XRootD CMSD SelByLoad Analysis

January 9, 2024

Tom Byrne

1 Introduction

At RAL, we've been having some issues trying to get the XRootD CMSD loadbalancing to work well for Echo, our Ceph storage cluster with ~20 XRootD gateway nodes. In particular there are issues with gateway nodes 'hotspotting' - receiving huge numbers of transfers and then failing functional tests (and real transfers) as they run out of physical resources. Although we have made progress in some places with configuration changes dramatically improving this, we continue to observe situations where some nodes receive large numbers of redirects from the CMSD managers and then become overloaded.

Part of the investigation has been looking at the underlying load balancing algorithm to try and understand some of the decisions that were made. After reading the code twice and coming to two different conclusions of how the algorithm is designed to function, I decided to implement the XrdCmsCluster::SelByLoad algorithm in Python with a lightweight testing wrapper around it, to enable us to understand the algorithms behaviour, and identify weaknesses (if any exist).

Here is the current SelByLoad algorithm implemented in python. See the end of this document for the full test framework implementation.

[19]:
```
class Cluster_original(Cluster):
    def SelByLoad(self, NeedSpace):
        # sp is the selected server, which will be ultimately returned by the␣
↪algorithm
        sp = None

        # loop over all servers in the list, the iterator is 'np' (next␣
↪'p'-something)
        for np in self.servers:
            # skip overloaded nodes
            # real algo has other if ... continue blocks here for other basic␣
↪checks (offline, bad etc.)
            if np.load >= self.maxload:
                continue

            # is we haven't got one selected, select the first node that didn't␣
↪fail the above checks
```

```
            if (sp == None):
                sp = np
            else:
                # reads and writes are handled differently due to the writes␣
↪needing to take space into account, but the logic is very similar in both␣
↪cases
                if NeedSpace:
                    # uses 'mass' instead of load in the xrootd code
                    # as mass ~= load * diskutil and diskutil is the same in␣
↪all cases
                    # I think we can just use load
                    if (abs(sp.load - np.load) <= self.fuzz):
                        if (sp.RefW > (np.RefW + disklinger)):
                            sp = np
                    elif (sp.load > np.load):
                        sp = np
                else:
                    # we're doing a read
                    # if the absolute difference between the selected and next␣
↪node load is within a fuzz,
                    if (abs(sp.load - np.load) <= self.fuzz):
                        # pick the one that has had less reads in the current␣
↪RefReset interval
                        if (sp.RefR > np.RefR):
                            sp = np
                    # else if the selected load is greater than the next node␣
↪load
                    elif (sp.load > np.load):
                        sp = np
        # return the selected node
        return sp
```

2 SelByLoad analysis

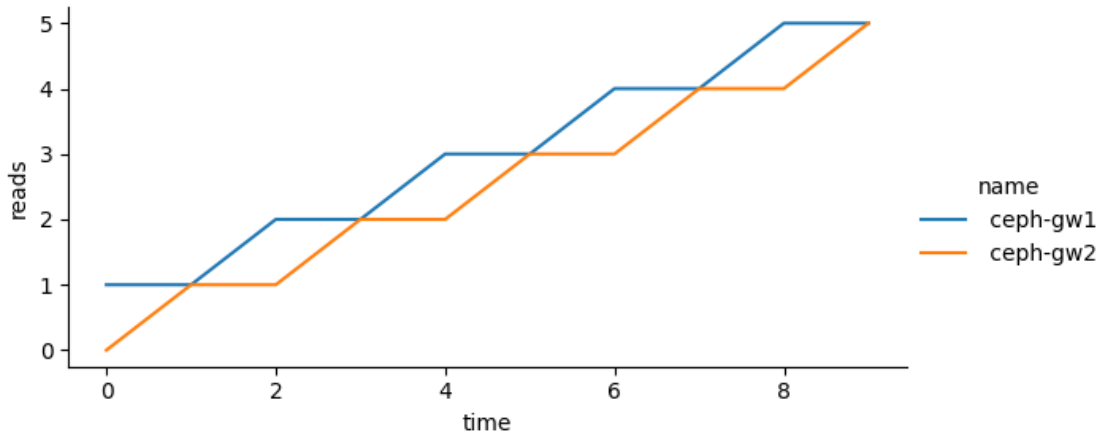
2.1 Basic load balancing validation

To validate the load balancing works as expected and introduce the tooling, lets start with two gateways with equal load score, and one read per second for ten seconds.

