Connection Strings Demystified

A Deep Dive into Network Timeouts and TNS Internals

SOUG Day 1/2024

2024-04-17

Christoph Lutz

swisscom

Oracle SQLNet Timeouts ...

SQLNET.SEND_TIMEOUT

SQLNET.DOWN_HOSTS_TIMEOUT

TCP.CONNECT_TIMEOUT

TRANSPORT_CONNECT_TIMEOUT

NAMES.LDAP_CONN_TIMEOUT

CONNECT_TIMEOUT

SQLNET.INBOUND_CONNECT_TIMEOUT

SQLNET.RECV_TIMEOUT

INBOUND_CONNECT_TIMEOUT_listener_name

SQLNET.OUTBOUND_CONNECT_TIMEOUT

TNS Connection Strings – How do the Timeouts and Options Work?

```
MY_TEST.WORLD =
  (DESCRIPTION =
   (TRANSPORT_CONNECT_TIMEOUT=4)(CONNECT_TIMEOUT=9)(ENABLE=BROKEN)
    (ADDRESS_LIST =
       (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan01.mydomain.net)(PORT = 1521))
    )
    (ADDRESS_LIST =
       (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan02.mydomain.net)(PORT = 1521))
    )
    (CONNECT_DATA =
       (SERVICE_NAME = MY_TEST_RW.WORLD)
    )
  )
```

How do these timeouts really work?

Is there anything special about ADDRESS_LISTs?

Are these timeouts cumulative or not?

Are they for SCAN and node listeners?

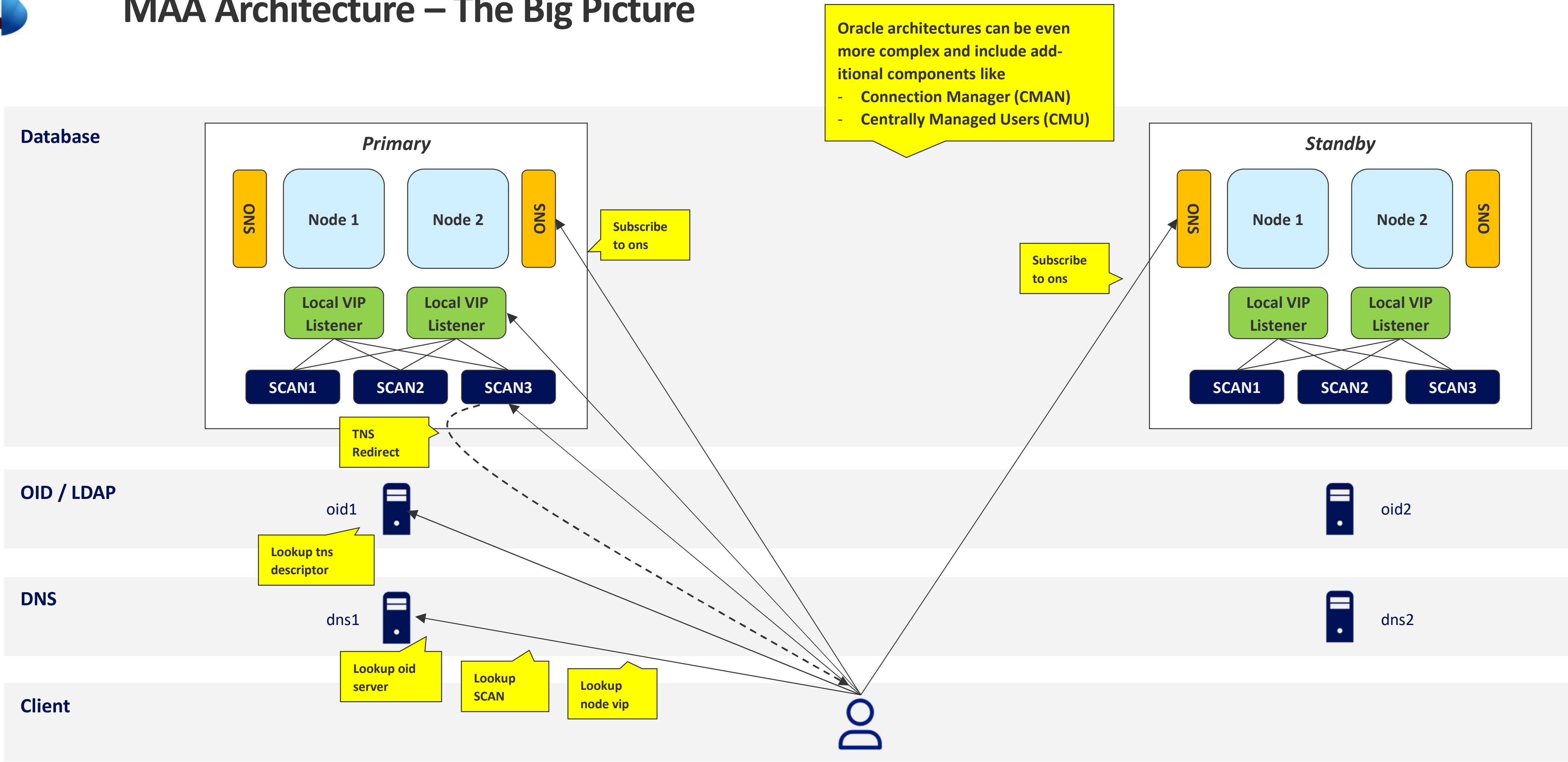
A Word of Caution

This presentation covers low-level internal and undocumented behavior.

This means:

- Things can and will change across different versions and patch levels
- My observations, findings and interpretations may be inaccurate or wrong
- Use the techniques shown in this presentation at your own risk!
- All examples were tested with **Oracle 19.20 on OEL 8.8**, other versions may differ!

MAA Architecture – The Big Picture



DNS Quirks & Oddities

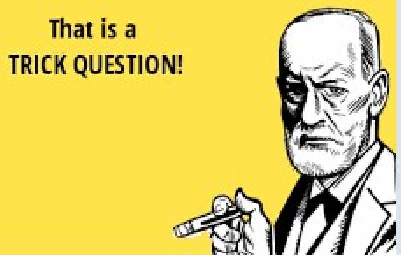
DNS Lookups – How Many Requests Will We Get?

Example Connection String

```
MY_TEST.WORLD =
  (DESCRIPTION =
   (FAILOVER=ON)(TRANSPORT_CONNECT_TIMEOUT=4)(CONNECT_TIMEOUT=9)(ENABLE=BROKEN)
       (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan01.mydomain.net)(PORT = 1521))
       (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan02.mydomain.net)(PORT = 1521))
    (CONNECT_DATA =
       (SERVICE_NAME = MY_TEST_RW.WORLD)
    )
  )
```

```
/etc/resolv.conf

options attempts:2
options timeout:4
…
nameserver 1.2.3.4
nameserver 4.3.2.1
```

a) One Lookup?
b) Two Lookups?
c) Three Lookups
d) More than three lookups?
e) It depends… on what?!



Answer

It depends!

(either 8 or 10)

- **my-scan01: 6 lookups**
- **my-scan02: 2 lookups**
- **(db node : 2 lookups)**

There are various reasons for this that will be explained in the following slides.

How To Analyze DNS Lookup Requests?

1. Tcpdump

```
tcpdump --immediate-mode -i any -nn -v host "(my-scan01 or my-scan02) and udp and port 53"
```

2. Wireshark / Tshark

```
tshark -i any                                          \
-n                                                     \
-l                                                     \
-T fields                                              \
-e dns.qry.name -f 'src port 53'                       \
-Y 'dns.qry.name contains "my-scan01" or               \
    dns.qry.name contains "my-scan02"'
```

We'll use this filter to analyze the behavior of SCAN lookups.

DEMO

DNS Oddity #1

The output shows an interesting pattern and behavior, can you spot it?

Example Output

```
21:44:35.416008 xxx.xxx.xxx.xxx.19371 > 1.2.3.4.53: 16347+ A? my-scan01.mydomain.net. (42)
21:44:35.416029 xxx.xxx.xxx.xxx.19371 > 1.2.3.4.53: 38111+ AAAA? my-scan01.mydomain.net. (42)
21:44:35.420833 xxx.xxx.xxx.xxx.14217 > 4.3.2.1.53: 18490+ A? my-scan01.mydomain.net. (42)
21:44:35.420838 xxx.xxx.xxx.xxx.14217 > 4.3.2.1.53: 56612+ AAAA? my-scan01.mydomain.net. (42)
21:44:35.425665 xxx.xxx.xxx.xxx.46735 > 1.2.3.4.53: 11703+ A? my-scan02.mydomain.net. (42)
21:44:35.425671 xxx.xxx.xxx.xxx.46735 > 1.2.3.4.53: 44214+ AAAA? my-scan02.mydomain.net. (42)
21:44:35.661922 xxx.xxx.xxx.xxx.15511 > 4.3.2.1.53: 65229+ A? my-scan01.mydomain.net. (42)
21:44:35.661939 xxx.xxx.xxx.xxx.15511 > 4.3.2.1.53: 30144+ AAAA? my-scan01.mydomain.net. (42)
```

The Oracle client performs IPv4 and IPv6 DNS lookups. Why this?!

DNS Oddity #1 – Dual Stack IPv4 and IPv6 Lookups

```
int getaddrinfo(const char* hostname,
                const char* service,
                const struct addrinfo* hints,
                struct addrinfo** res);
```

```
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    socklen_t ai_addrlen;
    struct sockaddr* ai_addr;
    char* ai_canonname;
    struct addrinfo* ai_next;
};
```

ai_family:
Specifies the desired address family.

Valid field values:

- **AF_INET (IPv4)**
- **AF_INET6 (IPv6)**
- **AF_UNSPEC (any)**

AF_UNSPEC indicates that **getaddrinfo**() should return socket addresses for any address family (either IPv4 or IPv6).

**The getaddrinfo behavior is driven by the flags set in the application.
Disabling the OS IPv6 network stack (ipv6.disable=1) makes no difference!**

getaddrInfo:

getaddrinfo is a standard **libc function** that converts domain names, hostnames and IP addresses between human-readable text representations and structured binary formats for the Linux socket API.

See also "man getaddrinfo".

DNS Oddity #1 – Oracle Client getaddrinfo Options

BPF Tracing Script using uprobes

```
./snlinGetAddrInfo <pid>
```

Default

```
-> snlinGetAddrInfo
  -> getaddrinfo
       host: my-scan01.mydomain.net
       hints:
         ai_flags=2
           AI_CANONNAME
         ai_family=0
           AF_UNSPEC
  <- getaddrinfo: 0
<- snlinGetAddrInfo: 0
```

The Oracle client uses ai_family=AF_UNSPEC by default and this results in dual-stack DNS lookups!

IP=V4_ONLY

```
-> snlinGetAddrInfo
  -> getaddrinfo
       host: my-scan01.mydomain.net
       hints:
         ai_flags=2
           AI_CANONNAME
         ai_family=2
           AF_INET
  <- getaddrinfo: 0
<- snlinGetAddrInfo:
```

When the TNS IP=V4_ONLY clause is used, Oracle uses the AF_INET flag, which results in IPv4 lookups only.

Oracle Client getaddrinfo Options

ai_family

The Oracle client uses **AI_UNSPEC** by default and this is the reason why it performs dual-stack dns lookups.

DNS Oddity #1 – IP=V4_ONLY Option

```
MY_TEST.WORLD =
  (DESCRIPTION =
   (FAILOVER=ON)(TRANSPORT_CONNECT_TIMEOUT=4)(CONNECT_TIMEOUT=9)(ENABLE=BROKEN)
        (ADDRESS =
            (PROTOCOL = TCP)(HOST = my-scan01.mydomain.net)(PORT = 1521)(IP=V4_ONLY))
        (ADDRESS =
            (PROTOCOL = TCP)(HOST = my-scan02.mydomain.net)(PORT = 1521)(IP=V4_ONLY))
      (CONNECT_DATA =
        (SERVICE_NAME = MY_TEST_RW.WORLD)
      )
  )
```

IP=V4_ONLY

The connection string on the left will result in a total of **four** DNS lookups:

- **my-scan01: 3 lookups**
- **my-scan02: 1 lookups**

DEMO

DNS Oddity #2

Even without IPv6 lookups we still observe more lookups than expected!

Example Output

21:44:35.416008 xxx.xxx.xxx.xxx.19371 > 1.2.3.4.53: 16347+ A? my-scan01.mydomain.net. (42)

~~21:44:35.416029 xxx.xxx.xxx.xxx.19371 > 1.2.3.4.53: 38111+ AAAA? my-scan01.mydomain.net. (42)~~

21:44:35.420833 xxx.xxx.xxx.xxx.14217 > 4.3.2.1.53: 18490+ A? my-scan01.mydomain.net. (42)

~~21:44:35.420838 xxx.xxx.xxx.xxx.14217 > 4.3.2.1.53: 56612+ AAAA? my-scan01.mydomain.net. (42)~~

21:44:35.425665 xxx.xxx.xxx.xxx.46735 > 1.2.3.4.53: 11703+ A? my-scan02.mydomain.net. (42)

~~21:44:35.425671 xxx.xxx.xxx.xxx.46735 > 1.2.3.4.53: 44214+ AAAA? my-scan02.mydomain.net. (42)~~

21:44:35.661922 xxx.xxx.xxx.xxx.15511 > 4.3.2.1.53: 65229+ A? my-scan01.mydomain.net. (42)

~~21:44:35.661939 xxx.xxx.xxx.xxx.15511 > 4.3.2.1.53: 30144+ AAAA? my-scan01.mydomain.net. (42)~~

Even without IPv6 lookups there are still 4 lookups. How come?!

DNS Oddity #2 – Oracle snlinGetAddrInfo Function

MOS Doc ID 1449843.1 mentions something very interesting:

DNS requests to resolve the scan name are made by the Oracle Net function **snlinGetAddrInfo**.

When an **ADDRESS_LIST** is **not used** in the connect descriptor, snlinGetAddrInfo invokes **two separate DNS queries** during the connection process.
...
When **ADDRESS_LIST** syntax **is used** snlinGetAddrInfo will make **only one query**.

The client's DNS lookup behavior changes with an ADDRESS_LIST but the reason for this is unknown!

Source: My Oracle Support, Reducing Client Connection Delays When DNS Is Unresponsive (Doc ID 1449843.1)

snlinGetAddrInfo:

System **N**etworking **Lin**ux **getaddrinfo**

snlinGetAddrInfo seems to be a wrapper around the libc getaddrinfo function.

How To Make DNS Behave As Expected?

```
MY_TEST.WORLD =
  (DESCRIPTION =
   (FAILOVER=ON)(TRANSPORT_CONNECT_TIMEOUT=4)(CONNECT_TIMEOUT=9)(ENABLE=BROKEN)
      (ADDRESS_LIST =
        (ADDRESS =
            (PROTOCOL = TCP)(HOST = my-scan01.mydomain.net)(PORT = 1521)(IP=V4_ONLY))
      )
      (ADDRESS_LIST =
        (ADDRESS =
            (PROTOCOL = TCP)(HOST = my-scan02.mydomain.net)(PORT = 1521)(IP=V4_ONLY))
      )
      (CONNECT_DATA =
        (SERVICE_NAME = MY_TEST_RW.WORLD)
      )
  )
```

Only these settings result in the behavior we would expect!

The connection string on the left will result in a total of two DNS lookups:

- **my-scan01: 1 lookup**
- **my-scan02: 1 lookup**

DEMO

DNS – Expected Behavior

Example Output

```
21:44:35.416008 xxx.xxx.xxx.xxx.19371 > 1.2.3.4.53: 16347+ A? my-scan01.mydomain.net. (42)
21:44:35.416029 xxx.xxx.xxx.xxx.19371 > 1.2.3.4.53: 38111+ AAAA? my-scan01.mydomain.net. (42)
21:44:35.420833 xxx.xxx.xxx.xxx.14217 > 4.3.2.1.53: 18490+ A? my-scan01.mydomain.net. (42)
21:44:35.420838 xxx.xxx.xxx.xxx.14217 > 4.3.2.1.53: 56612+ AAAA? my-scan01.mydomain.net. (42)
21:44:35.425665 xxx.xxx.xxx.xxx.46735 > 1.2.3.4.53: 11703+ A? my-scan02.mydomain.net. (42)
21:44:35.425671 xxx.xxx.xxx.xxx.46735 > 1.2.3.4.53: 44214+ AAAA? my-scan02.mydomain.net. (42)
21:44:35.661922 xxx.xxx.xxx.xxx.15511 > 4.3.2.1.53: 65229+ A? my-scan01.mydomain.net. (42)
21:44:35.661939 xxx.xxx.xxx.xxx.15511 > 4.3.2.1.53: 30144+ AAAA? my-scan01.mydomain.net. (42)
```

ADDRESS_LIST and IP=V4_ONLY give the expected results

DNS Oddities – Summary

No ADDRESS_LIST, no IP=V4_ONLY

1. lookup my-scan01 IPv4
2. lookup my-scan01 IPv6
3. SCAN expansion my-scan01 IPv4
4. SCAN expansion my-scan01 IPv6
5. SCAN expansion my-scan02 IPv4
6. SCAN expansion my-scan02 IPv6
7. lookup my-scan02 IPv4
8. lookup my-scan02 IPv6
9. [lookup dbhost01-v IPv4]
10. [lookup dbhost01-v IPv6]

These lookups only occur when a hostname is used in the LOCAL_LISTENER setting.

IP=V4_ONLY

1. lookup my-scan01 IPv4
3. SCAN expansion my-scan01 IPv4
5. SCAN expansion my-scan02 IPv4
7. lookup my-scan02 IPv4
9. [lookup dbhost01-v IPv4]

ADDRESS_LIST and IP=V4_ONLY

3. SCAN expansion my-scan01 IPv4
5. SCAN expansion my-scan02 IPv4
9. [lookup dbhost01-v IPv4]

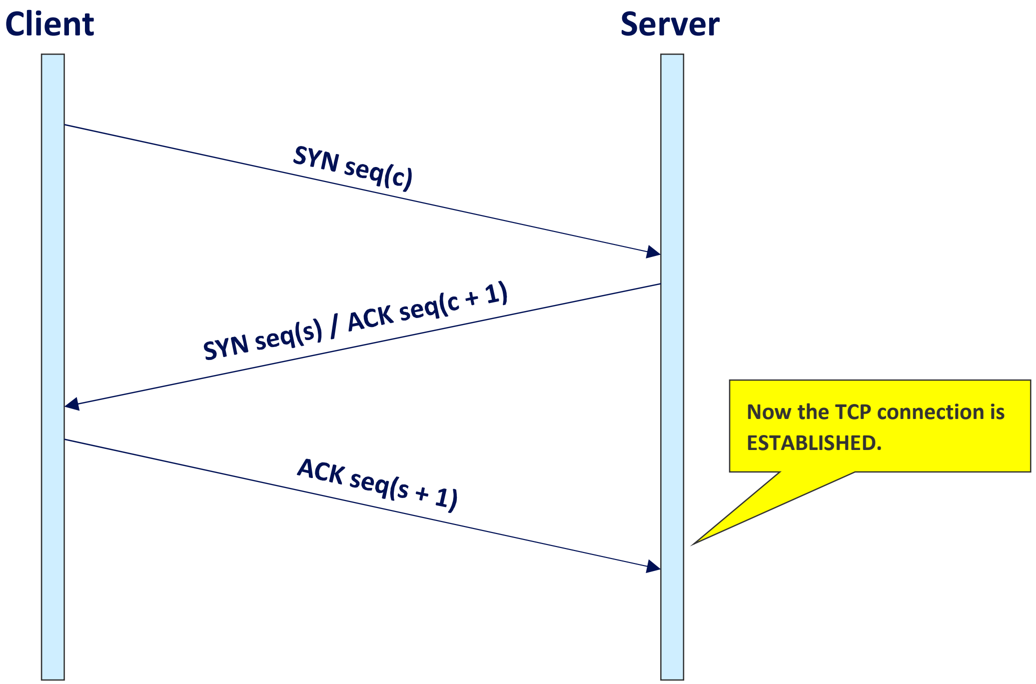
TCP Timeouts – New Connections

TCP Timeouts – New Connections

This section is about this setting!

```
MY_TEST.WORLD =
  (DESCRIPTION =
   (FAILOVER=ON)(TRANSPORT_CONNECT_TIMEOUT=4)(CONNECT_TIMEOUT=9)(ENABLE=BROKEN)
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan01.mydomain.net)(PORT = 1521))
      )
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan02.mydomain.net)(PORT = 1521))
      )
      (CONNECT_DATA =
        (SERVICE_NAME = TEST_SERVICE_RW.WORLD)
      )
  )
)
```

TCP Connection Establishment – Three-Way Handshake



Sequence Numbers

All bytes in a TCP connection are numbered with a sequence number and **the initial sequence number (ISN) is randomly chosen.**

The sequence number is the byte number of the first byte of data in the TCP packet sent (also called a TCP segment).

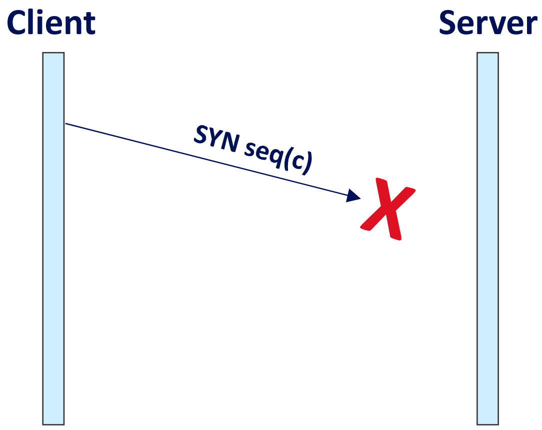
Acknowledgment Numbers

The acknowledgement number is the sequence number of the next byte the receiver expects to receive.

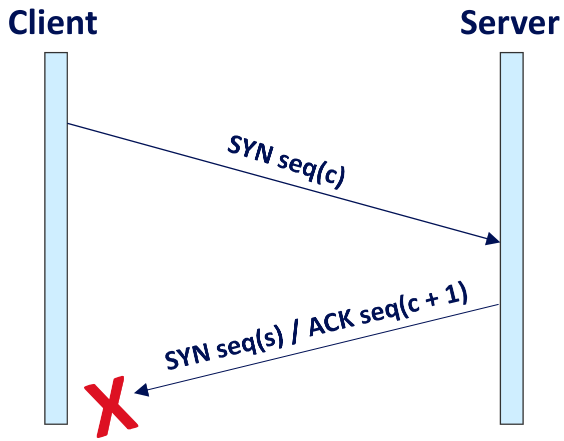
Acknowledgment of sequence number <n> means the receiver acknowledges the receipt of all bytes less than (not including) byte number <n>.

TCP Packet Loss – Initial RTO Problem Scenarios

SYN gets lost



SYN/ACK gets lost



From a client's perspective, there is no difference between SYN and SYN/ACK loss when a new connection is initiated.

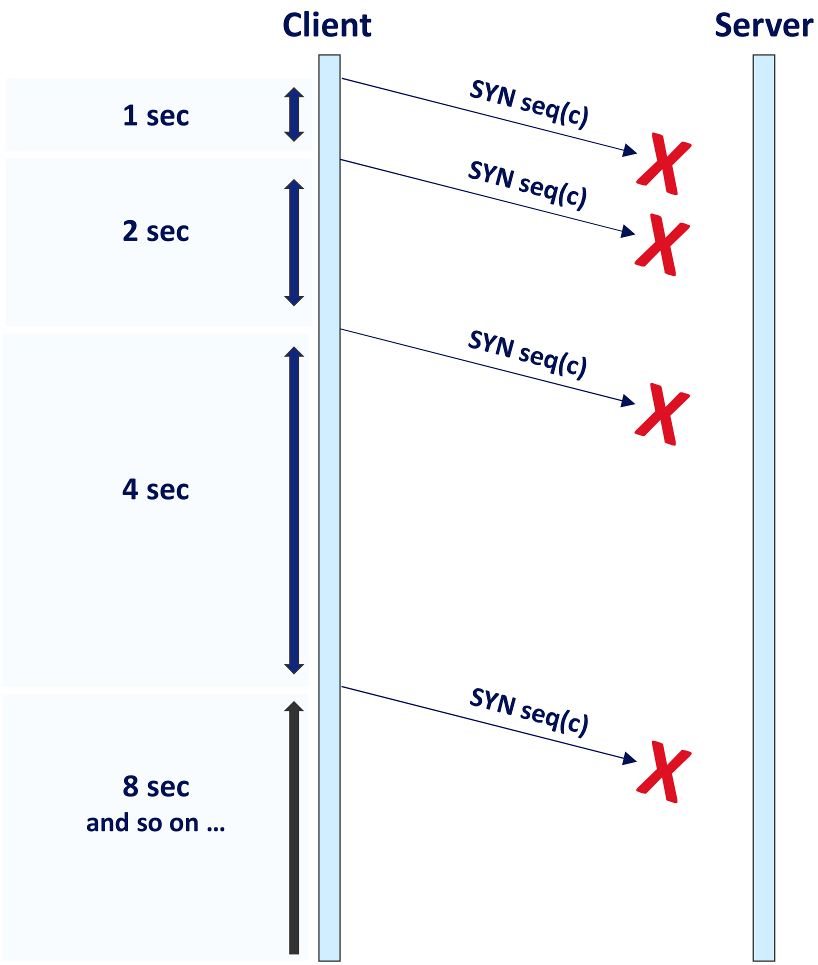
How should a client handle this kind of situation?

=> Retransmit the SYN after an initial Retransmit Timeout (RTO)

How do different operating systems handle this kind of situation?

- **Linux: 1 sec Initial RTO**
- **Windows: 1 sec initial RTO**

TCP Packet Loss – Initial Retransmit Timeout (RTO)



On Linux, the initial RTO is 1 sec* and the max number of SYN retries is de-fined by the following tunable that defaults to 6 on OEL 8:

net.ipv4.tcp_syn_retries

Linux uses an exponential backoff algo-rithm that doubles the timeout on every retransmission.

