4.1 RDMA transport livelock(传输活锁)

> RDMA transport protocol is designed around the assumption that packets are not dropped due to network congestion. This is achieved by PFC in RoCEv2. However, packet losses can still happen for various other reasons, including FCS errors, and bugs in switch hardware and software [21]. Ideally, we want RDMA performance to degrade gracefully in presence of such errors. Unfortunately, we found that the performance of RDMA degraded drastically even with a very low packet loss rate. We illustrate this with the following simple experiment. We connected two servers A and B, via a single switch (W), and carried out three experiments for RDMA SEND, WRITE, and READ. In the first experiment, A used RDMA SENDs to send messages of size 4MB each to B as fast as possible. The second experiment was similar, except A used RDMA WRITE. In the third experiment B used RDMA READ to read 4MB chunks from A as fast as possible. The switch was configured to drop any packet with the least significant byte of IP ID equals to 0xff. Since our NIC hardware generates IP IDs sequentially, the packet drop rate was 1/256 (0.4%). We found that even with this low packet drop rate, the application level goodput was zero. In other words, the system was in a state of livelock – the link was fully utilized with line rate, yet the application was not making any progress. The root cause of this problem was the go-back-0 algorithm used for loss recovery by the RDMA transport. Suppose A is sending a message to B. The message is segmented into packets 0, 1, · · · , i, · · · , m. Suppose packet i is dropped. B then sends an N AK(i) to A. After A receives the NAK, it will restart sending the message from packet 0. This go-back-0 approach caused live-lock. A 4MB message is segmented into 4000 packets. Since the packet drop rate is a deterministic 1/256, one packet of the first 256 packets will be dropped. Then the sender will restart from the first packet, again and again, without making any progress. Note that TCP and RDMA transport make different assumptions on the network. TCP assumes a best-effort network, in which packets can be dropped. Thus, TCP stacks incorporate sophisticated retransmission schemes such as SACK [24] to deal with packet drops. On the other hand, RDMA assumes a lossless network, hence our vendor chose to implement a simple go-back-0 approach. In go-back-0, the sender does not need to keep any state for retransmission. This experiment clearly shows that for large network like ours, where packet losses can still occur despite enabling PFC, a more sophisticated retransmission scheme is needed. Recall however, that the RDMA transport is implemented in the NIC. The resource limitation of the NIC we use meant that we could not implement a complex retransmission scheme like SACK. SACK would also be overkill, as packet drops due to network congestion have been eliminated by PFC. Our solution is to replace the go-back-0 with a goback-N scheme. In go-back-N, retransmission starts from the first dropped packet and the previous received packets are not retransmitted. Go-back-N is not ideal as up to RT T × C bytes , where C is the link capacity, can be wasted for a single packet drop. But go-back-N is almost as simple as go-back-0, and it avoids livelock. We worked with our NIC provider to implement the go-back-N scheme, and since doing that, we have not observed livelock in our network. We recommend that the RDMA transport should implement go-back-N and should not implement go-back-0.