[85]:
```
ral1 = Cluster_original(config = "simple", fuzz = 15, maxload = 80, disklinger␣
↪= 0)
ral1.addServer("ceph-gw1", load = 10, RefR = 0, RefW = 0)
ral1.addServer("ceph-gw2", load = 10, RefR = 0, RefW = 0)

test1 = TestRun(cluster = ral1)
```

```
results = test1.run(duration = 10, refReset = 10, readspersec = 1, writespersec␣
 ↪= 0, progress = True)
df = pd.read_csv(StringIO(results), header = None, names = test1.headers())
display(df[['time', 'name','reads']].head());
sns.relplot(data=df, x="time", y="reads", kind='line', hue='name', aspect=2,␣
 ↪height=3);
```

	time	name	reads
0	0	ceph-gw1	1
1	0	ceph-gw2	0
2	1	ceph-gw1	1
3	1	ceph-gw2	1
4	2	ceph-gw1	2



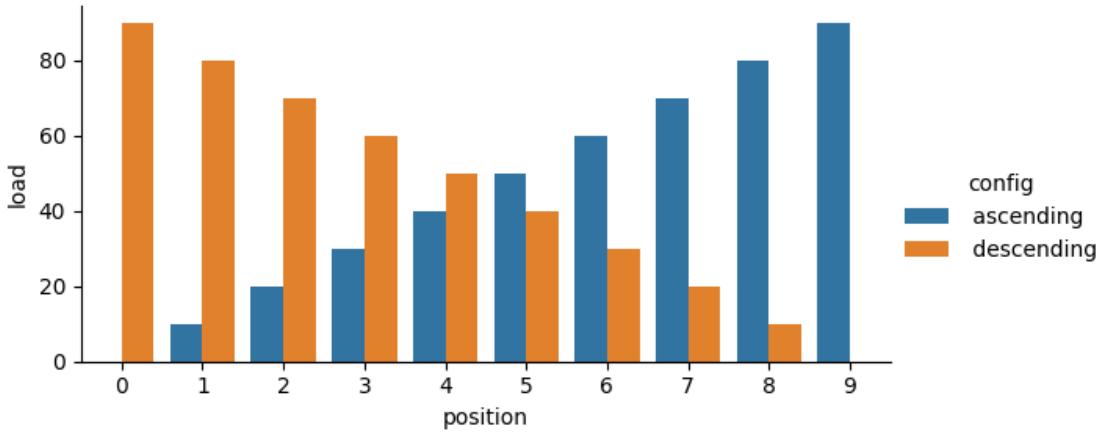
Plotting reads per server over time, we can see the expected round robin placement behaviour of two servers with equal load scores. The first read hits the first server in the list, which increments the *RefR* value for that node, which means the next read hits the second server, and so on.

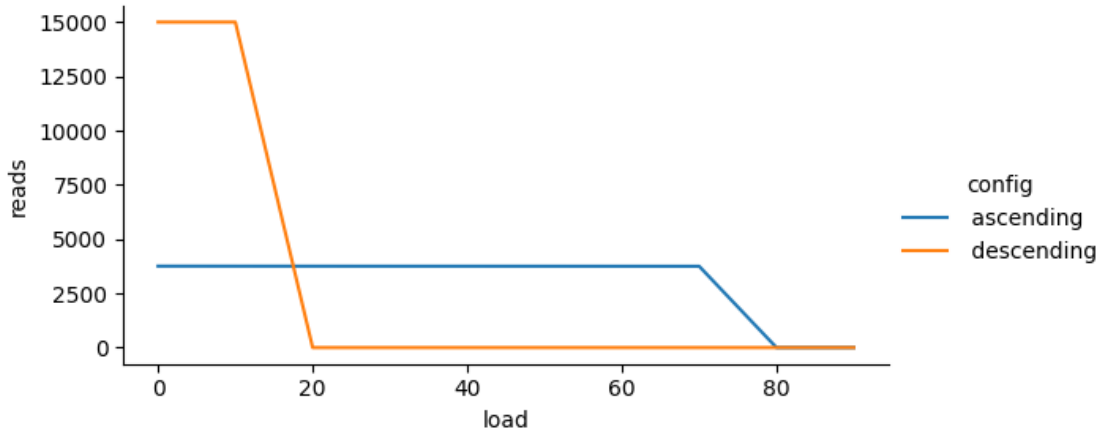
2.2 More servers and varying loads

Lets start looking at greater numbers of servers with varying loads across the set. We start with comparing two CMSD clusters, one with ascending load order and the other with descending load order. Both clusters contain 10 servers with loads evenly spaced between 0 and 90 (e.g. *0,10,20,…,80,90*). The test performs 50 reads a second for 10 minutes (with *RefReset* at 10 minutes, and *fuzz = 15*).