With an initial RTO of 1 sec and 6 re-tries, the timeout is:

1 + 2 + 4 + 8 + 16 + 32 + 64 = 127 sec

Or, more generally:

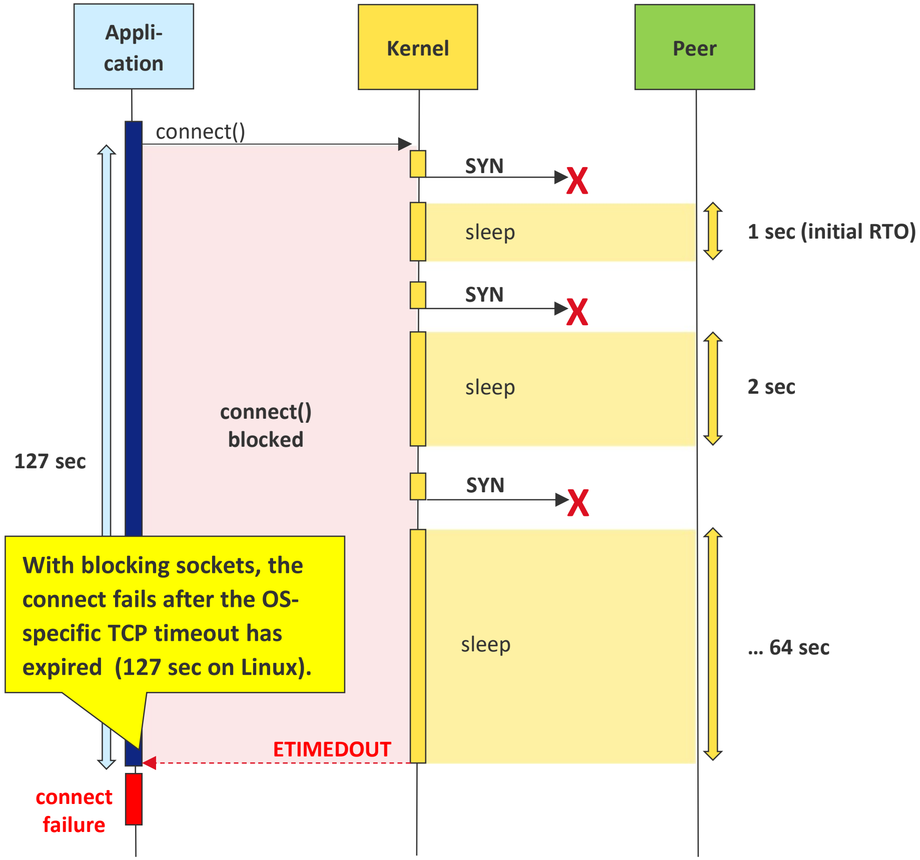
timeout = 2 ^ (tcp_syn_retries + 1) - 1

* Defined by kernel macro TCP_TIMEOUT_INIT in include/net/tcp.h:

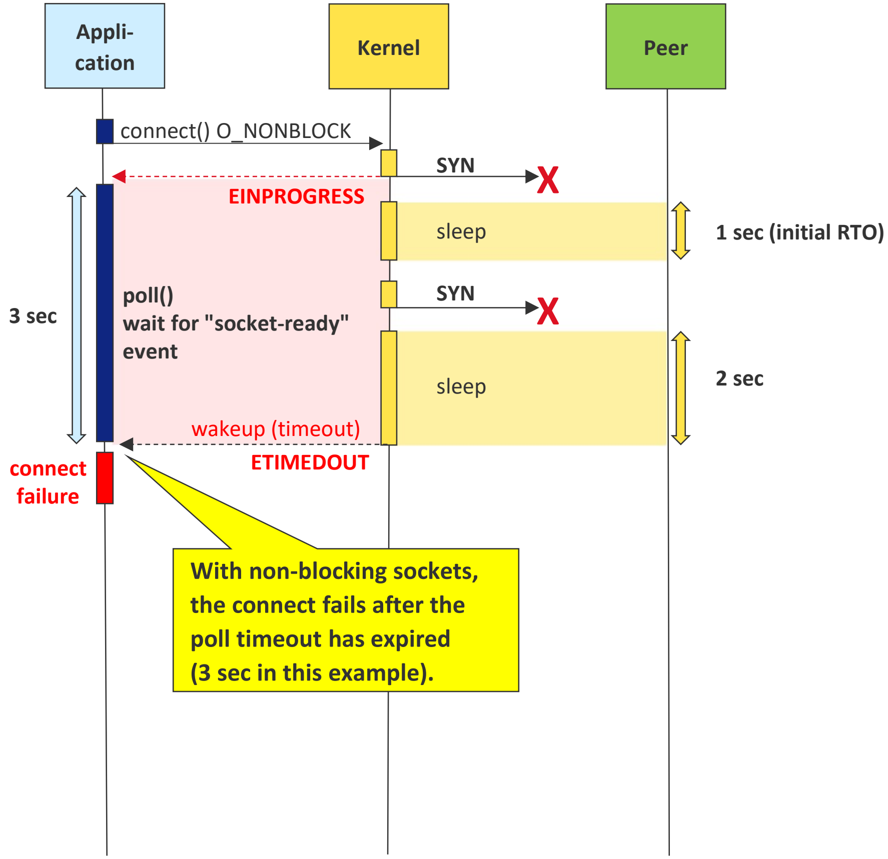
```
#define TCP_TIMEOUT_INIT
             ((unsigned)(1*HZ))
```

Blocking vs Non-Blocking Sockets

Blocking Socket



Non-Blocking Socket



TRANSPORT_CONNECT_TIMEOUT

Connection String

```
MY_TEST.WORLD =
 (DESCRIPTION =
  (FAILOVER=ON)(TRANSPORT_CONNECT_TIMEOUT=4)(CONNECT_TIMEOUT=9)(ENABLE=BROKEN)
   …
```

The TRANSPORT_CONNECT_TIMEOUT defines the time poll() should block.

System Calls (strace)

```
strace -e trace=fcntl,socket,connect,poll -p <sqlplus_pid>

…
fcntl(9, F_SETFL, O_RDONLY|O_NONBLOCK)  = 0

connect(9, {sa_family=AF_INET, sin_port=htons(1521),
        sin_addr=inet_addr("xxx.xxx.xxx.xxx")}, 16) = -1 EINPROGRESS
                                                     (Operation now in progress)

poll([{fd=9, events=POLLOUT}], 1, 4000) = 1 ([{fd=9, revents=POLLOUT}])
…
```

The timeout value used by poll() can be different from the Transport Connect Timeout due to processing delays or timer granularity effects! Refer to Appendix C for further details.

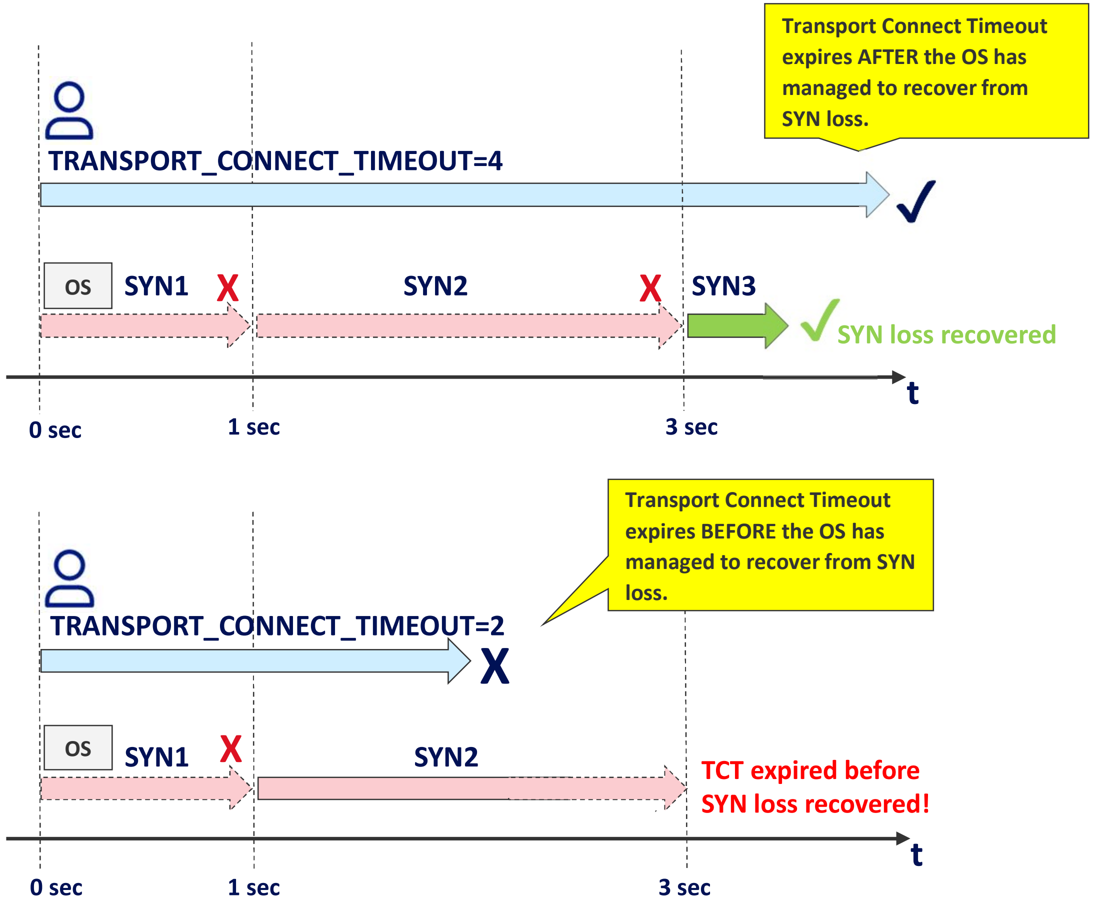
TRANSPORT_CONNECT_TIMEOUT

The TRANSPORT_CONNECT_TIMEOUT is the maximum time the Oracle client waits for a socket to become ready.

With OCI based clients on Linux, Oracle uses the **poll()** system call, so loosely speaking, the transport connect timeout is the **poll timeout**.

Implementations in other clients, li-braries (JDBC, ODP.NET) and operating systems may vary and may use differ-ent mechanisms (like timer threads).

Transport Connect Timeout and TCP Initial RTO



Use a Transport Connect Timeout with some headroom to recover from packet loss!

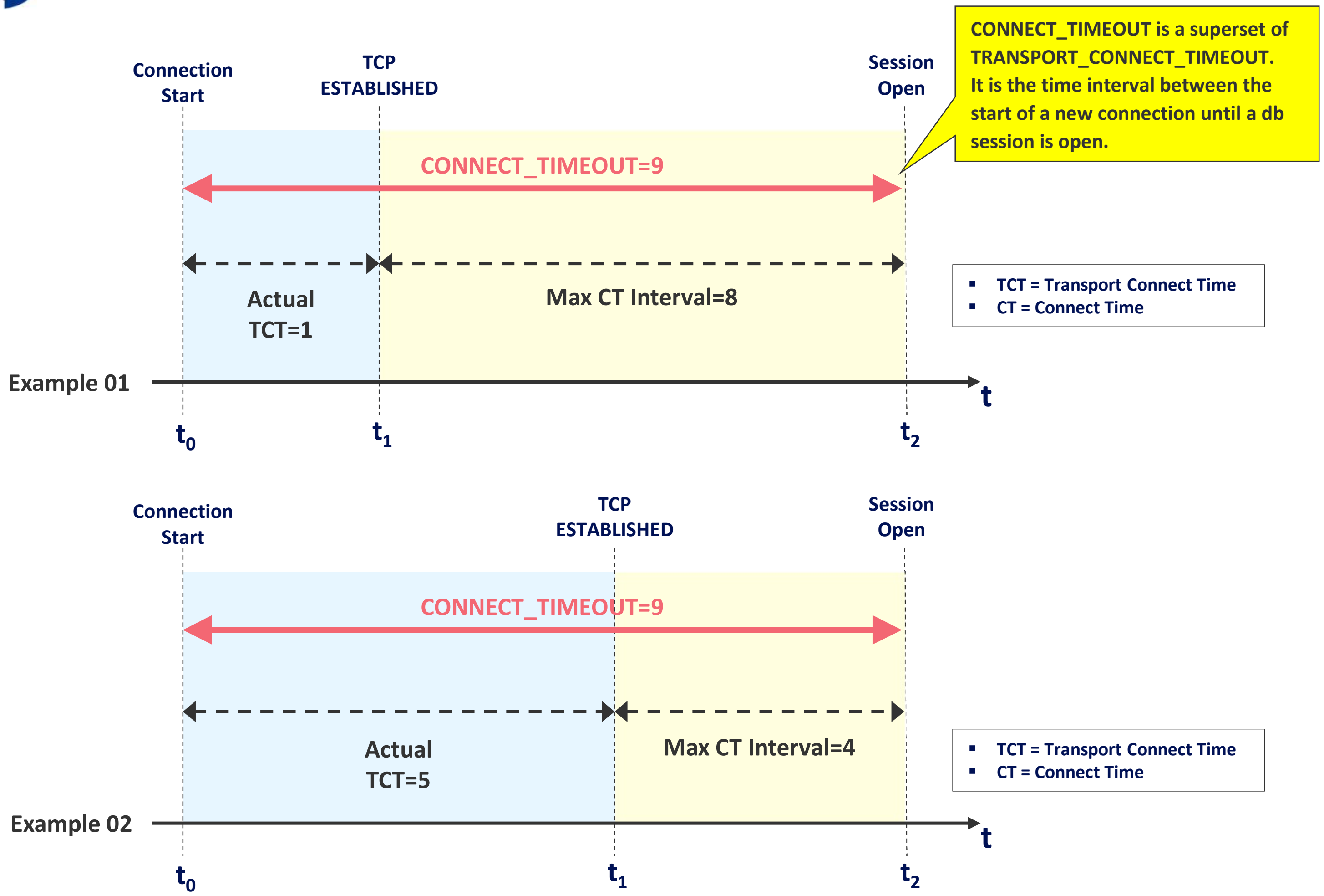
Transport Connect Timeout & TCP RTO

If the TRANSPORT_CONNECT_TIMEOUT expires before the initial TCP RTO had a chance to "recover" from a packet loss, the client will give up and cancel the connection attempt even though a TCP connection may have been established shortly after a successful retransmit.

Setting TRANSPORT_CONNECT_TIMEOUT to a value greater than the initial RTO on the OS side, will give a client **more head-room to recover from processing delay situations** (which may occur during tem-porary load bursts on the network or on the server side).

Of course, the ideal setting depends on the TCP defaults of the client and how gracefully it can handle packet loss errors.

Transport Connect Timeout and Connect Timeout



CONNECT_TIMEOUT is a superset of the TRANSPORT_CONNECT_TIMEOUT

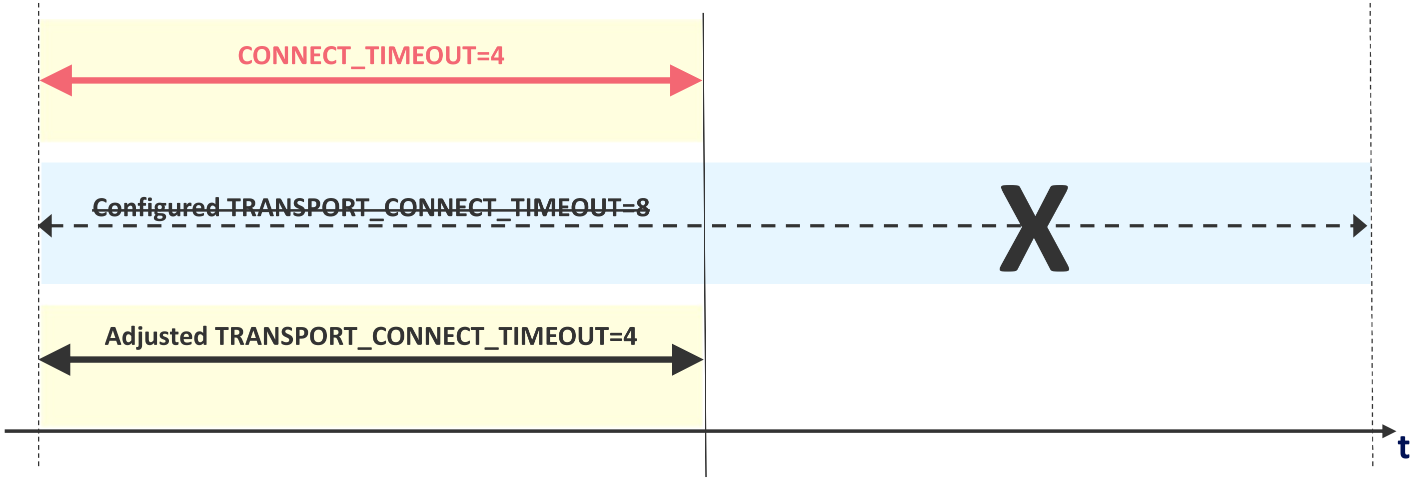
Connect Timeouts

The CONNECT_TIMEOUT is a superset of the TRANSPORT_CONNECT_TIMEOUT.

That means, the CONNECT_TIMEOUT defines the time interval between the start of a new connection request until a database session is open.

If the establishment of a new TCP connection incurs a delay, the delay time is subtracted from the CONNECT_TIMEOUT. In other words, **network delays during connection establishment, reduce the max CONNECT_TIME interval.**

Connect Timeout < Transport Connect Timeout



Connection String

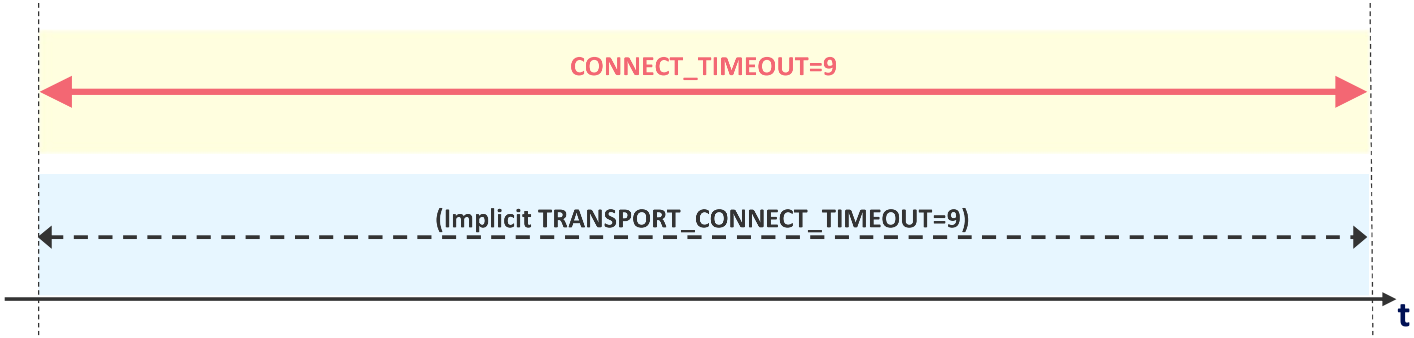
```
MY_TEST.WORLD =
   (DESCRIPTION =

 (FAILOVER=ON)(TRANSPORT_CONNECT_TIMEOUT=8)(CONNECT_TIMEOUT=4)(ENABLE=BROKEN)
     …
```

Connect Timeout < Transport Connect TO

When the Connect Timeout value is set lower than the Transport Connect Time-out value, Oracle will silently cap and adjust the Transport Connect Timeout value at run-time, so that:

Transport Connect TO = Connect Timeout

Transport Connect Timeout not Specified



Connection String

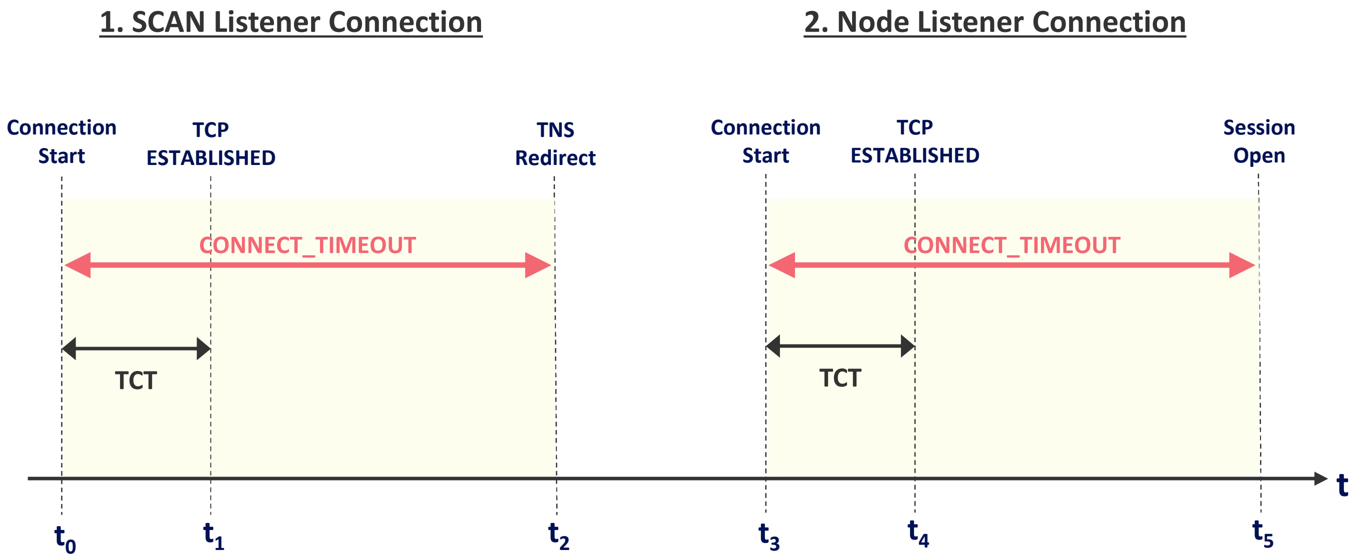
```
MY_TEST.WORLD =
   (DESCRIPTION =
  (FAILOVER=ON)(CONNECT_TIMEOUT=9)(ENABLE=BROKEN)
     …
```

Transport Connect Timeout not set

When the Transport Connect Timeout is not set, Oracle implicitly sets ist value to the value of the Connect Timeout:

Transport Connect TO = Connect Timeout

SCAN and Node Listener Timeouts



The timeouts apply on a per-connection basis. So, there are separate time-outs for the connection to the SCAN and the node listener.

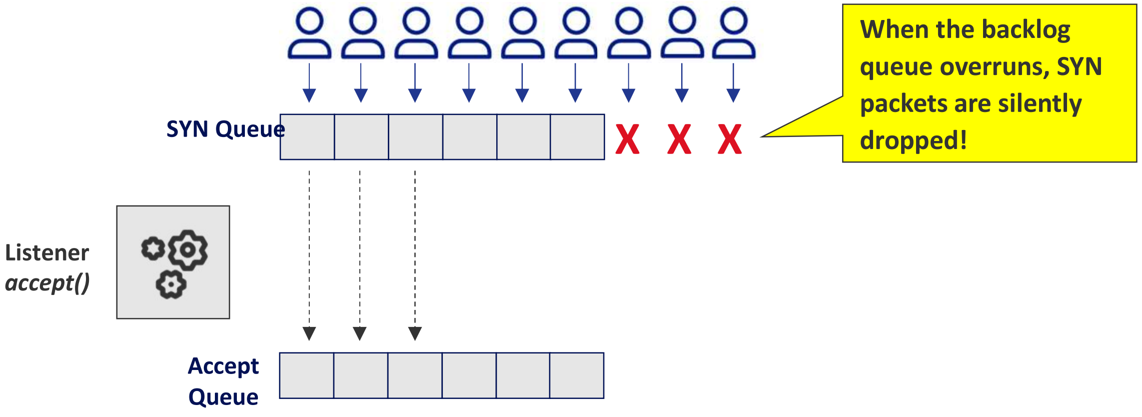
Transport Connect Timeout & Connect Timeout

The Oracle client opens new connections for connecting to the SCAN listener and the node listener.

The Transport Connect Timeout and the Connect Timeout settings therefore apply to every connection separately; that is, Oracle starts a new timeout before open-ing a connection to either a SCAN or a node listener.

Moreover, the client internally expands the SCAN and constructs an ADDRESS entry for every SCAN IP. **If a connection attempt results in a timeout, the client will retry and iterate over all SCAN IPs of all ADDRESS_LIST clauses** (s. also Appendix B, slides SCAN Host Expansion and Connection Attempts and Retries).

TCP Backlog Queue – Listener QUEUESIZE



In consolidated environments, it's highly recommended to increase the listener's default QUEUESIZE (to 1024 or higher).

Note: for SCAN listeners you must use the **TCP.QUEUESIZE** parameter in sqlnet.ora!

Listener QUEUESIZE

The size of the listener's TCP backlog queue is defined by the **QUEUESIZE** parameter and defaults to **128** on Linux.

This default may not be enough in the following situations:

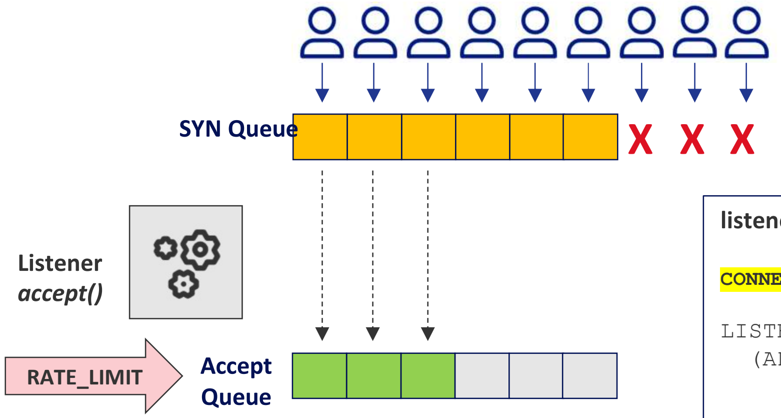
- **Connection bursts**
- **High system load**

When the TCP backlog queue fills up faster than the listener can complete accept() calls, the backlog queue will eventually overrun and SYN requests get dropped and silently ignored!

In such a situation, clients will retransmit their SYN requests.

For further information on how to configure the TCP backlog queue, refer to TCP Backlog Queue – Details in Appendix B

TCP Backlog Queue – Listener Rate Limit



listener.ora:

```
CONNECTION_RATE_LISTENER=50

LISTENER=
  (ADDRESS=
    (PROTOCOL=TCP)
      (HOST=mydbhost)(PORT=1521)(RATE_LIMIT=yes)
  )
```

The Rate Limit feature throttles the rate at which the listener accepts and processes new connections.

This will increase backpressure on the TCP backlog queue.

With rate limiting, you should also consider increasing the TCP backlog queue size!

Listener Rate Limit

The listener rate limit can protect from getting overloaded because of sudden connection bursts.

