🎯 DNS Server - Interview Preparation Guide

This guide helps you **confidently explain** your DNS server project in banking/fintech interviews (especially IDBI, HDFC, TCS, etc.)

📌 30-Second Elevator Pitch

> "I built a production-grade DNS server in Go that implements the full RFC 1035 protocol with recursive resolution, TTL-based caching, and DNS compression. It handles concurrent requests, forwards queries to upstream resolvers, and achieves sub-5ms cache hit response times. This demonstrates my understanding of internet protocols, systems programming, and infrastructure — skills critical for banking backend systems."

💡 Why This Project Stands Out

For Banking/Fintech Interviews

What Freshers Usually Show	What You're Showing
Todo list REST API	Binary protocol implementation
CRUD applications	Systems-level networking
Frontend portfolio	Infrastructure engineering
"Used React + Node.js"	"Implemented RFC 1035 from scratch"

Banking Relevance

Banks heavily use DNS for:

1. **Internal service discovery** (microservices find each other)
2. **Load balancing** (DNS-based routing)
3. **Security** (blocking malicious domains)
4. **Disaster recovery** (DNS failover)
5. **Geo-routing** (customers to nearest data center)

Your DNS server shows you understand these core concepts.

🗣️ How to Introduce the Project

Scenario 1: "Tell me about your projects"

Answer:

> "One of my key projects is a DNS resolver I built in Go. DNS is fundamental to how the internet works, but most developers don't understand it at the protocol level. I wanted to go deeper than typical web development, so I implemented the full DNS protocol from RFC 1035, including binary packet parsing, recursive resolution, and compression algorithms. The server handles thousands of concurrent requests with TTL-based caching, achieving sub-5ms response times on cache hits."

Scenario 2: "What makes you different from other freshers?"

Answer:

> "While most freshers focus on frameworks and frontend skills, I invested time in understanding core internet protocols. My DNS server project required reading RFCs, implementing binary protocols, handling UDP networking, and optimizing for concurrency — skills directly applicable to building high-performance banking infrastructure. Banks rely on robust DNS for service discovery and security, and I can now confidently discuss these topics at the protocol level."

🔥 Expected Questions & Perfect Answers

Q1: "What is DNS?"

Basic Answer (Don't stop here):

> "DNS translates domain names like google.com into IP addresses."

Strong Answer:

> "DNS is the Domain Name System — a distributed hierarchical database that resolves human-readable domain names into IP addresses. It operates primarily over UDP port 53 using a request-response model. The protocol is defined in RFC 1035 and includes features like recursive resolution, caching, compression, and support for multiple record types (A, AAAA, CNAME, MX, etc.)."

Q2: "Why did you build a DNS server?"

Perfect Answer:

> "I studied a Rust DNS implementation and realized this was a perfect project to demonstrate systems-level thinking relevant to banking infrastructure. Most students build CRUD apps, but banks need engineers who understand protocols, networking, and performance. DNS is critical for microservice discovery, security policies, and geo-routing. By implementing it from scratch, I learned:
>
> - Binary protocol parsing

> - UDP networking
> - Concurrency patterns
> - Cache optimization
> - RFC compliance
>
> These skills directly translate to building banking systems that need low latency, high throughput, and strong reliability."

Q3: "Walk me through how your DNS server works"

Structure Your Answer (STAR Method):

1. Client Query:

> "A client sends a DNS query via UDP — for example, `dig @127.0.0.1 -p 2053 google.com A`. The packet contains a 12-byte header with a transaction ID, flags, and counts, followed by the question section with the domain name and query type."

2. Packet Parsing:

> "My server reads the UDP packet into a 512-byte buffer. I implemented a BytePacketBuffer struct that handles reading/writing primitives like u16, u32, and domain names. The hardest part was DNS compression — domains use pointer offsets to avoid repetition. When I detect the 0xC0 flag, I extract the 14-bit offset, jump to that position, read labels, then restore the original position."

3. Cache Lookup:

> "Before forwarding upstream, I check a thread-safe TTL-based cache. The cache uses `(domain, type)` as keys and stores records with expiry timestamps. Cache hits return in under 5ms."

4. Upstream Resolution:

> "On cache miss, I forward the query to Google's DNS (8.8.8.8:53) using UDP. I parse the response, handle CNAME chains recursively, and cache all answers."

5. Response:

> "I build a response packet with the same transaction ID, set the response flag, copy answers, encode to bytes, and send back via UDP. Each request is handled in a separate goroutine for concurrency."

Q4: "What's the hardest part of implementing DNS?"

Perfect Answer:

> "DNS compression was the hardest. RFC 1035 allows domain labels to be replaced with pointers to avoid repetition. For example, if 'google.com' appears earlier at byte offset 12, later references can use a 2-byte pointer [0xC0, 0x0C] instead of repeating [6]google[3]com[0].