I'm switching to just examining the final state of each cluster for simplicity, and we are interested in the number of reads the servers with a particular load received in each configuration. If the load ordering of the servers in the server list does not affect the algorithms decision making, we would expect both configuration to produce the same result.

[86]:
```
loads = [0,10,20,30,40,50,60,70,80,90]

ral1 = Cluster_original(config = "ascending", fuzz = 15, maxload = 80,␣
 ↪disklinger = 0)
for i, load in enumerate(loads):
    ral1.addServer(f"gw{i}", load, 0, 0)


ral2 = Cluster_original(config = "descending", fuzz = 15, maxload = 80,␣
 ↪disklinger = 0)
for i, load in enumerate(loads[::-1]):
    ral2.addServer(f"gw{i}", load, 0, 0)

test1 = TestRun(cluster = ral1)
test2 = TestRun(cluster = ral2)

results = test1.run(duration = 600, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)
results += "\n" + test2.run(duration = 600, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)

df = pd.read_csv(StringIO(results), header = None, names = test1.headers())

sns.catplot(data=df, kind='bar', x="position", y="load", hue='config',␣
 ↪aspect=2, height=3);
sns.relplot(data=df, x="load", y="reads", kind='line', hue='config', aspect=2,␣
 ↪height=3);
```





The first plot is just the visual representation of the load score of each server in relation to its position in the server list, i.e. the first node in the ascending configurations node list has a score of 0 (no load), and the first node in the descending configurations node list has a score of 90 (heavily loaded). To summarise what we are testing here, both sets of hosts have the same distribution of loaded servers (e.g. one node with load 0, one with load 10 etc.), the only difference being how the servers are arranged in the server list, and therefore which order they are iterated through by the *SelByLoad* algorithm.

The second plot shows the number of reads that were sent to each load score for the two configurations. For example the node with a load score of 40 received ~4000 reads when the server loads were arranged in an ascending configuration, and no reads with the descending configuration. This demonstrates wildly different placement decisions depending on the server load ordering in the server list. This is problematic as nodes are not reordered by load in the server list (This was confirmed with the help of Jyothish using GDB on a live CMSD instance). Of particular interest is the 'ascending' load order resulting in an essentially perfect round robin balancing of requests between all servers with $load < maxload$. It's worth noting that the 'descending' configuration has an expected placement distribution - a round robin placement on nodes within a fuzz value of the best scoring node (including the best scoring node).

2.3 Demonstrating walk up behaviour

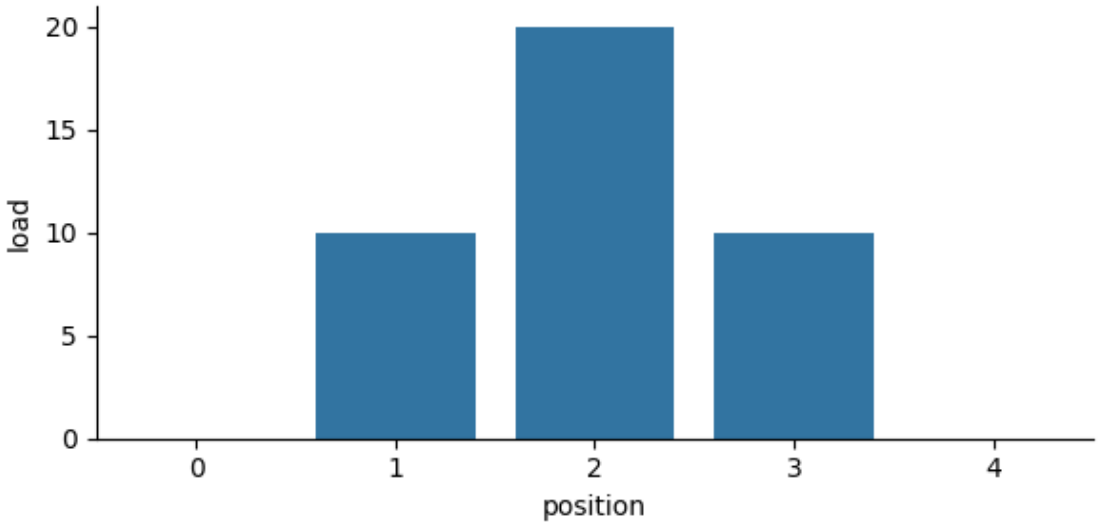
The ascending order behaviour occurs because the algorithm allows higher load servers to be picked over a currently selected lower loaded one provided: 1. they are within a fuzz of each other 2. the higher loaded one has been picked less often in the current *RefReset* interval (i.e. has a lower *RefR/RefW* value) This allows the algorithm to quickly walk up from low load scored servers at the beginning of the server list to servers with significantly higher load scores, assuming there are many decisions made per *RefReset* interval.

