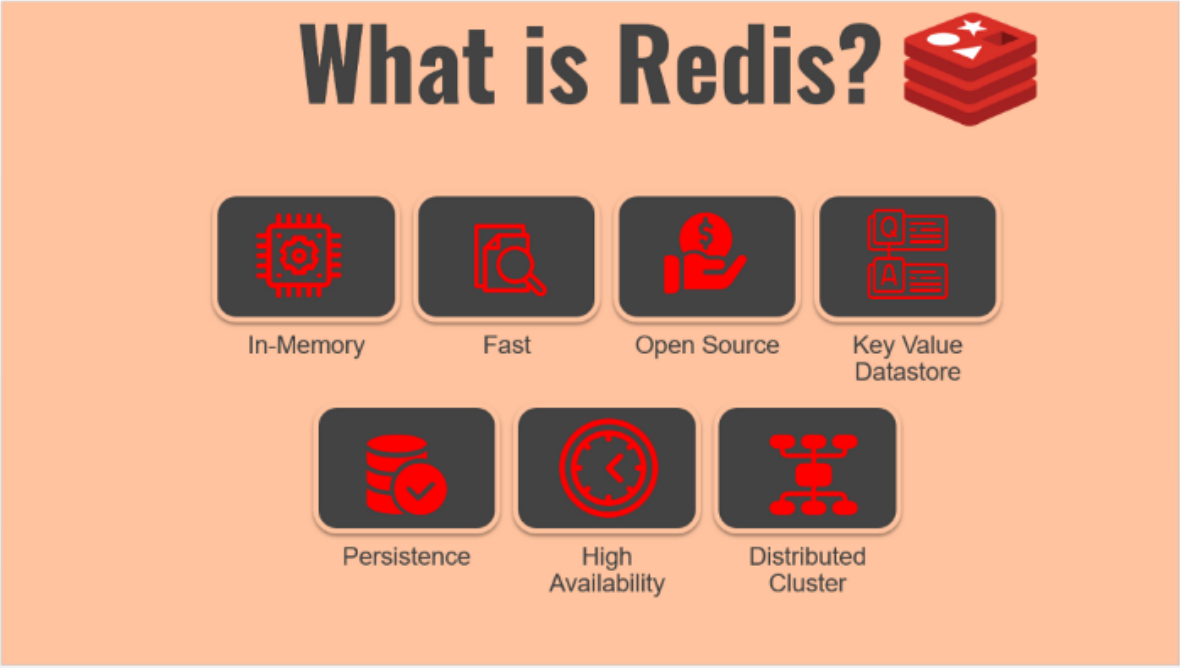
gauravtayal.medium.com

Redis Internals: why is it so fast ? - Gaurav Tayal - Medium

Gaurav Tayal

7-8 minutes



Introduction

Redis is often used as a cache. It can also be used as a storage when the consistency requirements are not high.In addition, redis also provides message subscription, transaction, index and other features. We can also use cluster features to build distributed storage services and realise non strong consistency distributed lock services.

Redis has a common advantage when using the above scenarios, that is, fast processing speed (high performance).

How fast is redis ?

To understand how fast redis is, you need to have an assessment tool.

Fortunately, redis provides such a tool and gives some experience data of commonly used hardware platforms.

1. Redis benchmark can be used to evaluate the performance of redis. The command line provides the function of evaluating the performance of specific commands in the normal / pipeline mode and under different pressures.
2. Redis has excellent performance. As a key value system, the maximum load level is 10W / s, and the set and get time consumption levels are 10ms and 5ms. Using pipeline can improve the performance of redis operation.

   redis-benchmark -t set,lpush -n 100000 -qSET: 97087.38 requests per second //Processing 97000 set requests per second
   LPUSH: 101112.23 requests per second //Process 100000 lpush requests per second
   ### Script execution times
   redis-benchmark -n 100000 -q script load
   "redis.call('set','foo','bar')"

   SCRIPT load redis.call('set','foo','bar'): 101317.12 requests per second, p50=0.255 msec

   By default, redis benchmark uses 100000 requests, 50 clients, and a payload of 3 bytes for testing.

Why is redis so fast ?

Redis is a single thread application, which means redis uses a single thread to process the client's requests.

