Scalable Distributed Rate Limiter Design Documentation

1. Overview, Requirements, and Assumptions

A. Functional Requirements:

- Enforce per-user, per-IP, and per-API key rate limits (e.g., 1,000 requests per minute).
- Support global rate limits in addition to per-client limits.
- Allow burst capacity via token bucket algorithms while enforcing steady average limits.
- Provide immediate feedback to clients when the rate limit is exceeded (HTTP 429).

B. Nonfunctional Requirements:

- Low Latency: Each rate limit check should occur in sub-millisecond to a few milliseconds.
- High Throughput: The system must handle millions of requests per second and scale horizontally.
- High Availability: Must be fault tolerant with no single point of failure.
- Distributed Operation: Must operate globally with consistency across regions if needed.

C. Assumptions:

- The application handles massive traffic with millions of API calls per second.
- Rate limits may need to be applied at multiple dimensions (user, IP, API key, and global).
- A distributed in-memory data store (like Redis) is available for fast, atomic operations.
- Sticky session routing can be used if strong consistency is required across regions.

2. Rate Limiting Algorithms and Implementation

A. Algorithm Choice:

- Token Bucket: Allows bursts by maintaining tokens that are replenished at a steady rate. When a request arrives, a token is removed; if none are available, the request is rejected.
- Sliding Window: Optionally, a more precise sliding window counter can be used to smooth out spikes.

B. Implementation Mechanism:

- Use Redis for atomic operations via Lua scripts, which perform increments and TTL checks in a single transaction.
- The rate limiter runs as a distributed microservice, exposing an API (e.g., isRequestAllowed(clientID, currentTime)).

3. High-Level Architecture and Flow

A. API Gateway Integration:

- All API requests first hit the API Gateway, which then forwards them to the Rate Limiter Service for evaluation.

- The gateway uses HTTPS externally and calls the rate limiter service using gRPC/REST over secure channels.

B. Distributed Rate Limiter Service:

- This stateless microservice receives a client identifier and current timestamp, then performs an atomic check against a Redis cluster.
- Keys are sharded by clientID to distribute load across the Redis cluster.

C. Redis Cluster for Atomic Operations:

- Rate limit counters are maintained as keys with a TTL corresponding to the rate limit period.
- Lua scripts execute the check (current token count), decrement (or add tokens), and set the TTL atomically.

D. Response Flow:

- The Rate Limiter returns an allow/deny decision. In the case of denial, the API Gateway responds with an HTTP 429 error.
- Allowed requests proceed to the downstream application services.

4. Scalability, Fault Tolerance, and Distributed Considerations

A. Scalability:

- API Gateways and Rate Limiter services scale horizontally via autoscaling groups to meet traffic demand.
- Redis clusters are sharded using consistent hashing, enabling the system to handle millions of operations per second.

B. Fault Tolerance:

- The Redis cluster is replicated (replication factor of 3) to ensure data durability and availability.
- The rate limiter microservices are stateless and can be replaced on failure without loss of state.

C. Global Distribution:

- The system can be deployed in multiple regions; sticky sessions or global rate limit keys ensure that requests for the same client ID are directed to the same Redis shard if strong consistency is required.
- Eventually consistent designs might allow fast local rate limiting with periodic global reconciliation.

5. Protocols and External Infrastructure

A. Communication Protocols:

- Client-to-Server: HTTPS for incoming requests.
- Interservice Communication: gRPC or REST over secure TCP with mutual TLS ensures fast, secure message passing.

B. External Infrastructure:

- A Redis Cluster provides the in-memory data store needed for atomic rate limit operations.
- Global load balancers and DNS routing ensure that requests are directed to the nearest rate limiter instance.
- Monitoring tools (like Prometheus/Grafana) track request rates, latency, and errors for autoscaling and alerting.

6. Final Thoughts

This design for a distributed rate limiter is built to support extremely high traffic volumes in a very huge application. It features:

- Low latency through in-memory atomic operations using Redis and Lua scripting.
- Horizontal scalability by deploying the rate limiter as a stateless microservice, combined with a sharded Redis cluster.
- High availability and fault tolerance via replication and autoscaling.
- Flexibility to enforce rate limits on multiple dimensions (user, IP, API key, global) and support burst capacity with the token bucket algorithm.

By integrating this distributed rate limiter into an application's API gateway, the system can effectively prevent abuse, maintain fair resource usage, and scale to handle millions of requests per second across the globe.