Homa

Harsh Kapadia

Boston University — Spring 2023

Table of Contents

NOTE	Latest (HTML) version of this document: networking.harshkapadia.me/files/homa/report

Introduction

- Homa is a new transport protocol designed specifically for Data Centers and operates at the Transport Layer of the OSI Model.
- It aims to replace the Transmission Control Protocol (TCP) as the transport protocol in Data Centers and claims to fix all of TCP's problems.
- Homa's primary goal is to provide the lowest possible latency for short messages at high network load.

Data Center Environment

Since Homa is a transport protocol meant for Data Centers, it is important to understand a typical Data Center structure and the requirements of protocols operating in such an environment.

Topology

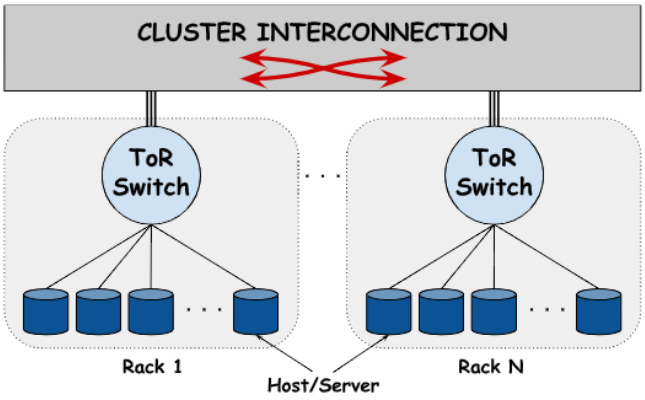

Fig. 1. A typical datacenter cluster.

Figure 1. A typical Data Center cluster

Considering a generalised Data Center topology:

- A Data Center typically consists of various clusters. Each cluster consists of multiple racks and each rack consists of multiple machines (hosts/peers).
- Each rack has a switch that connects all the machines in that rack, called the Top of Rack (ToR) switch.

Protocol Requirements

- Low latency
  - Tail latency should be good.

    - Packet processing overheads should be low.
- High throughput
    - Both data throughput (amount of data sent) and message throughput (no. of messages sent) should be high.
- Reliable delivery
    - Data should be delivered regardless of failures.
- Congestion Control
    - Congestion (packet buildup in buffers) needs to be kept at a minimum to ensure low latency.
- Efficient Load Balancing
    - Links and processor cores should be be hot spots.

Message vs Packet

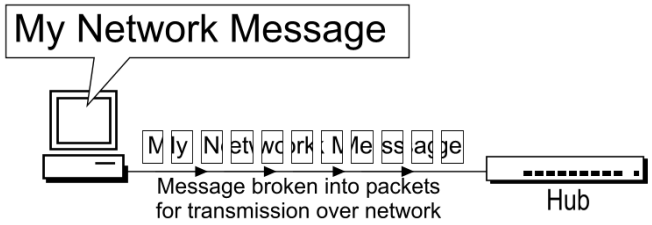

Figure 2. A message is transmitted by splitting it into packets.

- Message: A logically sensible, complete and dispatchable data unit that makes sense in its entirety and can be processed.
- Packet: A chunk/part of a message that can be combined with other packets of that message to form the complete message.

Problems with TCP

Stream Orientation

- TCP is a Stream-Oriented Protocol. It blindly creates packets of a certain length as and when the Session Layer (OSI Model) above it gives it data, without having any knowledge of a logical chunk of data (a message).
    - Content (Message) lengths present at higher layers (like the Application Layer HTTP protocol `Content-Length` header) are not visible to TCP.
- So TCP is not aware of the message size. It is only aware of the length of the current packet.
- This means that the receiver does not have immediate knowledge as to when it can start processing data. It has to figure that out by looking through and parsing the request data sent.
- The receiver also does not know how much data it is going to receive, so the sender is responsible to not overwhelm the receiver.

- As TCP is a stream, it prefers to follow one path to its destination (Flow-Consistent Routing), to prevent too much packet reordering. This leads to Load Balancing issues, because multiple TCP flows along the same path can cause congestion and TCP Head-of-Line Blocking (HoLB).
- Another issue with streaming is that there is a chance of multiple application threads on the receiver reading data not meant for them, because the receiver doesn't immediately know where a message starts or ends at the Transport Layer. With protocols like HTTP/2 which multiplex multiple requests on the same TCP connection, this problem becomes even more apparent.
  - HTTP/2 also suffers from TCP HoLB.
    - The HTTP/3 protocol solves this issue by using the QUIC transport protocol in place of TCP, but all the other issues with streaming remain.