When a listener rate limit is active and the maximum number of concurrent connec-tions per second has been reached, the lis-tener will no longer call **accept()** to process new connections. New connections will therefore be queued in the listener's backlog queue.

So, when enforcing a listener rate limit, also consider increasing the listener's **QUEUESIZE**.

However, there's a limit to everything. While a system may be able to queue thousands of connection requests in the TCP backlog queue, it may not be able to process those requests fast enough (that is, before the client timeouts expire)!

TCP Backlog Queue – Monitoring (tcpsynbl.bt)

Listener QUEUESIZE 128 (Default)

```
@backlog[xxx.xxx.xxx.xxx, 1521, 128]:
[0]                    3 |@@                                                  |
[1]                    1 |                                                    |
[2, 4)                 3 |@@                                                  |
[4, 8)                 7 |@@@@@@                                              |
[8, 16)               18 |@@@@@@@@@@@@@@@@@                                   |
[16, 32)              19 |@@@@@@@@@@@@@@@@@@                                  |
[32, 64)              53 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[64, 128)             21 |@@@@@@@@@@@@@@@@@@@@                                |

11:20:12 Dropping a SYN to xxx.xxx.xxx.xxx :1521
11:20:12 Dropping a SYN to xxx.xxx.xxx.xxx :1521
11:20:12 Dropping a SYN to xxx.xxx.xxx.xxx :1521
…
```

TCP backlog queue is full, requests are getting dropped!

Listener QUEUESIZE 1024

```
@backlog[xxx.xxx.xxx.xxx, 1521, 1024]:
[0]                   13 |@@@@@@@@                                            |
[1]                    5 |@@@                                                 |
[2, 4)                 2 |@                                                   |
[4, 8)                 8 |@@@@@                                               |
[8, 16)               11 |@@@@@@@                                             |
[16, 32)              21 |@@@@@@@@@@@@@@                                      |
[32, 64)              48 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                    |
[64, 128)             77 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[128, 256)            73 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   |
```

TCP backlog queue is adequately sized.

How to monitor the TCP Backlog Queue?

With conventional tools, the utilization of the TCP backlog queue can only be monitored system-wide and not on a per-socket basis.

To monitor the TCP backlog queue on a per-socket based, use the BPF based **tcpsynbl.bt** script (or the slightly enhanced **tcpsynbl2.bt** script).

DEMO

Request Processing Cost – Listener Only

```
strace -c -p <listener_pid>
```

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 75.99    0.000861         861         1           wait4
  3.44    0.000039           9         4           epoll_wait
  3.27    0.000037           2        14         1 read
  2.65    0.000030           3        10           times
  2.12    0.000024           4         6           epoll_ctl
  1.50    0.000017           2         6           write
  1.50    0.000017          17         1           rt_sigaction
  1.50    0.000017           1        10           fcntl
  1.41    0.000016          16         1           accept
  1.32    0.000015           1         9           close
  1.15    0.000013           2         5           lstat
  1.06    0.000012           3         4           openat
  0.71    0.000008           4         2           getsockname
  0.62    0.000007           1         4           stat
  0.53    0.000006           3         2           setsockopt
  0.53    0.000006           3         2           getsockopt
  0.26    0.000003           3         1         1 getpeername
  0.18    0.000002           2         1           lseek
  0.09    0.000001           0         2           getpid
  0.09    0.000001           0         2           geteuid
  0.09    0.000001           1         1           gettid
  0.00    0.000000           0         2           pipe
  0.00    0.000000           0         1           clone
------ ----------- ----------- --------- --------- ----------------
100.00    0.001133          12        91         2 total
```

Listener Request Processing

Listener request processing is relatively light-weight and incurs a very moderate system call footprint.

Request Processing Cost – Listener + FG Process

```
strace -c -f -p <listener_pid>
```

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ------------------
 51.23    0.013822          30       454         2 read
  8.77    0.002366          10       223           mmap
  7.27    0.001962        1962         1           restart_syscall
  7.02    0.001894           5       371       112 openat
  3.53    0.000952           8       108           mprotect
  2.92    0.000787           2       272           close
  2.08    0.000562           9        59         7 ioctl
  1.83    0.000494         247         2           clone
  1.73    0.000467           3       132        21 stat
  1.56    0.000420         420         1           wait4
  1.05    0.000282           3        79        30 recvmsg
  1.01    0.000272           2       113           fstat
  0.86    0.000231           1       116           lseek
  0.79    0.000213           2        94           geteuid
  0.70    0.000190           4        41           sendmsg
  0.62    0.000166           3        49           poll
  0.50    0.000134           3        35           write
  0.43    0.000117           6        19        12 access
  0.42    0.000114           1        67           fcntl
  0.42    0.000113           2        41           rt_sigaction
  0.39    0.000106           1        63           getpid
  0.31    0.000083           1        45           rt_sigprocmask
  0.27    0.000074           8         9           madvise
  0.26    0.000069           3        22           brk
  0.23    0.000063           3        18         1 lstat
  0.23    0.000062           8         7           epoll_wait
  0.23    0.000062           3        17           epoll_ctl
  […]
------ ----------- ----------- --------- --------- ------------------
100.00    0.026980           9      2753       204 total
```

This can add up to a lot of unecessary work. Note that the number of system calls can vary across different releases (output from 19.20)

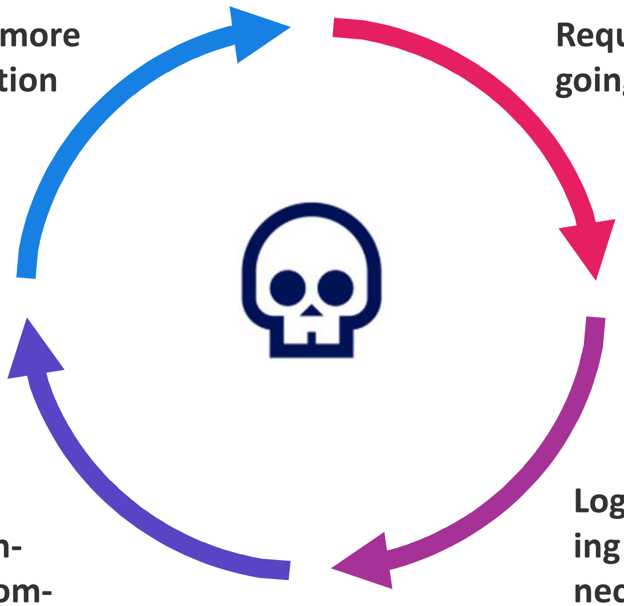
Spawning Dedicated Server Processes

Spawning a new dedicated FG process is a very expensive operation that incurs thousands of system calls!

If any of those system calls gets slowed down or delayed, the client may run into a connect timeout, which may look like a "network problem" at first glance even though the root cause could be something completely different!

If you open and close a new connection on every request, you're doing it wrong – completely wrong! This is a recipe for disaster!

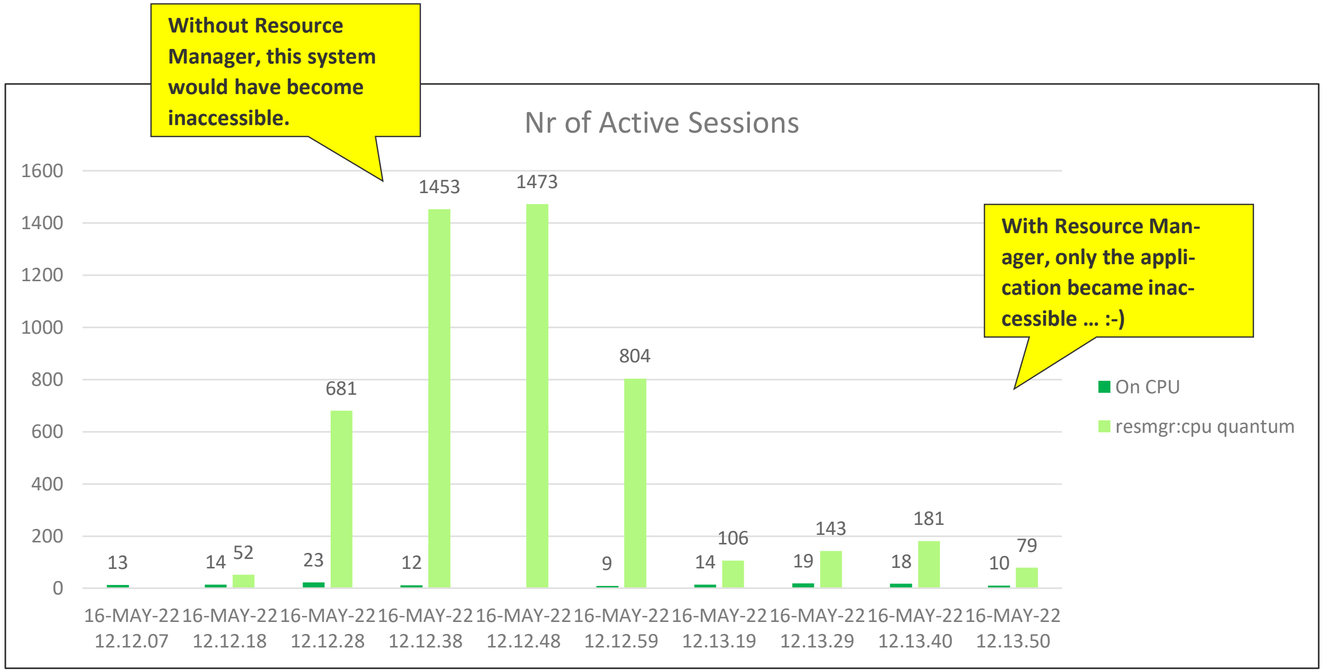
Connection & Logon Storms – Vicious Circle



Vicious Circle

Applications that spawn new con-nections to "compensate" for slowness can bring a system to its knees!

Connection & Logon Storms – Active Sessions



The Listener Rate Limiter feature (RATE_LIMIT) will only help in situations when lots of NEW connections are getting opened.

It will not help against existing database sessions (from a connection pool) that are already open and idle and that suddenly become all active at the same time!

Connection & Logon Storms

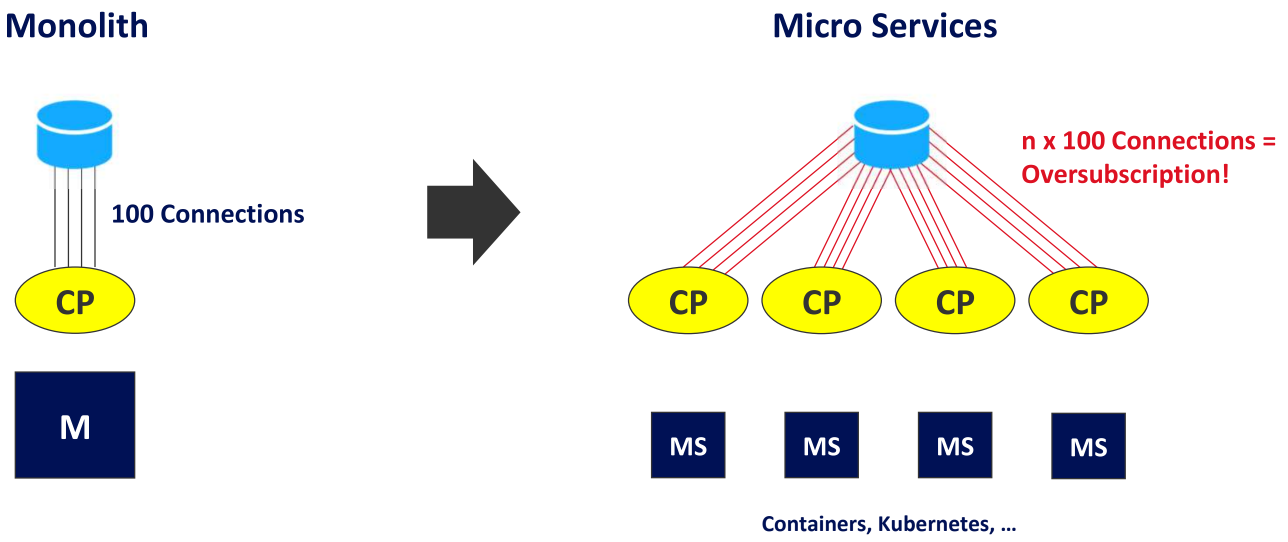
With connection pools, **Oracle recommends 1-10 connections per CPU core!**

Moreover, the Oracle Real-World Performance group recommends creating a static pool of connections to the data-base by setting the minimum and maximum number of connections to the same value.

This prevents connection storms by keeping the number of database connections constant to a predefined value.

Source: Oracle 21, Universal Connection Pool Developer's Guide, Section: About Optimizing Real-World Performance with Static Connection Pools

Connection & Logon Storms – Micro Services



If you split your application into multiple micro-services, you should split your connection pool into multiple "micro-pools" as well!

CPU Oversubscription

If an application is split into multiple micro services, make sure that the total number of all active connections / active sessions across all services does not result in a CPU oversubscription on the database system.

CPU oversubscription leads to significant performance problems and can even result in system crashes or node evictions!

The **database resource manager (dbrm)** can be used to protect the system from getting overloaded, but may not help to protect application users from getting bad performance and response times.

Recommendations:

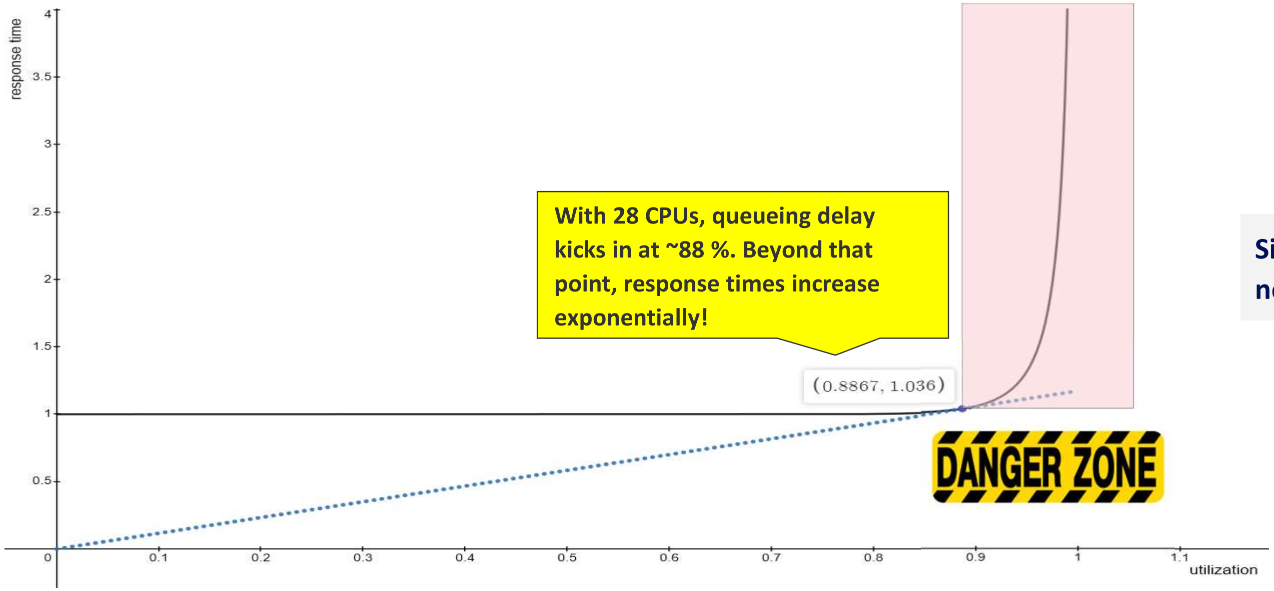
- Use a static connection pool (min = max number of sessions).
- Maximum 10 connections per cpu core on the database system

[Source: Oracle 21, Universal Connection Pool Developer's Guide, Section: About Optimizing Real-World Performance with Static Connection Pools]

CPU Oversubscription – The Knee in the Curve

```
--time-- procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
          r  b   swpd     free   buff      cache   si    so    bi    bo    in      cs us sy id wa st
...
12:11:06 39  0   8504 48798124  42288 215586016   0     0     6   137 69634 161953 62  4 34  0  0
12:11:11 43  0   8504 49408448  42288 215586752   0     0    16   515 72534 175330 79  5 16  0  0
12:11:16 78  0   8504 49316252  42288 215588448   0     0     6   504 91720 195323 81  8 10  0  0
12:11:21 71  0   8504 48885032  42288 215591136   0     0   268  1302 94536 202266 87  8  5  0  0
12:11:26 61  0   8504 49080848  42288 215594000   0     0     6   380 80673 193100 87  6  7  0  0
12:11:32 32  0   8504 49055484  42288 215596896   0     0   187  2163 75637 191474 84  6 10  0  0
12:11:37 38  0   8504 49038896  42288 215599872   0     0    10   253 79153 174189 79  5 16  0  0
12:11:42 53  0   8504 49026524  42288 215453760   0     0    10  1888 77789 171116 76  5 18  0  0
12:11:47 54  0   8504 48945376  42288 215454000   0     0     8   241 84896 193286 80  6 14  0  0
```

System busy!



Situations like this may result in con-nection drops and network timeouts!

Image source: https://www.desmos.com/calculator/cqh81xgspq

Connection & Logon Storms – System Statistics

logons cumulative

Shown as "logons" in AWR reports!

This statistic is incremented **every time a process starts.**

It includes non-user calls such as parallel query secondary calls, and job queue processes calls (in the case of Parallel Execution it will increment each time a new parallel worker process starts).

user logons cumulative

This statistic tracks **"real"** application/end user logons.

AWR Logons Statistic

In AWR reports, the Load Profile section shows a **logons** statistics.

This statistic represents **logons cumulative** and is therefore not a reliable indicator for the number of application/end users logons.

To track effective end user logons, use the statistics **user logons cumulative** and its complement **user logouts cumulative**.

Client and Server Side TNS Timeouts ("Connect Timeouts")

TNS Timeouts – Overview

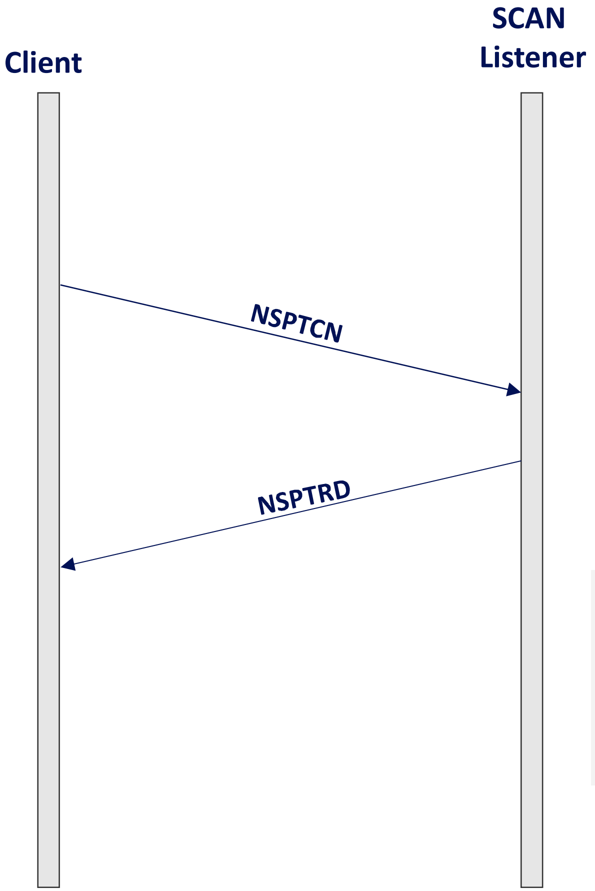
Client	tnsnames.ora: TRANSPORT_CONNECT_TIMEOUT sqlnet.ora: SQLNET.OUTBOUND_CONNECT_TIMEOUT
Listener	listener.ora: INBOUND_CONNECT_TIMEOUT_listener_name
Server Processes	sqlnet.ora: SQLNET.INBOUND_CONNECT_TIMEOUT

TNS Connect Timeout

This section is mainly about this setting!

```
MY_TEST.WORLD =
  (DESCRIPTION =
   (FAILOVER=ON)(TRANSPORT_CONNECT_TIMEOUT=4)(CONNECT_TIMEOUT=9)(ENABLE=BROKEN)
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan01.mydomain.net)(PORT = 1521))
      )
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan02.mydomain.net)(PORT = 1521))
      )
      (CONNECT_DATA =
        (SERVICE_NAME = MY_TEST_RW.WORLD)
      )
  )
)
```

TNS Handshake – SCAN and Node Listeners



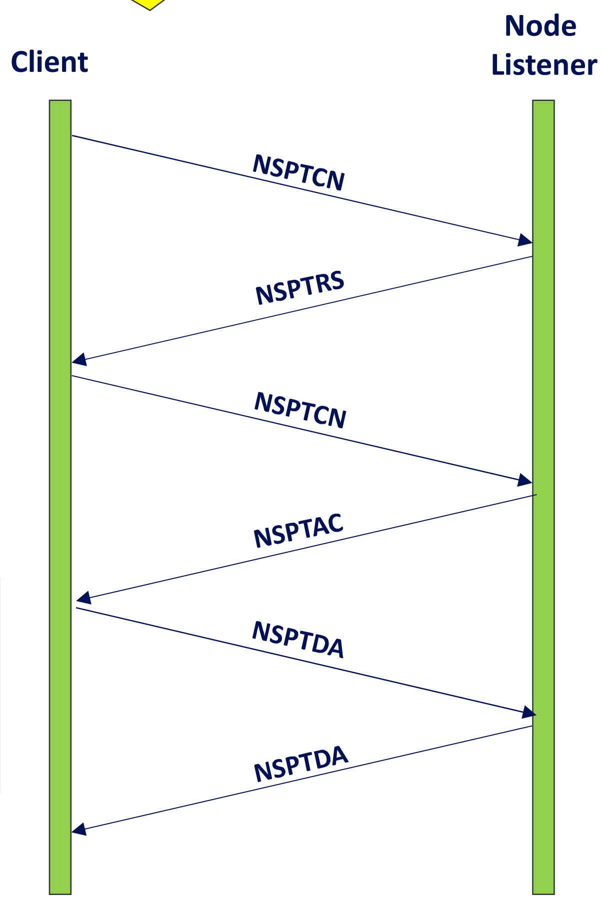
Packet Types

NSPTCN: Connect Packet
NSPTRD: Redirect Packet
NSPTRS: Resend Packet
NSTPAC: Accept Packet
NSPTDA: Data Packet

The SCAN listener's main job is to redirect clients to a node listener.

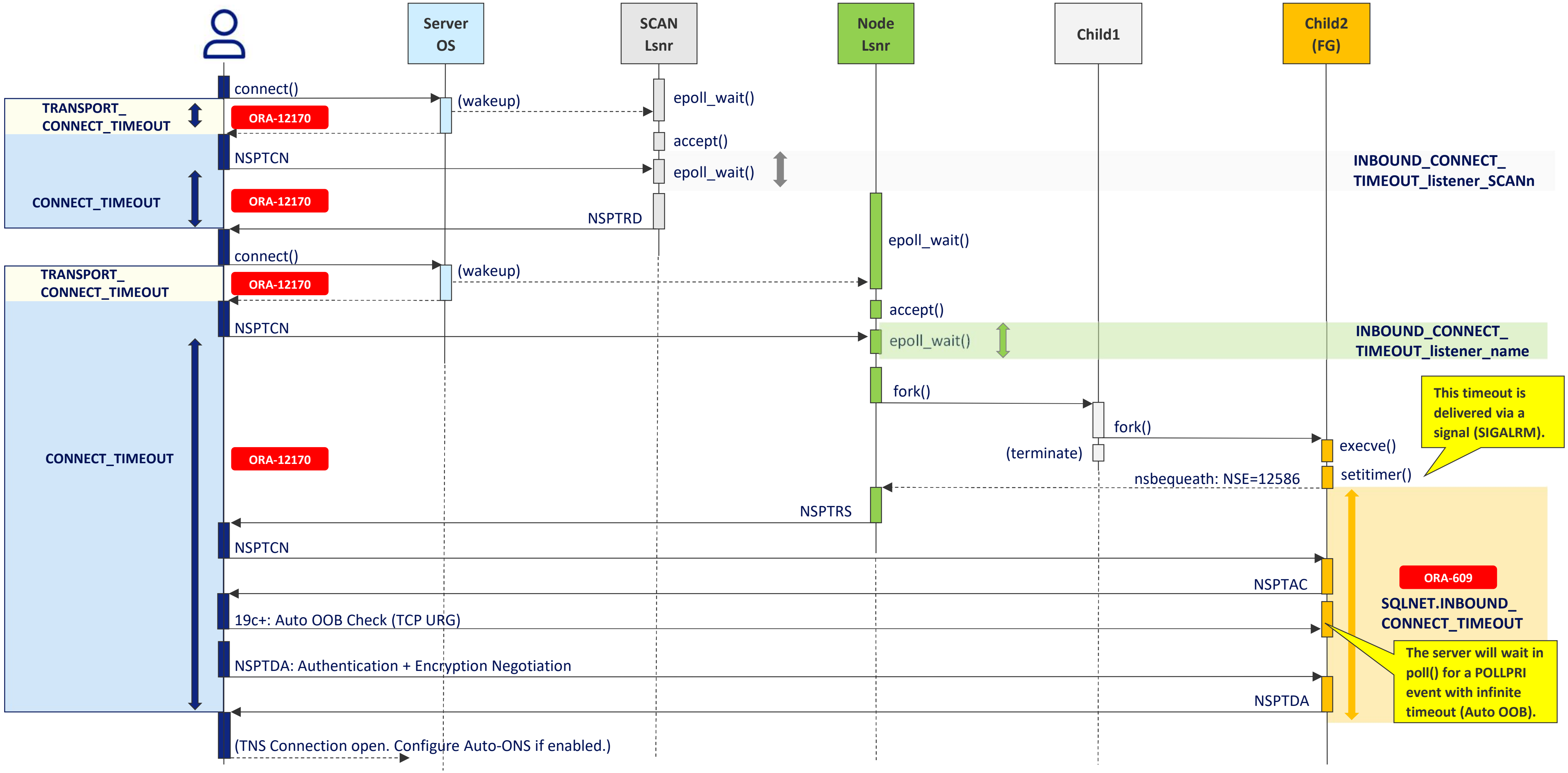
The redirect target address is defined by the **LOCAL_LISTENER** setting.

