Google Cloud

TCP Memory Isolation on Multi-tenant Servers

Shakeel Butt <shakeelb@google.com>
Christian Warloe <cwarloe@google.com>
Wei Wang <weiwan@google.com>

Linux Plumbers Conference Sep 13, 2022

Work done in collaboration with:

Yuchung Cheng <ycheng@google.com>

Eric Dumazet <edumazet@google.com>

Coco Li <lixiaoyan@google.com>

Arjun Roy <arjunroy@google.com>

Soheil Hassas Yeganeh <soheil@google.com>

Vinny Zhang <vwz@google.com>

and others ...

Outline

- What is TCP memory?
- How TCP memory is accounted?
- Problems in existing TCP memory accounting
- Solution
- Challenges in deploying the solution
- Work-in-progress
- Conclusion

What is TCP memory?

- **TCP memory:** Memory holding the packets in flight
- **TX/Send**
  - Keep the data in memory until the receiver has ACKed the data
- **RX/Receive**
  - Keep the data in memory until the user application has consumed it

How is TCP memory accounted?

- Accounting TCP memory
  - Single global counter: **`tcp_memory_allocated`**
  - Visible through **`/proc/net/sockstat[6]`** and **`/proc/net/protocols`**
- Limiting TCP memory
  - System wide shared limit: **`/proc/sys/net/ipv4/tcp_mem`** (Array of 3 long integers)
    - **Enter** TCP pressure state: **`tcp_memory_allocated > tcp_mem[1]`**
    - **Leave** TCP pressure state: **`tcp_memory_allocated <= tcp_mem[0]`**
    - **Hard** TCP usage limit: **`tcp_memory_allocated > tcp_mem[2]`**

What happens on TCP pressure?

- Reduce (or prevent increasing) the send or receive buffers for the sockets
- On **RX**
  - May coalesce packets
  - May drop packets preferably out-of-order packets
  - Wakes up the userspace application to consume the incoming packets
- On **TX**
  - May throttle the current thread of the sender

Current TCP memory accounting causes isolation issues

Problem 1: Shared unregulated tcp_mem limit

When the TCP memory usage hit the TCP limit:

1. Sockets of arbitrary jobs will see reduced send and receive buffer.
2. Packets of arbitrary jobs will be drops.
3. Threads of arbitrary jobs will get throttled.

Low priority jobs can hog TCP memory and adversely impact higher priority jobs

Current TCP memory accounting causes isolation issues

Problem 2: Disconnect between TCP memory & system memory

When the system is **OOM** but TCP memory usage is in normal range:

1. TCP pressure mechanisms do not get triggered and allow network bursts.
2. A network burst can cause atomic allocation failures negatively impacting arbitrary jobs.
3. A network burst steals memory and CPU from arbitrary jobs doing memory reclaim.
4. A network burst keeps the system in OOM state for longer negatively impacting almost all the jobs.

Negatively impacts the ability to provide differentiated services to jobs of different priorities

Current TCP memory accounting causes isolation issues

Problem 2: Disconnect between TCP memory & system memory

When the system is **OOM** but TCP memory usage is in normal range:

1. TCP pressure mechanisms do not get triggered and allow network bursts.
2. A network burst can cause atomic allocation failures negatively impacting arbitrary jobs.
3. A network burst steals memory and CPU from arbitrary jobs doing memory reclaim.
4. A network burst keeps the system in OOM state for longer negatively impacting almost all the jobs.

Negatively impacts the ability to provide differentiated services to jobs of different priorities

Solution is still in WIP: we will discuss some ideas at the end

Solving Problem 1

- **Remove shared global TCP limit**
- **Start charging jobs for their TCP memory usage**
  - Use memory cgroups to start charging TCP memory
  - The memcg limit of jobs will limit their TCP memory usage

Solution: TCP memory accounting using memory cgroups (TCP-memcg)

- TCP memory accounting has different semantics in memcg-v1 vs memcg-v2
- In memcg-v1, TCP memory is accounted separately from the memcg memory usage
  - Added complexity to provision another resource
  - Off by default and inefficient
- In memcg-v2, TCP memory is accounted as regular memory
  - Aligns with our cgroup v2 migration journey
- We **ported** memcg-v2 TCP accounting into our memcg-v1 deployment

Solution: TCP memory accounting using memory cgroups (TCP-memcg)

- TCP memory accounting has different semantics in memcg-v1 vs memcg-v2
- In memcg-v1, TCP memory is accounted separately from the memcg memory usage
  - Added complexity to provision another resource
  - Off by default and inefficient
- In memcg-v2, TCP memory is accounted as regular memory
  - Aligns with our cgroup v2 migration journey
- We **ported** memcg-v2 TCP accounting into our memcg-v1 deployment