Connection Orientation

- TCP is a Connection-Oriented Protocol.
- TCP does a three-way handshake to establish a connection between a sender and a receiver, which takes ~1 RTT (Round-Trip Time), where data is not being sent.
- Each application might have hundreds or thousands of connections and a Data Center might have thousands or millions of applications!
- Thus, connection-orientation causes a lot of overheads in terms of storage, compute, goodput and latency.

Fair Scheduling

- TCP uses Fair Scheduling to share bandwidth between various connections, where all TCP streams/flows get an equal bandwidth.
- Under high load, all streams try to share bandwidth equally, which collectively slows down everyone.

Sender-Driven Congestion Control

- In the TCP model, the sender is responsible to implement Flow Control and Congestion Control, to not overwhelm the receiver and network switches respectively.
- Detection of congestion causes TCP to reduce its sending rate, which reduces throughput and Link Utilization.
- The sender mostly detects congestion by packet loss, duplicate acknowledgements and timeouts (among other parameters), which means:
  - Assumption of congestion
    - TCP assumes congestion on packet loss, whether that is the reality or not.
  - Inaccurate congestion detection (False positives)
    - TCP has an issue of falsely detecting congestion at times, because packet loss isn't always caused by congestion.

    - Detection only after congestion
        - If it detects congestion, it does so only after queuing/buffer buildup has started.
- All of the issues above mean that TCP bases its logic off of assumption and reduces its sending rate based on those inaccurate assumptions, which leads to throughput loss.

In-Order Packet Delivery

- TCP has a lower out-of-order packet tolerance than Homa.
- TCP prefers that packets are sent in-order and so on the same link (Flow-Consistent Routing).
- As discussed in the Stream Orientation sub-section above, Flow-Consistent Routing causes hot spots to develop (Load Balancing issues), which leads to congestion and TCP HoLB.

Sender vs Receiver

Depending on the direction of communication, a client and a server can be the sender and the receiver respectively, or vice versa.

Table 1. Direction of communication defining clients and servers as senders or receivers

	Client → Server (Request)	**Server → Client (Response/Reply)**
Sender	Client	Server
Receiver	Server	Client

Features

Table 2. Homa's features that fulfill a Data Center protocol's requirements

Protocol Requirement	**Homa Feature(s)**
Low latency	Connectionless protocol, Shortest Remaining Processing Time
High throughput	Receiver-driven Congestion control, Connectionless protocol, Shortest Remaining Processing Time, No per-packet acknowledgements
Reliable delivery	At-Lease-Once semantics
Congestion Control	Receiver-driven Congestion Control, Shortest Remaining Processing Time, High out-of-order packet tolerance
Efficient Load Balancing	Receiver-driven Congestion control, High out-of-order packet tolerance

Homa's features:

Message Orientation

- Homa is a Message-Oriented Protocol unlike TCP, which is a Stream-Oriented Protocol. This means that Homa is aware of the overall message length at the Transport Layer unlike TCP, which is only aware of packet lengths that it has to create.
- Homa implements Remote Procedure Calls (RPCs), which exposes a measurable dispatch unit (a 'message') to the transport layer.
- Now that the sender knows how much data it has to send to complete the RPC Request (= message length), it communicates that message length to the server in the first packet.
- This enables the receiver to know how much data it is expecting, how much data has arrived and how much is pending for every RPC.
- This is game changing, because Homa can now implement Receiver-Driven Congestion control as described in the next sub-point below.
- Knowing the full message length also lets the receiver know when it has received the message in its entirety right at the Transport Layer, so that the appropriate application thread can consume the message and carry out the required action(s). There is no fear of reading a packet from a different message, as for example with TCP in HTTP/2.