Note that the client sends a CONNECT packet twice. We'll see why in a minute …

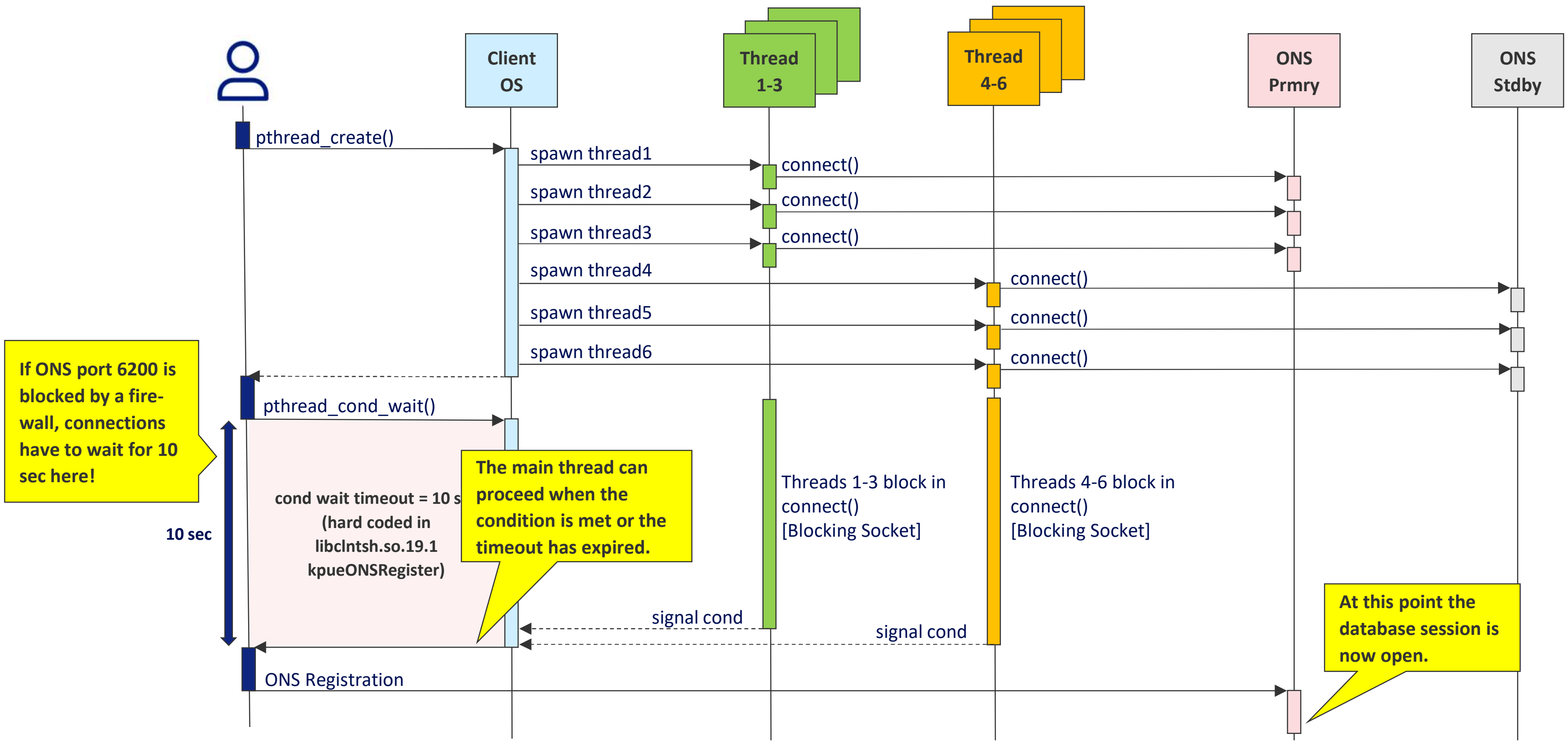



The Dance

Connect Timeouts – The Dance Between Client and Server



Connect Timeouts – The Dance Between Client and Server (Auto-ONS)



Summary

Network Timeouts Summary

If the network is slow, all these timeouts can add up!

Timeout	Default Duration	Parameters	Remark
DNS lookup timeout	Configurable. Default: timeout x attempts x searches	/etc/resolv.conf: timeout: <n> attempts: <n>	Further details in Appendix A
TCP Initial Retransmit Timeout	Linux: 1 sec Windows: 1 sec	Not configurable; defined by kernel constant TCP_TIMEOUT_INIT on Linux. Can be overriden with a BPF hook in in kernels 4.12+.	
TCP Initial Connection Timeout	Linux: 127 sec	tcp_syn_retries	Further details in Appendix B
Client TCP Idle Timeout without Keepalive	Inifinite		Further details in Appendix C
Client TCP Idle Timeout with Keepalive	> 2h	Client tnsnames.ora ENABLE=BROKEN	Further details in Appendix C
Server TCP Idle Timeout without Keepalive	Infinite		Further details in Appendix C
Server TCP Idle Timeout with Keepalive	Infinite	Server-side sqlnet.ora: SQLNET.EXPIRE_TIME	Further details in Appendix C
TCP Retransmit Timeout Established Connection	13 -60 min.	tcp_retries2	Further details in Appendix C
Oracle TCP Connect Timeout to OID/LDAP	15 sec	sqlnet.ora NAMES.LDAP_CONN_TIMEOUT	
Oracle Client TCP Connect Timeout to SCAN and Node Listener	60 sec	sqlnet.ora TCP.TRANSPORT_CONNECT_TIMEOUT tnsnames.ora TRANSPORT_CONNECT_TIMEOUT	Further details in Appendix B
Oracle Client TCP Connect Timeout to ONS	10 sec	None (hardcoded)	Further details in Appendix E
Oracle Client and Server Socket Send and Receive Timeout	Configurable	SQLNET.SEND_TIMEOUT SQLNET.RECEIVE_TIMEOUT	**You usually don't need these!** Further details in Appendix C

Connection String Format – Recommended Starting Point

Use ADRESS_LIST clauses to reduce the nr of DNS lookup requests and to enable Auto-ONS with FAN (s. details in Appendix).

Don't go too low, consider clients with a 3 sec initial TCP RTO and set the transport connect timeout slightly higher.

Give the server enough time and headroom in load burst situations (avoid cancelling TNS connection requests too early as this can trigger aggressive connection pool behavior).

```
MY_TEST.WORLD =
  (DESCRIPTION =
   (FAILOVER=ON)(TRANSPORT_CONNECT_TIMEOUT=4)(CONNECT_TIMEOUT=9)(ENABLE=BROKEN)
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan01.mydomain.net)(PORT = 1521))
      )
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan02.mydomain.net)(PORT = 1521))
      )
      (CONNECT_DATA =
        (SERVICE_NAME = MY_TEST_SERVICE_RW.WORLD)
      )
  )
```

Enable TCP keepalive on the client.

Add additional parameters like RETRY_COUNT and RETRY_DELAY as needed (not shown).

Modified from Source: Oracle Whitepaper, Application Checklist for Continuous Service for MAA Solutions, Oracle Technical Brief, 2023-07-23

Thank you for attending!

There's much more information in the Appendix!

Questions, feedback, comments?
I look forward to hearing from you!

christoph.lutz@swisscom.com

@chris_skyflier

Symposium 42 | @sym_42

References

References (1/2)

Arthur Chiao, TCP Socket Listen: A Tale of Two Queues, 2022-08-28

Beat Ramseier, Are you fishing or catching? – Server-side SQL*Net tracing for specific clients, 2017-10-15

Brendan Gregg, Systems Performance Enterprise and the Cloud, Second Edition, Addison-Wesley, 2021

Chris Down, Creating controllable D state (uninterruptible sleep) processes, 2024-02-05

Franck Pachot, SQLNET.EXPIRE_TIME and ENABLE=BROKEN, 2020-02-15

Jim Cromie, Linux Kernel Newbies Mailing List, Thread: getconf CLK_TCK and CONFIG_HZ, 2011-03-12

Marco Pracucci, Linux TCP_RTO_MIN, TCP_RTO_MAX and the tcp_retries2 sysctl, 2018-04-27

My Oracle Support: 12c Client: Approximate 9 - 10 Second Delay in Connecting via Remote SQL*Plus (Doc ID 2218140.1)

My Oracle Support: Reducing Client Connection Delays When DNS Is Unresponsive (Doc ID 1449843..1)

My Oracle Support, Oracle Net Listener Trace shows NSE=12586, is This an Error ? (Doc ID 738724.1)

Oracle 19c, Net Services Administrator's Guide

Oracle 21, Universal Connection Pool Developer's Guide, Section: About Optimizing Real-World Performance with Static Connection Pools

References (2/2)

Oracle Net8 Administrator's Guide, Release 8.1.5, Section 6: Configuring Naming Methods and the Listener

Oracle Whitepaper, Application Checklist for Continuous Service for MAA Solutions, Oracle Technical Brief, 2023-07-23

Oracle Whitepaper, Fast Application Notification (FAN), December 2016

RedHat Customer Portal, How to tune the DNS client resolver library through /etc/resolv.conf, 2016-02-25

Appendix A: DNS

DNS – getaddrinfo API

```
int getaddrinfo(const char* hostname,
                const char* service,
                const struct addrinfo* hints,
                struct addrinfo** res);
```

Hostname to look up.

Service name or port.

Hints / options to direct the getaddrinfo operation.

List of getaddrinfo results.

struct addrinfo

```
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    socklen_t ai_addrlen;
    struct sockaddr* ai_addr;
    char* ai_canonname; /* canonical name */
    struct addrinfo* ai_next; /* this struct
                               * can form a
                               * linked list
                               */
};
```

DNS – getaddrinfo Example Program

```
int main(int argc, char *argv[]) {

    struct addrinfo hints, *res, *result;
    void *ptr;
    char addrstr[INET_ADDRSTRLEN];
    int err;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags |= AI_CANONNAME;

    err = getaddrinfo("my-scan01.mydomain.net", NULL, &hints, &result);

    if (err != 0) {
      fprintf(stderr, "error in getaddrinfo: %s\n", gai_strerror(err));
      exit(EXIT_FAILURE);
    }

    for (res = result; res != NULL; res = res->ai_next) {

        switch(res->ai_family) {
            case AF_INET:
                ptr = &((struct sockaddr_in *) res->ai_addr)->sin_addr;
                break;
            case AF_INET6:
                ptr = &((struct sockaddr_in6 *) res->ai_addr)->sin6_addr;
                break;
            }

            inet_ntop (res->ai_family, ptr, addrstr, INET_ADDRSTRLEN);
            printf("IPv%d address: %s (%s)\n",  res->ai_family == PF_INET6 ? 6 : 4,
                   addrstr, res->ai_canonname);
    }

    freeaddrinfo(result);
    return 0;
}
```

Provide hints (options) to direct the getaddrinfo operation:
- AF_UNSPEC: Allow IPv4 or IPv6
- SOCK_STREAM: TCP Socket

Call getaddrinfo with a SCAN hostname

Loop over addrinfo structures returned by getaddrinfo

Check if result is an IPv4 or IPv6 address.

Print result

getaddrinfo API

getaddrinfo()

```
int getaddrinfo(const char* hostname,
                const char* service,
                const struct addrinfo* hints,
                struct addrinfo** res);
```

struct addrinfo

```
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    socklen_t ai_addrlen;
    struct sockaddr* ai_addr;
    char* ai_canonname; /* canonical name */
    struct addrinfo* ai_next; /* this struct
                               * can form a
                               * linked list
                               */
};
```

Example Program Output

```
./ex_gai

IPv4 address: xxx.xxx.xxx.36 (my-scan01 …)
IPv4 address: xxx.xxx.xxx.38 ((null))
IPv4 address: xxx.xxx.xxx.37 ((null))
```

Resolver Logic – Basic Mechanism

/etc/resolv.conf

```
options attempts:2
options timeout:4
options rotate

search mydomain.net

nameserver 1.2.3.4
nameserver 4.3.2.1
```

The rotate option will change (shuffle) the order in which name-servers are iterated.

Resolver Logic

```
for each attempt:
   for each DNS server:
      lookup "hostname":
         connect()
         sendto()
         poll(... timeout(ms))
         recvfrom() if there is reply
      if reply (success or fail), return
```

Source: RedHat Customer Portal, How to tune the DNS client resolver library through /etc/resolv.conf, 2016-02-25

As per the resolver logic on the left, the total number of lookups is determined as follows:

total lookups = attempts x nameservers

The resolver's configuration may aggravate the Oracle client's default DNS lookup behavior.

The more IP addresses you need to lookup and the more retries you have configured, the longer the potential delay when DNS is slow or unrespon-sive!

Anyway, let's verify this …

Resolver Logic – DNS Timeouts & Retries

Test Case: Block incoming DNS Traffic & run the test program

```
iptables -A INPUT -p udp --sport 53 -j DROP

strace -e trace=sendto,poll ./ex_gai
```

```
/etc/resolv.conf

options attempts:2
options timeout:4
…
Nameserver 1.2.3.4
nameserver 4.3.2.1
```

With this configuration, we'd expect a total number of requests of 4 : attempts x nameservers = 2 x 2 = 4

Req#	Trace
1	`sendto(3, "Z\261\1\0\0\1\0\0\0\0\0\0\vmy-scan01\10mydomain"..., 42, MSG_NOSIGNAL, NULL, 0) = 42` `poll([{fd=3, events=POLLIN}], 1, 4000)  = 0 (Timeout)`
2	`sendto(4, "Z\261\1\0\0\1\0\0\0\0\0\0\vmy-scan01\10mydomain"..., 42, MSG_NOSIGNAL, NULL, 0) = 42` `poll([{fd=4, events=POLLIN}], 1, 4000)  = 0 (Timeout)`
3	`sendto(3, "Z\261\1\0\0\1\0\0\0\0\0\0\vmy-scan01\10mydomain"..., 42, MSG_NOSIGNAL, NULL, 0) = 42` `poll([{fd=3, events=POLLIN}], 1, 4000)  = 0 (Timeout)`
4	`sendto(4, "Z\261\1\0\0\1\0\0\0\0\0\0\vmy-scan01\10mydomain"..., 42, MSG_NOSIGNAL, NULL, 0) = 42` `poll([{fd=4, events=POLLIN}], 1, 4000)  = 0 (Timeout)`
5	`sendto(3, "D'\1\0\0\1\0\0\0\0\0\0\vmy-scan01\10mydomain"..., 55, MSG_NOSIGNAL, NULL, 0) = 55` `poll([{fd=3, events=POLLIN}], 1, 4000)  = 0 (Timeout)`
6	`sendto(4, "D'\1\0\0\1\0\0\0\0\0\0\vmy-scan01\10mydomain"..., 55, MSG_NOSIGNAL, NULL, 0) = 55` `poll([{fd=4, events=POLLIN}], 1, 4000)  = 0 (Timeout)`
7	`sendto(3, "D'\1\0\0\1\0\0\0\0\0\0\vmy-scan01\10mydomain"..., 55, MSG_NOSIGNAL, NULL, 0) = 55` `poll([{fd=3, events=POLLIN}], 1, 4000)  = 0 (Timeout)`
8	`sendto(4, "D'\1\0\0\1\0\0\0\0\0\0\vmy-scan01\10mydomain"..., 55, MSG_NOSIGNAL, NULL, 0) = 55` `poll([{fd=4, events=POLLIN}], 1, 4000)  = 0 (Timeout)`
n/a	`error in getaddrinfo: Name or service not known`

We can clearly see the request timeout of 4 sec used, but the total number of requests is 8 and there-fore twice to what is expected!

What is going on?!

Resolver Logic – Search List: What If No Result?

Resolver Logic: Function __res_context_query Arguments (res_query.c)

```
Breakpoint 2, __GI___res_context_query (ctx=ctx@entry=0x602010,
    name=name@entry=0x400950 "my-scan01.mydomain.net", …
) at res_query.c:113

Breakpoint 2, __GI___res_context_query (ctx=ctx@entry=0x602010,
    name=name@entry=0x7fffffffed110 "my-scan01.mydomain.net.mydomain.net", …
) at res_query.c:113
```

The resolver appends the default domain and then performs an additional search.

Resolver Logic: Function __res_context_search (res_query.c)

```
/*
 * We do at least one level of search if
 *  - there is no dot and RES_DEFNAME is set, or
 *  - there is at least one dot, there is no trailing dot,
 *    and RES_DNSRCH is set.
 */

 if ((!dots && (statp->options & RES_DEFNAMES) != 0) ||
    (dots && !trailing_dot && (statp->options & RES_DNSRCH) != 0)) {
    …
```

If the resolver fails to get a result, it will always try at least one search and the search will send a request to all nameservers again!

What if there is no result?

The resolver's logic will always try at least one search if a lookup doesn't return a result and if the following conditions are met:

- Hostname contains a domain part
- Hostname contains no trailing dot (non absolute domain name)
- RES_DNSRCH is enabled (default, s. below)

So, worst case, the total number of lookups is as follows:

total lookups = 2 x nameservers x attempts

This behavior cannot be changed via configuration options in /etc/resolv.conf!

RES_DNSRCH

If set, **res_search**() will search for hostnames in the current domain and in parent domains. This option is used by gethostbyname(3). [Enabled by default].

Resolver Logic – Behavior in Case of No Result

Resolver Logic in Case of No Result

```
for each attempt:
  for each DNS server:
    check DNS server for "hostname":
      connect()
      sendto()
      poll(... timeout(ms))
      recvfrom() if there is reply
    if reply (success), return

for each attempt:
  for each DNS server:
    check DNS server for "hostname.DEFAULT_DOMAIN":
      connect()
      sendto()
      poll(... timeout(ms))
      recvfrom() if there is reply
    if reply (success), return
```

/etc/resolv.conf

```
options attempts:2
options timeout:4
options rotate

search mydomain.net

nameserver 1.2.3.4
nameserver 4.3.2.1
```

Resolver Logic – Oracle Client

```
#0  0x00007f6f6491a080 in sendmmsg ()
#1  0x00007f6f64bf25dd in __res_context_send ()
#2  0x00007f6f64bef394 in __res_context_query ()
#3  0x00007f6f64bf01e0 in __res_context_search ()
#4  0x00007f6f63ddcf09 in _nss_dns_gethostbyname4_r ()
#5  0x00007f6f64900364 in gaih_inet.constprop.8 ()
#6  0x00007f6f64901704 in getaddrinfo ()
#7  0x00007f6f66d9c8f3 in snlinGetAddrInfo ()
#8  0x00007f6f66dda95c in nttbnd2addr ()
#9  0x00007f6f66cdd6b3 in ntacbbnd2addr ()
#10 0x00007f6f66cdd429 in ntacbnd2addr ()
#11 0x00007f6f66c8998f in nsgettrans_bystring ()
#12 0x00007f6f66cf5beb in niotns ()
#13 0x00007f6f66d00a1d in osncon ()
#14 0x00007f6f66ba0a77 in kpuadef ()
#15 0x00007f6f66b86699 in upiini ()
#16 0x00007f6f66b9f253 in kpuatch ()
#17 0x00007f6f66b7a705 in OCIServerAttach ()
#18 0x00007f6f6a8d890d in afictatt ()
#19 0x00007f6f6a8d806b in afictcon ()
#20 0x00007f6f6a8de797 in aficoncon ()
#21 0x00007f6f6a8dce16 in aficon ()
#22 0x00007f6f6a8d5c8e in aficmd ()
#23 0x00007f6f6a8d4617 in aficfd ()
#24 0x00007f6f6a8d315c in aficdr ()
#25 0x00007f6f6a900aa6 in afidrv ()
#26 0x0000000000400d60 in main ()
```

libc.so.6 (#0)
libresolv.so.2 (#1–#3)
libnss_dns.so.2 (#4)
libc.so.6 (#5–#6)
libclntsh.so.19.1 (#7–#17)
libsqlplus.so (#18–#26)

glibc (#0–#6)
Oracle (#7–#26)

If dns lookups fail, the Oracle client tries again a second time.

The Oracle client has its own retry logic and similar to the libc resolver, tries to look up a SCAN a second time when the first lookup fails.

Worst case, the total number of request is as follows by default:

((2 x SCANS) + 1)) x nameservers x attempts x searches x 2

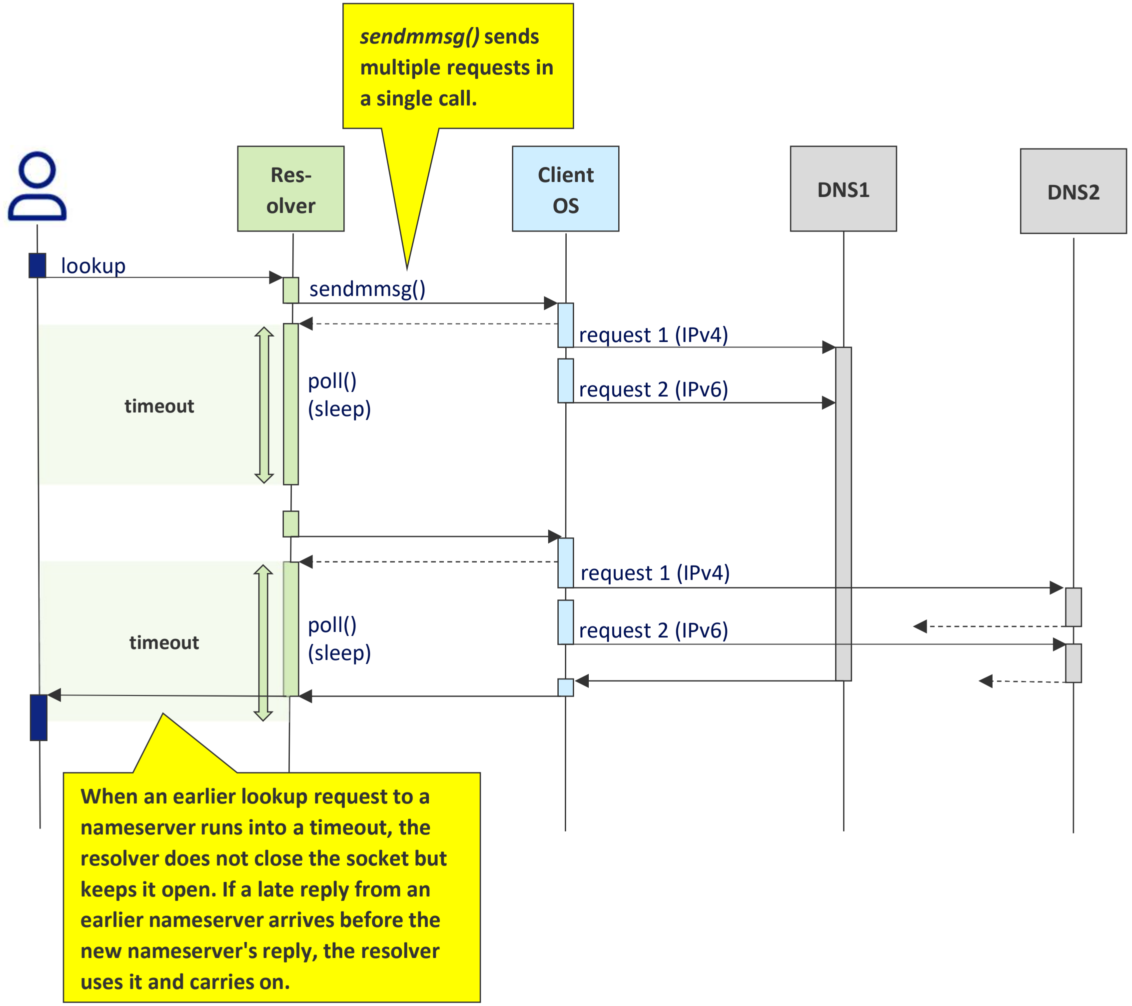
Ultimately, if none of the lookups will succeed, the Oracle client will fail with the following error message:

ORA-12545: Connect failed because target host or object does not exist

Resolver Logic – Examples

	Normal Case	Worst Case
Default Behavior	lookup requests = (SCANs + 1) x 2 - The Oracle client resolves the first SCAN twice (two calls to snlinGetAddrinfo / getaddrinfo for the first SCAN) - The resolver performs an additional IPv6 lookup for every request.	lookup requests = ((2 x SCANS) + 1)) x nameservers x attempts x searches x 2 = 5 x 2 x 2 x 2 x 2 = **80** [Assuming 2 nameservers and 2 attempts]
ADDRESS_LIST	lookup requests = SCANs x 2 - The Oracle client resolves every SCAN once (one call to snlinGetAddrInfo / getaddrinfo per SCAN) - The resolver performs an additional IPv6 lookup for every request	lookup requests = (2 x SCANs) x nameservers x attempts x searches x 2 = (2 x 2) x 2 x 2 x 2 x 2 = **64** [Assuming 2 nameservers and 2 attempts]
ADDRESS_LIST and IP=V4_ONLY	Lookup requests = SCANs - The Oracle client resolves every SCAN once (one call to snlinGetAddrInfo / getaddrinfo per SCAN) - The resolver performs an IPv4 lookup only	lookup requests = (2 x SCANs) x nameservers x attempts x searches = (2 x 2) x 2 x 2 x 2 = **32** [Assuming 2 nameservers and 2 attempts]

Resolver Logic – Optimizations



Resolver Optimizations

The libc resolver has neat low-level optimizations that aim to improve efficiency.

DNS & EZConnect – History

HOSTNAME Adapter (Oracle 8i days)

Configuration File	Configuration Option
Client: sqlnet.ora	NAMES.DIRECTORY_PATH=(HOSTNAME)
Client: /etc/hosts or Server: DNS	xxx.xxx.xxx.45 myhostdb03-v.mydomain.net mydomain.net
Server: listener.ora	GLOBAL_DBNAME = MYDOMAIN.NET

If this resolves via /etc/hosts or DNS, no tnsnames.ora is needed!

Note:

In 11g+ the HOSTNAME adapter will only work when **either** of the following is configured:

- DEFAULT_SERVICE_LISTENER_listener_name (server-side)
- HOSTNAME.DEFAULT_SERVICE_IS_HOST=1 (client-side)

With the HOSTNAME adapter, clients resolve a database global name via DNS or /etc/hosts and don't need a tnsnames.ora configuration file to connect to a database.

Oracle Net8 Administrator's Guide, Release 8.1.5, Section 6: Configuring Naming Methods and the Listener

Host Naming Method

Before the introduction of EZConnect naming in Oracle 10g, the "Host Naming Method" provided a "simple connectivity" mechanism.