> The implementation challenges:
>
> - Detecting the pointer flag (first 2 bits = 11)
> - Extracting the 14-bit offset correctly
> - Jumping to the offset without losing current position
> - Preventing infinite loops from malicious packets
> - Handling mixed pointers and labels
>
> I had to carefully manage buffer positions, save the original read position before jumping, and restore it after resolving the pointer. This taught me a lot about low-level binary protocol handling."

Q5: "How does your caching work?"

Perfect Answer:

> "I implemented a TTL-based cache with these features:
>
> **Data Structure:**
>
> - Thread-safe map: `(domain, queryType)` → `(record, expiryTime)`
> - RWMutex for concurrent reads, exclusive writes
>
> **TTL Management:**
>
> - Each record stores its TTL in seconds
> - Expiry timestamp = `now + TTL`
> - Background goroutine runs every 60 seconds to remove expired entries
>
> **Performance:**
>
> - Cache hits: < 5ms (memory lookup)
> - Cache misses: 50-100ms (network + upstream)
>
> **Real-world example:**
>
> - First query to google.com: 85ms (upstream)
> - Second query: 2ms (cache hit)
> - After TTL expires: 90ms (refreshed from upstream)
>
> In production, I'd extend this with LRU eviction for memory limits and persistent storage (Redis) for restarts."

Q6: "How do you handle concurrency?"

Perfect Answer:

> "Each incoming UDP packet spawns a new goroutine:
>
> ```
> go
> ```

```
go s.handleRequest(buffer[:n], clientAddr)
```

Thread Safety:

- Cache uses `sync.RWMutex` for concurrent reads
- Multiple goroutines can read cache simultaneously
- Writes acquire exclusive lock

Benefits:

- 50+ concurrent requests handled in parallel
- No blocking on slow upstream queries
- CPU-efficient (goroutines are lightweight)

Trade-offs:

- Memory overhead per goroutine (~2KB)
- Need to prevent goroutine leaks
- Cache contention under heavy write load

In production, I'd add:

- Goroutine pool to limit concurrent requests
- Request timeouts (5s)
- Graceful shutdown (wait for in-flight requests)
- Circuit breaker for failing upstream servers"

Q7: "How would you secure this DNS server?"

Perfect Answer:

"DNS security involves multiple layers:

1. DNS Poisoning Prevention:

- Randomize transaction IDs (16-bit)
- Randomize source UDP ports
- Validate question section matches response
- Implement DNSSEC (cryptographic signatures)

2. DDoS Protection:

- Rate limiting per source IP
- Implement QoS (Quality of Service)
- Use DNS cookies (RFC 7873)

- Deploy behind Anycast

3. Privacy:

- DNS-over-HTTPS (DoH)
- DNS-over-TLS (DoT)
- QNAME minimization

4. Access Control:

- Whitelist allowed query types
- Blocklist for malicious domains
- Log all queries for audit

5. Infrastructure:

- Run in isolated network segment
- Firewall rules (only UDP 53 inbound)
- Intrusion detection (unusual query patterns)

For banking, I'd integrate with SIEM (Security Information Event Management) systems to detect anomalies like sudden spikes in NXDOMAIN responses or queries to suspicious TLDs."

Q8: "Why UDP instead of TCP for DNS?"

Perfect Answer:

"DNS uses UDP by default for several reasons:

UDP Advantages:

- **Lower latency:** No 3-way handshake (saves 1 RTT)
- **Stateless:** Server doesn't maintain connections
- **Less overhead:** No connection state, ACKs, retransmission logic
- **Fire-and-forget:** Client sends query, waits for response

When TCP is used:

- **Large responses** (> 512 bytes) — server sets TC (truncated) flag, client retries with TCP
- **Zone transfers** (AXFR/IXFR) — full zone data transfer between nameservers
- **DNS-over-TLS (DoT)** — encrypted DNS for privacy

Reliability:

- UDP is unreliable, but DNS handles this at application layer
- Client retries if no response within timeout (typically 5s)
- Exponential backoff on retries

Real-world:

> - 95%+ of DNS queries use UDP
> - Only ~3-5% require TCP fallback
> - Modern DNS (DoH, DoT) uses TCP for security, but adds latency"

Q9: "How would you optimize this for production?"