Receiver-Driven Congestion control

- As discussed before, Sender-Driven Congestion Control is inaccurate, is based on assumptions and detects congestion only after it has occurred.
- Using Homa, the receiver knows the total data each RPC is going to send it, so it can implement mechanisms to implement Flow and Congestion Control.
    - As mentioned before, the sender sends the message length to the server in the first packet.
- Letting the receiver control each RPC sender's flow is more accurate and not based on assumptions, because the receiver knows how much data is going to receive from all RPCs trying to send data to it.
- The receiver can make decisions of whether to grant permission to a RPC to send data to it based on its buffer occupancy, available bandwidth, observed RTT, etc. These decisions can be made in real-time as well and new instructions can be communicated to the sender whenever the receiver deems fit.

Connectionless Protocol

- Homa uses RPCs and so it doesn't require explicit connection establishment between the sender and receiver. This reduces connection setup overhead, in terms of storage, compute, goodput and latency.

Shortest Remaining Processing Time

- Homa implements Shortest Remaining Processing Time (SRPT) Scheduling to queue messages to send, rather than TCP's Fair Scheduling.

- 'Processing time' corresponds to the amount of the message left to be transmitted/received.
- The lesser the data left to be transmitted/received (i.e., the lesser the processing time), the earlier the message will be sent (i.e., the higher the priority the message will have).
- Thus, SRPT prevents short messages from starving behind long messages in queues on both ends, which solves the TCP HoLB problem.

No Per-Packet Acknowledgements

- Homa does not send out explicit acknowledgements for every packet, thus reducing almost half the packets that have to be sent per message in comparison to TCP.
- This reduces transmission overheads and conserves bandwidth.
- Homa does send some Control Packets to regulate the protocol, but they are not nearly as frequently sent as TCP acknowledgements.
    - More details about Homa's packets and working can be found below.

High Out-of-Order Packet Tolerance

- Packet Spraying
    - Packet Spraying is a technique in which packets of one flow are sent over multiple short paths to the destination, rather than just using one path for the entire flow (as in TCP's Flow-Consistent Routing).
    - Packet Spraying is advantageous because it aids in Load Balancing packets over multiple links, avoiding network traffic hot spot creation on particular paths/links. This keeps congestion to the minimum.
    - The problem with different packets being sent on different links is that the packets will reach the destination at different times and out of order, so there will be an increased reordering of packets.
        - Excessive reordering of packets causes unnecessary timeouts, which causes unnecessary retransmissions and that wastes bandwidth.
- Homa can reap Packet Spraying's Load Balancing benefits without worrying too much about the added reordering causing retransmissions, because it has a higher out-of-order packet tolerance than TCP. In extreme cases where a RPC has to be aborted, Homa will restart the communication as well.

At-Least-Once Semantics

- Homa is a reliable protocol and implements At-Least-Once semantics to provide reliability.
- In case of failures or losses, Homa does have mechanisms to ensure retransmission, so packets are sent at least once, but can be sent more times in case of issues, to ensure delivery.

Peers, RPCs and Receivers

A peer (host/machine) can be a sender or a receiver and can have multiple RPCs.

Table 3. The relation between peers, RPCs and Receivers

Sender ID	RPC ID	Receiver ID
A	RPC1, RPC2	B
B	RPC3	A
C	RPC4, RPC5, RPC6	B

Priorities

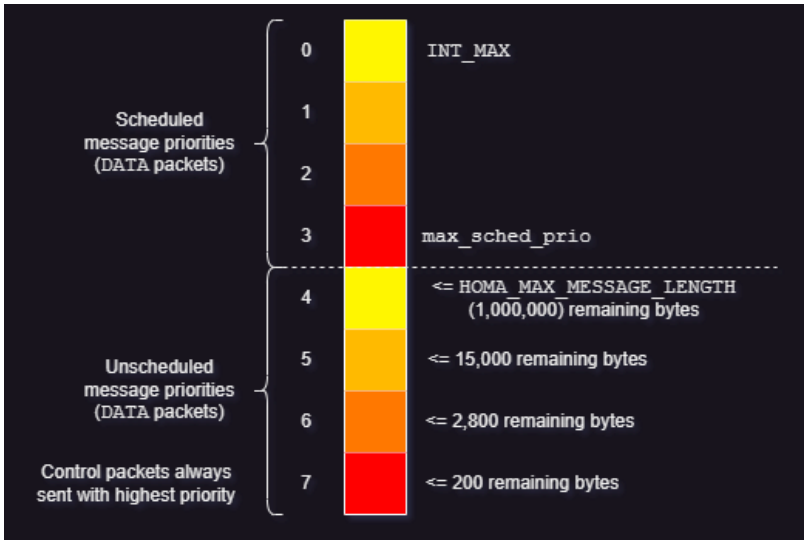

Figure 3. Initial values for Homa's default priority array (Priority increases from Level 0 to Level 7.)