When configured, the HOSTNAME adapter allows database clients to resolve the global database name via DNS or /etc/hosts, which eliminates the need to maintain TNS connect descriptors in a tnsnames.ora file.

DNS & EZConnect – Name Lookups

The HOSTNAME naming method has been carried over into EZCONNECT.

With EZCONNECT and a connection string like this, Oracle will try to resolve the TNS service name via DNS.

```
connect username/password@MYDOMAIN.NET
```

EZConnect will try to resolve this via DNS by default!

Why should we care?!

If no service name is not found in DNS, the resolver will iterate over all domain entries in the search list and this can generate a lot of DNS lookup requests!

Be careful with the naming method order in NAMES.DIRECTORY_PATH.

Always put the main naming method(s) first!

```
NAMES.DIRECTORY_PATH(EZCONNECT,TNSNAMES)
```

This way, EZConnect will be tried first which can result in lots of unecessary DNS requests before the service name is found in tnsnames.ora.

[Oracle 19c, Net Services Administrator's Guide, Section 8.1.5 Configuring Easy Connect Naming to Use a DNS Alias]()

Naming Method Order

Always put the main naming methods(s) in NAMES.DIRECTORY_PATH first as otherwise the client may attempt to unecessarily resolve the service name via DNS.

Appendix B:
TCP Timeouts – New Connections

Socket API – Blocking Socket Example

```
int main(int argc, char *argv[]) {

    int sockfd = 0;
    struct sockaddr_in serv_addr;

    memset(&serv_addr, '0', sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    inet_pton(AF_INET, argv[1], &serv_addr.sin_addr);
    serv_addr.sin_port = htons(atoi(argv[2]));

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
    printf("Connected.\n");

    close(sockfd);
    return 0;
}
```

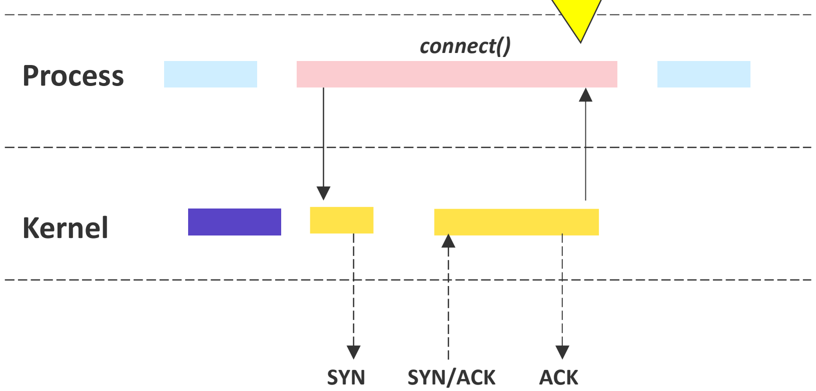
Declare socket data structures.

Initialize serv_addr. This structure will contain the destination IP and port.

Create a new TCP socket.

Open a new TCP connection to the target IP and port.

Now the application is aware of the established connection and can proceed.



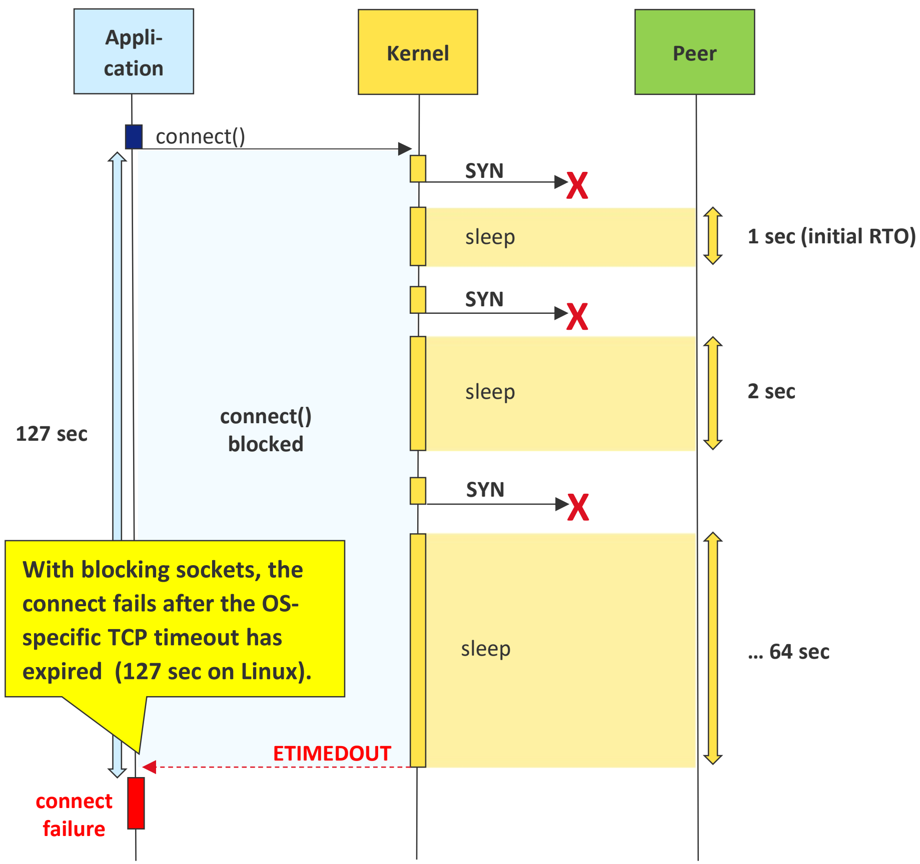
Blocking Sockets– connect()

When an application calls **connect(),** it will block until the kernel has managed to establish a TCP connection.

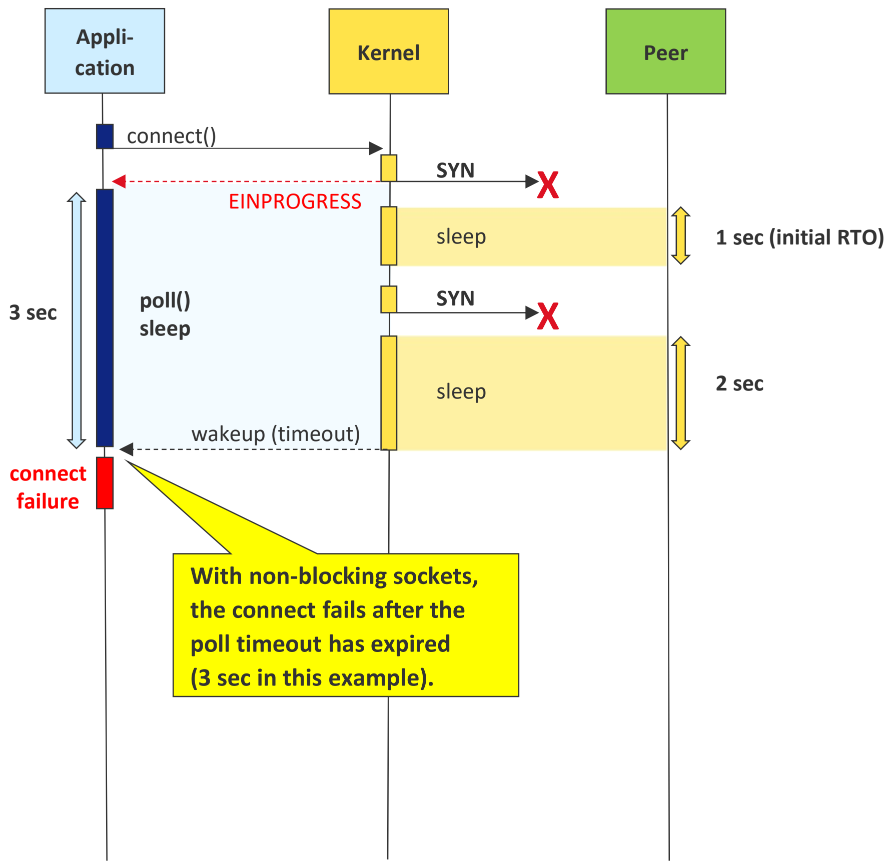
If the communication endpoint does not respond to the connection request (SYN), the kernel will retry and retrans-mit the SYN multiple times until hitting an OS-specific timeout!

Blocking vs Non-Blocking Sockets (Fully Animated)

Blocking Socket



Non-Blocking Socket



Socket API – Non-Blocking Socket Example

```
if((fcntl(sockfd, F_SETFL, fcntl(sockfd, F_GETFL, 0) | O_NONBLOCK)) == -1) {
    printf("Error: fcntl\n");
    return 1;
}

struct pollfd pfds[1];
pfds[0].fd = sockfd;
pfds[0].events = POLLOUT;

if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) == -1) {

    if(errno == EINPROGRESS) {

        sock_ready = poll(pfds, 1, 3000);

        if(sock_ready > 0 && pfds[0].revents & POLLOUT) {
            printf("Connection is ready.\n");
        }
        else if (sock_ready == 0) {
            printf("Connection timed out.\n");
        }
        else {
            printf("poll failed. Aborting.\n");
            return 1;
        }
    }
}
```

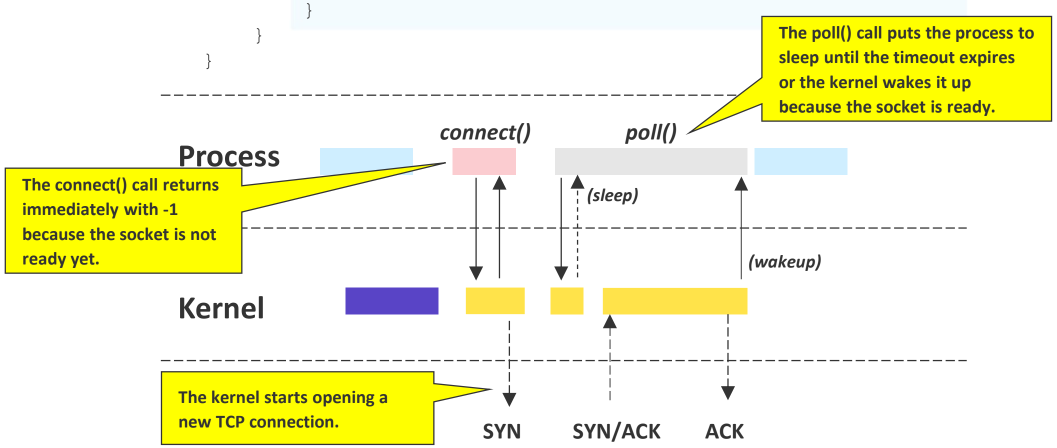
Make the socket non blocking with the O_NONBLOCK flag.

Declare and initialize the pollfd structure.

The connect() will return an error with errno set to EINPROGRESS. This means the socket is not ready yet.

Call poll() with a timeout of 3000 ms.

Check the poll() resullt. If successful, the socket is ready, if not we may have run into the timeout.

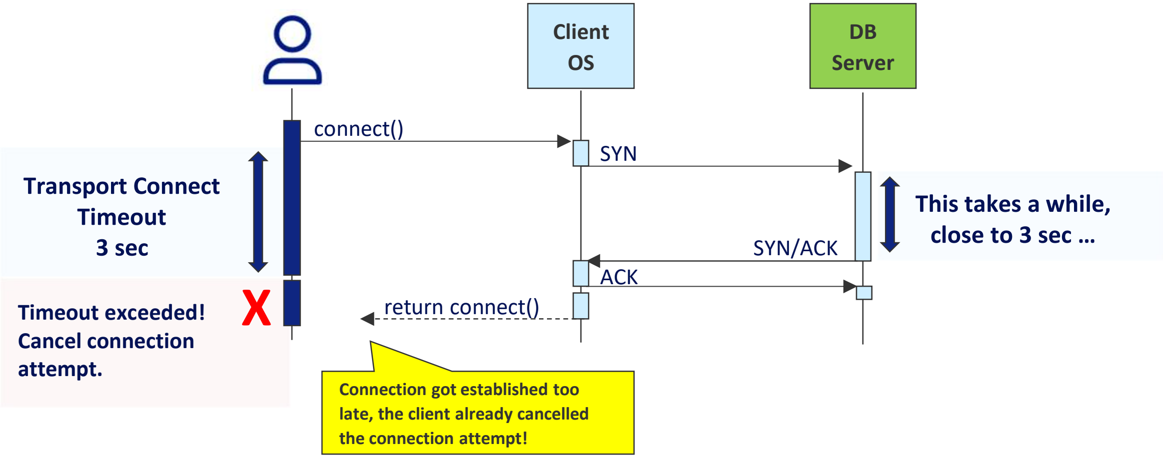


Non-Blocking Sockets – connect()

When an application calls **connect()**, the call will not block and return immediately with -1 and errno set to **EINPROGRESS**.

This signals to the calling process that the socket is not ready yet and that it can wait for the socket to become ready with *poll()*, which will put the calling process to sleep until the socket is ready or until the timeout exceeds.

TCP Packet Loss – TCT <= Initial RTO



- **TRANSPORT_CONNECT_TIMEOUT: 3 sec**
- **TCP Initial RTO: 3 sec (Windows)**

Ideally set the TRANSPORT_CONNECT_TIMEOUT to a higher value than the initial RTO on the OS side! This will give the client a chance to recover a connection when there are processing delays.

TRANSPORT_CONNECT_TIMEOUT

If the TRANSPORT_CONNECT_TIMEOUT expires before the initial RTO got a chance to "recover" from a TCP packet loss, the client will give up and cancel the connection attempt even though a TCP connection would have been established shortly after a successful retransmit.

Setting TRANSPORT_CONNECT_TIMEOUT to a value higher than the initial RTO on the OS side, will give a client more headroom to recover from processing delay situations (which may occur during temporary load bursts on the network or on the server side).

TCP Packet Loss – Real Life Scenario

Application Log (Windows Java Client)

```
2023-07-08T11:20:12.186: Checking ip: xxx.xxx.xxx.xxx:1521
```

TRANSPORT_CONNECT_TIMEOUT = 3

```
2023-07-08T11:20:15.190:
*** Java Stack Trace ***
*** Exception caught ***
Message:  connect timed out
```

Transport Connect Timeout expires before application becomes aware of established TCP connection!

Tcpdump (Server)

```
11:20:12.184377 IP ccc.ccc.ccc.ccc.56873 > xxx.xxx.xxx.xxx.1521:
Flags [S], seq 1762175761
```

Connection attempt (SYN).

```
11:20:15.185235 IP ccc.ccc.ccc.ccc.56873 > xxx.xxx.xxx.xxx.1521:
Flags [S], seq 1762175761

11:20:15.185262 IP xxx.xxx.xxx.xxx.1521 > ccc.ccc.ccc.ccc.56873:
Flags [S.], seq 2104104716, ack 1762175762

11:20:15.186823 IP ccc.ccc.ccc.ccc.56873 > xxx.xxx.xxx.xxx.1521:
Flags [.], ack
```

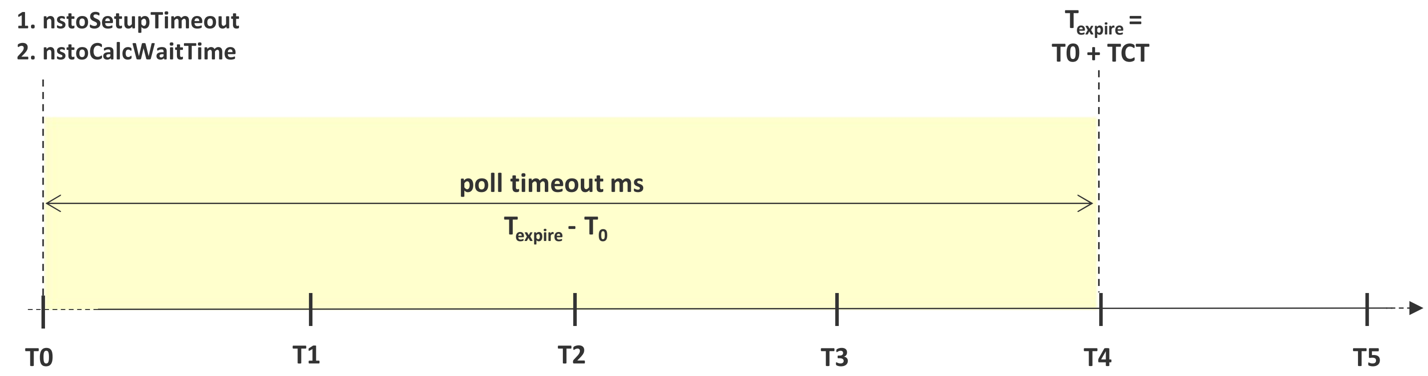
Retry succeeds 3 sec later after initial RTO of 3 sec

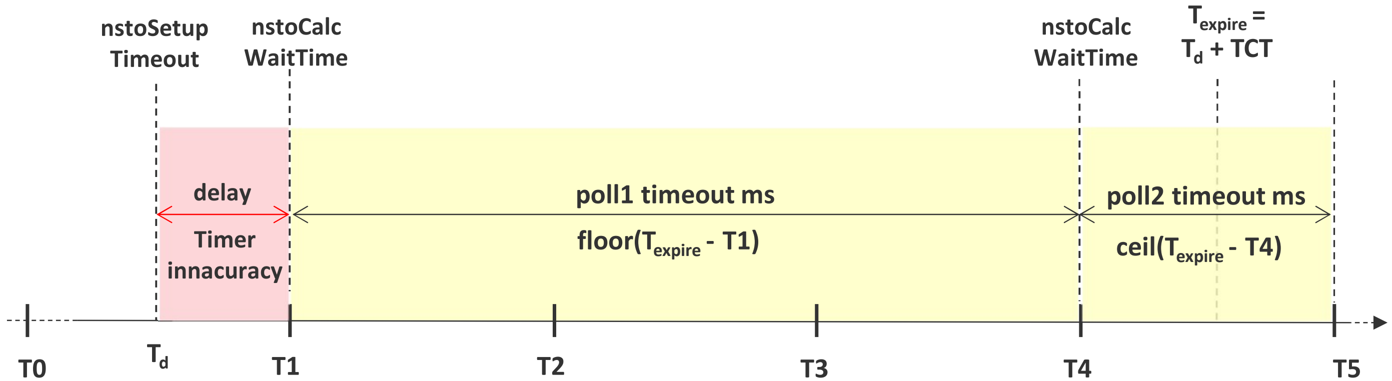
vmstat

```
--time-- procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
           r  b   swpd      free   buff     cache   si   so    bi    bo     in     cs us sy id wa st
11:20:07 102  0  59904 125475856  15392 224277552    0    0    18  2756 277874 274871 84 12  4  0  0
11:20:12 128  2  59904 125257056  15392 224281312    0    0    56  3870 283630 290438 85 12  3  0  0
11:20:18 123  0  59904 125012976  15392 224288320    0    0    15   466 256425 263106 85 11  4  0  0
11:20:23 112  0  59904 124652552  15392 224295280    0    0    20  3778 232582 204669 88  9  3  0  0
```

System very busy and saturated at during problem time!

Transport Connect Timeout Calculation – Oddities





Timing Oddities

nstoSetupTimeout calculates an expi-ration time by adding the transport connect timeout value to the current time (T_{expire} = $T_{current}$ + TCT).

nstoCalcWaitTimeout calculates how much wait time is left by subtracting the current time from the expiration time (timeout = T_{expire} - $T_{current}$).

When nstoSetupTimeout and nstoCalcWaitTime execute at "the same time" (**within the same clock interval** returned by the times system call), the timeout in ms (T_{expire} – $T_{current}$) is an integer multiple in sec. However, when execution incurs a delay or the functions don't execute within the same clock interval, the timeout is not an integer multiple in sec and gets rounded down.

When that happens, the client issues calls to poll() again until the expiration time T_{expire} has exceeded.

Transport Connect Timeout Calculation – Example

Connection String

```
MY_TEST.TEST.DBS =
  (DESCRIPTION =
   (FAILOVER=ON)(TRANSPORT_CONNECT_TIMEOUT=4)(CONNECT_TIMEOUT=9)(ENABLE=BROKEN)
   …
```

```
./ora_connect2.bt --unsafe <client_pid>

23:27:07 180006/180006: connect: fd=9, xxx.xxx.xxx.xxx:1521

23:27:07 180006/180006: nstoControlTTO: entry
23:27:07 180006/180006: times: ret=593455102
23:27:07 180006/180006: nstoSetupTimeout: timeout_ms=4000, ticks_curr_ms=5934551020
23:27:07 180006/180006: nstoSetupTimeout: ticks_curr_ms+timeout_ms=5934551020+4000=5934555020
23:27:07 180006/180006: nstoSetupTimeout+130: timeout_ms=4000
23:27:07 180006/180006: nstoSetupTimeout+133: t_expire_ms=5934555020
23:27:07 180006/180006: nstoControlTTO: ret=0

23:27:07 180006/180006: nstoCalcWaitTime: entry
23:27:07 180006/180006: times: ret=593455103
23:27:07 180006/180006: nstoCalcWaitTime+39: ticks_curr_ms=5934551030, t_expire_ms=5934551020
23:27:07 180006/180006: nstoCalcWaitTime+39: t_expire_ms-ticks_curr_ms=5934555020-5934551030=3990
23:27:07 180006/180006: nstoCalcWaitTime: ret=3990

23:27:07 180006/180006: poll: fd=9, event=POLLOUT, timeout=3000

23:27:09 180006/180006: nstoCalcWaitTime: entry
23:27:09 180006/180006: times: ret=593455303
23:27:09 180006/180006: nstoCalcWaitTime+39: ticks_curr_ms=5934554030, t_expire_ms=5934551020
23:27:09 180006/180006: nstoCalcWaitTime+39: t_expire_ms-ticks_curr_ms=5934555020-5934554030=990
23:27:09 180006/180006: nstoCalcWaitTime: ret=990

23:27:09 180006/180006: poll: fd=9, event=POLLOUT, timeout=1000
```

nstoControlTTO() calculates the timeout expiration time by adding the timeout to the current time (it converts ticks to milliseconds)

nstoCalcWaitTime() calculates the wait time / poll timeout in ms, but this gets rounded down later when passed to poll()!

Sometimes you'll find the timeout settings and poll() timeouts do not match, they're mostly off by 1 sec. Why is this?

If poll() completes and the expiration time has not yet exceeded, the client calculates a new timeout and calls poll() again (if < 1 sec, the timeout gets rounded up to 1 sec).

Timeout Calculation

The Oracle client uses the **times()** sytem call to calculate the expected timeout expiration time.

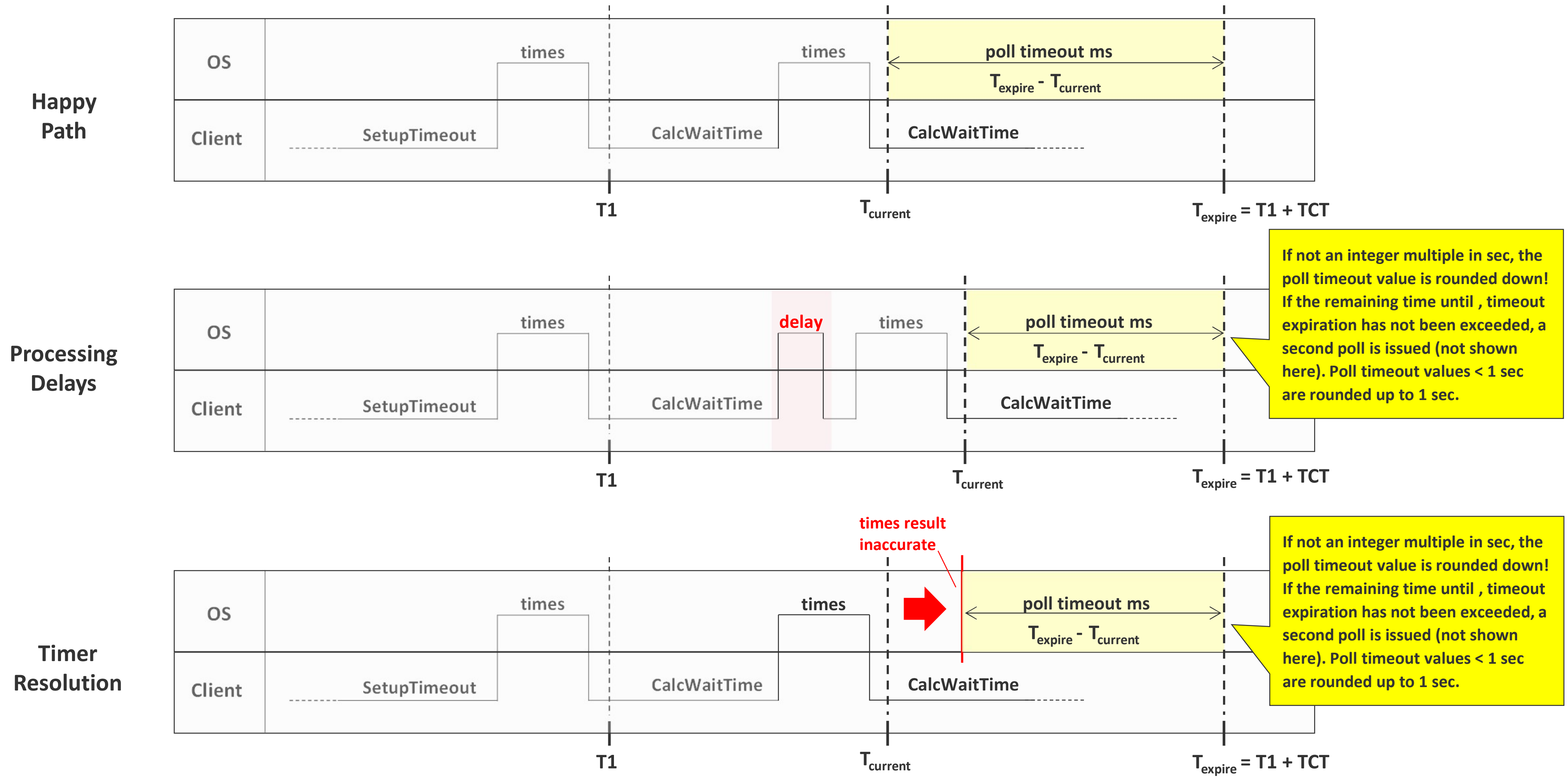
This system call returns the number of clock ticks since an arbitrary point in the past (on OEL8, CLK_TCK returns 100 Hz, which means times() uses a tick granularity of 1 cs).

