |  ((flags & MSG\_EOR) ? M\_EOR : 0));  if (top == NULL) {  error = EFAULT; /\* only possible error \*/  goto release;  }  space -= resid - uio->uio\_resid;  resid = uio->uio\_resid;  }  if (dontroute) {  SOCK\_LOCK(so);  so->so\_options |= SO\_DONTROUTE;  SOCK\_UNLOCK(so);  }  /\*  \* XXX all the SBS\_CANTSENDMORE checks previously  \* done could be out of date. We could have received  \* a reset packet in an interrupt or maybe we slept  \* while doing page faults in uiomove() etc. We  \* could probably recheck again inside the locking  \* protection here, but there are probably other  \* places that this also happens. We must rethink  \* this.  \*/  VNET\_SO\_ASSERT(so);   pru\_flag = (flags & MSG\_OOB) ? PRUS\_OOB :  /\*  \* If the user set MSG\_EOF, the protocol understands  \* this flag and nothing left to send then use  \* PRU\_SEND\_EOF instead of PRU\_SEND.  \*/  ((flags & MSG\_EOF) &&  (so->so\_proto->pr\_flags & PR\_IMPLOPCL) &&  (resid <= 0)) ?  PRUS\_EOF :  /\* If there is more to send set PRUS\_MORETOCOME. \*/  (flags & MSG\_MORETOCOME) ||  (resid > 0 && space > 0) ? PRUS\_MORETOCOME : 0;  #ifdef KERN\_TLS  pru\_flag |= tls\_pruflag; #endif   error = (\*so->so\_proto->pr\_usrreqs->pru\_send)(so,  pru\_flag, top, addr, control, td);   if (dontroute) {  SOCK\_LOCK(so);  so->so\_options &= ~SO\_DONTROUTE;  SOCK\_UNLOCK(so);  }  #ifdef KERN\_TLS  if (tls != NULL && tls->mode == TCP\_TLS\_MODE\_SW) {  /\*  \* Note that error is intentionally  \* ignored.  \*  \* Like sendfile(), we rely on the  \* completion routine (pru\_ready())  \* to free the mbufs in the event that  \* pru\_send() encountered an error and  \* did not append them to the sockbuf.  \*/  soref(so);  ktls\_enqueue(top, so, tls\_enq\_cnt);  } #endif  clen = 0;  control = NULL;  top = NULL;  if (error)  goto release;  } while (resid && space > 0);  } while (resid);  release:  sbunlock(&so->so\_snd); out: #ifdef KERN\_TLS  if (tls != NULL)  ktls\_free(tls); #endif  if (top != NULL)  m\_freem(top);  if (control != NULL)  m\_freem(control);  return (error); } |

1.判断 uio 是否为 NULL，以及数据包 top 是否设置了 M\_PKTHDR 标志，来计算剩余发送数据的长度 resid。如果没有设置 M\_PKTHDR 标志，则通过 m\_length() 函数计算数据包长度。

2.对 resid 进行检查，如果其小于 0 或者 flags 中设置了 MSG\_EOR，并且套接字类型为 SOCK\_STREAM，则将错误码设置为 EINVAL 并跳转到 out 标签处进行错误处理。

3.判断是否需要启用路由绕过选项（MSG\_DONTROUTE 和 SO\_DONTROUTE），并将结果保存到 dontroute 变量中。如果当前线程的进程描述符 td 不为 NULL，则将 ru\_msgsnd 计数器加一。

如果需要传递消息控制数据，则计算消息控制数据的长度 clen。

4.调用 sblock() 函数对套接字的发送缓冲区进行加锁，等待发送空间可用。如果加锁失败，则将错误码设置为相应的值，并跳转到 out 标签处进行错误处理。

5.循环发送

5.1调用 SOCKBUF\_LOCK() 函数对套接字的发送缓冲区进行加锁，然后检查发送缓冲区状态和套接字错误状态。如果发送缓冲区已经被关闭，则将错误码设置为 EPIPE 并跳转到 release 标签处进行释放操作。

5.2如果套接字未连接，且所需的协议需要连接，但是未提供目标地址，则将错误码设置为 ENOTCONN 或 EDESTADDRREQ，具体取决于协议是否需要目标地址。如果套接字已经连接，则检查发送缓冲区是否有足够的空间来容纳传输的数据和控制信息。如果发送缓冲区的空间不足，则根据套接字的阻塞标志和非阻塞标志进行等待或者返回 EWOULDBLOCK。如果发送缓冲区的空间足够，就释放套接字的发送缓冲区锁。