- Homa makes use of priorities to implement SRPT Scheduling, which prevents shorter messages from starving behind longer messages (HoLB).
- The lesser the bytes of a message still to arrive at the receiver, the higher is that message's priority.
  - In case of a tie on the number of bytes left to be transmitted, the older message gets the higher priority.
- Homa ranks (gives a priority order to) all RPCs in every peer based on the above criteria and then ranks all the peers based on each one's highest priority RPC.

Sender

- A Homa sender divides all messages into two parts
  - A small unscheduled part.
    - It is sent blindly (without waiting for anything) to the receiver to inform it of the total message length.
  - A larger scheduled part.

        - It is sent part-by-part as and when the receiver permits (grants) it.
- Unscheduled data has higher priority than scheduled data.
- Homa has eight priority levels by default, from Level 0 (lowest priority) to Level 7 (highest priority).
- The eight levels are split into two parts by a user-defined parameter (`max_sched_prio`). The part with the higher priorities is used for unscheduled data, while the other part with lower priorities is used for scheduled data.
    - Unscheduled data has higher priority because those packets have to reach the receiver as soon as possible to inform it of the total message length.
- All packets other than the packets that carry message data bytes are called Control Packets and are always sent at the highest priority, as they help regulate the protocol.

Receiver

- A Homa receiver dynamically decides the priority of a message, because
    - It knows the amount of data yet to arrive for all the RPCs that it has.
    - It is aware of the load, bandwidth, free buffer space, etc. it has available.
- A receiver will inform its senders of the unscheduled and scheduled data priorities it should use.
- A receiver usually computes new priorities for messages when it needs to send new permission granting packets to RPCs.

Sending Data

- To send unscheduled data, a Homa sender checks if it has received any unscheduled data priorities from the receiver.
    - If it has, it uses those priorities to send unscheduled data packets.
    - If not, then it uses its own initialized values, which will trigger the receiver to send it its updated values for later unscheduled data.
- Once unscheduled data is sent, the receiver usually grants permission to the sender to send (scheduled) data to it and includes the priority level to be used for those scheduled data packets in the permission granting packet itself.
- To send scheduled data, the sender adds the priority level that it received from the receiver to the new packets with the scheduled data and then sends them across.

Packet Types

Homa's packet types:

DATA Packet

```
DATA(rpc_id, data_bytes, data_offset, self_priority, message_length)
```

- Sent by the sender.

GRANT Packet

```
GRANT(rpc_id, expected_data_offset, expected_scheduled_data_priority)
```

- Sent by the receiver.
- Indicates that the sender may now transmit all bytes in the message up to a given offset.

RESEND Packet

```
RESEND(rpc_id, data_offset, expected_data_length, expected_data_priority)
```

- Sent by the sender or receiver.
- Indicates that the sender should retransmit a given range of bytes within a message.

UNKNOWN Packet

```
UNKNOWN(rpc_id)
```

- Sent by the sender or receiver.
- Indicates that the RPC for which a packet was received is unknown to it.

BUSY Packet

```
BUSY(rpc_id)
```

- Sent by the sender.
- Indicates that a response to `RESEND` will be delayed and is used to prevent timeouts.
    - The sender might be busy transmitting higher priority messages or another RPC operation is still being executed.

`CUTOFFS` Packet

```
CUTOFFS(rpc_id, expected_unscheduled_data_priority)
```

- Sent by the receiver.
- Indicates priority values that the sender should use for unscheduled packets.

`ACK` Packet

```
ACK(rpc_id)
```

- Sent by the sender.
- Explicitly acknowledges the receipt of a response message for one or more RPCs.

`NEED_ACK` Packet

```
NEED_ACK(rpc_id)
```

- Sent by the receiver.
- Indicates an explicit requirement for an `ACK` packet for a particular RPC.