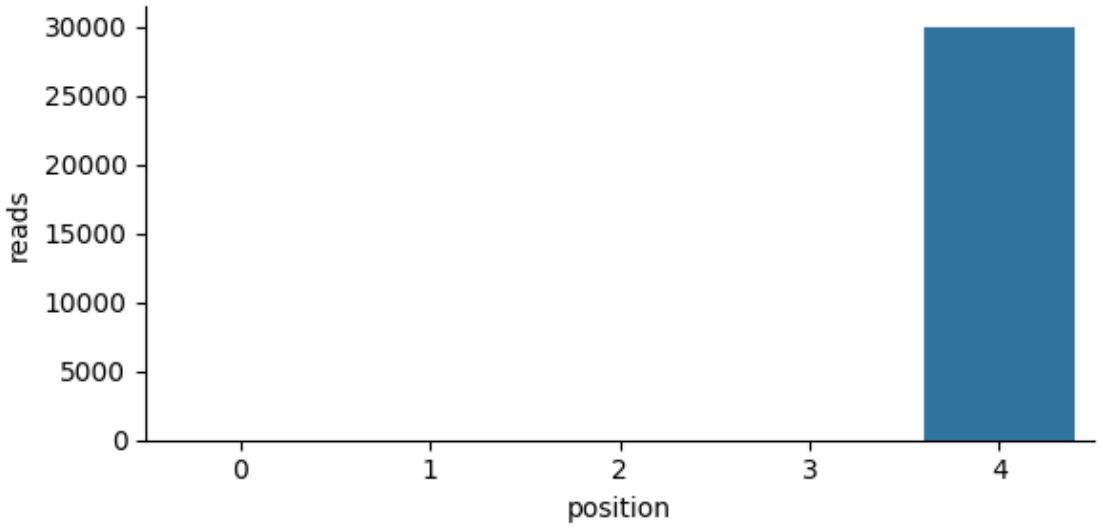
This means a server with a low load at the start of the server list is generally less favoured then one at the end. It is trivial to demonstrate the favouritism with a peak style load distribution:

[87]:
```
loads = [0,10,20,10,0]
```

```
ral1 = Cluster_original(config = "peak", fuzz = 15, maxload = 80, disklinger =␣
 ↪0)
for i, load in enumerate(loads):
    ral1.addServer(f"gw{i+1}", load, 0, 0)

test1 = TestRun(cluster = ral1)

results = test1.run(duration = 600, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)

df = pd.read_csv(StringIO(results), header = None, names = test1.headers())

sns.catplot(data=df, kind='bar', x="position", y="load", aspect=2, height=3);
sns.catplot(data=df, kind='bar', x="position", y="reads", aspect=2, height=3);
display(df[['name','load','reads']]);
```

