Ruby master - Bug #3540

IO.copy_stream fails to detect client disconnect w/sendfile

07/06/2010 05:12 AM - normalperson (Eric Wong)

Status: Closed
Priority: Normal
Assignee:
Target version: 2.0.0
Backport:

ruby -v:
ruby 1.9.3dev (2010-07-04 trunk 28540)
x86_64-linux

Description

sendfile() may return with a short write upon a client disconnect. Instead of retrying and getting an error, Ruby tries to force a select() on the descriptor which fails to detect the disconnect. This causes IO.copy_stream to hang, (possibly until TCP keepalives kick in). IO.copy_stream should raise immediately.

Attached are:

- a patch to fix the issue
- a script that reproduces the issue with sendfile (under Linux 2.6.34)

History

#1 - 07/06/2010 12:39 PM - akr (Akira Tanaka)

sendfile() may return with a short write upon a client disconnect. Instead of retrying and getting an error, Ruby tries to force a select() on the descriptor which fails to detect the disconnect. This causes IO.copy_stream to hang, (possibly until TCP keepalives kick in). IO.copy_stream should raise immediately.

Thank you for the reproducible script and fix.

I'll commit your fix.

However I think the Linux select behavior which doesn't notify writability on disconnected TCP socket is suspicious.

```
linux% ruby -rsocket -e 'serv = TCPServer.open("127.0.0.1", 8888)
s1 = TCPSocket.open("127.0.0.1", 8888)
s2 = serv.accept
s2.close
s1.write "a" rescue p $!
s1.write "a" rescue p $!
p IO.select(nil, [s1], nil, 0)
'
# nil
```

FreeBSD and Solaris notify writability.

```
freebsd% ruby -rsocket -e 'serv = TCPServer.open("127.0.0.1", 8888)
s1 = TCPSocket.open("127.0.0.1", 8888)
s2 = serv.accept
s2.close
s1.write "a" rescue p $!
s1.write "a" rescue p $!
p IO.select(nil, [s1], nil, 0)
'
```

08/07/2021
I think select should notify writability when write would not block. Clearly write doesn't block on disconnected socket.

Linux also notify writability for UNIX domain socket pair.

```ruby
# I tested Linux 2.6.26.
--
Tanaka Akira

=end
```

Sendfile() may return with a short write upon a client disconnect. Instead of retrying and getting an error, Ruby tries to force a select() on the descriptor which fails to detect the disconnect. This causes IO.copy_stream to hang, (possibly until TCP keepalives kick in). IO.copy_stream should raise immediately.

Thank you for the reproducible script and fix.

I'll commit your fix.

Thank you for looking into this.

However I think the Linux select behavior which doesn't notify writability on disconnected TCP socket is suspicious.

FreeBSD and Solaris notify writability.

I think select should notify writability when write would not block. Clearly write doesn't block on disconnected socket.

Linux also notify writability for UNIX domain socket pair.

UNIX domain sockets are easy to do notification for since they're always on the same host. TCP might be harder to detect (and thus the Linux folks choose not to bother at all) because the client is on a different machine and it might lose a physical connection.

How does FreeBSD or Solaris behave if a client is on a different machine.
and has the network cable pulled out? In the case of physically disconnected network cable, the client TCP stack has no way to notify the server of a disconnect. "kill -9" or even normal OS shutdown would give the TCP stack a chance to properly shutdown the connection.

There are a few more instances of "errno = EAGAIN" assignments in io.c that look suspicious to me. My proposed fixes are below, but I'm having trouble reproducing the badness I was seeing with IO.copy_stream in these code paths:

diff --git a/io.c b/io.c
index 5129a14..108af7e 100644
--- a/io.c
+++ b/io.c
@@ -649,7 +649,7 @@ io_fflush(rb_io_t *fptr)
  if (0 <= r) {
   fptr->wbuf_off += (int)r;
   fptr->wbuf_len -= (int)r;
-  errno = EAGAIN;
+  errno = EAGAIN;
   goto retry;   }   if (rb_io_wait_writable(fptr->fd)) {       rb_io_check_closed(fptr);
  }

-    Eric Wong
=end

#3 - 07/06/2010 03:32 PM - akr (Akira Tanaka)

=begin
2010/7/6 Eric Wong normalperson@yhbt.net:

UNIX domain sockets are easy to do notification for since they’re always on the same host. TCP might be harder to detect (and thus the Linux folks choose not to bother at all) because the client is on a different machine and it might lose a physical connection.