`RTTbytes`

- While sending unscheduled data and in `GRANT` packets, Homa has to send a certain amount of data and a data offset respectively.
- The amount of data or data offset is approximately set to `RTTbytes`, where RTT stands for 'Round-Trip Time'.
- The value `RTTbytes` is the number of bytes that can be kept transmitting from the point of sending one packet until the point another packet is received and processed.
- By the time `RTTbytes` are transmitted, a `GRANT` packet will most probably have arrived from the receiver and will have been processed, so the sender can keep transmitting more data without interrupting its sending, thus keeping Link Utilization at 100%.
    - Sending and receiving can be done at the same time, because links are usually Full Duplex.

Working

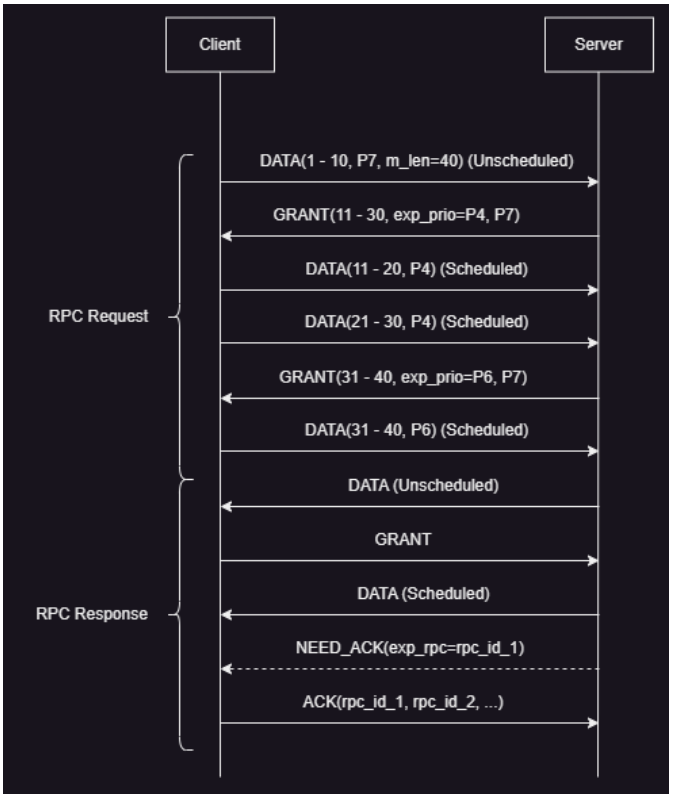

Figure 4. A general Homa communication sequence diagram (Priority increases from Level P0 to Level P7.)

Client

Sender

- Whenever Homa (operating at the Transport Layer) receives a message (RPC Request) from the layer above it (Session Layer) at the client, it divides the message (data) into a small 'unscheduled data' part and a larger 'scheduled data' part.
- The client is starting the communication with the server, so it behaves as the sender.
- The unscheduled bytes of the message are sent blindly to the receiver (server) in `DATA` packets as a RPC Request. (`DATA` packets contain the message length.)
- If the receiver thinks that it can receive packets from that RPC, it will send a `GRANT` packet to the sender, which will contain the amount of data that the sender can send (`RTTbytes`) and the priority at which the packets should be sent.
- The sender now sends all the data it was granted permission to send, in one or more packets, depending on the MTU (Maximum Transmission Unit).
- `GRANT` packets keep coming in until the sender has finished sending all its data or till the receiver is able to accept data, whichever comes first.
    - If the receiver cannot accept more data, it will just not send a `GRANT` packet for a certain time period. Further information in the Message Sequence Scenarios section.

Receiver

- Once the RPC Request has been full transmitted, the sender starts acting as a receiver and waits for a response (RPC Response/Reply) to its request.

- Once the complete RPC Response has been received, an `ACK` packet is sent to signal the receipt of the message and the end of that RPC's communication.

Server

Receiver

- On receiving unscheduled `DATA` packets through a RPC, the receiver (here, the server) checks if it has the capacity to accept connections.
- If the receiver can accept data, then it calculates
  - The priority at which data has to be sent to it.
  - The amount of data that the receiver can accept (`RTTbytes`).
- On computing that, the receiver adds that data to a `GRANT` packet and sends it to the sender asking for more data.
- The receiver gets `DATA` packets from the sender for all the data it granted.
- Once the receiver receives all the packets with the data it had asked for from the sender, it sends another `GRANT` packet with a newly computed priority and data offset.
- The receiver keeps receiving `DATA` packets and keeps sending `GRANT` packets until the entire message has arrived or until it can accept more data, whichever comes first.
- Once the entire message (RPC Request) has arrived, the receiver hands it off to the appropriate application thread for processing.