5.3将剩余的数据长度 resid 减去发送的数据长度，并将控制信息长度 clen 设置为 0。如果 uio 不为 NULL，将数据从用户空间复制到一个 mbuf 链中。如果 uio 为 NULL，则设置 resid 为 0，并将数据包 top 的 M\_EOR 标志设置为 MSG\_EOR 标志的值。如果启用了 TLS，则调用 ktls\_frame() 函数对 top 进行加密处理。如果启用了路由绕过选项，则将 SO\_DONTROUTE 选项设置为 1。然后调用协议的 pru\_send() 函数将数据发送出去，并将返回值保存在 error 变量中。最后释放控制信息和数据包。

**pru\_send()/tcp\_usr\_send**

|  |
| --- |
| C tatic int tcp\_usr\_send(struct socket \*so, int flags, struct mbuf \*m,  struct sockaddr \*nam, struct mbuf \*control, struct thread \*td) {  struct epoch\_tracker et;  int error = 0;  struct inpcb \*inp = NULL;  struct tcpcb \*tp = NULL; #ifdef INET #ifdef INET6  struct sockaddr\_in sin; #endif  struct sockaddr\_in \*sinp = NULL; #endif #ifdef INET6  int isipv6 = 0; #endif  u\_int8\_t incflagsav;  u\_char vflagsav;  bool restoreflags;  TCPDEBUG0;   /\*  \* We require the pcbinfo "read lock" if we will close the socket  \* as part of this call.  \*/  NET\_EPOCH\_ENTER(et);  inp = sotoinpcb(so);  KASSERT(inp != NULL, ("tcp\_usr\_send: inp == NULL"));  INP\_WLOCK(inp);  vflagsav = inp->inp\_vflag;  incflagsav = inp->inp\_inc.inc\_flags;  restoreflags = false;  if (inp->inp\_flags & (INP\_TIMEWAIT | INP\_DROPPED)) {  if (control)  m\_freem(control);  /\*  \* In case of PRUS\_NOTREADY, tcp\_usr\_ready() is responsible  \* for freeing memory.  \*/  if (m && (flags & PRUS\_NOTREADY) == 0)  m\_freem(m);  error = ECONNRESET;  goto out;  }  tp = intotcpcb(inp);  if (flags & PRUS\_OOB) {  if ((error = tcp\_pru\_options\_support(tp, PRUS\_OOB)) != 0) {  if (control)  m\_freem(control);  if (m && (flags & PRUS\_NOTREADY) == 0)  m\_freem(m);  goto out;  }  }  TCPDEBUG1();  if (nam != NULL && tp->t\_state < TCPS\_SYN\_SENT) {  switch (nam->sa\_family) { #ifdef INET  case AF\_INET:  sinp = (struct sockaddr\_in \*)nam;  if (sinp->sin\_len != sizeof(struct sockaddr\_in)) {  if (m)  m\_freem(m);  error = EINVAL;  goto out;  }  if ((inp->inp\_vflag & INP\_IPV6) != 0) {  if (m)  m\_freem(m);  error = EAFNOSUPPORT;  goto out;  }  if (IN\_MULTICAST(ntohl(sinp->sin\_addr.s\_addr))) {  if (m)  m\_freem(m);  error = EAFNOSUPPORT;  goto out;  }  if (ntohl(sinp->sin\_addr.s\_addr) == INADDR\_BROADCAST) {  if (m)  m\_freem(m);  error = EACCES;  goto out;  }  if ((error = prison\_remote\_ip4(td->td\_ucred,  &sinp->sin\_addr))) {  if (m)  m\_freem(m);  goto out;  } #ifdef INET6  isipv6 = 0; #endif  break; #endif /\* INET \*/ #ifdef INET6  case AF\_INET6:  {  struct sockaddr\_in6 \*sin6;   sin6 = (struct sockaddr\_in6 \*)nam;  if (sin6->sin6\_len != sizeof(\*sin6)) {  if (m)  m\_freem(m);  error = EINVAL;  goto out;  }  if ((inp->inp\_vflag & INP\_IPV6PROTO) == 0) {  if (m != NULL)  m\_freem(m);  error = EAFNOSUPPORT;  goto out;  }  if (IN6\_IS\_ADDR\_MULTICAST(&sin6->sin6\_addr)) {  if (m)  m\_freem(m);  error = EAFNOSUPPORT;  goto out;  }  if (IN6\_IS\_ADDR\_V4MAPPED(&sin6->sin6\_addr)) { #ifdef INET  if ((inp->inp\_flags & IN6P\_IPV6\_V6ONLY) != 0) {  error = EINVAL;  if (m)  m\_freem(m);  goto out;  }  if ((inp->inp\_vflag & INP\_IPV4) == 0) {  error = EAFNOSUPPORT;  if (m)  m\_freem(m);  goto out;  }  restoreflags = true;  inp->inp\_vflag &= ~INP\_IPV6;  sinp = &sin;  in6\_sin6\_2\_sin(sinp, sin6);  if (IN\_MULTICAST(  ntohl(sinp->sin\_addr.s\_addr))) {  error = EAFNOSUPPORT;  if (m)  m\_freem(m);  goto out;  }  if ((error = prison\_remote\_ip4(td->td\_ucred,  &sinp->sin\_addr))) {  if (m)  m\_freem(m);  goto out;  }  isipv6 = 0; #else /\* !INET \*/  error = EAFNOSUPPORT;  if (m)  m\_freem(m);  goto out; #endif /\* INET \*/  } else {  if ((inp->inp\_vflag & INP\_IPV6) == 0) {  if (m)  m\_freem(m);  error = EAFNOSUPPORT;  goto out;  }  restoreflags = true;  inp->inp\_vflag &= ~INP\_IPV4;  inp->inp\_inc.inc\_flags |= INC\_ISIPV6;  if ((error = prison\_remote\_ip6(td->td\_ucred,  &sin6->sin6\_addr))) {  if (m)  m\_freem(m);  goto out;  }  isipv6 = 1;  }  break;  } #endif /\* INET6 \*/  default:  if (m)  m\_freem(m);  error = EAFNOSUPPORT;  goto out;  }  }  if (control) {  /\* TCP doesn't do control messages (rights, creds, etc) \*/  if (control->m\_len) {  m\_freem(control);  if (m)  m\_freem(m);  error = EINVAL;  goto out;  }  m\_freem(control); /\* empty control, just free it \*/  }  if (!