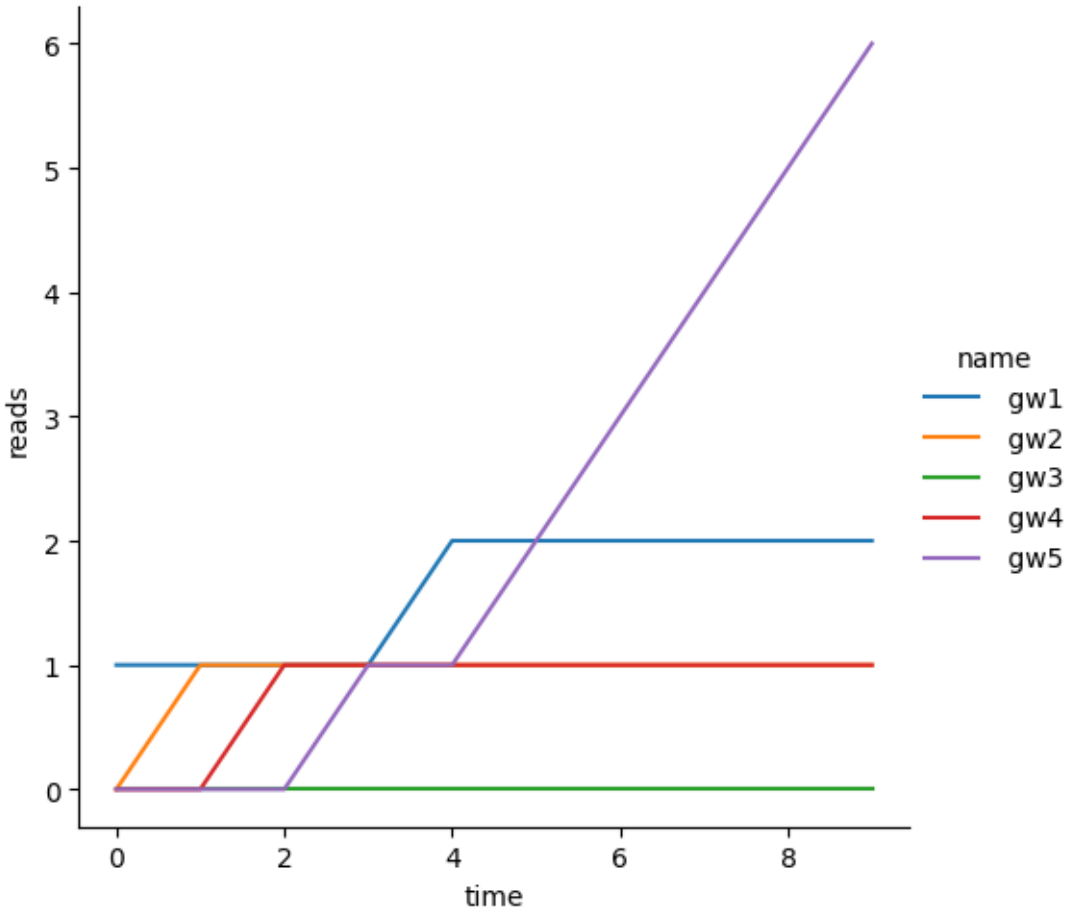
	name	load	reads
0	gw1	0	2
1	gw2	10	1
2	gw3	20	0
3	gw4	10	1
4	gw5	0	29996





The zero load node at the end of the list is favoured dramatically, getting all but four of the 30,000 reads of the test. Changing the test parameters to enable us to observe each read decision individually makes it more clear why this happens:

[88]:
```
loads = [0,10,20,10,0]

ral1 = Cluster_original(config = "peak", fuzz = 15, maxload = 80, disklinger =␣
 ↪0)
for i, load in enumerate(loads):
    ral1.addServer(f"gw{i+1}", load, 0, 0)

test1 = TestRun(cluster = ral1)

results = test1.run(duration = 10, refReset = 600, readspersec = 1,␣
 ↪writespersec = 0, progress = True)

df = pd.read_csv(StringIO(results), header = None, names = test1.headers())

sns.relplot(data=df, x="time", y="reads", kind='line', hue='name');
```



We can see the first 5 reads are somewhat evenly distributed between the gateways, but from read 6 onward, all further reads are sent to gw5. I found it useful to step through this process and understand the logic for the first 5 reads:

1) **gw1** gets the 1st read as there is no node that has a lower *RefR* and it was first to be selected (being at the top of the list).
2) **gw2** gets the 2nd read as it has a lower *RefR* than gw1 and is within the fuzz of it
3) **gw4** gets the 3rd read as it is the first node that is within the fuzz of gw1 whilst having a lower *RefR* (gw2 is excluded for *RefR*, gw3 is excluded for fuzz)
4) **gw5** gets the 4th read as it is the first node that is within the fuzz of gw1 whilst having a lower *RefR* (gw2 and gw4 are excluded for *RefR*, gw3 is excluded for fuzz)
5) **gw1** gets the 5th read as there are no nodes with lower *RefR* that are within a fuzz of it

At this point the selection algorithm has constructed a staircase for the algorithm to climb, and will always follow the same loop:

1) select **gw1**, as it is first in the list
2) select **gw2**, as it is the first node to have a lower *RefR* ($1 < 2$) than gw1 whilst being within a fuzz of gw1
3) select **gw3**, as it is the first node to have a lower *RefR* ($0 < 1$) than gw2 whilst being within

a fuzz of gw2

4) pass over **gw4** as it has a higher *RefR* than gw3 whilst being within a fuzz value of it
5) select **gw5**, as it has a score that is better than gw3 by over the fuzz value

As all reads will continue to hit gw5, there will be no changes to the *RefR* values of the other nodes and therefore no change to the behaviour. The algorithm will always end up with gw3 as the selected node and therefore will always end up selecting gw5.