Sender

- Once the application gets back with a response message, the receiver (the server) starts acting as a sender.
- Once the RPC Response has been transmitted, an `ACK` packet is expected, to indicate message receipt and as a trigger to get rid of all the state related to that RPC.
  - If an `ACK` packet is not received in time, a `NEED_ACK` packet can be sent, requesting for an `ACK` packet.

Message Sequence Scenarios

Error-Free Communication

Explained in the Working section above.

Scheduled Data Loss

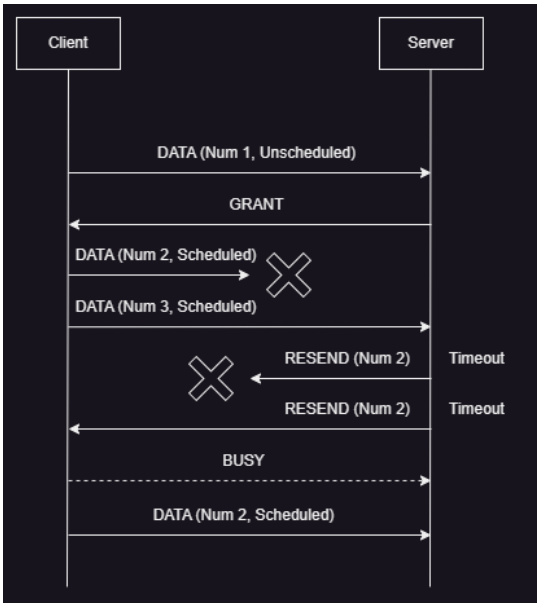

Figure 5. A RPC Request where lost scheduled `DATA` packets trigger `RESEND` packets.

- A RPC Request is shown in the image above.
- Here, a scheduled `DATA` packet is lost and the `RESEND` packet for that missing data is lost as well, but the next `RESEND` packet that is sent after a timeout makes it to the sender.
- The sender can either immediately respond with the missing data in a `DATA` packet or if it is busy transmitting other higher priority packets, then it can send a `BUSY` packet to the receiver to prevent a timeout (like a 'keep-alive' indicator) and can send the `DATA` packet once it is free.
    - Avoiding timeouts helps prevent unnecessary `RESEND` packets and abortion of the RPC.

Aborted RPC

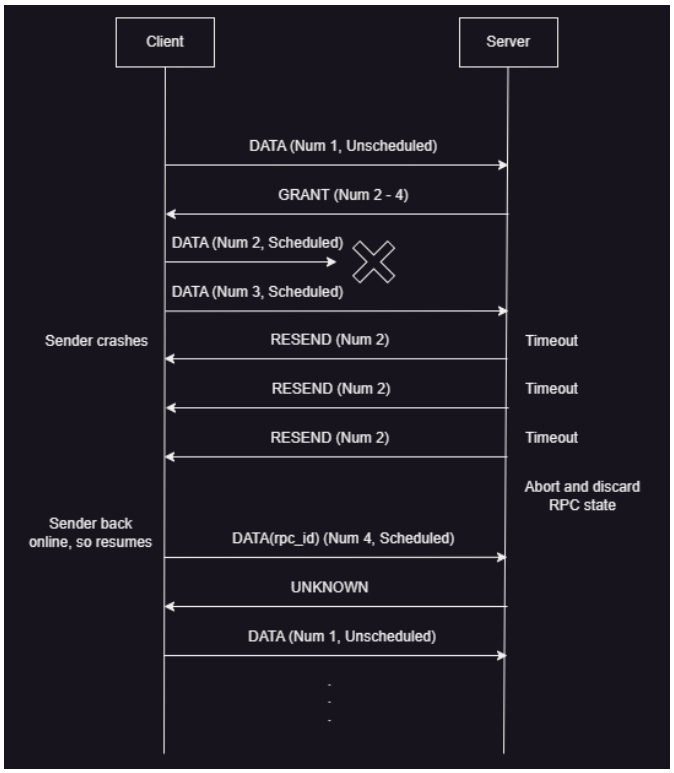

Figure 6. A RPC Request where the sender crashes, which causes multiple timeouts and the receiver eventually aborts the RPC. Once online, the sender is forced to restart the communication.