(flags & PRUS\_OOB)) {  sbappendstream(&so->so\_snd, m, flags);  if (nam && tp->t\_state < TCPS\_SYN\_SENT) {  /\*  \* Do implied connect if not yet connected,  \* initialize window to default value, and  \* initialize maxseg using peer's cached MSS.  \*/ #ifdef INET6  if (isipv6)  error = tcp6\_connect(tp, nam, td); #endif /\* INET6 \*/ #if defined(INET6) && defined(INET)  else #endif #ifdef INET  error = tcp\_connect(tp,  (struct sockaddr \*)sinp, td); #endif  /\*  \* The bind operation in tcp\_connect succeeded. We  \* no longer want to restore the flags if later  \* operations fail.  \*/  if (error == 0 || inp->inp\_lport != 0)  restoreflags = false;   if (error)  goto out;  if (IS\_FASTOPEN(tp->t\_flags))  tcp\_fastopen\_connect(tp);  else {  tp->snd\_wnd = TTCP\_CLIENT\_SND\_WND;  tcp\_mss(tp, -1);  }  }  if (flags & PRUS\_EOF) {  /\*  \* Close the send side of the connection after  \* the data is sent.  \*/  socantsendmore(so);  tcp\_usrclosed(tp);  }  if (TCPS\_HAVEESTABLISHED(tp->t\_state) &&  ((tp->t\_flags2 & TF2\_FBYTES\_COMPLETE) == 0) &&  (tp->t\_fbyte\_out == 0) &&  (so->so\_snd.sb\_ccc > 0)) {  tp->t\_fbyte\_out = ticks;  if (tp->t\_fbyte\_out == 0)  tp->t\_fbyte\_out = 1;  if (tp->t\_fbyte\_out && tp->t\_fbyte\_in)  tp->t\_flags2 |= TF2\_FBYTES\_COMPLETE;  }  if (!(inp->inp\_flags & INP\_DROPPED) &&  !(flags & PRUS\_NOTREADY)) {  if (flags & PRUS\_MORETOCOME)  tp->t\_flags |= TF\_MORETOCOME;  error = tp->t\_fb->tfb\_tcp\_output(tp);  if (flags & PRUS\_MORETOCOME)  tp->t\_flags &= ~TF\_MORETOCOME;  }  } else {  /\*  \* XXXRW: PRUS\_EOF not implemented with PRUS\_OOB?  \*/  SOCKBUF\_LOCK(&so->so\_snd);  if (sbspace(&so->so\_snd) < -512) {  SOCKBUF\_UNLOCK(&so->so\_snd);  m\_freem(m);  error = ENOBUFS;  goto out;  }  /\*  \* According to RFC961 (Assigned Protocols),  \* the urgent pointer points to the last octet  \* of urgent data. We continue, however,  \* to consider it to indicate the first octet  \* of data past the urgent section.  \* Otherwise, snd\_up should be one lower.  \*/  sbappendstream\_locked(&so->so\_snd, m, flags);  SOCKBUF\_UNLOCK(&so->so\_snd);  if (nam && tp->t\_state < TCPS\_SYN\_SENT) {  /\*  \* Do implied connect if not yet connected,  \* initialize window to default value, and  \* initialize maxseg using peer's cached MSS.  \*/   /\*  \* Not going to contemplate SYN|URG  \*/  if (IS\_FASTOPEN(tp->t\_flags))  tp->t\_flags &= ~TF\_FASTOPEN; #ifdef INET6  if (isipv6)  error = tcp6\_connect(tp, nam, td); #endif /\* INET6 \*/ #if defined(INET6) && defined(INET)  else #endif #ifdef INET  error = tcp\_connect(tp,  (struct sockaddr \*)sinp, td); #endif  /\*  \* The bind operation in tcp\_connect succeeded. We  \* no longer want to restore the flags if later  \* operations fail.  \*/  if (error == 0 || inp->inp\_lport != 0)  restoreflags = false;   if (error)  goto out;  tp->snd\_wnd = TTCP\_CLIENT\_SND\_WND;  tcp\_mss(tp, -1);  }  tp->snd\_up = tp->snd\_una + sbavail(&so->so\_snd);  if (!(flags & PRUS\_NOTREADY)) {  tp->t\_flags |= TF\_FORCEDATA;  error = tp->t\_fb->tfb\_tcp\_output(tp);  tp->t\_flags &= ~TF\_FORCEDATA;  }  }  TCP\_LOG\_EVENT(tp, NULL,  &inp->inp\_socket->so\_rcv,  &inp->inp\_socket->so\_snd,  TCP\_LOG\_USERSEND, error,  0, NULL, false); out:  /\*  \* If the request was unsuccessful and we changed flags,  \* restore the original flags.  \*/  if (error != 0 && restoreflags) {  inp->inp\_vflag = vflagsav;  inp->inp\_inc.inc\_flags = incflagsav;  }  TCPDEBUG2((flags & PRUS\_OOB) ? PRU\_SENDOOB :  ((flags & PRUS\_EOF) ? PRU\_SEND\_EOF : PRU\_SEND));  TCP\_PROBE2(debug\_\_user, tp, (flags & PRUS\_OOB) ? PRU\_SENDOOB :  ((flags & PRUS\_EOF) ? PRU\_SEND\_EOF : PRU\_SEND));  INP\_WUNLOCK(inp);  NET\_EPOCH\_EXIT(et);  return (error); } |

