kindred

Connection Pooling Demystified

Priit Piipuu

WHOAMI: Priit Piipuu

- Started as a DBA twenty years ago
- Currently database performance engineer at Kindred Group
- Main job is to help developers use Oracle technologies in best way possible
- Blog: https://priitp.wordpress.com, twitter: @PPiipuu

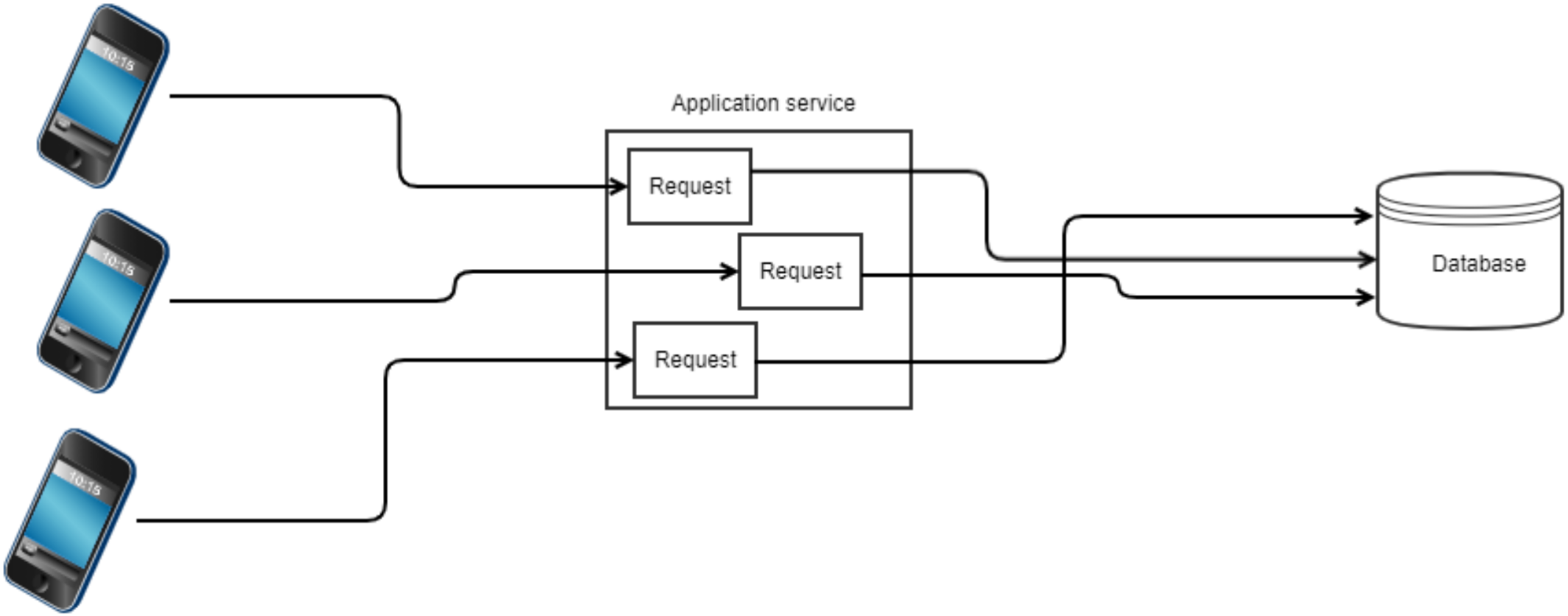
In this presentation…

- Why connection pooling
  - Problems it solves
  - Problems it creates
- Extra features in connection pools
  - Oracle Universal Connection Pool
  - Database Resident Connection Pool
- Connection pool sizing & configuration