- A RPC Request is shown in the image above.
- The sender crashes after sending two of its three scheduled `DATA` packets.
- The first scheduled `DATA` packet is lost as well, which causes a timeout on the receiver, causing it to send a `RESEND` packet for the missing data.
- As the sender has crashed, the `RESEND` packet does not get an expected `DATA` packet response, which leads to timeouts and more `RESEND` packets.
- After multiple `RESEND` packets not receiving responses, the receiver determines that the sender is non-responsive and discards all of the state related to that RPC ID.
- On coming back online, the sender looks at its previous state and tries to resume by sending the third scheduled `DATA` packet that it had not sent, but the receiver sends an `UNKNOWN` packet on receipt of that `DATA` packet, as it had already discarded all information related to that RPC ID.
- The sender has to restart the communication with the receiver.

Unscheduled Data Loss

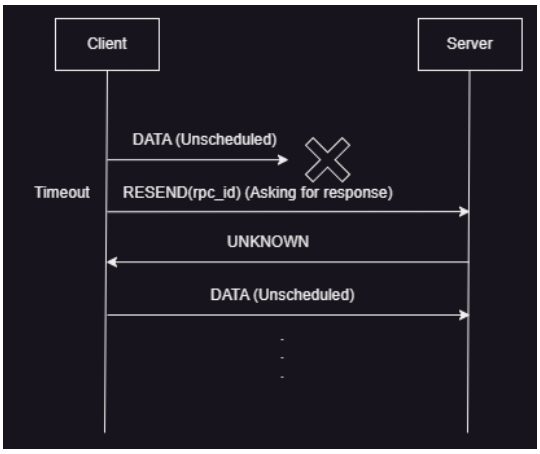

Figure 7. A RPC Request where the initial unscheduled data itself gets lost, which eventually leads to the client having to restart the communication.

- A RPC Request is shown in the image above.
- If the blindly sent unscheduled `DATA` packets don't reach the receiver due to loss, overload, congestion or other reasons, then the sender times out waiting for a response from the receiver.
- On timing out, the sender sends a `RESEND` packet to the receiver, asking for a response.
- When the `RESEND` packet reaches the receiver, it will respond with an `UNKNOWN` packet, because it never got the initial packets and was never aware of the RPC.
- The sender has to restart the communication with the receiver.

Conclusion

- In conclusion, Homa is wonderful a study in understanding the shortcomings of TCP, designing a protocol to fix those shortcomings to lead to better performance.
- TCP is one of the most widely used transport protocols in Data Centers and displacing it with Homa is a long way away, but in experiments, Homa has been able to achieve a significantly

better performance than TCP, which can be an impetus to adopt it in Data Centers.

Acknowledgements

I would like to thank Prof. Dr. Abraham Matta for his guidance, understanding and help throughout the study.

I would also like to thank Prof. Dr. John Ousterhout for writing the Homa research papers and Linux Kernel module, and for his help setting up Homa and answering my queries regarding the protocol.

Contact

Feel free to reach out to the author of this document, Harsh Kapadia, at harshk@bu.edu or through links.harshkapadia.me.

References

- Detailed explanations: networking.harshkapadia.me/homa
- Research papers
  - J. Ousterhout. *It's Time to Replace TCP in the Datacenter.*
  - B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. *Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities.* In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, pages 221 - 235, New York, NY, USA, 2018. Association for Computing Machinery.
  - J. Ousterhout. *A Linux Kernel Implementation of the Homa Transport Protocol.* In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 99 - 115. USENIX Association, July 2021.
  - M. Noormohammadpour and C. S. Raghavendra. *Datacenter Traffic Control: Understanding Techniques and Tradeoffs.* In IEEE Communications Surveys & Tutorials, vol. 20, no. 2, pp. 1492-1525, Secondquarter 2018, doi: 10.1109/COMST.2017.2782753.
- Homa Linux Kernel module
- Homa experimentation
- Detailed references: networking.harshkapadia.me/homa#resources

```
<style>
  .imageblock > .title {
    text-align: inherit;
  }
</style>
```