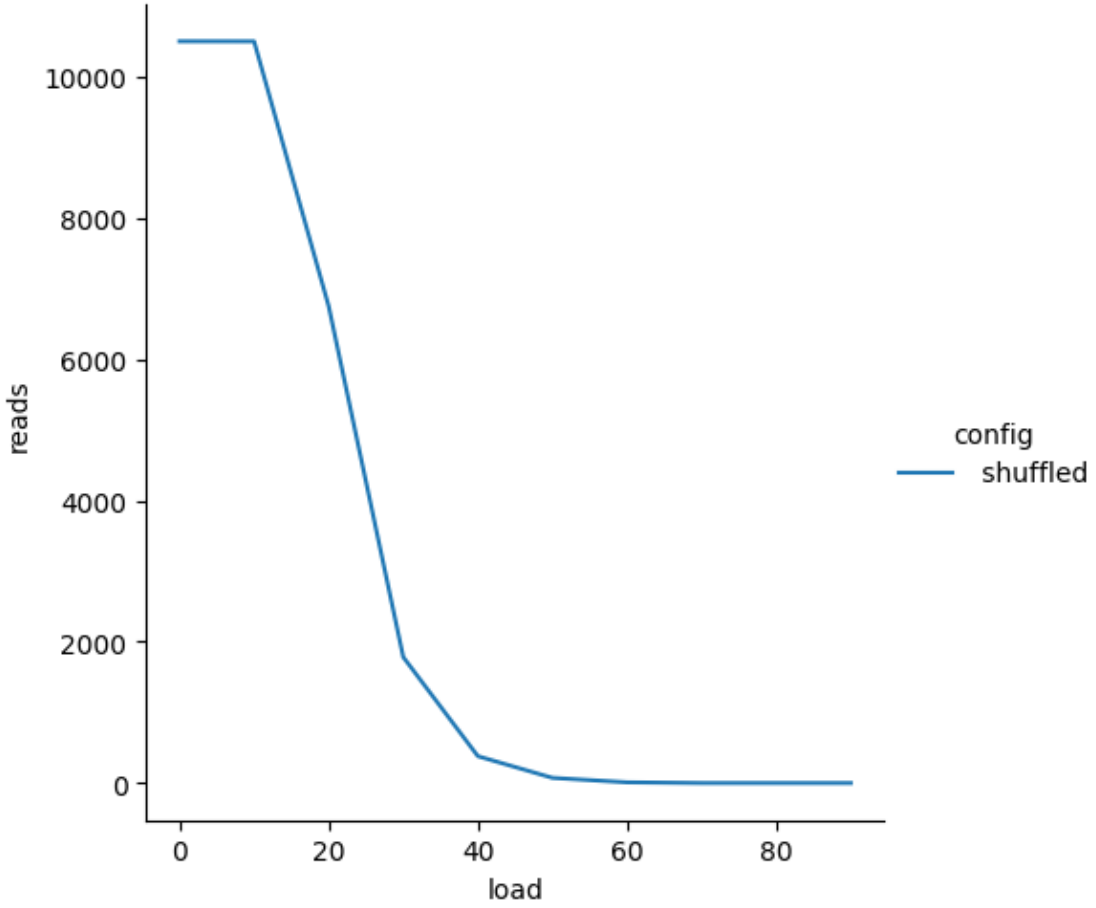
This is problematic from an operational perspective as that the server order is 'hidden', so it's very hard to understand why one node with a particular load gets x transfers in a time period, while another with a similar load gets y. This in turn makes it hard to make good observations about the tuning of the loadbalancing.