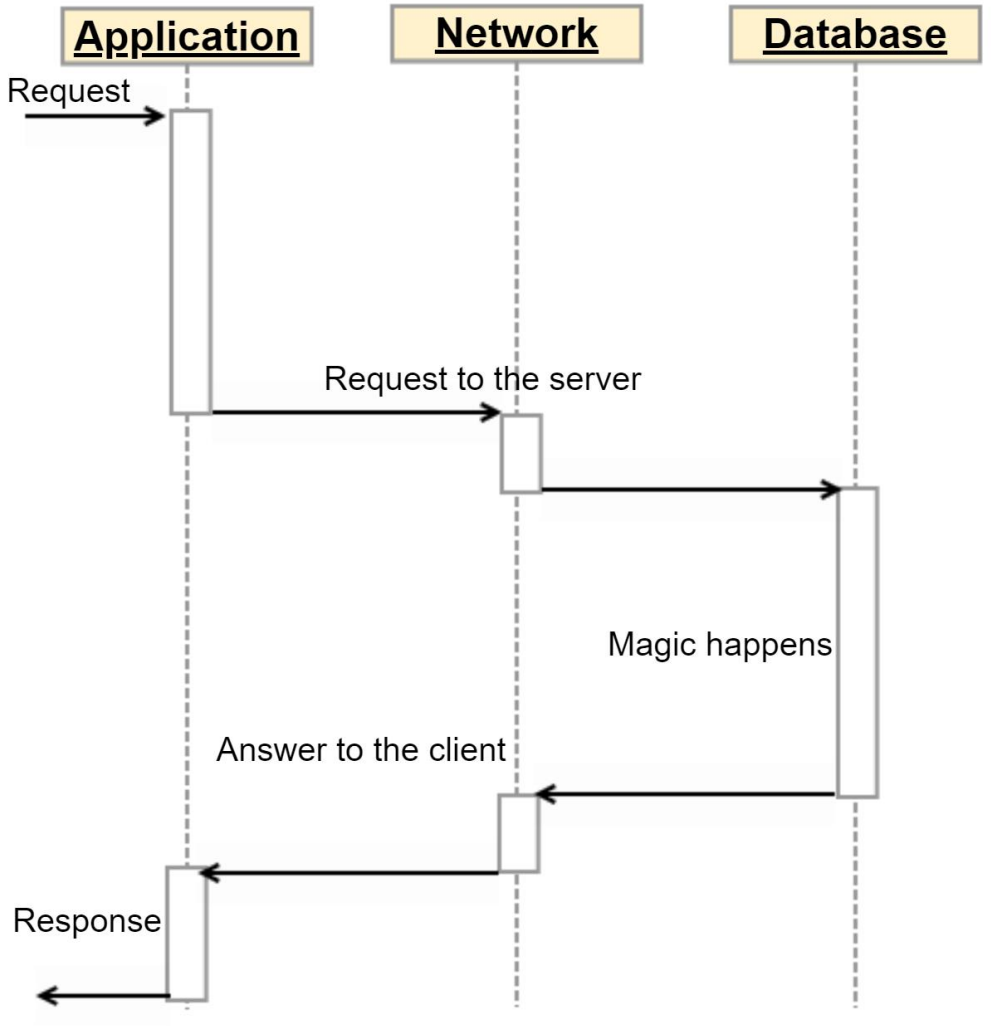
Modern application architecture

- Request-response model
  - One of the basic ways how computers communicate
  - Has been around for a while
  - One computer sends a request while second computer answers to that request
  - In simple cases communication is syncronous: think of typical web service
  - In case of web service
    - Large number of requests
    - Expectation of low response time