The client internally converts the number of ticks to milliseconds and calculates how much timeout time is left. If the calculated timeout value is not an integer multiple in sec, it is rounded down to the nearest integer value in seconds and used in the call to **poll().**

The timeout value used by poll() can be different from the timeout value set in TRANSPORT_CONNECT_TIMEOUT due to processing delays or timer granularity effects!

Note that the CONNECT_TIMEOUT behaves in a similar way.

Transport Connect Timeout Calculation – Timing Effects



Transport Connect Timeout – Timing Effects Example

Happy Path

connect()

nstoSetupTimeout():

$TCT_{expire} = T_{curr} + TCT_{tns}$

8370 = 4730 + 4000

nstoCalcWaitTime():

T_{curr} = times()

$T_{poll} = TCT_{expire} - T_{curr}$

4000 = 8730 – 4730

poll(): timeout = T_{poll} (4000)

Processing Delays

connect()

nstoSetupTimeout():

$TCT_{expire} = T_{curr} + TCT_{tns}$

8370 = 4730 + 4000

Processing Delay (1 ms)

nstoCalcWaitTime:

T_{curr} = times()

$T_{poll} = TCT_{expire} - T_{curr}$

3999 = 8730 – **4731**

poll(): timeout = T_{poll} (**3000**)

Poll timeout rounded down to nearest integer value in sec!

If rounded down, a second poll() will be issued (not shown here). If the remaining time until timeout expiration is < 1 sec, the second poll timeout value will be rounded up to 1 sec.

Timer Resolution

connect()

nstoSetupTimeout():

$TCT_{expire} = T_{curr} + TCT_{tns}$

8370 = 4730 + 4000

nstoCalcWaitTime():

T_{curr} = times() **=> times returns cs!**

$T_{poll} = TCT_{expire} - T_{curr}$

3990 = 8730 – **4740**

poll(): timeout = T_{poll} (**3000**)

Poll timeout rounded down to nearest integer value in sec!

If rounded down, a second poll() will be issued in this case as well (not shown here).

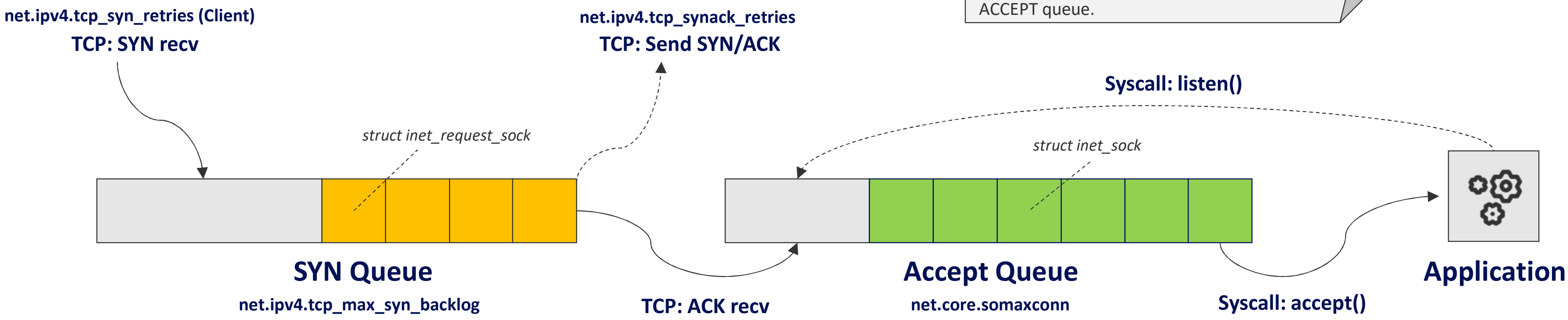
Time

TCP Backlog Queue – Details

int listen(int sockfd**,** int backlog**);**

The **backlog** argument defines the maximum length to which the queue of pending connect-ions for **sockfd** may grow.

Starting with Linux kernel version 4.3, the **backlog** argument defines the length of the SYN and ACCEPT queue.



How to monitor the Queue Lengths?

SYN Queue:
ss -plnt state syn-recv sport = :1521

Accept Queue:
ss -plnt sport = :1521

nstat Counters:
nstat -az |grep -i listen

BPF (bpftrace):
tcpsynbl.bt

Oracle Listener
QUEUESIZE = 128 (default)

Change Default Queuesize (listener.ora)

listener_name= (DESCRIPTION= (ADDRESS=(PROTOCOL=tcp)(HOST=server)(PORT=1521)(**QUEUESIZE=1024**)))

Image modified from source Brendan Gregg, Systems Performance Enterprise and the Cloud, Second Edition, Addisson-Wesley, 2021, p. 520

TCP Timeouts – SCAN Host Expansion

```
MY_TEST.WORLD =
  (DESCRIPTION =
   (FAILOVER=ON)(TRANSPORT_CONNECT_TIMEOUT=4)(CONNECT_TIMEOUT=9)(ENABLE=BROKEN)
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan01.mydomain.net)(PORT = 1521))
      )
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan02.mydomain.net)(PORT = 1521))
      )
      (CONNECT_DATA =
        (SERVICE_NAME = MY_TEST_RW.WORLD)
      )
  )
```

The client attempts to connect to every IP before giving up. If a RETRY_COUNT has been specified, the client will try every SCAN IP RETRY_COUNT times again after the first iteration!

nlad_expand_hst: Result:

SCAN1 IP addresses.

```
(DESCRIPTION=
    (FAILOVER=ON)(TRANSPORT_CONNECT_TIMEOUT=4)(CONNECT_TIMEOUT=9)(ENABLE=BROKEN)
        (ADDRESS_LIST=
            (ADDRESS=(PROTOCOL=TCP)(HOST=xxx.xxx.xxx.38)(PORT=1521)(HOSTNAME=my-scan01.mydomain.net))
            (ADDRESS=(PROTOCOL=TCP)(HOST=xxx.xxx.xxx.37)(PORT=1521)(HOSTNAME=my-scan01.mydomain.net))
            (ADDRESS=(PROTOCOL=TCP)(HOST=xxx.xxx.xxx.36)(PORT=1521)(HOSTNAME=my-scan01.mydomain.net)))
        (ADDRESS_LIST=
            (ADDRESS=(PROTOCOL=TCP)(HOST=yyy.yyy.yyy.134)(PORT=1521)(HOSTNAME=my-scan02.mydomain.net))
            (ADDRESS=(PROTOCOL=TCP)(HOST=yyy.yyy.yyy.133)(PORT=1521)(HOSTNAME=my-scan02.mydomain.net))
            (ADDRESS=(PROTOCOL=TCP)(HOST=yyy.yyy.yyy.132)(PORT=1521)(HOSTNAME=my-scan02.mydomain.net)))
        (CONNECT_DATA=(SERVICE_NAME=MY_TEST_RW.WORLD)
        (CID=(PROGRAM=sqlplus)(HOST=myhostdb01.mydomain.net)(USER=oracle))
    )
)
```

SCAN2 IP addresses.

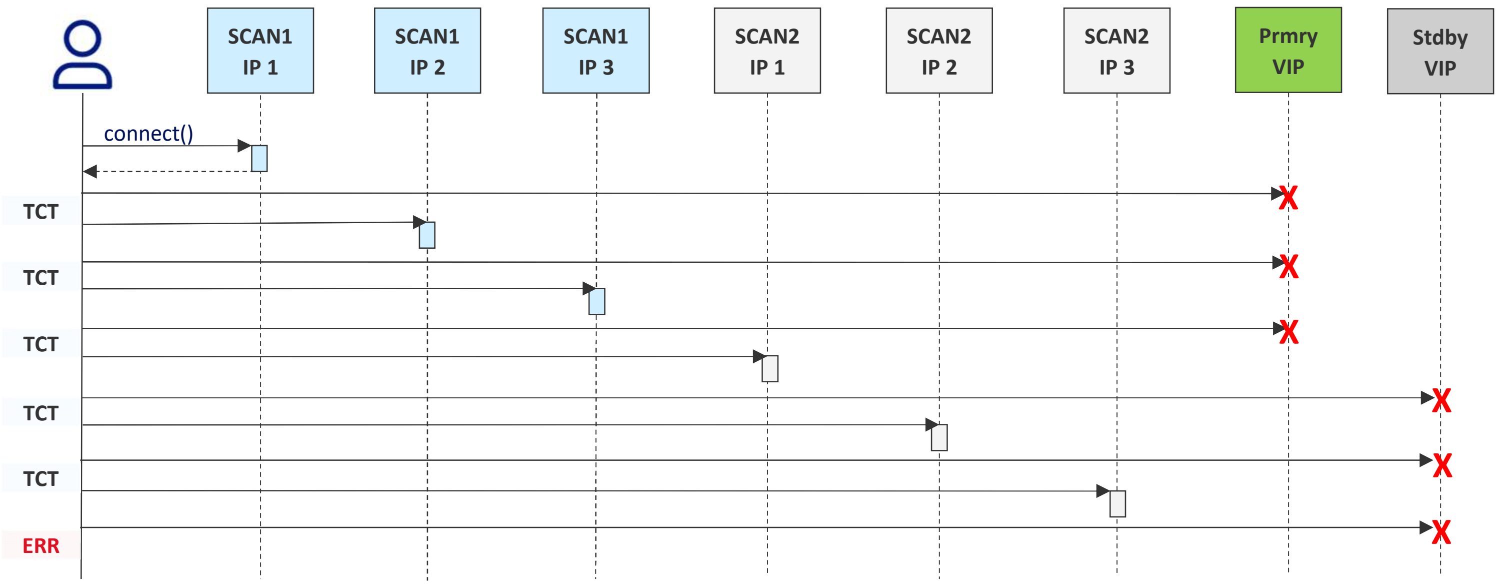
SCAN Host Expansion

The client internally expands the SCAN and constructs an ADDRESS entry for every SCAN IP. If a connection attempt results in a timeout, the client will retry and iterate over all SCAN IPs of all ADDRESS_LIST clauses (note also that the expanded TNS descriptor uses the undocumented HOSTNAME clause).

When a RETRY_COUNT has been speci-fied, the client will attempt to connect to every SCAN IP 1 + RETRY_COUNT times in total.

Depending on timeout settings, you may not need to specify RETRY_COUNT as you always get implicit retries with a SCAN!

TCP Timeouts – DB Connection Attempts & Retries



The error depends on the problem situation:

- If standby vip is unreachable: ORA-12170
- If standby vip is reachable but the listener not aware of the service: ORA-12514

Happy Path (not illustrated)

1. Connect to the First SCAN IP (Round Robin)
2. Get a TNS Redirect Packet (NSPTRD)
3. Connect to the Node Listener

Unhappy Path

1. Connect to SCAN Listener (Round Robin)
2. Get a TNS Redirect Packet (NSPTRD)
3. Connect to the Primary Node Listener
4. Connect to the next Primary SCAN IP
5. Connect to the Node Listener
6. Connect to the next SCAN IP
7. Connect to the Primary Node Listener
8. Repeat steps 1-7 on the Standby
9. Fail with Error

Other TCP Timeouts

OID TCP Timeouts

- 15 sec (default for NAMES.LDAP_CONN_TIMEOUT)
- In case of multiple DIRECTORY_SERVERS, each one is tried in order
- The list of DIRECTORY_SERVERS servers is iterated 5 times maximum (total attempts = 5 x DIRECTORY_SERVERS)
- Note: OID connections are not load balanced (if the first server in the list is not reachable, connections will hang).

ONS TCP Timeouts:

- 10 sec (hardcoded)

The TRANSPORT_CONNECT_TIMEOUT only applies to TCP connections to SCAN and node listeners. It does not apply to other connections!

OID Timeouts

For TCP connections to OID/LDAP servers, there is the **NAMES.LDAP_CONN_TIMEOUT**, which **defaults to 15 sec**.

OID does not support load balancing across multiple different servers.

If multiple OID/LDAP servers are config-ured, the client attempts to connect to the first one in the list of DIRECTORY_SERVERS and fails over to the next one in case of an error or a timeout (**LDAP_CONN_TIMEO**UT).

It will iterate over the list of **DIRECTORY_SERVERS** for a maximum of five times. If none of the connection attempts succeeds, the client will fail with " ORA-12154: TNS:could not resolve the connect identifier specified".

ONS Timeouts

Connections to ONS use a (hardcoded) default **timeout of 10 sec**. However, as will be shown later, this is implemented differently than the TCP timeouts we've looked at so far.

TNS Timeouts – Documentation Review

Client	**TRANSPORT_CONNECT_TIMEOUT | TCP.CONNECT_TIMEOUT** - Timeout duration in ms, sec, or min for a client to establish an Oracle Net connection to an Oracle database. **SQLNET.OUTBOUND_CONNECT_TIMEOUT (sqlnet.ora):** - Time for a client to establish an Oracle Net connection to the database server. - The outbound connect timeout interval is a superset of the TCP connect timeout interval - This parameter is overriden by the CONNECT_TIMEOUT in the address description
Listener	**INBOUND_CONNECT_TIMEOUT_listener_name (listener.ora):** - Time for a client to complete its connect request to the listener after the network connection had been established. - Set the value of INBOUND_CONNECT_TIMEOUT_listener_name parameter to a lower value than the SQLNET.INBOUND_CONNECT_TIMEOUT.
Server Processes	**SQLNET.INBOUND_CONNECT_TIMEOUT (sqlnet.ora):** - Time for a client to connect with the database server and provide the necessary authentication information.

This is rather a TCP connection.

Connect timeout is additive (the client starts a new connect timeout whenever it connects to a SCAN or node listener).

This is true!

Plus Auto OOB check and encryption negotiation.

Sources:
Oracle Database 19c, Database Net Services Reference, Section 6.11: Timeout Parameters
Oracle Database 19c, Database Net Services Reference, Section 5.2: sqlnet.ora Profile Parameters
Oracle Database 19c, Database Net Services Reference, Section 7.4 Control Parameters

Appendix C:
TCP Timeouts – Established Connections

TCP Timeouts – Established Connections

This section explains this setting (among other things).

```
MY_TEST.WORLD =
  (DESCRIPTION =
   (FAILOVER=ON)(TRANSPORT_CONNECT_TIMEOUT=3)(CONNECT_TIMEOUT=9) (ENABLE=BROKEN)
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan01.mydomain.net)(PORT = 1521))
      )
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = TCP)(HOST = my-scan02.mydomain.net)(PORT = 1521))
      )
      (CONNECT_DATA =
        (SERVICE_NAME = MY_TEST_RW.WORLD)
      )
  )
)
```

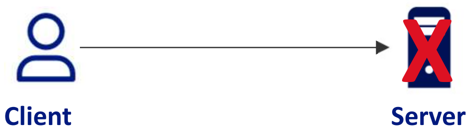
Established Connections – TCP Timeout Scenarios

Established Connection

Scenario #1
Active Connection
(TCP Retransmission)

Scenario #2
Idle Connection
(TCP Keepalive)

Established Connections – TCP Packet Loss Scenarios



The server can fail unexpectedly.

The client can fail unexpectedly.

A firewall (or other network device) can close connections unexpectedly.

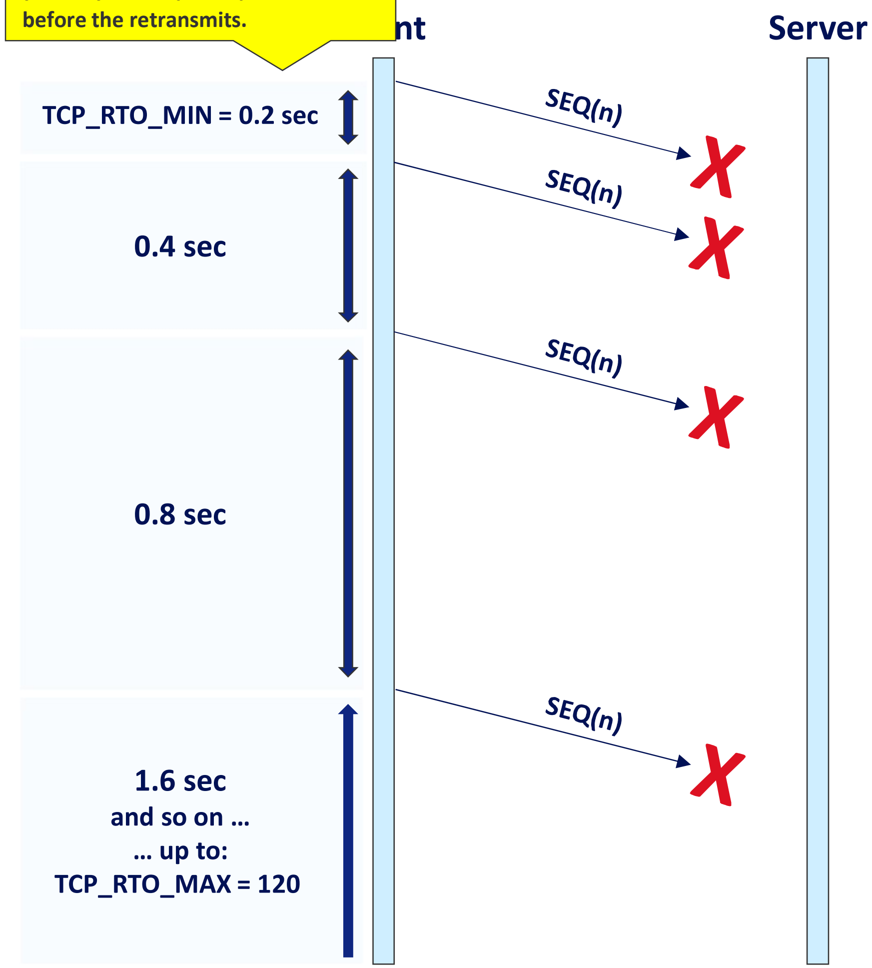
Networks and communication peers are unreliable and can potentially fail any time. TCP copes with this kind of unreliability in different ways depending on whether a connection is "active" or "idle".

Established Connections

- **"Active" Connections: Retransmits**
- **"Idle" Connections: TCP Keepalive**

TCP Retransmits – RTO Established Connections

When Tail Loss Probe (TLP) is enabled (tcp_early_retrans >= 3) , you may see a "probe packet" here before the retransmits.



Retrans-mission	RTO ms	Time ela sec
1	200	0.2
2	400	0.6
3	800	1.4
4	1'600	3.0
5	3'200	6.2
6	6'400	12.6
7	12'800	25.4
8	25'600	51.0
9	51'200	102.2
10	102'400	204.6
11	120'000	324.6
12	120'000	444.6
13	120'000	564.6
14	120'000	684.6
15	120'000	804.6
16	**120'000**	**924.6**

The connection is detected as broken when the last retry expires (15th retry) but it takes another RTO to notify the upper layers (16th retry)!

The kernel dynamically adjusts the RTO at runtime depending on RTT. Therefore, retransmit timeouts can take up to 32 min (16 x 120 sec).

You can check the effective RTO and RTT of a connection with: "`ss -i`".

Source: https://pracucci.com/linux-tcp-rto-min-max-and-tcp-retries2.html

TCP Retransmits

On Linux, the max number of retransmits for unacknowledged TCP packets is defined by the following tunable that defaults to 15 on OEL 7:

net.ipv4.tcp_retries2

With a min RTO of 0.2 sec, a max RTO of 120 sec and 15 retries, the timeout is 924.6 sec (~15.4 min).

Note that the connection is detected as broken after the 15th retry but it takes another RTO of 120 sec to notify the upper layers!

The constants TCP_RTO_MIN and TCP_RTO_MAX are defined in include/net/tcp.h:

#define TCP_RTO_MIN ((unsigned)(HZ/5))

#define TCP_RTO_MAX ((unsigned)(120*HZ))

TCP Retransmits – How To Detect?

Tail Loss Probe (TLP)

Show TCP retransmits incl. TLP in real-time

```
./tcpretrans -l

TIME     PID   IP LADDR:LPORT              T> RADDR:RPORT              STATE
22:10:46 0     4  xxx.xxx.xxx.xxx:13470    L> yyy.yyy.yyy.yyy:1521     ESTABLISHED
22:10:46 0     4  xxx.xxx.xxx.xxx:13470    R> yyy.yyy.yyy.yyy:1521     ESTABLISHED
22:10:46 42454 4  xxx.xxx.xxx.xxx:13470    R> yyy.yyy.yyy.yyy:1521     ESTABLISHED
…
```

Count TCP retransmits per TCP stream

```
./tcpretans -c

LADDR:LPORT                  RADDR:RPORT                RETRANSMITS
[xxx.xxx.xxx.xxx]#1521 <-> [yyy.yyy.yyy.yyy]#50456           4564
…
```

How to detect TC Retranmits?

TCP retransmits are easiest to detect with the following BPF based tools:

- **BCC: tcpretrans**
- **bpftrace: tcpretrans.bt**

The BCC tool **tcpretrans** is very power-ful and can even detect TLP probes and count the number of retransmits per TCP stream (s. examples on the left)!

If BPF tools are not available, you can revert to conventional packet capture based tools like tcpdump and wire-shark.

TCP Retransmission – Optimizations

TCP Retransmission

Timeout Based Retransmission

Timeout Range:
[200 ms, 120 sec]

Benefit: Simple ☺

Drawback: Slow

Fast Retransmission

Goal:
Trigger retransmission faster than the timeout based mechanism (after receival of duplicate ACKs).

Selective ACK (SACK):
Optimization - only retransmit the missing data segments.

Benefit:
Suitable for "hole loss".

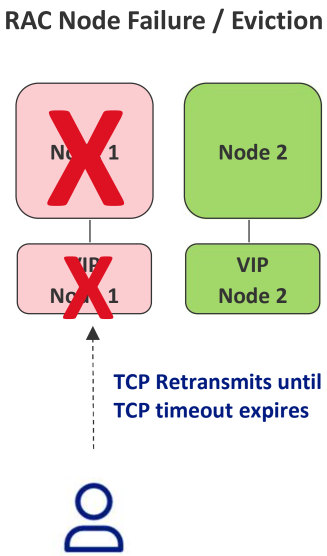
Drawback:
No improvement for tail loss.

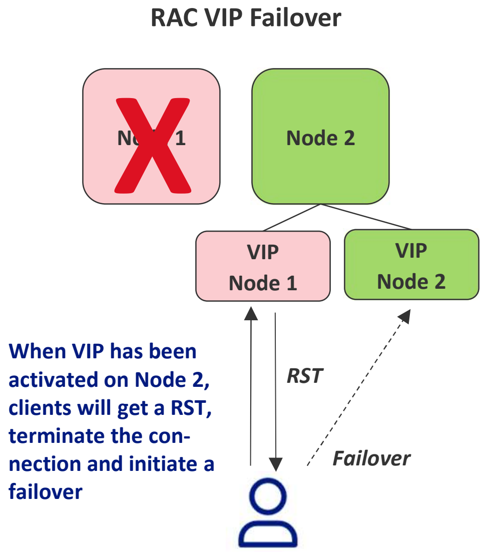
Tail Loss Probe (TLP)

Goal:
Avoid long RTOs. If no ACK has been received within a short Probe Timeout (PTO), retransmit the last segment to trigger fast recovery (SACK).

PTO Range:
max(2 * SRTT, 10 ms)

TCP Timeouts – RAC VIP Failover





When all RAC nodes are down or unreachable, clients have to wait until the timeouts expire before they will initiate a failover!

RAC VIP Failover

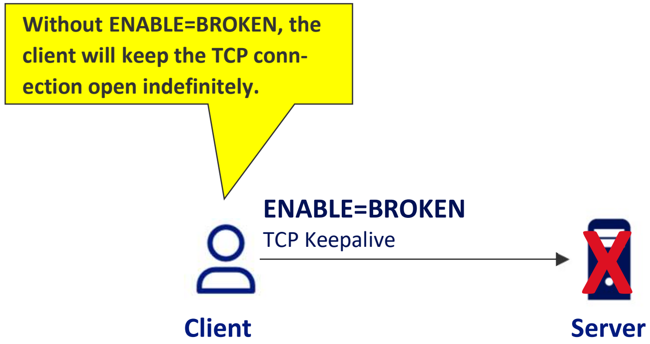
When a RAC node fails or gets evicted, its VIP fails over to another remaining node.

As soon as the VIP gets activated on a remaining node, the remaining node will reply with a RST to all clients that send packets to the VIP that failed over.

This way, clients do not need to wait until the TCP connect or retransmit timeouts expire and can initiate a failover immediately.

Note that this will only work as long as one RAC node is available and reachable. If all nodes are down or unreachable, clients have to wait until the timeouts expire before they can initiate a failover.

Idle Connections – TCP Keepalive: Client to Server



The server can fail unexpectedly.

- **"Active" connections will time out after 924.6 sec by default (tcp_retries2 = 15)**
- **With ENABLE=BROKEN, "idle" connections will time out after >2h, as defined by the following tunables:**

Tunable	Description	Default Value Exadata	Default Value Linux
ipv4.tcp_keepalive_time	Time in sec a connection must be idle before the first keepalive probe is sent.	900	7200
ipv4.tcp_keepalive_probes	Number of unacknowledged keepalive probes to send before considering the connection dead.	20	9
ipv4.tcp_keepalive_intvl	Interval in sec between keepalive probes.	75	75

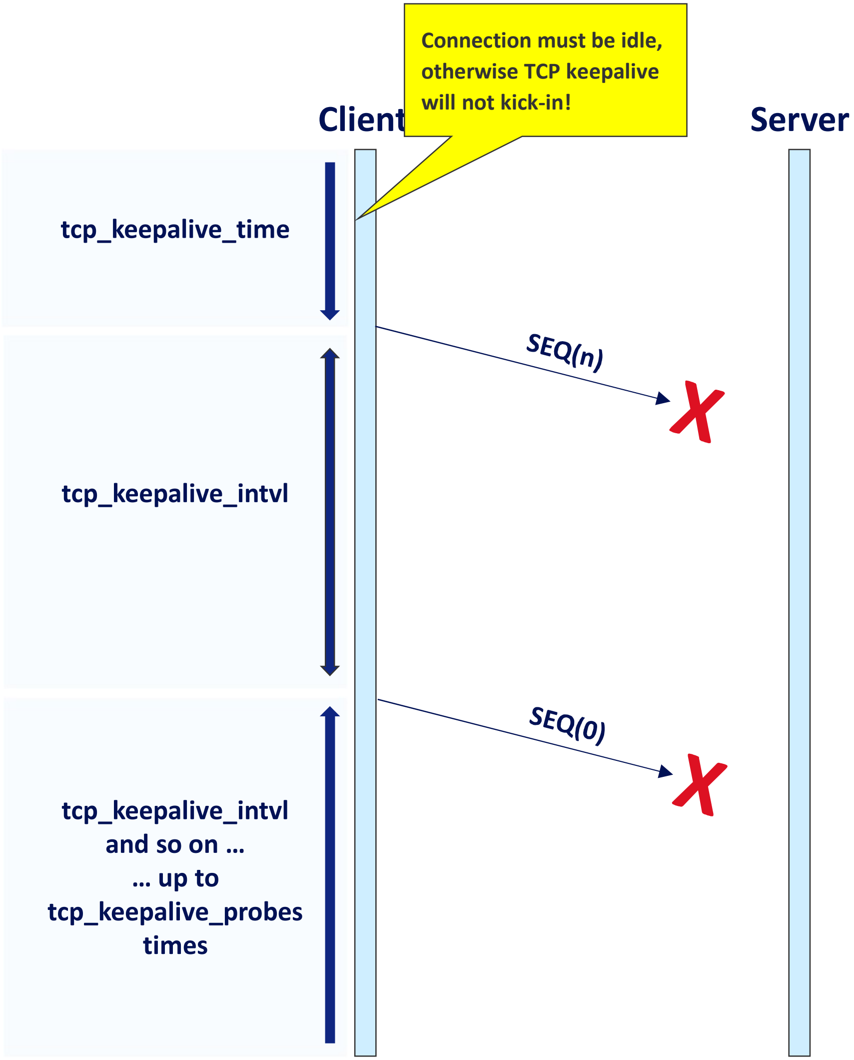
ENABLE=BROKEN

The ENABLE=BROKEN clause will enable TCP keepalive on the TCP socket.