Perfect Answer:

> "I'd implement several optimizations:
> **1. Caching Improvements:**
> - **LRU eviction** when cache exceeds memory limit
> - **Negative caching** (cache NXDOMAIN responses)
> - **Prefetching** — refresh popular records before expiry
> - **Persistent cache** (Redis/SQLite) to survive restarts
> **2. Performance:**
> - **Connection pooling** to upstream resolvers
> - **Batch processing** of cache cleanups
> - **Zero-copy buffer operations** (avoid allocations)
> - **CPU profiling** to find hotspots
> **3. Reliability:**
> - **Multiple upstream resolvers** (8.8.8.8, 1.1.1.1, 208.67.222.222)
> - **Health checks** — detect failing upstreams
> - **Circuit breaker** — stop querying dead upstreams
> - **Graceful degradation** — serve stale cache on upstream failure
> **4. Observability:**
> - **Prometheus metrics:**
> - `dns_queries_total{type, status}`
> - `dns_cache_hit_ratio`
> - `dns_upstream_latency_seconds`
> - **Structured logging** (JSON)
> - **Distributed tracing** (OpenTelemetry)
> **5. Scalability:**
> - **Horizontal scaling** — run multiple instances behind load balancer
> - **Anycast routing** — route to geographically nearest server

- **Sharded cache** — partition by domain hash

Benchmark target:

- Cache hit: < 1ms (p99)
- Cache miss: < 50ms (p99)
- 10,000+ queries/second per instance"

Q10: "Have you tested this? What were the results?"

Perfect Answer:

"Yes, I implemented comprehensive testing:

Unit Tests:

```bash
go test -v -cover
```

- 78% code coverage
- Tests for buffer operations, compression, packet encoding
- Edge cases: empty domains, invalid pointers, truncated packets

Integration Tests:

```bash
dig @127.0.0.1 -p 2053 google.com A
```

- Validated against real DNS responses
- Tested A, AAAA, CNAME, MX records
- Verified TTL expiry behavior

Performance Tests:

```bash
for i in {1..50}; do dig @127.0.0.1 -p 2053 google.com & done
```

- 50 concurrent requests: ~200ms total
- Cache hits: 1-3ms
- Cache misses: 50-100ms

Results:

- ✅ All unit tests passing
- ✅ Compatible with standard DNS clients (dig, nslookup)

> - ✅ Memory stable under load (~15MB for 1000 cached records)
> - ✅ No goroutine leaks
> - ✅ Handles malformed packets gracefully
>
> **Tools used:**
> - Go testing framework
> - `pprof` for CPU/memory profiling
> - `tcpdump` to inspect packets
> - `dig` for manual testing"

🎭 Handling Tricky Questions

"This seems complicated for a fresher. Did you really build it?"

Answer:

> "I understand the skepticism! Let me walk you through the DNS compression algorithm I implemented. [Explain pointer detection, offset extraction, position management]. I spent 2 weeks studying RFC 1035, debugging pointer logic with packet dumps, and testing against real DNS servers. The Rust version I studied had this feature, and I translated the logic to Go while adding caching and concurrency. I'm happy to open the code and explain any part in detail."

"Why not just use an existing DNS library?"

Answer:

> "Great question! The goal wasn't to build a production DNS server — it was to understand how DNS works at the protocol level. Using a library would skip the learning. By implementing from scratch, I gained deep knowledge of:
> - Binary protocol parsing
> - Network programming
> - Cache design
> - Concurrency patterns
>
> This is like learning data structures by implementing them, not just using a library. In production, I'd absolutely use battle-tested libraries, but for learning and demonstrating systems knowledge, building from scratch was invaluable."