1. RDMA 传输协议是围绕数据包不会因网络拥塞而丢失的假设而设计的。这是通过 RoCEv2 中的 PFC 实现的。然而，由于各种其他原因，数据包丢失仍然可能发生，包括 FCS 错误以及交换机硬件和软件中的错误 [21]。
2. 理想情况下，我们希望 RDMA 性能在出现此类错误时能够适度降低。不幸的是，我们发现即使丢包率非常低，RDMA 的性能也会急剧下降。
3. 我们用下面的简单实验来说明这一点。我们通过单个交换机 (W) 连接两台服务器 A 和 B，并对 RDMA SEND、WRITE 和 READ 进行了三个实验。在第一个实验中，A 使用 RDMA SEND 尽可能快地向 B 发送每条大小为 4MB 的消息。第二个实验类似，除了 A 使用 RDMA WRITE。在第三个实验中，B 使用 RDMA READ 尽可能快地从 A 读取 4MB 块。交换机配置为丢弃 IP ID 最低有效字节等于 0xff 的任何数据包。由于我们的 NIC 硬件按顺序生成 IP ID，因此丢包率为 1/256 (0.4%)。
4. 我们发现，即使丢包率如此之低，应用程序级别的吞吐量也为零。换句话说，系统处于活锁状态——链路以线速充分利用，但应用程序没有任何进展。
5. 该问题的根本原因是 RDMA 传输用于丢失恢复的 go-back-0 算法。假设 A 正在向 B 发送一条消息。该消息被分段为数据包 0、1、⋯⋯、i、⋯、m。假设数据包 i 被丢弃。然后，B 向 A 发送 N AK(i)。A 收到 NAK 后，将重新从数据包 0 开始发送消息。这种 go-back-0 方法导致了活锁。
6. 一条 4MB 的消息被分割成 4000 个数据包。由于丢包率是确定性的 1/256，因此前 256 个数据包中的一个数据包将被丢弃。然后发送方将从第一个数据包重新开始，一次又一次，没有任何进展。
7. 请注意，TCP 和 RDMA 传输对网络做出不同的假设。 TCP 假设一个尽力而为的网络(尽最大努力交付)，在该网络中数据包可以被丢弃。因此，TCP 堆栈结合了复杂的重传方案（例如 SACK [24]）来处理数据包丢失。
8. 另一方面，RDMA 假定无损网络，因此我们的供应商选择实施简单的 go-back-0 方法。在go-back-0中，发送方不需要为重传保留任何状态。
9. 这个实验清楚地表明，对于像我们这样的大型网络，尽管启用了 PFC，仍然可能发生丢包，因此需要更复杂的重传方案。但请记住，RDMA 传输是在 NIC 中实现的。我们使用的NIC的资源限制意味着我们无法实现像SACK这样复杂的重传方案。 SACK 也没有必要，因为 PFC 已经消除了由于网络拥塞导致的数据包丢失。
10. 我们的解决方案是用 goback-N 方案替换 go-back-0。在go-back-N中，重传从第一个丢弃的数据包开始，之前接收到的数据包不再重传。
11. Go-back-N 并不理想，因为最多 RTT × C 字节（其中 C 是链路容量）可能会因单个数据包丢失而被浪费（好好理解这句话什么含义，为什么是RTT*C）。
12. 但是 go-back-N 几乎和 go-back-0 一样简单，并且它避免了活锁。我们与 NIC 提供商合作实施了 go-back-N 方案，自此之后，我们没有在网络中观察到活锁。我们建议 RDMA 传输应实现 go-back-N，而不应实现 go-back-0。

FCS errors （FCS 错误）

FCS错误是数据通信中的一种错误类型，FCS代表帧校验序列（Frame Check Sequence）。FCS错误表示在数据传输过程中，接收方计算出的数据帧的校验序列与接收到的数据帧中的校验序列不匹配。这表明数据在传输过程中可能发生了损坏或错误，通常是由于噪声、干扰、信号衰减或其他通信问题引起的。

FCS错误是一种校验机制，用于检测数据传输中的错误，以确保数据的完整性。当接收方检测到FCS错误时，通常会要求发送方重新发送数据帧，以确保数据的准确传输。这有助于提高数据通信的可靠性。

ps1: 上述检测是否匹配的原理是什么：

FCS（帧校验序列）错误的检测原理是通过使用循环冗余校验（CRC）算法来验证数据帧的完整性。以下是FCS错误的检测原理：

1. 发送方计算FCS：在数据帧的发送端，将数据帧中的数据部分通过CRC算法进行处理，生成一个FCS值。这个FCS值是数据帧的校验序列，它包含了数据的校验信息。
2. 数据传输：数据帧被传输到接收端。在传输过程中，数据帧可能受到干扰、噪声或其他问题的影响，导致其中的数据位发生变化。
3. 接收方计算FCS：在数据帧的接收端，接收到数据帧后，再次将数据帧中的数据部分通过相同的CRC算法进行处理，生成一个新的FCS值。
4. 比较FCS：接收方计算得到的新FCS值与数据帧中原始的FCS值进行比较。如果这两个FCS值不匹配，就会发生FCS错误。这表示在数据传输过程中，数据帧的内容已发生变化，数据帧可能已损坏。
   如果FCS值匹配，表示数据帧的完整性得到了保障，没有发生损坏。但如果FCS值不匹配，接收方通常会丢弃该数据帧并要求发送方重新发送，以确保数据的正确性。
   CRC是一种强大的校验算法，通常用于检测数据传输中的错误。它在计算FCS值时将数据帧的所有位都考虑在内，因此能够高效地检测各种类型的错误，包括单位比特错误和多位比特错误。这使得FCS错误检测非常可靠。