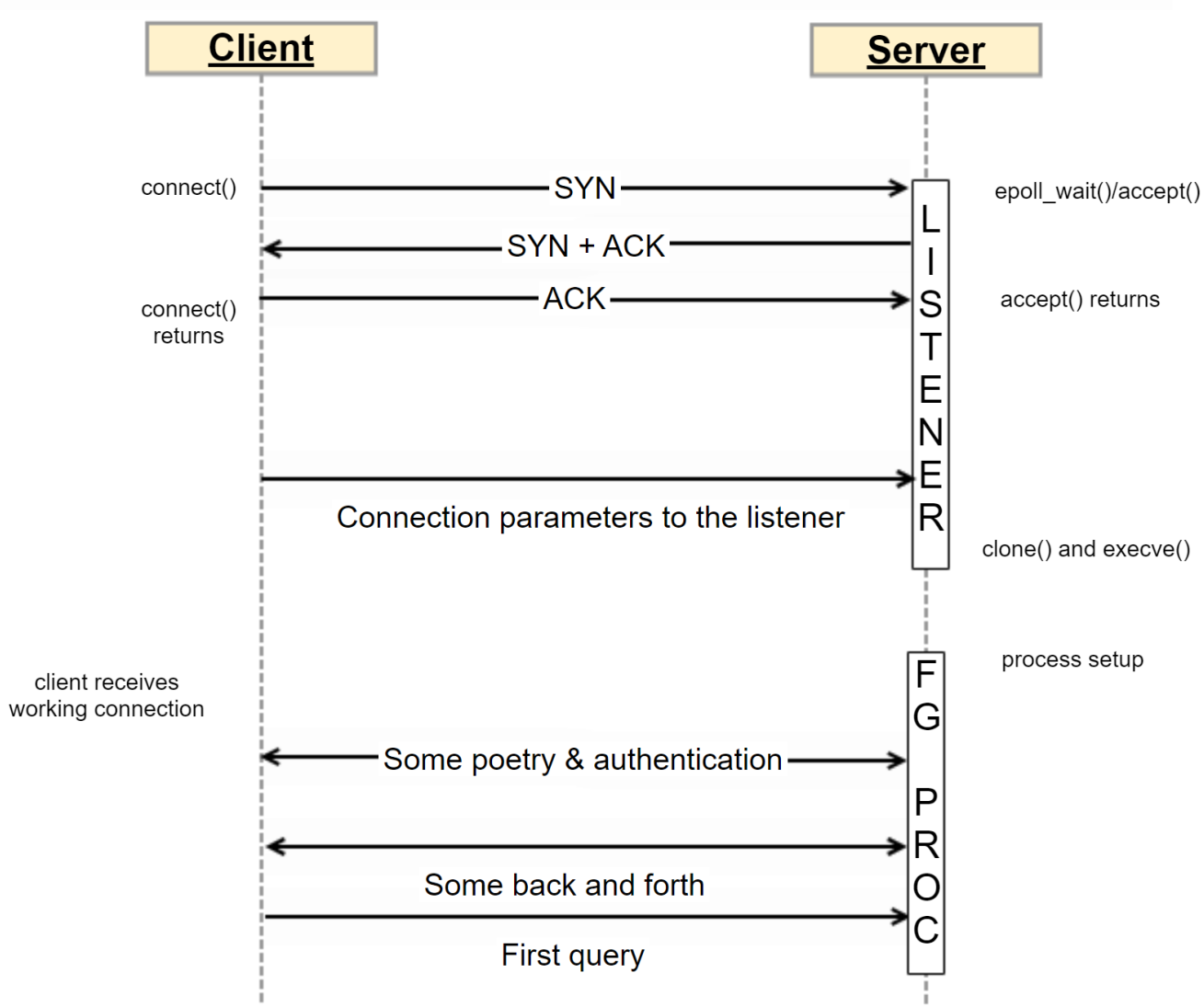
Request-response model



Request-response model



Birth of a connection



Is it worth it?

		Median	p99
Borrowing connection from UCP	Without connection validation	2.6us	13.5us
	With connection validation	562us	990us
Creating new connection	Local	46ms	56ms
	Remote	49ms	62ms
	Local with DRCP	19.9ms	28ms
	Remote with DRCP	22ms	33ms

Is it worth it?

		Median	p99
Connection close	UCP	9us	34us
	Local	780us	1ms
	Remote	1.1ms	2.5ms

- Test case
  - Single threaded, talks to single database instance and local listener
  - Private cloud, vm's in different networks
  - Opens and closes connection, builds histograms on response times
  - N = 1m

Long-running sessions

- Good
  - Reduces network connections
  - Predictable response times for fast queries
  - Cursor caching in the foreground process
  - Statement caching in application

- Bad
  - Network isn’t reliable
  - RAC: Troubles with planned downtimes, and no connection time load balancing

Dead connection detection (I)

- Lightweight connection validation in Oracle JDBC
  - Sends empty packet to the database
  - Does not wait for reply
  - Available since 18c, specific to JDBC thin
  - Connection.isValid method has timeout
- In UCP
  - Connection validation is by default off
  - When enabled, default is lightweight validation
    - Toggled with PoolDataSource.setValidateConnectionOnBorrow method
  - Lightweight validation is disabled when SQL statement for validation is set
    - PoolDataSource.setSQLForValidateConnection method
  - No timeout for lightweight connection validation

Dead connection detection (II)

- Fast Application Notification
  - Based on Oracle Notification Service
  - Notifies subscribers of service configuration and status changes
  - Needs Clusterware, in case of multiple clusters messages can be forwarded by GDS
  - Starting with UCP 12.2 FAN and FCF are enabled automatically
  - Starting with 19c, in-Band FAN for planned outages

- Fast Connection Failover
  - Planned outages: affected connections are closed gracefully
  - Unplanned events: affected connections are closed immediately

From developer’s perspective

- Similar API compared to vanilla JDBC or OCI
- Single point of configuration
  - Number of connections
  - Statement caching
- Automates some of the connection management tasks
  - Dead connection detection
  - Handling stale connections
- Extra features in UCP:
  - Support for Fast Application Notification and Fast Connection Failover
  - Runtime connection rebalancing and connection affinity
  - Application continuity