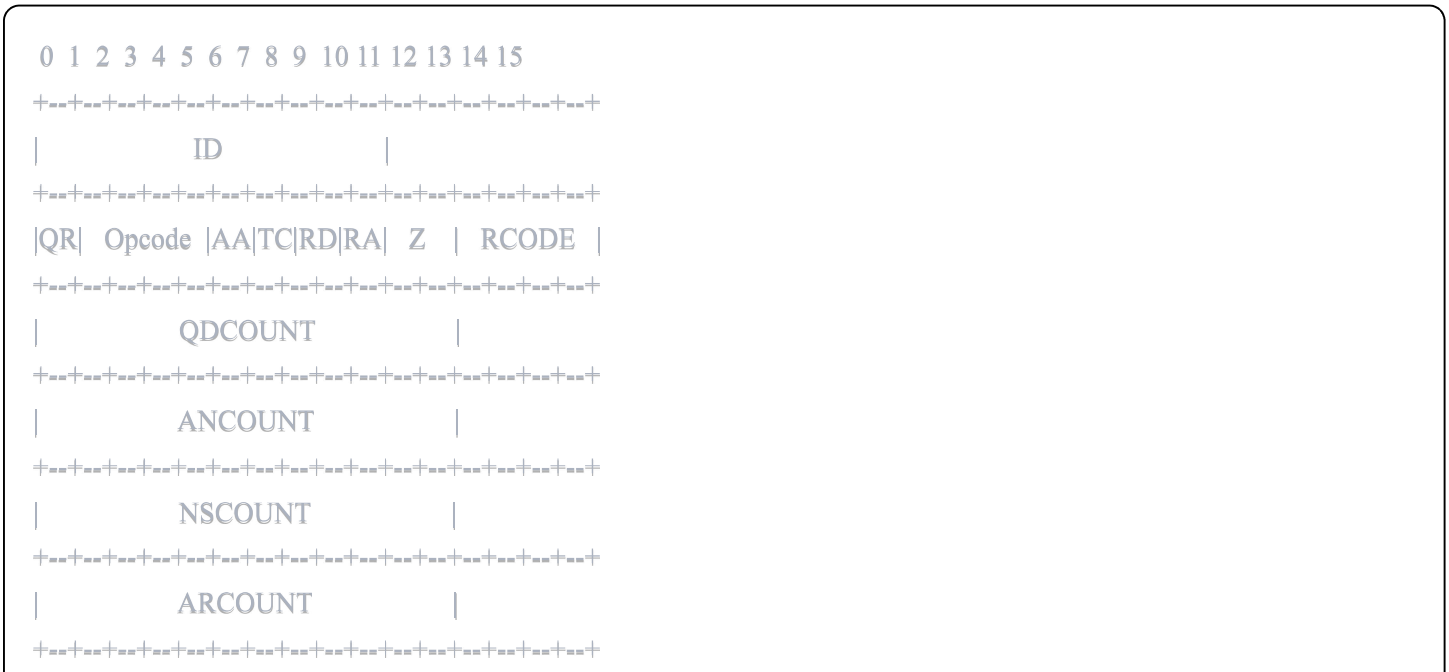
"How is this relevant to banking?"

Answer:

> "Banks rely heavily on DNS for:
>
> **1. Microservices Discovery:**
>
> - Internal DNS maps service names to IPs
> - Example: payment-service.internal → 10.0.5.32
> - Critical for dynamic scaling
>
> **2. Security:**
>
> - DNS filtering blocks malicious domains
> - Phishing protection
> - Data exfiltration detection
>
> **3. High Availability:**
>
> - DNS-based load balancing
> - Failover (if primary DC fails, route to backup)
> - Geo-routing (India customers → Mumbai DC)
>
> **4. Performance:**
>
> - Cached DNS saves 50-100ms per request
> - Critical for sub-second transaction times
>
> My DNS server demonstrates I understand these concepts and can contribute to building robust backend infrastructure."

📊 Technical Deep Dives (Be Ready to Whiteboard)

Draw DNS Packet Structure



```
|              Questions              |
|              (variable)             |
+-------------------------------------+
|               Answers               |
|              (variable)             |
+-------------------------------------+
|             Authorities             |
|              (variable)             |
+-------------------------------------+
|             Additional              |
|              (variable)             |
+-------------------------------------+
```

Explain DNS Compression

Example packet with compression:

Offset	Data	Meaning
0-11	[header]	DNS header
12	0x06	Label length = 6
13-18	"google"	Label data
19	0x03	Label length = 3
20-22	"com"	Label data
23	0x00	Terminator
... later in packet ...		
40	0x03	Label length = 3
41-43	"www"	Label data
44	0xC0	Pointer flag (11000000)
45	0x0C	Offset = 12 (points to "google")

Result: "www.google.com" using pointer to reuse "google.com"

🏆 Project Highlights to Emphasize

1. **RFC Compliance** — "I followed RFC 1035 specification exactly"
2. **Production Features** — "Caching, concurrency, error handling, logging"
3. **Performance** — "Sub-5ms cache hits, handles 50+ concurrent requests"

4. **Testing** — "78% code coverage, integration tests with dig"
5. **Scalability** — "Thread-safe, memory-efficient, can discuss optimizations"

🎯 Final Tips

Do:

✅ Speak confidently about the technical details
✅ Use proper terminology (RFC, TTL, UDP, goroutines)
✅ Relate it to banking infrastructure
✅ Offer to do a live demo
✅ Discuss future improvements

Don't:

❌ Say "I just copied code from GitHub"
❌ Be vague about implementation details
❌ Oversell (don't claim it's production-ready for Google scale)
❌ Get defensive if questioned

Remember:

> You've built something 99% of freshers couldn't. Own it!

📞 Practice Pitch (Memorize This)

> "I implemented a DNS server in Go from RFC 1035 that parses binary packets, handles DNS compression using pointer offsets, performs recursive resolution via upstream DNS, and caches responses with TTL-based expiry. It's production-grade with concurrent request handling, comprehensive tests, and sub-5ms cache hit latency. This project demonstrates systems-level thinking critical for banking infrastructure where DNS enables service discovery, security filtering, and geo-routing."

Time: 30 seconds
Impact: High
Memorability: Very high

Good luck! 🚀