ps2: 介绍一下CRC算法及原理：

Cyclic Redundancy Check

循环冗余校验（CRC）算法是一种校验和校验方法，用于检测和纠正数据传输中的错误。CRC算法主要用于数字通信和数据存储领域，以验证数据的完整性，尤其在网络通信、存储介质（如硬盘、光盘）等方面广泛应用。

CRC的工作原理如下：

1. 选择一个特定的多项式：CRC算法的关键是选择一个多项式，通常称为生成多项式（或CRC多项式）。这个多项式的位数和系数是根据特定的CRC标准来确定的。不同的CRC标准使用不同的生成多项式。
2. 数据帧处理：要计算CRC值，首先需要将数据帧分成两部分：数据部分和附加的CRC字段。数据部分是要传输的数据，CRC字段是在发送端计算的校验值。
3. CRC计算：生成多项式被用来对数据部分进行数学运算（模2除法）。这个过程产生一个余数，也就是CRC校验值。
4. 附加CRC：CRC校验值附加到数据帧的末尾。
5. 数据传输：带有CRC校验值的数据帧被发送到接收端。
6. CRC校验：接收端使用相同的生成多项式对接收到的数据帧进行CRC计算。如果计算得到的CRC值与接收到的CRC值相匹配，那么数据被认为是完整的。如果CRC值不匹配，就表示数据帧可能已损坏。

CRC算法具有以下特点：

- 高效性：CRC校验是一种高效的校验方法，能够快速检测数据传输中的错误。
- 可靠性：CRC算法可以检测多种错误，包括单比特和多比特错误。
- 简单性：CRC计算是基于模2的二进制除法，计算相对简单。
  不过，需要注意的是，CRC算法主要用于检测错误，而不是修复错误。如果CRC校验失败，通常的做法是要求重新传输数据或采取其他纠错措施，以确保数据的正确性。因此，CRC在数据传输的可靠性和完整性方面发挥着重要的作用。不同的应用场景和标准可以使用不同的CRC多项式来满足特定的需求。

ps3: 提供一个简单CRC计算示例

CRC（循环冗余校验）的计算涉及多项式运算，通常使用模2二进制除法进行。下面将详细说明CRC的计算过程，并提供一个简单的示例。

1. 选择CRC多项式：首先，选择一个特定的CRC多项式，该多项式通常以二进制表示，并在计算中用于进行除法操作。不同的CRC标准使用不同的多项式。
2. 数据和CRC字段的排列：数据帧被分成两部分，数据部分和附加的CRC字段。数据部分是要传输的数据，CRC字段是在发送端计算的校验值。
3. 数据部分附加0：为了进行CRC计算，数据部分的末尾通常会附加一定数量的零比特，这些比特的数量通常等于CRC多项式的位数减一。这确保了CRC计算的正确性。
4. CRC计算：接下来，进行模2二进制除法运算。这是一个逐位的异或（XOR）操作。具体步骤如下：
   a. 将数据部分和附加的零比特组合成一个整体，作为初始的除数。
   b. 将CRC多项式作为除数。
   c. 开始从数据帧的左侧（高位）向右侧（低位）逐位地处理，逐位地进行异或操作。
   d. 如果当前处理的位是1，则执行异或操作。如果是0，则不执行异或操作。
   e. 持续处理数据帧的每一个位，直到数据帧的所有位都被处理完。
   f. 最后，得到的余数就是CRC校验值。

5. 附加CRC：CRC校验值被附加到数据帧的末尾。
6. 数据传输：带有CRC校验值的数据帧被发送到接收端。
7. CRC校验：接收端使用相同的CRC多项式对接收到的数据帧进行CRC计算，包括数据部分和接收到的CRC字段。如果计算得到的CRC校验值与接收到的CRC校验值相匹配，数据被认为是完整的。如果CRC值不匹配，就表示数据帧可能已损坏。