TCP pressure for memcg is still in WIP: some discussion at the end

Challenges in deploying TCP Memcg

- No historical data on how much TCP memory each job is allocating
- Additional memory need to be provisioned for the jobs.
  - Failure to correctly provision can lead to OOM or network degradation
  - TCP usage is spiky, making it hard for users to predict usage increase
- Each job owner may have different priorities and timelines
  - A single user can become a long pole and may delay the deployment of TCP-memcg
- Enabling new functionality exposes untested scenario and potentially new bugs

Deploying TCP memcg: Data Collection

- Implemented a new mode "measure" of TCP memory accounting
    - Measure network memory usage by the jobs ***without*** charging them
- Deployed across the fleet and collected TCP memory usage data of all jobs

Deploying TCP memcg: Memory resource provisioning

- Identify all the jobs that are under provisioned
  - Migrate users to load-shaping mechanisms
    - Userspace traffic throttling
    - Dynamic job sizing
  - Or, raise memory limits
    - Manual, error prone and can become long pole for adoption

Deploying TCP memcg: Fine-grained Control

- Toggle network memory charging per container
- Opt-in
  - Allow jobs to experiment ahead of time and de-risk rollout
- Opt-out
  - Quickly mitigate job specific issues during rollout
  - Prevent rollout from being blocked on single users
- This capability de-risks the TCP-memcg deployment by not letting small set of users who are slow to opt-in.

Deploying TCP memcg: new kernel bugs

1. Unwarranted memcg OOMs
2. Machines getting hard locked up on memcg OOM of network intensive jobs

Bug#1: Unwarranted memcg OOMs

- On opting-in to TCP-memcg one specific job started seeing higher rate of memcg OOMs
- On closer look at the OOM report:
  - TCP memory usage was very high
  - Job's memory usage plus free memory on the system was larger than DRAM on system

Two decade old TCP pre-charge optimization

- Kernel implements per-socket pre-charge cache (`sk->sk_forward_alloc`) to reduce contention on SMP machines
    - On allocation, adds more than requested size to global counter and cache the difference to make subsequent allocations fulfilled without access to global counter.
    - On deallocation, deposit to local cache up to a certain limit.
- What happens for applications with thousands of sockets?
    - For some scenarios, a socket can cache up to 1MiB of charge.
    - Large amount of fragmented charges (# of sockets * 1 MiB)
    - Can cause memcg OOMs as these are not reclaimable

Solution to the fragmented pre-charges

- Move from per-socket cache to per-cpu cache
  - Number of CPUs limit the amount of cached charge. (# of CPUs * 1MiB)
  - See "net: reduce tcp_memory_allocated inflation" patch series.
- Side effect of the solution
  - Memory cgroup becomes the performance bottleneck (upstream report) for TCP memory accounting.
  - Posted memory cgroup charging optimizations.

Bug#2: Hardlock up by memcg OOMing network intensive job

- The scenario triggering the hardlock up:
  - The job had a lot of threads in **`epoll_wait()`**.
  - The job exceeded its limit and gets OOM-killed.
  - A burst of incoming packets keep trying to wake up the job's threads creating contention of Read/Write spinlock of the eventpollfd.
  - Linux Read/Write spinlock bias towards readers in IRQ context and thus put fuel to this fire.
- Root cause
  - Wakers have to travel the linked list containing all the epoll waiter while holding locks to find the thread they waked
  - All the waiting threads have their status changed on SIGKILL, so wakers have to traverse the whole linked list
- Solution: Remove the thread from the linked list irrespective of their status

WIP: System level TCP pressure

- **Problem:** Disconnect between TCP memory and system memory
- How about dynamically change tcp_mem based on `MemFree` from `/proc/meminfo` ?
- Two challenges:
  - Is `MemFree` the metric? What if there is a lot of easily cold reclaimable memory?
  - The TCP throttling mechanisms does not differentiate between jobs of different priorities
- Possible solution:
  - `MemFree` in the presence of proactive reclaimer (uswapd)
  - Define different TCP throttling thresholds for jobs of different priorities
  - Possibly BPF based implementation

WIP: Memcg level TCP pressure

- Currently **vmpressure** is used to trigger TCP pressure for memcg
    - Not good if job lacks reclaimable user memory
- What about PSI?
    - PSI is oblivious to the source of memory pressure
- What about **`memory.high`**?
    - CPU throttling on memory.high can make TCP pressure worse
- Possible solution: Something similar to **`memory.high`** but without CPU throttling.

Conclusion

- For multi-tenant servers, static tcp_mem is **harmful**.
- If you run multi-tenant systems in your infra or if you are planning to migrate from cgroup v1 to v2 then you will face similar challenges.
- **Takeaways** from our experience of deploying TCP-memcg:
    a. Changing fundamental part of the system will break old assumptions and expose new bugs.
    b. The capability to opt-in or out individual jobs enabled us to do more aggressive deployment.
    c. Dynamic right sizing and load balancing technologies drastically reduce