If the kernel cannot detect disconnect, how the kernel causes EPIPE?

How does FreeBSD or Solaris behave if a client is on a different machine and has the network cable pulled out? In the case of physically disconnected network cable, the client TCP stack has no way to notify the server of a disconnect. "kill -9" or even normal OS shutdown would give the TCP stack a chance to properly shutdown the connection.

I don't say about such physical disconnection.

I described about the situation that the kernel knows the connection is disconnected.

The connection is disconnected by RST packet. The RST packet is generated by a normal packet is sent to closed port.

% ruby -rsocket -e 'def netstat
s = netstat -n
s.each_line { |line| puts line if /State.*$|127.0.0.1:8888/ =~ line }
p
end
serv = TCPServer.open("127.0.0.1", 8888)
s1 = TCPSocket.open("127.0.0.1", 8888)
s2 = serv.accept
netstat
s2.close
netstat
s1.write "a" rescue $!
netstat
s1.write "a" rescue $!
P
IO.select(nil, [s1], nil, 0)
'
Proto Recv-Q Send-Q Local Address           Foreign Address
State
When first netstat call, the TCP states of s1 (the local address is 127.0.0.1:8888) and s2 (the local address is 127.0.0.1:34516) are ESTABLISHED.

s2.close sends a FIN packet to s1.
s1 receives it and send an ACK packet to s2.
This changes s1 to FIN_WAIT2 and s2 to CLOSE_WAIT.

The first s1.write "a" sends a normal data packet to s2.
Since the write system call doesn't wait the result of the packet, the system call itself succeeds.
But s2 is CLOSE_WAIT and no data acceptable.
So s2 sends back a RST packet to s1 and change state of s2 to CLOSED.
Then s1 receives the RST packet. It changes the state of s1 to CLOSED.

The second s1.write "a" fails with EPIPE.
This is because the kernel knows s1 is CLOSED.

Now the kernel knows write() for s1 doesn't block.
(It causes an error immediately)
So FreeBSD and Solaris notify it with select().
But Linux doesn't.
I think it is a problem of Linux.

Tanaka Akira

Unix domain sockets are easy to do notification for since they're always on the same host. TCP might be harder to detect (and thus the Linux folks choose not to bother at all) because the client is on a different machine and it might lose a physical connection.

If the kernel cannot detect disconnect, how the kernel causes EPIPE?

How does FreeBSD or Solaris behave if a client is on a different machine and has the network cable pulled out? In the case of physically disconnected network cable, the client TCP stack has no way to notify the server of a disconnect. "kill -9" or even normal OS shutdown would give the TCP stack a chance to properly shutdown the connection.

I don't say about such physical disconnection.

I described about the situation that the kernel knows the connection is disconnected.

The connection is disconnected by RST packet.
The RST packet is generated by a normal packet is sent to closed port.
When first netstat call, the TCP states of s1 (the local address is 127.0.0.1:8888) and s2 (the local address is 127.0.0.1:34516) are ESTABLISHED.

s2.close sends a FIN packet to s1. s1 receives it and send an ACK packet to s2. This changes s1 to FIN_WAIT_2 and s2 to CLOSE_WAIT.

The first s1.write "a" sends a normal data packet to s2. Since the write system call doesn't wait the result of the packet, the system call itself succeeds. But s2 is CLOSE_WAIT and no data acceptable. So s2 sends back a RST packet to s1 and change state of s2 to CLOSED. Then s1 receives the RST packet. It changes the state of s1 to CLOSED.

The second s1.write "a" fails with EPIPE. This is because the kernel knows s1 is CLOSED.

Now the kernel knows write() for s1 doesn't block. (It causes an error immediately) So FreeBSD and Solaris notify it with select(). But Linux doesn't. I think it is a problem of Linux.

Ah ok, thanks for the clarification. I missed the second write failing with EPIPE entirely :x

I think my second patch to remove "errno = EAGAIN" assignments might be needed for some corner cases, too, because we need a second write() to detect EPIPE under Linux.

--
Eric Wong
=end
This issue was solved with changeset r28557.
Eric, thank you for reporting this issue.
Your contribution to Ruby is greatly appreciated.
May Ruby be with you.

Can we get this backported to 1.9.2? I noticed it wasn't in rc2.
Malicious clients can exploit this bug and DoS servers this way.

Thanks.
--
Eric Wong