“TCP assumes a best-effort network, in which packets can be dropped. Thus, TCP stacks incorporate sophisticated retransmission schemes such as SACK [24] to deal with packet drops” TCP 假设一个尽力而为的网络，在该网络中数据包可以被丢弃。因此，TCP 堆栈结合了复杂的重传方案（例如 SACK [24]）来处理数据包丢失

该陈述准确描述了传输控制协议（TCP）的行为。TCP旨在通过潜在不可靠的网络提供可靠的面向连接的通信。它假设一个尽力而为的网络，在这种网络中，数据包可能会丢失、延迟、重复或无序交付。

为了应对数据包丢失的可能性，TCP结合了各种可靠数据传输的机制，包括：

1. 重传：当发送方在一定的超时期内没有收到已发送数据包的确认（ACK）时，它会假定数据包丢失，并重新发送。
2. 选择性确认（SACK）：SACK是TCP的扩展，允许接收方通知发送方有关丢失或无序段的信息，从而更有效地重传丢失的数据而不是整个流。
3. 流控制：TCP使用流控制机制来防止拥塞，并确保数据以接收方可以处理的速率发送。这可以防止网络过载，从而导致数据包丢失。**【滑动窗口】**
4. 拥塞控制：TCP还包括拥塞控制算法，以适应网络的可用容量，减少由于拥塞而导致的数据包丢失的可能性。
   这些机制共同确保在不可靠网络上实现可靠的数据传输。虽然TCP尽最大努力提供可靠的通信通道，但需要注意，在高度拥挤或极不可靠的网络中，它可能无法防止数据包丢失的情况。

ps：解释一下TCP的流控制技术

TCP的流控制是一种机制，用于确保数据在发送方和接收方之间以一种平衡和可控的速率传输，以避免拥塞和数据包丢失。流控制的主要目标是确保发送方不会以太快的速度向接收方发送数据，从而超过接收方的处理能力。
下面是TCP流控制的一些关键要点：

1. 滑动窗口：TCP使用滑动窗口机制来实现流控制。发送方和接收方都维护一个窗口大小（以字节数或数据包数量为单位）的值。发送方可以发送的数据量不会超过接收方报告的窗口大小。
2. 接收方通告窗口：接收方通过TCP首部中的通告窗口字段通知发送方当前可以接收的数据量。通告窗口的大小是动态调整的，根据接收方的可用缓冲区空间和处理能力。
3. 发送方的发送窗口：发送方的发送窗口是发送方能够维护的未确认数据的最大数量。发送方会根据接收方通告的窗口大小来调整它发送的数据量。
4. 动态调整：流控制是动态的，允许适应网络条件的变化。如果接收方的缓冲区变得拥塞或处理能力受限，它可以减小通告窗口的大小，从而减慢数据传输速率。如果接收方有更多可用的缓冲区空间，它可以增加通告窗口的大小，允许发送方发送更多数据。
5. 防止拥塞：流控制有助于防止网络拥塞，因为发送方会根据接收方的能力来调整发送速率。这有助于保持网络的稳定性，减少数据包丢失的风险。
   总之，TCP的流控制是一种动态机制，通过维护发送窗口和接收窗口大小来确保数据在网络中以一种可控和平衡的方式传输，以避免拥塞和数据包丢失。这有助于提供可靠的通信，尤其是在不可靠的网络环境中。

"Ingress" 和 "egress" 是网络术语，通常用于描述数据包在计算机网络中的流动方向。它们的含义如下：

1. Ingress（入口）：Ingress port 指的是数据包进入网络的物理或逻辑接口或端口。这是数据包从外部网络流入本地网络的地方。Ingress 端口通常用于描述网络设备（如路由器、交换机、防火墙）上用于接收传入数据包的接口。在入口处的数据包通常会经过一系列的处理和过滤，以确定它们是否应该进一步传输到网络中的其他部分。
2. Egress（出口）：Egress port 指的是数据包离开网络的物理或逻辑接口或端口。这是数据包从本地网络流出到外部网络的地方。Egress 端口通常用于描述网络设备上用于发送数据包到外部网络的接口。在出口处的数据包可能会经过一些额外的处理和路由，然后被传递到目标网络或设备。