Note though that TCP keepalive will only be used on idle sockets, that is, sockets without activity. If there is activity on a socket, the normal TCP retransmit timeouts apply.

Moreover, ENABLE=BROKEN relies on the OS level keepalive settings and depending on those, it may still take a long time before a client will consider a connection dead and clean it up (> 2h by default on Linux)!

TCP Keepalive – Idle Connections



TCP Keepalive

TCP keepalive is only triggered when a connection is inactive and idle (tcp_keepalive_time).

When the target host is unavailable or unreachable, a keepalive probe will generate no response. After an interval, (tcp_keepalive_intvl) the sender continues to repeat the probe multiple times (tcp_keepalive_probes).

When the connection is identified as down, the kernel frees up connection resources but will not notify the application about the timeout. Instead, the next read or write operation on the socket will fail with ETIMEDOUT.

TCP Keepalive – SQL*Plus Example

Enable keepalive on the TCP socket.

```
nsconbrok: asking transport to enable NTOBROKEN
nttctl: entry
setsockopt(9, SOL_SOCKET, SO_KEEPALIVE, [1], 4) = 0 <0.000013>
```

Send keepalive probe packets.

```
4 IP xxx.xxx.xxx.xxx.18426 > yyy.yyy.yyy.yyy.1521: Flags [.], ack 2643439806, win 401, length 0
21:45:04    003 IP xxx.xxx.xxx.xxx.18426 > yyy.yyy.yyy.yyy.1521: Flags [.], ack 1, win 401, length 0
21:45:07.43   05 IP xxx.xxx.xxx.xxx.18426 > yyy.yyy.yyy.yyy.1521: Flags [.], ack 1, win 401, length 0
21:45:10.446001 IP xxx.xxx.xxx.xxx.18426 > yyy.yyy.yyy.yyy.1521: Flags [.], ack 1, win 401, length 0
21:45:13.454002 IP xxx.xxx.xxx.xxx.18426 > yyy.yyy.yyy.yyy.1521: Flags [.], ack 1, win 40
21:45:16.462002 IP xxx.xxx.xxx.xxx.18426 > yyy.yyy.yyy.yyy.1521: Flags [.], ack 1, win 40
21:45:19.470004 IP xxx.xxx.xxx.xxx.18426 > yyy.yyy.yyy.yyy.1521: Flags [.], ack 1, win 40
21:45:22.478004 IP xxx.xxx.xxx.xxx.18426 > yyy.yyy.yyy.yyy.1521: Flags [.], ack 1, wi
21:45:25.486018 IP xxx.xxx.xxx.xxx.18426 > yyy.yyy.yyy.yyy.1521: Flags [.], ack 1, win 401, length 0
21:45:28.494002 IP xxx.xxx.xxx.xxx.18426 > yyy.yyy.yyy.yyy.1521: Flags [.], ack 1, win 401, length 0
21:45:31.502003 IP xxx.xxx.xxx.xxx.18426 > yyy.yyy.yyy.yyy.1521: Flags [R.], seq 1, ack 1, win 401, length 0
```

Reset connection when all keepalive probes remain un-acknowledged.

```
nsdofls: sending NSPTDA packet
nspsend: entry
nspsend: plen=396, type=6
nttwr: entry
write(9, "\0\0\1\214\6$\0\0\0\0\177\...") -1 ETIMEDOUT (Connection timed out)
```

The next write on the socket after the connection had been reset, fails with ETIMEDOUT.

```
ERROR at line 1:
ORA-03113: end-of-file on communication channel
Process ID: 394790
Session ID: 1072 Serial number: 31992
```

Oracle SQL*Plus fails with an ORA-03113 error.

TCP Keepalive Details

TCP Keepalive on a socket is enabled with the **setsockopt()** system call.

The first keepalive probe begins by transmitting a previously ACK'ed TCP segment that has a sequence number one less than the current sequence number. All keepalive packets have a length of 0.

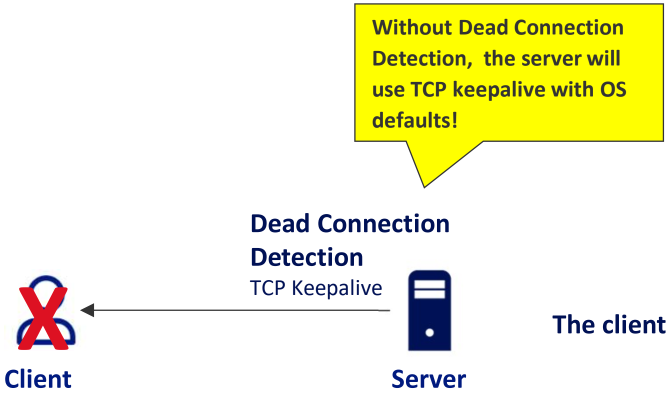
If the target host maintains an active connection, the sender receives an ACK and knows the connection is alive.

When the target host is unavailable or unreachable, the keepalive probes will not get acknowledged, the sender will eventually identify the connection as down / broken and send a RST.

Settings used in this example:

- ipv4.tcp_keepalive_time = 300
- Ipv4.tcp_keepalive_probes = 10
- ipv4.tcp_keepalive_intvl = 3

Idle Connections – TCP Keepalive: Server to Client



The client can fail unexpectedly.

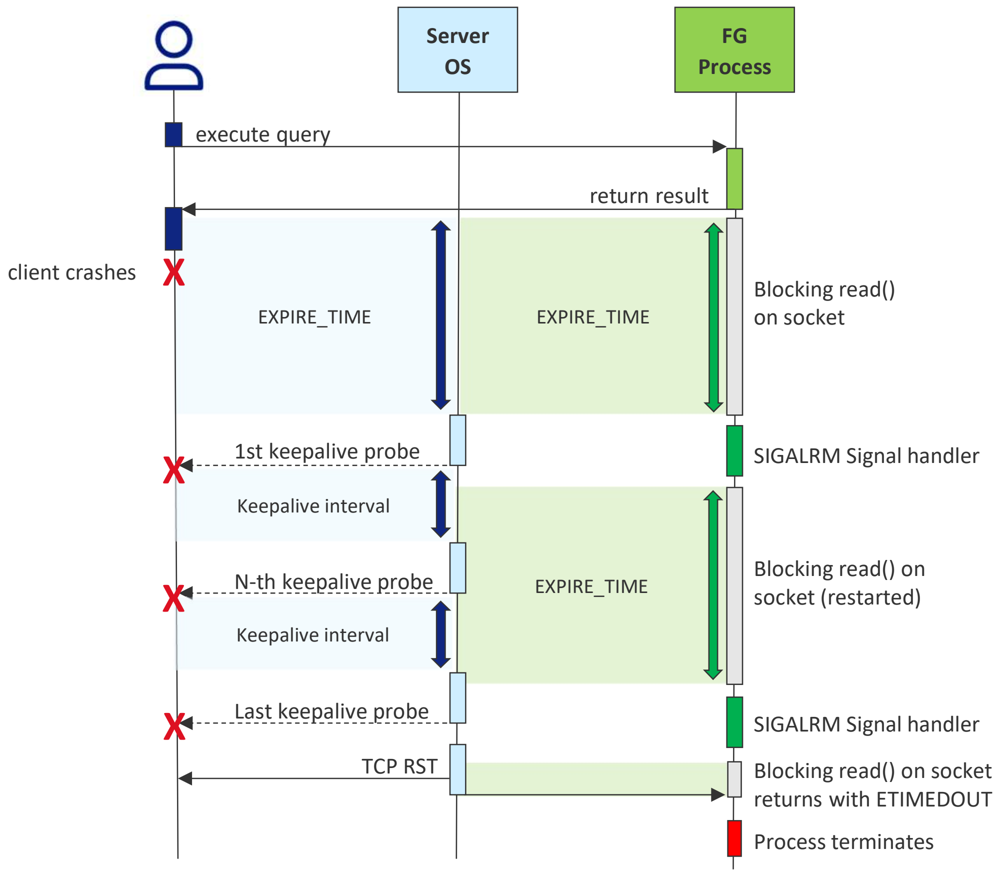
- **"Active" connections will time out after 924.6 sec by default (tcp_retries2 = 15)**
- **On the server-side, TCP keepalive is always enabled by default but uses the OS default settings with which detection of dead connections may take a long time!**
- **With SQLNET.EXPIRE_TIME (in minutes), the "idle" time after which connections are checked with keepalive probes can be configured. The number of probes and interval time between the probes is hardcoded (10 probes, 6 seconds interval).**
- **If the keepalive probes remain unacknowledged, the server process will terminate.**

Dead Connection Detection

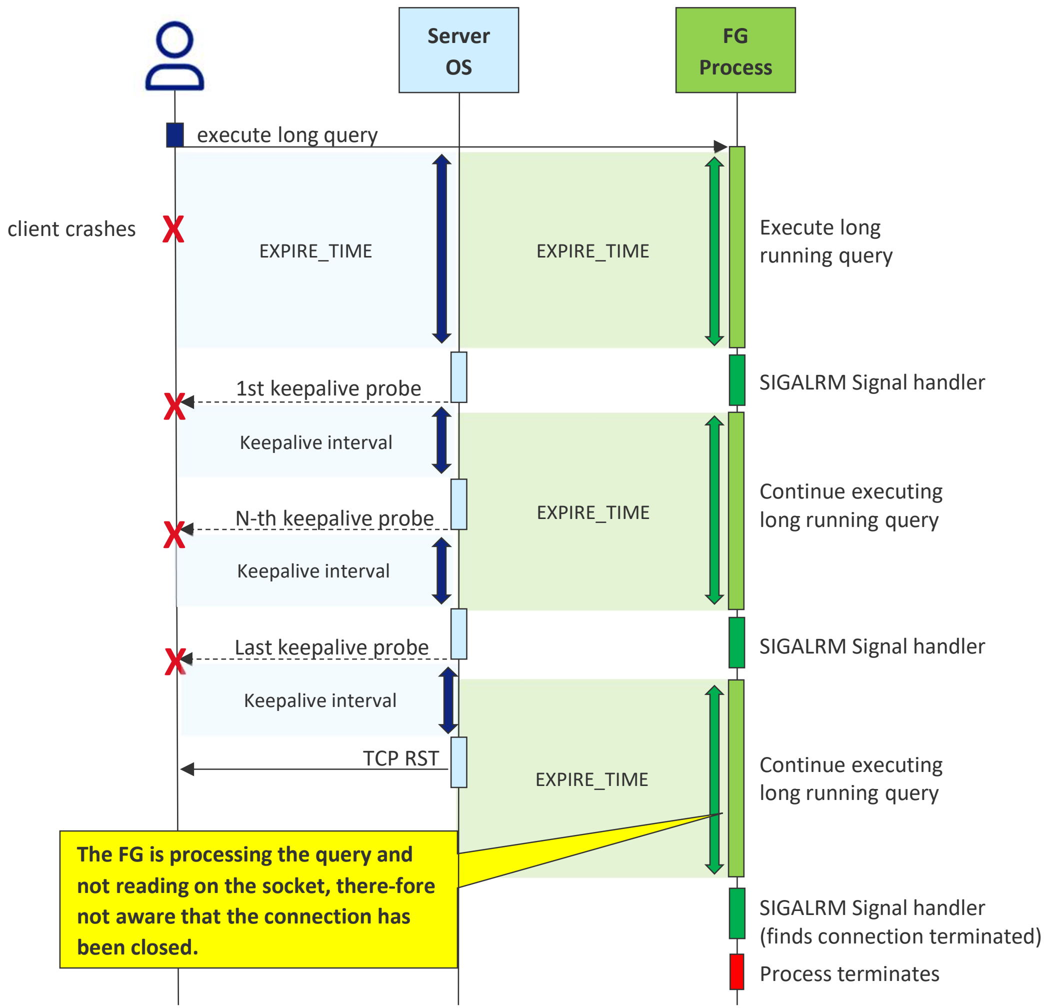
Dead Connection Detection is a server-side mechanism to identify and clean up dead TCP connections.

It is activated via the **SQLNET.EXPIRE_TIME** parameter in sqlnet.ora

Idle Connections – TCP Keepalive: Server to Client



Case 1: Socket idle + Session Idle



Case 2: Socket idle but Session Active

Dead Connection Detection – Example (1/3)

```
nttctl: entry
setsockopt(14, SOL_SOCKET, SO_KEEPALIVE, [1], 4) = 0

nttctl: entry
setsockopt(14, SOL_TCP, TCP_KEEPIDLE, [600], 4) = 0 <0.000008>
setsockopt(14, SOL_TCP, TCP_KEEPINTVL, [6], 4) = 0 <0.000007>
setsockopt(14, SOL_TCP, TCP_KEEPCNT, [10], 4) = 0 <0.000007>
nsconbrok: OS keep-alive options tuned
```

Enable TCP keepalive on the TCP socket.

The server sends 10 TCP keepalive probes

```
IP yyy.yyy.yyy.yyy.1521 > xxx.xxx.xxx.xxx.41262: Flags [.], ack 1301082556, win 668, length 0
IP yyy.yyy.yyy.yyy.1521 > xxx.xxx.xxx.xxx.41262: Flags [.], ack 1, win 668, length 0
IP yyy.yyy.yyy.yyy.1521 > xxx.xxx.xxx.xxx.41262: Flags [.], ack 1, win 668, length 0
IP yyy.yyy.yyy.yyy.1521 > xxx.xxx.xxx.xxx.41262: Flags [.], ack 1, win 668, length 0
IP yyy.yyy.yyy.yyy.1521 > xxx.xxx.xxx.xxx.41262: Flags [.], ack 1, win 668, length 0
IP yyy.yyy.yyy.yyy.1521 > xxx.xxx.xxx.xxx.41262: Flags [.], ack 1, win 668, length 0
IP yyy.yyy.yyy.yyy.1521 > xxx.xxx.xxx.xxx.41262: Flags [.], ack 1, win 668, length 0
IP yyy.yyy.yyy.yyy.1521 > xxx.xxx.xxx.xxx.41262: Flags [.], ack 1, win 668, length 0
IP yyy.yyy.yyy.yyy.1521 > xxx.xxx.xxx.xxx.41262: Flags [.], ack 1, win 668, length 0
IP yyy.yyy.yyy.yyy.1521 > xxx.xxx.xxx.xxx.41262: Flags [.], ack 1, win 668, length 0
IP yyy.yyy.yyy.yyy.1521 > xxx.xxx.xxx.xxx.41262: Flags [R.], seq 1, ack 1, win 668, length 0
```

Reset connection when all keepalive probes remain un-acknowledged.

Dead Connection Detection Details

In versions 12c+, Oracle uses the TCP keepalive mechanism on the OS and sets the following socket options (on Linux):

```
TCP_KEEPIDLE = SQLNET.EXPIRE_TIME
TCP_KEEPINTVL = 6 (hard coded value)
TCP_KEEPCNT = 10 (hard coded value)
```

In versions < 12, Oracle used a mecha-nism based on SQL*Net probe packets. This pre-12c mechanism can be enabled by setting the following parameter in the server-side sqlnet.ora:

USE_NS_PROBES_FOR_DCD=TRUE

Dead Connection Detection – Example (2/3)

Case 1: Socket idle + Session Idle

The signal handler interrupts the *read()* call on the socket and restarts it after execution (s. below)

```
read(14, 0x7fffeff51ed6, 8208) = ? ERESTARTSYS (To be restarted if SA_RESTART is set) <60.000006>
```

The signal handler executes and arms a new alarm (using EXPIRE_TIME; 1 min in this example)

```
--- SIGALRM {si_signo=SIGALRM, si_code=SI_KERNEL} ---
poll([{fd=14, events=POLLIN|POLLRDNORM}], 1, 0) = 0 (Timeout) <0.000008>
rt_sigprocmask(SIG_BLOCK, [ALRM], NULL, 8) = 0 <0.000019>
setitimer(ITIMER_REAL, {it_interval={tv_sec=0, tv_usec=0}, it_value={tv_sec=60, tv_usec=0}}, NULL) = 0
rt_sigprocmask(SIG_UNBLOCK, [ALRM], NULL, 8) = 0 <0.000006>
rt_sigreturn({mask=[]}) = 0 <0.000006>
```

The read() on the socket resumes and receives a timeout when the OS has closed the connection.

```
read(14, 0x7fffeff51ed6, 8208) = -1 ETIMEDOUT (Connection timed out) <6.800357>
```

The server process aborts with TNS-12535 and TNS-12560 errors

```
nserror: nsres: id=0, op=68, ns=12535, ns2=12560; nt[0]=505, nt[1]=110
```

Dead Connection Detection Details

When the socket and the session both are idle, that is, when the session is waiting for input from the client, the *read()* on the socket blocks until it gets input from the client or until it hits an error (which is the case when the OS closes the underlying socket after all TCP keepalive probes have been unacknow-ledged).

Note that the signal handler still periodically runs (interval defined by SQLNET.EXPIRE_TIME) and checks the stocket's status with *poll()*. However, chances that the signal handler will detect the broken connection are low (it only detects the broken connection if the OS closes it before the call to *poll()*).

Dead Connection Detection – Example (3/3)

Case 2: Socket idle + Session Active

The signal handler interrupts whatever the FG process has been executing (running a query, for instance).

```
--- SIGALRM {si_signo=SIGALRM, si_code=SI_KERNEL} ---
```

The *poll()* returns with the "error" and "hangup" flags set, therefore the server now knows that the connection to the client has been closed.

```
poll([{fd=14, events=POLLIN|POLLRDNORM}], 1, 0) = 1 ([{fd=14, revents=POLLIN|POLLRDNORM|POLLERR|POLLHUP}])
```

The signal handler calls *recvfrom()* with the MSG_PEEK flags. This doesn't return any data but results in a timeout error. This "verifies" the broken connection.

```
recvfrom(14, 0x7ffffffe5920, 1, MSG_PEEK, NULL, NULL) = -1 ETIMEDOUT (Connection timed out) <0.000007>
```

```
rt_sigprocmask(SIG_BLOCK, [ALRM], NULL, 8) = 0 <0.000006>
setitimer(ITIMER_REAL, {it_interval={tv_sec=0, tv_usec=0}, it_value={tv_sec=60, tv_usec=0}}, NULL) = 0
rt_sigprocmask(SIG_UNBLOCK, [ALRM], NULL, 8) = 0 <0.000006>
rt_sigreturn({mask=[]}) = 0 <0.000007>
```

The signal handler arms the alarm again. This would not be necessary though.

The server process aborts with TNS-12535 and TNS-12560 errors

```
nserror: nsres: id=0, op=68, ns=12535, ns2=12560; nt[0]=505, nt[1]=110
```

Dead Connection Detection Details

When the socket is idle but the session (or process rather) is executing user activity, like running a query for instance, the signal handler periodically interrupts execution and checks if the connection is still alive.

It does this using *poll()* and *recvfrom()* system calls. The return codes and flags set by these system calls will inform the server about the status of the connection. If the connection is found to be dead, the server process will terminate.

Note: In this case, the server process is not reading on a socket and waiting for user input. Therefore, the only way for the server to learn that the connection is dead, is via the signal handler.

Established Connections – Firewalls

A firewall (or other network device) can close connections unexpectedly.

Client

Active Connections:
- **Retransmits with timeout after 924.6 sec by default (tcp_retries2 = 15)**

Idle Connections
- **With ENABLE=BROKEN: timeout after ~2h**
- **Without ENABLE=BROKEN: dead connections will linger indefinitely**

Server

Active Connections:
- **Retransmits with timeout after 924.6 sec by default (tcp_retries2 = 15)**

Idle Connections
- **With DCD: timeout after SQLNET.EXPIRE_TIME + 60 sec (10 probes every 6 sec)**
- **Without DCD: dead connections will linger until the OS default keepalive timeouts kick in!)**

Firewalls

When a firewall between client and server closes an established connection, the following will happen:

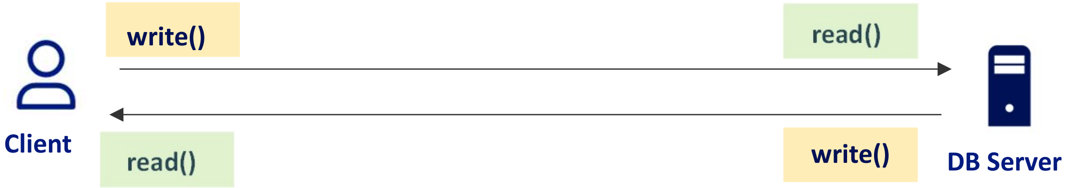
Active connections: the regular TCP retransmission mechanism will kick-in until the RTO expires (after 924.6 sec on Linux by default).

Idle connection: the TCP keepalive mechanism will identify dead con-nections and clean them up eventually.

On the server-side, Oracle always enables TCP keepalive implicitly and allows some fine tuning via Dead Connection Det-ection / SQLNET.EXPIRE_TIME.

On the client side, TCP keepalive is only enabled when ENABLE=BROKEN is used. Without that, dead client connections will linger around forever!

Established Connections – Send & Receive Timeouts



SQLNET.RECV_TIMEOUT

This sets the SO_RCVTIMEO option on the socket.

If data has been sent on a socket and a subsequent read blocks for the speci-fied period of time, it will fail with an error (EAGAIN).

SQLNET.SEND_TIMEOUT

This sets the SO_SNDTIMEO option on the socket.

If data has been received on a socket and a subsequent write blocks for the specified period of time, it will fail with an error (EAGAIN).

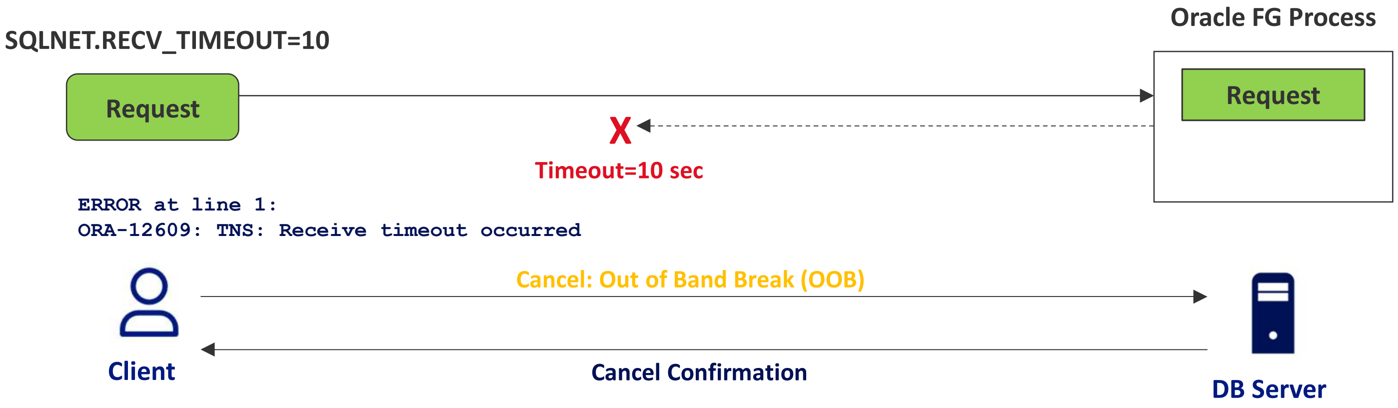
Send & Receive Timeouts

The send and receive timeouts will limit the time for the database server and client to complete send and receive operations.

The timeouts can be set in the server- and client-side sqlnet.ora via the RECV_TIMEOUT and SEND_TIMEOUT parameters.

Be careful with these settings, you probably don't need them often in practice!

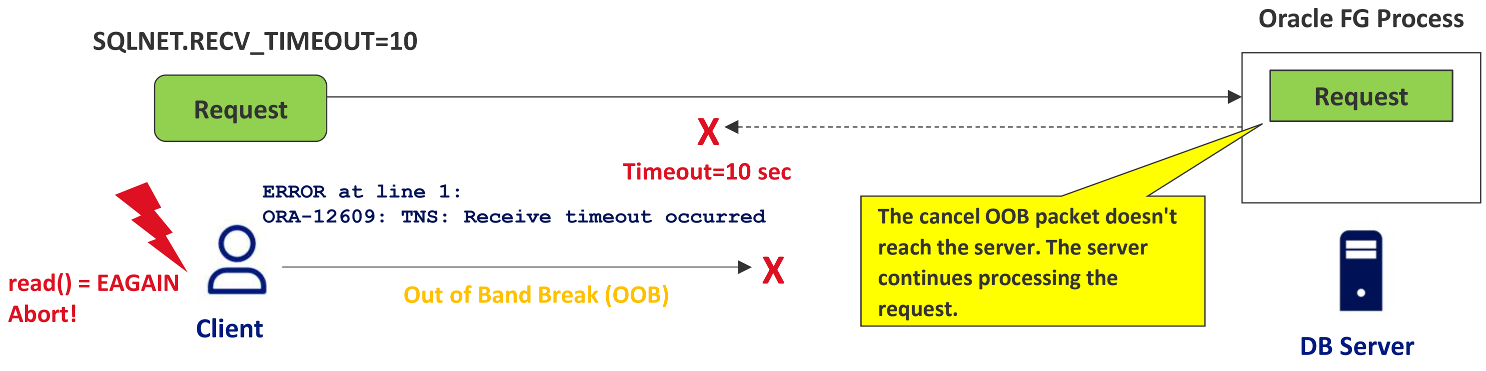
Established Connections – Receive Timeout: Happy Path




```
setsockopt(9, SOL_SOCKET, SO_RCVTIMEO, "\n\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 16) = 0 <0.000012>

read(9, 0x92e866, 8208) = -1 EAGAIN (Resource temporarily unavailable) <10.003321>

nserror: nsres: id=0, op=68, ns=12535, ns2=12609; nt[0]=0, nt"..., 145) = 145

nsdo: sending ATTN
sendto(9, "!", 1, MSG_OOB, NULL, 0) = 1 <0.000025>
nsdo: 1 urgent byte to transport

nioqrs:  state = interrupted (1)
nioqrs: nioqrs: sending reset marker...
nioqsm: entry\n
nioqsm:  Sending reset packet (2)...

read(9, "\0\0\0\374\6$..., 8208) = 252
```

The client and server sessions will remain open and the client can continue sending new requests to the server.

