Integrity of In-memory Data Mirroring in Distributed Systems

Tejas Wanjari
EMC Data Domain
Problem Definition

- In-memory data is changing
- Disk checksums are for the older state
- Mirroring cannot rely on disk checksums
- Undetected corruptions are not acceptable
- Reliability is prime (e.g. Backup/Recovery Systems)
Sources of Corruption during Mirroring

- System failure
  - “Clean” shutdown and reboot
- Hardware failure
  - Redundant failover
- Disks failure
  - Disk/filesystem checksums
- Process corruption
  - Avoiding copying without checksums
- Network corruption
  - Application/protocol checksums
TCP Checksum Vulnerability

- TCP Checksum: 4 bytes & weak
- Prone to False Positives (FPs)
  - Wrong data, correct checksum
- Failure probability: 1 in 16 million to 10 billion packets for 1526 bytes [Reference: [1] Stone et. al., When the CRC and TCP checksum disagree]
- Implies 1 undetected TCP corruption in 20GB to 1.2TB data, approximately
Strong checksum in Application?

- Performance overhead
  - Application data-structures different from network data-structures (e.g. B-tree data to fit into MTU)
  - “Interconnect or network” is the vulnerability, not the application
- End up reinventing transport protocol in application (over TCP!)
  - Handling retransmissions, in-order delivery, gaps, etc.
Ideal Solution

- Zero-copying: avoid multiple copies without checksums
- H/w redundancy for hardware failures
- Clean shutdowns on system failures
- Filesystem/block/disk checksums for disk reliability
- Bridging the integrity gap in network/interconnect
  - Protection in transport protocol
Why reinvent the wheel?

- RFC 2385: TCP MD5 Signature Option
- Implemented in Linux Kernel as TCP_MD5SIG socket option
- Linux implementation:
  - Efficient compute (uses kernel crypto-engine)
  - Retransmission on checksum mismatch
    - Implies seamless error-recovery
  - Reduces syscalls by calling /dev/crypto from within kernel. Thus, lesser copy_to_user/copy_from_user and smaller memory footprint.
Working of TCP_MD5SIG socketopt

- Both client and server must know each others’:
  - IP
  - Port
  - MD5 Key

before the connection is setup

- Client must bind() for the server to save the <IP,Port,MD5Key> mapping
TCP_MD5SIG over Socket

Server
10.2.1.5:7000
MD5 key = "foo"

socket()
bind(10.2.1.5:7000)
setsockopt(TCP_MD5SEG, 10.2.1.5:7000 => "foo")
listen()
accept()
read()
write()
write()
read()
close()

Client
10.2.1.22:8000
MD5 key = "foo"

socket()
bind(10.2.1.22:8000)
setsockopt(TCP_MD5SEG, 10.2.1.22:8000 => "foo")
connect()
read()
write()
close()

The MD5 key is known to both hosts prior to the connection.
TCP_MD5SIG works with IPv6 too.

TCP_MD5SIG option must be set before the server starts to listen. Also, the server can call the TCP_MD5SIG several times -- one per client. Each call saves the mapping of the client's endpoint to the client's MD5 key.

Save the Endpoint to MD5 key mapping.
Even the SYN is signed with MD5 key.

TCP_MD5SIG option must be set before the client does the connect. Client socket will call TCP_MD5SIG option only ones, since it will communicate with only one host, that is the server.

On either side, when the MD5 sign does not match, or the MD5 key is not found, the packet is dropped, leading to retransmission from the peer.
Evaluation: Latency

- **Round Trip Time (u-seconds)**
- **TCP payload (bytes)**
- **TCP payload (bytes):** 256, 512, 1024, 2048, 4096, 6144, 8192
- **MD5 Variations:**
  - no-MD5
  - TCP_MD5SIG
  - User MD5

Settings:
- 1 Thread - 1 Socket - 1 CPU
- 9000 bytes MTU
- 10Gbps Ethernet
Evaluation: Throughput (Single-threaded)

- Throughput (Gbps)
- TCP payload (bytes)

1 Thread - 1 Socket - 1 CPU
9000 bytes MTU
10Gbps Ethernet

- TCP_MD5SIG
- User MD5
Evaluation: Throughput - TCP_MD5SIG (Multi-threaded)

Throughput (Gbps)

TCP payload (bytes)

9000 bytes MTU
10Gbps Ethernet
60 CPU cores

- 10 threads
- 20 threads
- 30 threads
- 40 threads
- 50 threads
- 60 threads
Evaluation: Memory Footprint
(accessing /dev/crypto from userspace)

copy_from_user / copy_to_user
TCP_MD5SIG: 2 socket syscalls
User MD5: 2 socket syscalls
Use-Case: NVM Mirroring

Throughput (reqs/s)

TCP payload (bytes)

Throughput (Mbps)

TCP payload (bytes)
Conclusion

- Not a generic solution
  - First try other fits:
    - TCP checksum not good enough for the application?
    - Disk/filesystem checksum
    - Disk/flash mirroring

- But very effective for typical usecases
  - For line-speed mirroring of in-memory data:
    - Better throughput, memory footprint and same latency
    - Error detection and recovery seamless to application

- Future prospects: Persistent Memory
References


Thank-you!

Questions?