2.4 Shuffling the server list

One could argue that because the loads of servers will vary over time, this isn't important. Indeed, shuffling the server list with every decision will yield a reasonable looking average distribution of reads over loads. I've implemented a copy of the algorithm that shuffles the list for every decision:

[89]:
```
class Cluster_shuffle(Cluster):
    def SelByLoad(self, NeedSpace):
        sp = None
        for np in random.sample(self.servers, len(self.servers)):
            if np.load >= self.maxload:
                continue
            if (sp == None):
                sp = np # if we don't have a 'selected' node, pick the current␣
 ↪'next'.
            else:
                if NeedSpace:
                    # uses 'mass' instead of load in the xrootd code
                    # as mass ~= load * diskutil and diskutil is the same in␣
 ↪all cases
                    # I think we can just use load
                    if (abs(sp.load - np.load) <= self.fuzz):
                        if (sp.RefW > (np.RefW + disklinger)):
                            sp = np
                    elif (sp.load > np.load):
                        sp = np
                else:
                    if (abs(sp.load - np.load) <= self.fuzz):
                        if (sp.RefR > np.RefR):
                            sp = np
                    elif (sp.load > np.load):
                        sp = np
        # return the selected node
        return sp
```

```
loads = [0,10,20,30,40,50,60,70,80,90]

ral1 = Cluster_shuffle(config = "shuffled", fuzz = 15, maxload = 80, disklinger␣
 ↪= 0)
for i, load in enumerate(loads):
    ral1.addServer(f"ceph-gw{i}", load, 0, 0)

test1 = TestRun(cluster = ral1)
results = test1.run(duration = 600, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)
df = pd.read_csv(StringIO(results), header = None, names = test1.headers())
sns.relplot(data=df, x="load", y="reads", kind='line', hue='config');
```



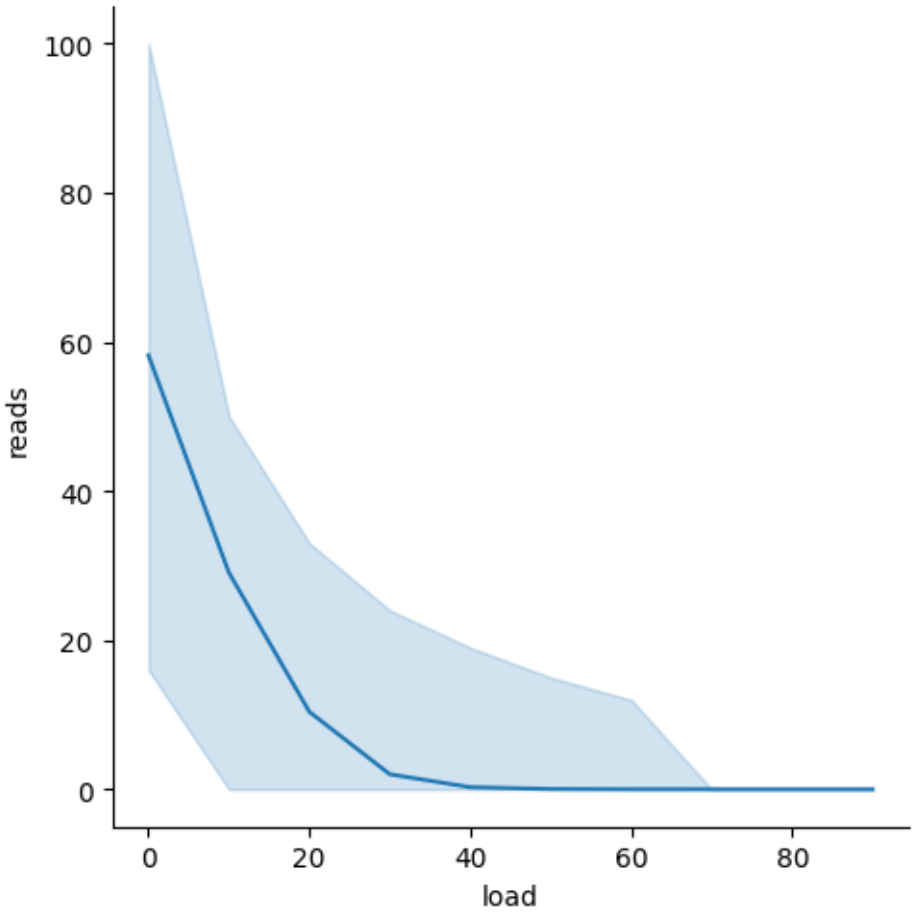
The average decision looks good, however this 'shuffling' isn't quite what happens in reality, as nodes are fixed in place in the server list whilst their loads update. The problem (as I see it) is that nodes with the same load score will be treated differently depending on their position in the list.

Additionally, although this evens out to a sensible placement profile over the long term, the "micro-

scale" decision making is just as important. If there are situations where a heavily loaded node can receive lots of transfers, or a lightly loaded node will not receive transfers then these will result in short term over or under loading. The former results in failures while the later reduces total capacity.