1. 变量初始化【struct epoch\_tracker 是 FreeBSD 操作系统中用于跟踪时序保护期的结构体。它用于在进入和离开时序保护期时跟踪上下文，并确保在进入时序保护期时禁止抢占。】
2. 【21-56】调用 NET\_EPOCH\_ENTER() 函数进入网络时序保护期，然后通过 sotoinpcb() 函数将套接字转换为与之关联的 struct inpcb，并通过 INP\_WLOCK() 函数对该结构进行写锁定。将 struct inpcb 转换为 struct tcpcb
3. 【57-96】IPV4地址合法检查，【98-178】IPV6地址合法检查
4. 控制信息检查，调用 sbappendstream() 函数将数据包 m 添加到套接字 so 的发送缓冲区中，并根据连接状态执行相应的初始化操作。调用套接字的底层输出函数 tfb\_tcp\_output() 发送数据
5. 处理带外数据（即 PRUS\_OOB 标志被设置）的情况

**tfb\_tcp\_output()/tcp\_output**

1. sendwin = min(tp->snd\_wnd, tp->snd\_cwnd);
2. 窗口更新【如果接收窗口大于 0，并且当前连接没有需要发送的 SYN 报文段，没有需要发送的 DELACK 报文段，并且 TCP 连接没有接收到 FIN 报文段，则进行窗口更新操作。首先计算可以增加的接收窗口大小 adv，该值考虑了窗口大小的缩放因子 tp->rcv\_scale。如果当前窗口大小已经是最大值 TCP\_MAXWIN，则不需要更新窗口。然后根据 adv 的大小进行判断，如果 adv 大于等于两个 MSS，并且 adv 大于等于接收缓冲区大小的 1/4 或者小于接收缓冲区大小的 1/8，或者接收缓冲区大小小于等于 8 个 MSS，或者 adv 大于等于 TCP 最大窗口大小，则需要发送窗口更新报文段。如果 adv 大于等于接收缓冲区大小的一半，则也需要发送窗口更新报文段。如果不需要发送窗口更新报文段，则跳转到 dontupdate 标签，否则跳转到 send 标签继续发送窗口更新报文段。】