Established Connections – Receive Timeout: Unhappy Path




```
setsockopt(9, SOL_SOCKET, SO_RCVTIMEO, "\n\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 16) = 0 <0.000012>

read(9, 0x92e866, 8208) = -1 EAGAIN (Resource temporarily unavailable) <10.003321>

nserror: nsres: id=0, op=68, ns=12535, ns2=12609; nt[0]=0, nt"..., 145) = 145

nsdo: sending ATTN
sendto(9, "!", 1, MSG_OOB, NULL, 0) = 1 <0.00
nsdo: 1 urgent byte to transport

nioqrs:  state = interrupted (1)
nioqrs: nioqrs: sending reset marker...
nioqsm: entry\n
nioqsm:  Sending reset packet (2)...

read(9, 0x16446c6, 8208) = -1 EAGAIN (Resource temporarily unavailable) <10.229542>
```

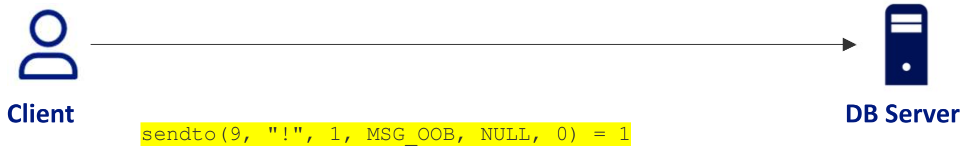
The client doesn't get the cancellation confirmed, waits for another RECV_ TIMEOUT interval and then aborts. This means it'll take 2 x RECV_TIMEOUT for the client to fail.

Appendix D: Out Of Band Breaks (OOB)

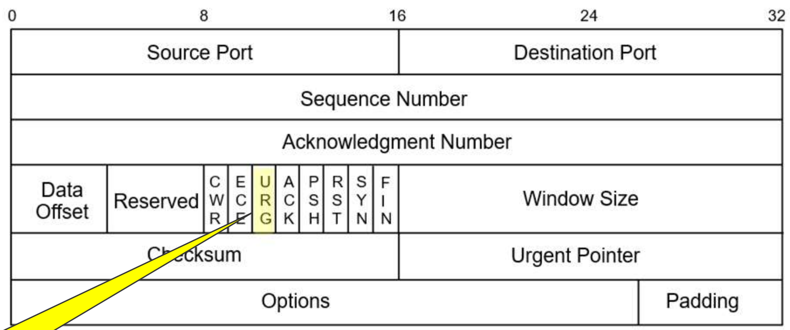
Out of Band Breaks (OOB)

Run Query

CTRL+C



`sendto(9, "!", 1, MSG_OOB, NULL, 0) = 1`

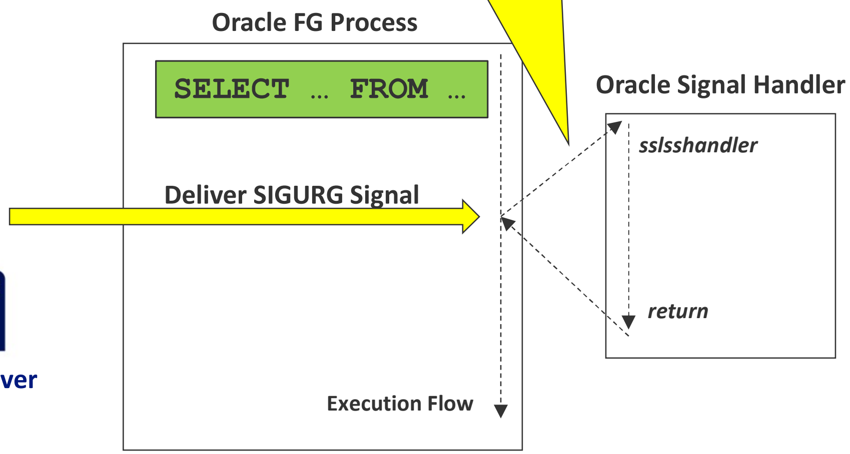


The client will send a 1-byte TCP packet with a payload of "!" and the URG flag set.

OOB is controlled by the DISABLE_OOB parameter in sqlnet.ora, which defaults to FALSE (OOB is enabled).

Note: This is a client-side only parameter! (setting it on the server will have no effect).

When the SIGURG signal is delivered, the process is interrupted and execution re-directed to the Oracle signal handler function sslsshandler.




```
#0  0x00000000042cd6b0 in sslsshandler ()
#1  <signal handler called>
#2  0x000000012c921c0 in smbget ()
#3  0x000000012c8ee7c in sorgetqbf ()
#4  0x000000012dfa9ce in qersoFetchSimple ()
#5  0x000000012df6d12 in qersoFetch ()
#6  0x000000000409ab98 in qerjoCartesianFetch ()
#7  0x000000012e01140 in qergsFetch ()
#8  0x000000012ba1f39 in opifch2 ()
#9  0x000000012c1c94b in kpoal8 ()
#10 0x000000012ba8f52 in opiodr ()
#11 0x000000012eda603 in ttcpip ()
#12 0x0000000002894d8c in opitsk ()
#13 0x0000000002899718 in opiino ()
#14 0x000000012ba8f52 in opiodr ()
#15 0x0000000002890b06 in opidrv ()
#16 0x000000000344dec5 in sou2o ()
#17 0x0000000000dce1a6 in opimai_real ()
#18 0x00000000034598e1 in ssthrdmain ()
#19 0x0000000000dcdfd0 in main ()
```

Auto OOB (new in 19c)

Server waiting for POLLPRI indefinitely.

```
poll([{fd=14, events=POLLIN|POLLPRI|POLLRDNORM}], 1, -1)
```

```
nsevdcall: Sending OOB and ATTN
sendto(9, "!", 1, MSG_OOB, NULL, 0)
```

Client sending OOB packet.

SQL*Net Trace Server (OOB ok)

```
nsaccept: Checking OOB Support
sntpoltsts: fd 14 need 43 readiness event, wait time -1
sntpoltsts: fd 14 has 4 readiness ev
sntpoltsts: exit
nsaccept: OOB is Reaching Perfectly
```

SQL*Net Trace Server (OOB failed)

```
nsaccept: Checking OOB Support
sntpoltsts: fd 14 need 43 readiness event, wait time -1
sntpoltsts: fd 14 has 2 readiness ev
sntpoltsts: exit
nttctl: entry
nsaccept: OOB is getting dropped
```

Notes:

- **DISABLE_OOB is a client- and DISABLE_OOB_AUTO a server-side parameter!**
- **Changing DISABLE_OOB_AUTO will NOT require a server restart.**

Oracle 19c Auto OOB

With Auto OOB in Oracle 19c, the server will check if OOB is supported at con-nect time.

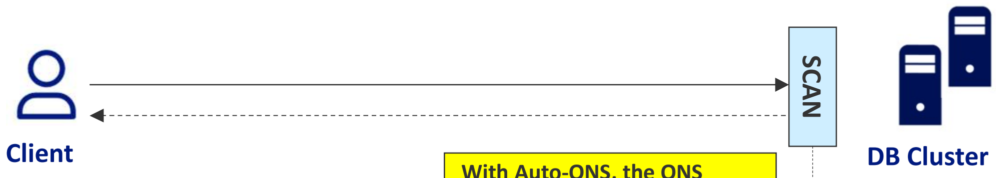
Technically, the Auto OOB support check consists of Oracle waiting on the **POLLPRI** event on the connection's socket. The Oracle server uses **poll()** with infinite timeout for that purpose.

If the server doesn't get a POLLPRI ev-ent on the connection's socket within the time limit defined by INBOUND_ CONNECT_TIMEOUT, the timer alarm will fire and the Oracle server process will fail with ORA-609 and TNS-12637 errors. This can occur if firewalls drop or clear TCP packets with the URG flag!

Auto OOB is controlled by the sqlnet.ora parameter DISABLE_OOB_AUTO, which defaults to FALSE (Auto OOB enabled). Note that this is a server-side only parameter (setting it on the client will have no effect).

Appendix E:
Fast Application Notification (FAN)

Fast Application Notification – ONS Auto Configuration



With Auto-ONS, the ONS Node Groups are created automatically based on the SCAN!

ONS Node Groups

```
# Primary
oracle.ons.nodes.001=node1a:6200,node1b:6200,node1c:6200

# Standby
oracle.ons.nodes.002=node2a:6200,node2b:6200,node2c:6200
```

In versions <12c, these ONS Node Groups had to be configured and maintained.

ONS Auto Configuration ("Auto-ONS")

In Oracle 12c and higher, FAN is auto-configured. This means that when the client starts, it queries the databases for the ONS end-points automatically.

The automatic configuration spans data centers and the client auto-matically receives an ONS configuration from each database listed in the connection URL. **No configuration of ONS is required at the client other than enabling FAN.**

The client will create multiple ONS Node Groups automatically to receive FAN events from primary and standby clusters
.

If network connections to ONS (port 6200) are slow or blocked by a firewall, the client will hang for 10 sec!

[Oracle Whitepaper: Fast Application Notification (FAN)]

Fast Application Notification – Prerequisites

For OCI based clients, FAN Notifications must be explicitly enabled! For other clients (JDBC), FAN is automatically configured and enabled in versions 12c and above!

1. Enable FAN Notifications (only for OCI based Clients)

```
srvctl modify service -d TESTDB19_011 -s my_test_rw -notification true
```

2. Check Configuration

```
srvctl config service -d TESTDB19_011 -s my_test_rw


Service name: MY_TEST_RW
Server pool:
Cardinality: 1
Service role: PRIMARY
Management policy: AUTOMATIC
DTP transaction: false
AQ HA notifications: true
Global: false
Commit Outcome: false
[…]
Preferred instances: TESTDB19_0113
Available instances: TESTDB19_0111,TESTDB19_0112,TESTDB19_0114
CSS critical: no
Service uses Java: false
```

Confusingly, this is still called "AQ HA notifications" in 19c because older FAN implementations used AQ as an event transport mecha-nism rather than ONS. Just ignore the "AQ" part in this name!

Fast Application Notification – fanWatcher Tool

1. fanWatcher Download

```
Download fanWatcher from this link (rename file to fanWatcher.java).
```

2. fanWatcher Installation

```
$ORACLE_HOME/jdk/bin/javac                                         \
-cp $ORACLE_HOME/jdbc/lib/ojdbc8.jar:$ORACLE_HOME/opmn/lib/ons.jar \
fanWatcher.java
```

You must include ojdbcN.jar, ons.jar and the current directory of fanWatcher (that is, include '.') in the classpath.

3. fanWatcher Basic Usage

```
$ORACLE_HOME/jdk/bin/java                                            \
-cp $ORACLE_HOME/jdbc/lib/ojdbc8.jar:$ORACLE_HOME/opmn/lib/ons.jar:.\
fanWatcher <config_type>

autoons:
Automatically configure ONS based on the TNS descriptor (ONS Auto-Config);
this option requires the environment variables user, password and url to be
set.

nodes="...":
Explicitly configure an ONS node list.
```

FAN & TNS Descriptors

No ADDRESS_LIST

```
(DESCRIPTION=
  (ADDRESS=(PROTOCOL=TCP)(HOST=my-scan01.mydomain.net)(PORT=1521))
  (ADDRESS=(PROTOCOL=TCP)(HOST=my-scan02.mydomain.net)(PORT=1521))
  (CONNECT_DATA=(SERVICE_NAME=MY_TEST_RW.WORLD))
)
```

```
Auto-ONS configuration=maxconnections.0001=0003
nodes.0001=MY-SCAN01.MDYDOMAN.NET:6200
maxconnections.0002=0003
nodes.0002=MY-SCAN02.MYDOMAIN.NET:6200
Opening FAN Subscriber Window ...
```

ADRESS_LIST

```
(DESCRIPTION=
  (ADDRESS_LIST=
    (ADDRESS=(PROTOCOL=TCP)(HOST=my-scan01.mydomain.net)(PORT=1521)))
  (ADDRESS_LIST=
    (ADDRESS=(PROTOCOL=TCP)(HOST=my-scan02.mydomain.net)(PORT=1521)))
  (CONNECT_DATA=(SERVICE_NAME=MY_TEST_RW.WORLD))
)
```

```
Auto-ONS configuration=maxconnections.0001=0003
nodes.0001=MY-SCAN01.MYDOMAIN.NET:6200
maxconnections.0002=0003
nodes.0002=MY-SCAN02.MYDOMAIN.NET:6200
Opening FAN Subscriber Window ...
```

EZConnect

```
my-scan01.mydomain.net:1521,
My-scan02.mydomain.net:1521/
MY_TEST_RW.WORLD
```

```
Subscribing to events of type:
Auto-ONS configuration=maxconnections.0001=0003
nodes.0001=MY-SCAN01.MYDOMAIN.NET:6200,MY-
SCAN02.MYDOMAIN.NET:6200
Opening FAN Subscriber Window ...
```

EZConnect does not support ADDRESS_LISTs. As a conseq-uence, auto-ons does not create a node group for the standby with EZConnect!

ONS Connections – TCP Timeouts Example Trace

```
./ora_connect.bt --unsafe <client_pid>

Tracing connect behavior (pid 334310). Hit ^C to stop.

[ Initial DNS & SCAN requests not shown… ]
```

Node Lsnr Request

```
23:21:01 334310/334310: connect: fd=11, nnn.nnn.nnn.nnn:1521
23:21:01 334310/334310: poll: fd=11, event=POLLOUT, timeout=4000
23:21:01 334310/334310: poll: fd=11, event=POLLIN, timeout=6000
23:21:01 334310/334310: poll: fd=11, event=POLLOUT, timeout=5000
23:21:01 334310/334310: poll: fd=11, event=POLLIN, timeout=5000
23:21:01 334310/334310: sendto: fd=11 (OOB check)
23:21:01 334310/334310: poll: fd=11, event=POLLIN, timeout=
```

Guess what, with ONS we'll get additional DNS requests… but we're done with DNS for now! :-)

DNS Lookup SCAN1

```
23:21:01 334310/334310: connect: fd=13, 1.2.3.4:53
23:21:01 334310/334310: poll: fd=13, event=POLLOUT, timeout=0
23:21:01 334310/334310: sendmmsg: fd=13, vlen=2, qname=my-scan01.mydomain.net
23:21:01 334310/334310: poll: fd=13, event=POLLIN, timeout=10000
23:21:01 334310/334310: poll: fd=13, event=POLLIN, timeout=9997
```

DNS Lookup SCAN2

```
23:21:01 334310/334310: connect: fd=13, 4.3.2.1:53
23:21:01 334310/334310: poll: fd=13, event=POLLOUT, timeou
23:21:01 334310/334310: sendmmsg: fd=13, vlen=2, qname=my-
23:21:01 334310/334310: poll: fd=13, event=POLLIN, timeout
23:21:01 334310/334310: poll: fd=13, event=POLLIN, timeout
```

The connections to ONS port 6200 are opened from newly spawned threads (s. pid and tid columns). The client spawns one thread per SCAN IP.

ONS Requests

```
23:21:01 334310/338778: connect: fd=14, xxx.xxx.xxx.36:6200
23:21:01 334310/338779: connect: fd=15, xxx.xxx.xxx.37:6200
23:21:01 334310/338780: connect: fd=16, xxx.xxx.xxx.38:6200
23:21:01 334310/338782: connect: fd=13, yyy.yyy.yyy.132:6200
23:21:01 334310/338781: connect: fd=17, yyy.yyy.yyy.134:6200
23:21:01 334310/139395: connect: fd=15, yyy.yyy.yyy.133:6200

23:21:01 334310/334310: ons_cond_timedwait_sec: entry now=1661250061,
                                                sec=1661250071, nsec=5000000
23:21:01 334310/334310: ons_cond_timedwait_sec: leave now=1661250061
```

The ONS connection timeout is implemented via threading mechanisms (*ons_cond_timedwait_sec()* is a wrapper around *pthread_cond_timedwait()*)

ONS Connections

The Oracle client spawns new threads for handling the ONS connections to the primary and the standby (one thread per SCAN IP).

The Oracle client's main thread will block on a condition variable for 10 sec with **pthread_cond_timedwait()**.

If the call to **pthread_cond_timedwait()** times out, the client proceeds without opening ONS connections!

Appendix F: SQL*Net Tracing

SQL*Net Tracing – Trace Settings

Client Trace (sqlnet.ora)

```
DIAG_ADR_ENABLED = OFF
TRACE_DIRECTORY_CLIENT = /path
TRACE_FILE_CLIENT = client
TRACE_LEVEL_CLIENT = SUPPORT
TRACE_TIMESTAMP_CLIENT = ON
TRACE_UNIQUE_CLIENT = ON
```

Listener Trace (listener.ora)

```
DIAG_ADR_ENABLED_LISTENER_name = OFF
TRACE_DIRECTORY_LISTENER_name = /path
TRACE_FILE_LISTENER_name = lsnr
TRACE_LEVEL_LISTENER_name = SUPPORT
```

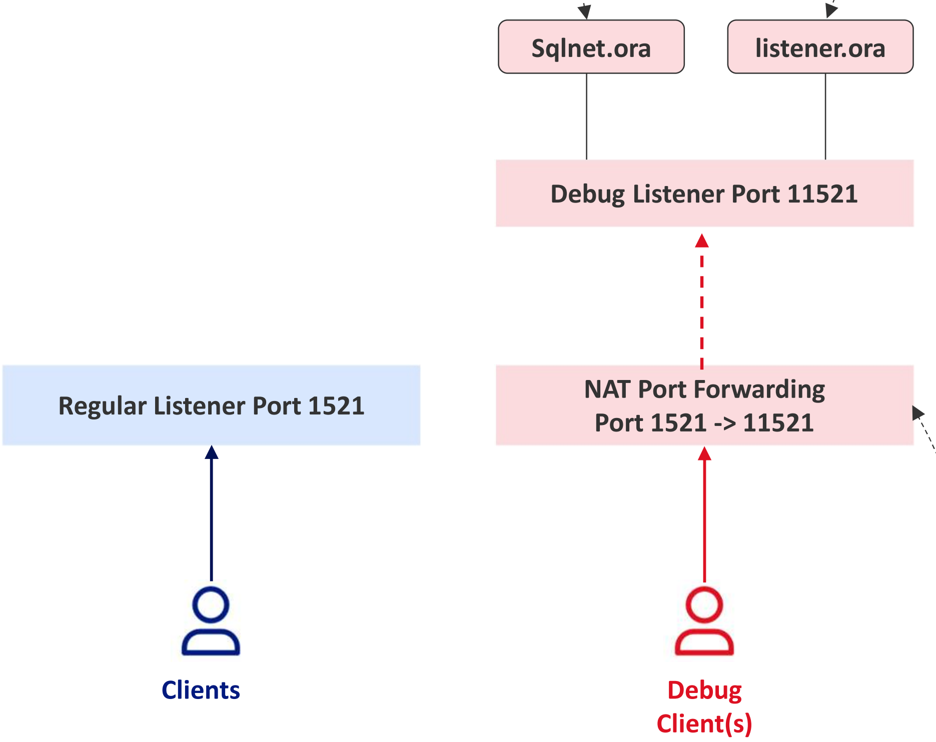
Server Trace (sqlnet.ora)

```
DIAG_ADR_ENABLED = OFF
TRACE_DIRECTORY_SERVER = /path
TRACE_FILE_SERVER = server
TRACE_LEVEL_SERVER = SUPPORT
TRACE_TIMESTAMP_SERVER = ON
TRACE_UNIQUE_SERVER = ON
```

SQL*Net Tracing – Redirect & Isolate Traffic to Debug Listener

```
DIAG_ADR_ENABLED = OFF
TRACE_DIRECTORY_SERVER = /path
TRACE_FILE_SERVER = server
TRACE_LEVEL_SERVER = SUPPORT
TRACE_TIMESTAMP_SERVER = ON
TRACE_UNIQUE_SERVER = ON
```

```
LISTENER_DEBUG =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = TCP)
          (HOST = myhostdb03-v.mydomain.net)
          (PORT = 11521)
      )
    )
  )

SID_LIST_LISTENER_DEBUG =
  (SID_LIST =
    (SID_DESC =
      (GLOBAL_DBNAME = MY_TEST_RW.WORLD)
      (ORACLE_HOME = /u01/app/oracle/product/19.0.0.0/dbhome_1920)
      (SID_NAME = TCTEST19_0113)
      (ENVS="TNS_ADMIN=/path/to/admin_debug")
    )
)

DIAG_ADR_ENABLED_LISTENER_DEBUG = OFF
TRACE_DIRECTORY_LISTENER_DEBUG = /path
TRACE_FILE_LISTENER_DEBUG = lsnr
TRACE_LEVEL_LISTENER_DEBUG = SUPPORT
```

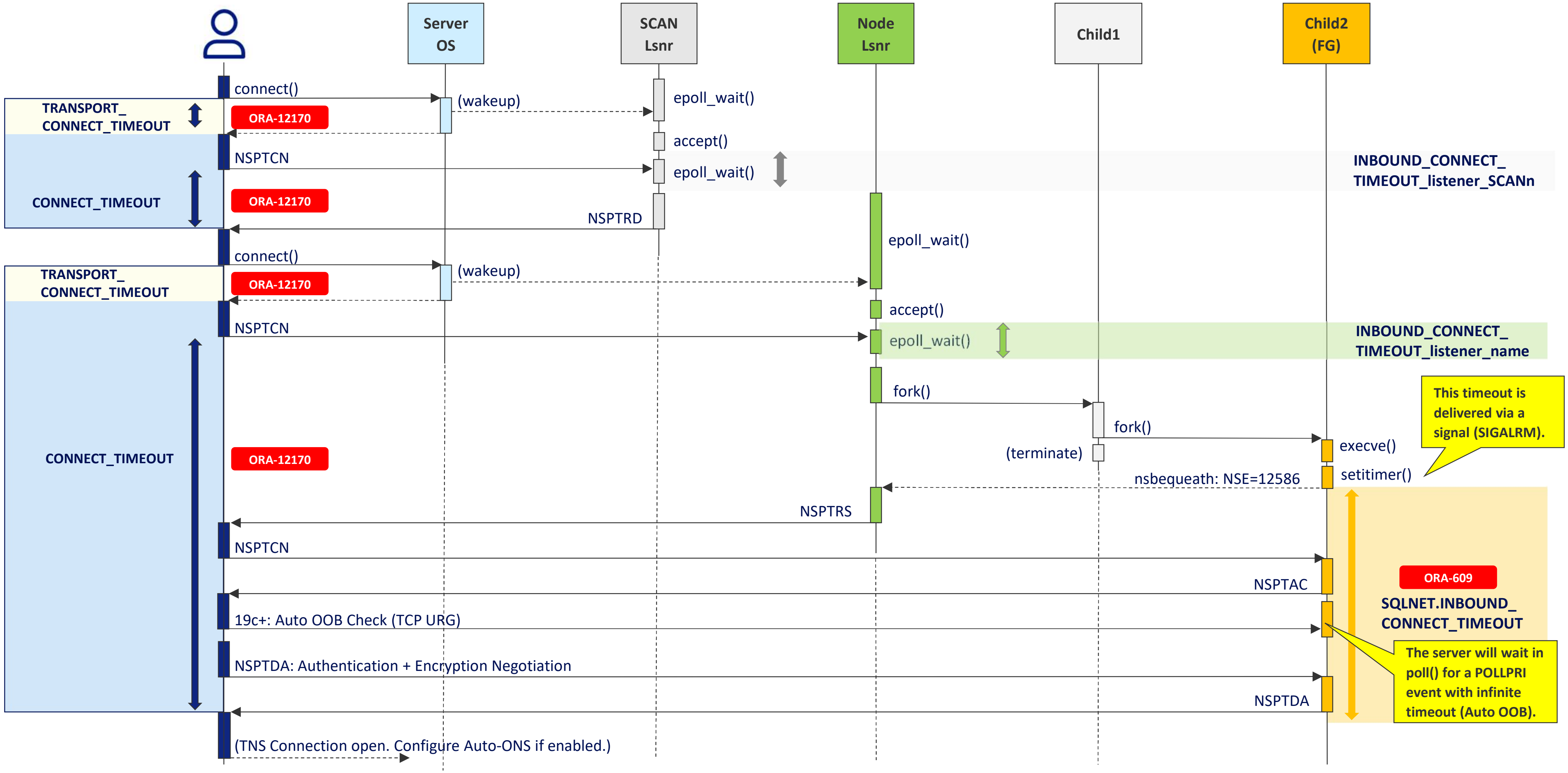



```
echo 1 > /proc/sys/net/ipv4/ip_forward

iptables -t nat -A PREROUTING -p tcp -s <source_ip> \
-d <destination_ip> --dport 1521 -j DNAT           \
--to-destination <destination_ip>:11521

iptables -A FORWARD -m state -p tcp -s <source_ip>  \
-d <destination_ip> --dport 11521                   \
--state NEW,ESTABLISHED,RELATED                     \
-j ACCEPT
```

Modified from source: Beat Ramseier, Are you fishing or catching? – Server-side SQL*Net tracing for specific clients, 2017-10-15

Appendix G – Connect Timeouts (Static Diagrams)

Connect Timeouts – The Dance Between Client and Server



Connect Timeouts – The Dance Between Client and Server (Auto-ONS)