The reasons for the high performance of redis are as follows:

- Memory storage: redis uses memory (in memory) storage, without disk IO overhead
- Single thread implementation: redis uses a single thread to process requests, avoiding the cost of thread switching and lock resource contention between multiple threads
- Non blocking IO: redis uses multiplexing IO technology to select the optimal IO implementation in poll, epool and kqueue
- Optimised data structure: redis has many optimised data structure implementations that can be directly applied. The application layer can directly use the native data structure to improve performance.

Single thread

Redis's core network model is implemented by single thread, which has caused many people's confusion at the beginning. Redis's official answer to this is:

> *It's not very frequent that CPU becomes your bottleneck with Redis, as usually Redis is either memory or network bound. For instance, using pipelining Redis running on an average Linux system can deliver even 1 million requests per second, so if your application mainly uses O(N) or O(log(N)) commands, it is hardly going to use too much CPU.*

What are the benefits of a single thread?

1. No consumption caused by thread creation or thread destruction
2. Avoid the CPU consumption caused by context switching
3. It avoids the competition problems between threads, such as adding locks, releasing locks, deadlocks, etc

   Also single-threaded mechanism greatly reduces the complexity of Redis's internal implementation. Hash's lazy Rehash, Lpush, and other "thread-unsafe" commands can be performed without lock.

I/O Model

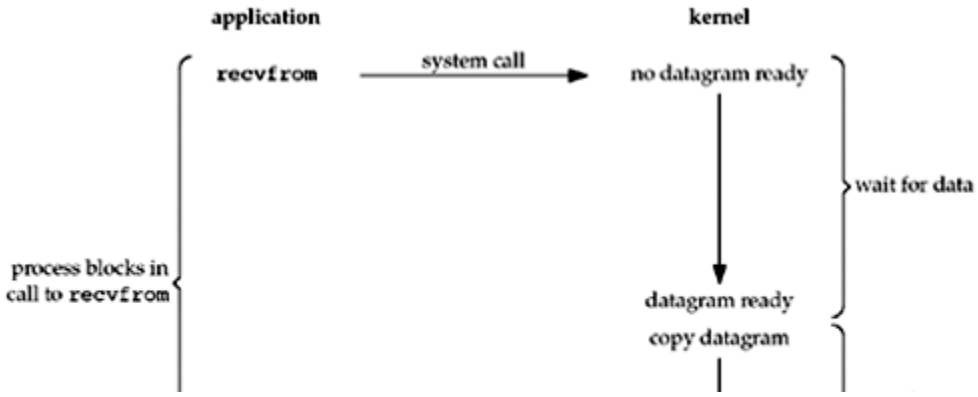
An IO operation is generally divided into two steps:

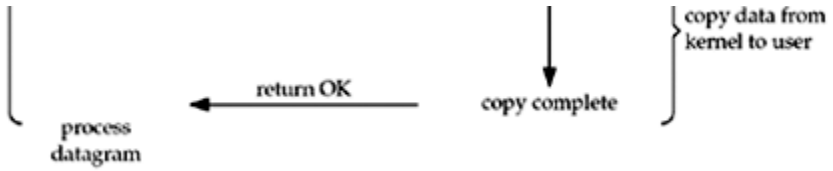
1. Wait for the data to arrive from the network, and then load it into the kernel space buffer
2. Data copied from kernel space buffer to user space buffer

   According to the two steps whether to block the thread, it can be divided into blocking / non blocking, synchronous / asynchronous.

Blocking I/O Model

The most prevalent model for I/O is the blocking I/O model, which we have used for all our examples so far in the text. By default, all sockets are blocking.