Connection pool as a circuit breaker

- With microservices slow requests will propagate to the upstream services
  - Worst case scenario: waiting on database or connection pool introduces cascading failure
- Query timeout for the rescue
  - Sets timeout on java.sql.Statement object
  - Timeout is in full seconds
- Not everything is query
  - Query timeout does not affect commits, metadata operations, connection creation
  - These can be handled by setting timeout on socket read (through java.sql.Connection)
  - Only way to achieve timeouts less than one second

Connection pool sizing(I)

- Connection pool limits max number of connections
  - Initial pool size: number of connections initially created, default is 0
  - Minimum pool size: minimum number of connections pool maintains
  - Maximum pool size: upper limit for the number of connections
  - Connection pool always tries to reach minimum pool size, unless minimum pool size is not reached
  - Limits are per connection pool instance. What about autoscaling?

- If max number of connections is reached, new requests will wait for connections to become available
  - Connection wait timeout: how many seconds application request waits for a new connection
  - Default is 3 seconds

Connection pool sizing (II)

- People like dynamically sized connection pools
  - Initial pool size/minimum pool size is set low
  - Maximum pool size is huge
  - During the rush hour application will create large number of new connections
  - Result is lousy response time and extra strain on systems
- Solution: use static connection pools
  - Initial pool size = minimum pool size = max pool size

Connection pool sizing (III)

- Little's law:
  - $L = \lambda W$
    - L -- average customers in stationary system,
    - λ -- average arrival rate
    - W -- is average response time
- Example:
  - Web service is expected to handle 600 requests per second on average (λ)
  - Average response time for database query is 15ms (W)
  - So on average there's 600*0.15 = 9 active database sessions
- Gives either minimum pool size or a good starting point for further tuning

Connection pool sizing (IV)

- Little's law and real world
  - Everything is long term average
  - It describes stationary process (average and variance do not change over time)
  - All requests are assumed independent, no variance due to the contention on common resources

Why so many idle connections?

- Example: transaction management in Spring Core
  - Transactional classes or methods are annotated with @Transactional
  - Easy to use, easy to get transaction demarcation wrong
  - When execution enters transactional method, database connection is allocated
  - Usually this is not the place where database connection is actually used
  - Result: many connections that are borrowed by the app, but idle

- Solution: LazyConnectionDataSourceProxy
  - Borrows connection only when it is actually used
  - Can reduce used connections significantly

Mitigation

- Resource manager
  - Can guarantee limited amount of CPU for important sessions
  - Active session pools: limits number of active sessions allowed for specific useg group
  - Limits amount of time session can be idle
- Oracle Connection Manager (cman)
  - Proxy Resident Connection Pool can reduce number of connections
- Database Resident Connection Pool (DRCP)
  - Can work with cman and UCP
- Esoterics
  - Shared servers
  - Threaded execution

DRCP on the client side (I)

- Connecion pool built into Oracle database

- For the developer
  - On client side connection type must be specified as POOLED
    - Example: jdbc:oracle:thin:@test.example.com:1521/testpdb.dbs:POOLED
  - Connection class name is set through oracle.jdbc.DRCPConnectionClass property

  - Server processes can be tagged, applications can attach to the tagged session
    - Application can control if DRCP can reuse the connection or not

DRCP on the client side (II)

- Application can specify if it wants a brand new session or reuses the old one
- Callback to fix database session state
- For execution queries, the foreground process needs to be reserved
  - oracle.jdbc.OracleConnection.attachServerConnection and detachServerConnection methods do that
  - UCP automates that
  - Same API is used with CMAN Proxy Resident Connection Pool as well

DRCP on server side (I)

- Default connection pool is created, but not started
- Managed through DBMS_CONNECTION_POOL package
- By default connection pools are created in CDB level
  - Can be managed on PDB level since 21c
- Does not support TCPS
- Has usual set of configuration parameters
  - MINSIZE and MAXSIZE
  - SESSION_CACHED_CURSORS -- # of cursors to cache in pooled session
  - INACTIVITY_TIMEOUT – how long pooled server can stay idle before terminated
  - MAX_THINK_TIME – how long client can stay idle before client connection is terminated
  - MAX_TXN_THINK_TIME – how long client with open transaction can stay idle

DRCP on server side (II)

- Database clients connect and authenticate with connection broker (CMON)
- CMON manages pooled foreground processes in DB instance
- When the client wants to do something, it asks for a process
- Broker allocates pooled process and hands it to the client
- Client talks directly to the pooled process
- When the pooled process is in use, it behaves like an ordinary foreground process
- Pooled process is handed back to the broker when the client is done