CSE221 Lecture 16

Aronya Baksy

November 2024

1 Implementing Remote Procedure Calls

1.1 Goals

- Make distributed computing easy with clean abstractions and powerful semantics of standard procedure calls
- Efficient implementation, without adding too much network overhead for RPC calls
- Secure communication, encrypted data in flight

1.2 RPC Structure

- Client and Server both have **stubs** and **RPC runtimes**.
- Client stub is responsible for placing a specification of the target procedure and the arguments into one or more packets and asking the RPC Runtime to transmit these reliably to the callee machine.
- server-stub unpacks args from the request message and calls the procedure on the server, then packages the return value into a response message
- RPC Runtime is responsible for retransmissions, acknowledgments, packet routing, and encryption.

1.3 RPC Binding

- two main questions: **naming** and **location**
- An interface name has 2 parts: *type* and *instance*
- Makes use of *grapevine* distributed database as a centralized store of all interfaces and locations
- Exporters of an interface publish to the grapevine DB, importers look up the database in the DB
- Grapevine can restrict access control to enforce which clients can export/import interfaces

1.4 Optimizations

- For **simple calls** where args fit in a single packet, no need of separate ACKs: piggyback the ACK along with the RPC response
- For **complex calls** requiring multiple packets, piggyback the ACK on the last response message, reduce total messages by 1
- Server maintains a pool of idle processes to serve incoming requests, use one process and return it to idle state (instead of forking new process each time there is a call)
- Maintain only stateless communication, allows for zero overhead of connection setup and teardown

2 Receive Livelock

2.1 Interrupts vs Polling

- **Polling**: user can control rate of polling, hence ensure constant load and maintain fairness between competing interfaces
- **Interrupts**: Work done proportionally increasess with input load, scales better for low input load, and lower latency compared to polling
- **MLFRR**: Max Loss Free Receive Rate, highest input load for which throughput of a well-designed system keeps up with the offered load
- Why deal with receive livelock - applications that have **high request rates** and **no rate limiting** (packet processing like routing/firewalls/VPNs/LoadBalancers), network monitoring sw and remote FS

2.2 What is Receive Livelock

- Behaviour when delivered throughput drops to zero while the input overload persists (ideally when input exceeds MLFRR, we want constant delivered throughput)
- no useful progress is being made, because some resource is entirely consumed with processing receiver interrupts

2.3 Solution

- Avoid livelock by:
  - Using interrupts only to initiate polling.
  - Using round-robin polling to fairly allocate resources among event sources.
  - Temporarily disabling input when feedback from a full queue, or a limit on CPU usage, indicates that other important tasks are pending.
  - Dropping packets early, rather than late, to avoid wasted work. Once a packet is received, try to process it to completion.
- Maintain high performance by
  - Re-enabling interrupts when no work is pending, to avoid polling overhead and to keep latency low.
  - doing as much work as possible in a kernel thread, rather than in the interrupt handler (keeping interrupt handlers light reduces interrupt sevice time)
  - Letting the receiving interface buffer bursts, to avoid dropping packets.
  - Eliminating the IP input queue, and associated overhead