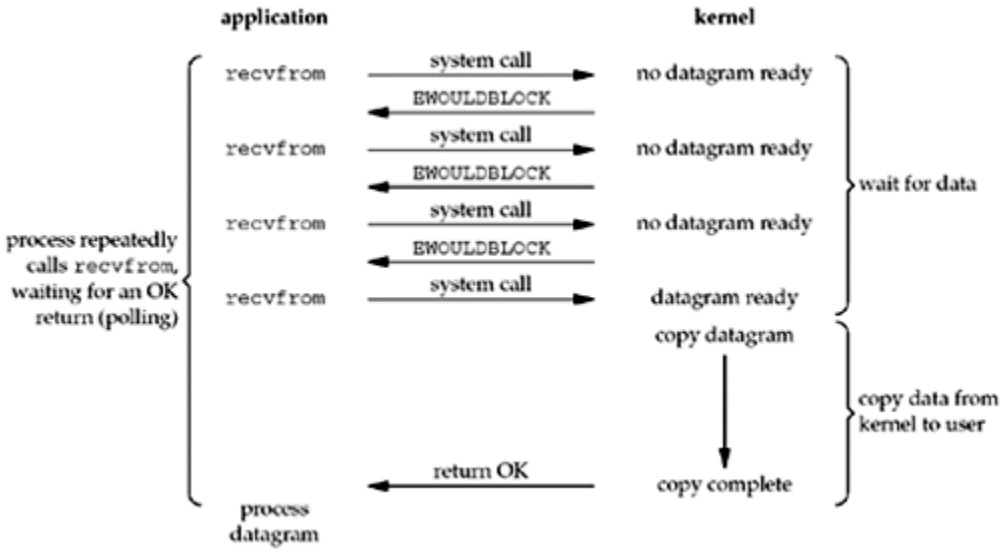


We use UDP for this example instead of TCP because with UDP, the concept of data being “ready” to read is simple: either an entire datagram has been received or it has not.

With TCP it gets more complicated, as additional variables such as the socket’s low-water mark come into play.

Non Blocking I/O Model

When we set a socket to be nonblocking, we are telling the kernel “when an I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead.”

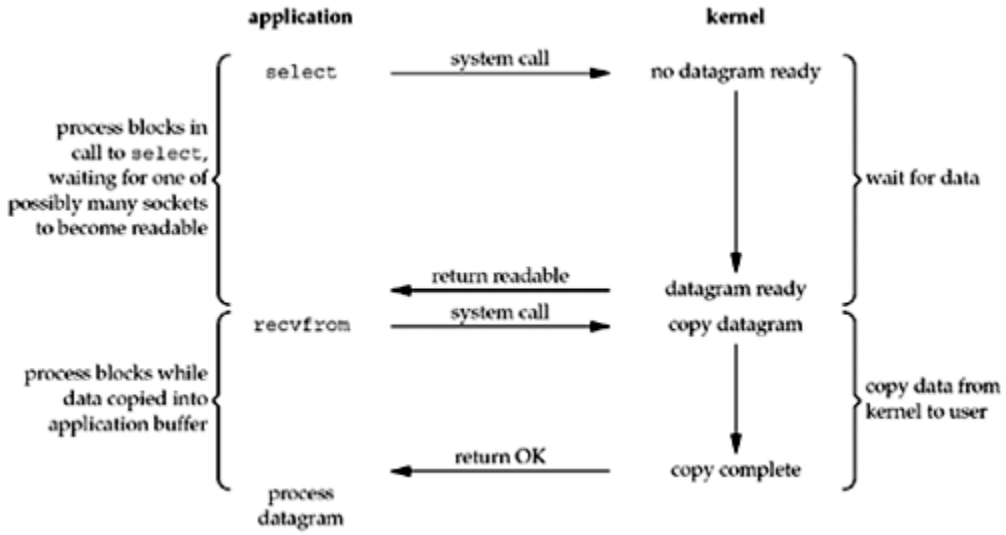


The first three times that we call *recvfrom*, there is no data to return, so the kernel immediately returns an error of *EWOULDBLOCK* instead. The fourth time we call *recvfrom*, a datagram is ready, it is copied into our application buffer, and *recvfrom* returns successfully. We then process the data.

When an application sits in a loop calling *recvfrom* on a nonblocking descriptor like this, it is called polling. The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

Multiplexing I/O Model

With I/O multiplexing, we call *select* or *poll* and block in one of these two system calls, instead of blocking in the actual I/O system call.



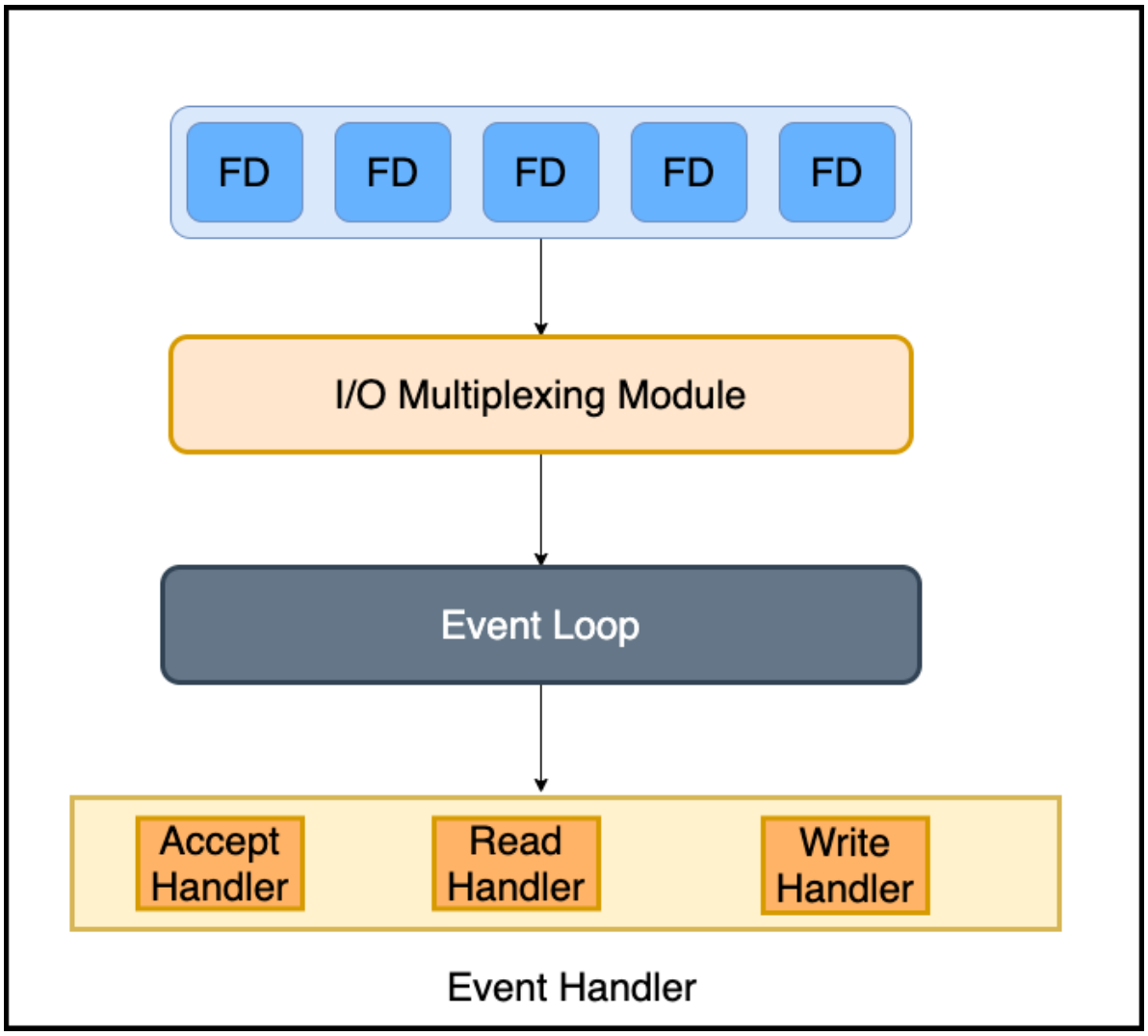
We block in a call to select, waiting for the datagram socket to be readable. When select returns that the socket is readable, we then call recvfrom to copy the datagram into our application buffer.

Comparing Multiplexing i/o to Blocking i/o, there does not appear to be any advantage, and in fact, there is a slight disadvantage because using select requires two system calls instead of one.

But the advantage in using select is that we can wait for more than one descriptor to be ready.

Now let's see how redis handles client connections ?

In general, redis uses a reactor design pattern that encapsulates multiple implementations (select, epoll, kqueue, etc.) to multiplex IO to handle requests from clients.



The reactor design pattern is often used to implement event driven. In addition, redis encapsulates different libraries for multiplexing io on different platforms.

Redis will preferentially choose the I / O multiplexing function with time complexity of O (1) as the underlying implementation, including the evport in Solaris 10, epoll in Linux and kqueue in Mac

OS / FreeBSD.

These functions all use the internal structure of the kernel and can serve hundreds of thousands of file descriptors.

However, if the current compilation environment does not have the above functions, select will be selected as an alternative. Because it will scan all the monitored descriptors when it is used, its time complexity is poor o (n), and it can only serve 1024 file descriptors at the same time, so it is not generally used as the first scheme.