This can be observed by looking at the average, and complete population interval of 10,000 different random load orders:

[90]:
```
loads = [0,10,20,30,40,50,60,70,80,90]
results = ""

for rand in range(1,10000):
    cluster = Cluster_original(config = f"random{rand}", fuzz = 15, maxload =␣
 ↪80, disklinger = 0)
    for i, load in enumerate(random.sample(loads, len(loads))):
        cluster.addServer(f"{i}", load, 0, 0)
    test = TestRun(cluster = cluster)
    results += "\n" + test.run(duration = 1, refReset = 600, readspersec = 100,␣
 ↪writespersec = 0, progress = False)

df = pd.read_csv(StringIO(results), header = None, names = test.headers())

sns.relplot(data=df, x="load", y="reads", kind='line', errorbar=('pi', 100));
```



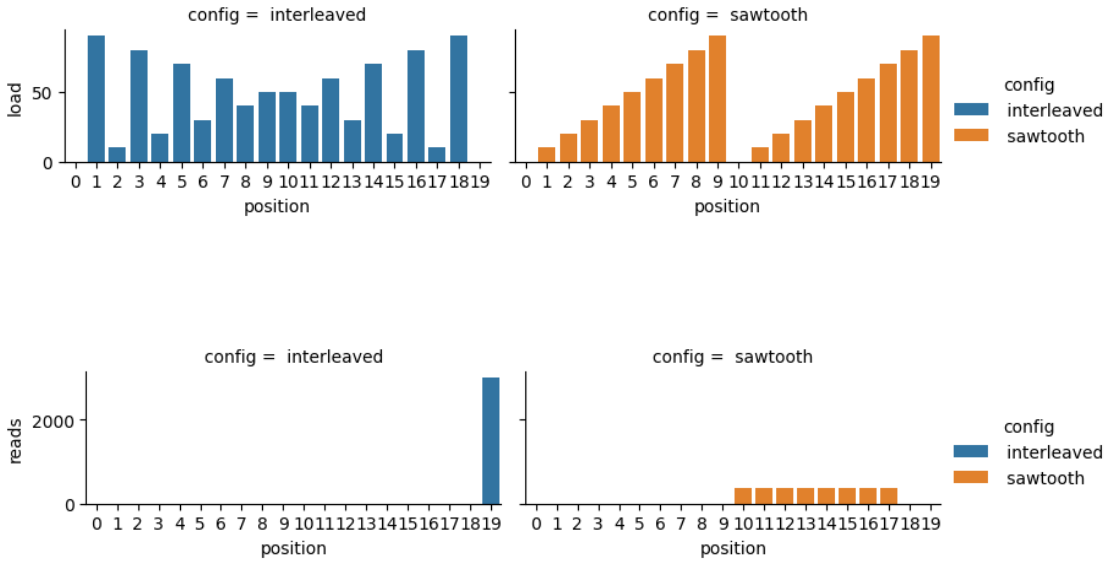
Although the average trend is for well distributed load, an individual run can have dramatically different placement decisions which makes it difficult to understand transient 'hotspotting' (or 'coldspotting').

2.5 Further edge cases

I've reconstructed some of the load distribution scenarios that have particularly odd placement decisions:

[91]:
```
ral1 = Cluster_original(config = "interleaved", fuzz = 15, maxload = 80,␣
 ↪disklinger = 0)
for i, load in␣
 ↪enumerate([0,90,10,80,20,70,30,60,40,50,50,40,60,30,70,20,80,10,90,0]):
    ral1.addServer(f"{i}", load, 0, 0)

ral2 = Cluster_original(config = "sawtooth", fuzz = 15, maxload = 80,␣
 ↪disklinger = 0)
for i, load in␣
 ↪enumerate([0,10,20,30,40,50,60,70,80,90,0,10,20,30,40,50,60,70,80,90]):
```

```
    ral2.addServer(f"{i}", load, 0, 0)

test1 = TestRun(cluster = ral1)
test2 = TestRun(cluster = ral2)

results = test1.run(duration = 60, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)
results += "\n" + test2.run(duration = 60, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)

df = pd.read_csv(StringIO(results), header = None, names = test1.headers())

sns.catplot(df, kind="bar", x="position", y="load", col="config", hue='config',␣
 ↪height=2, aspect=2);
sns.catplot(df, kind="bar", x="position", y="reads", col="config",␣
 ↪hue='config', height=2, aspect=2);
```



Here we can see the 'interleaved' configuration results in the two servers with zero load getting either all the transfers or none. the 'sawtooth' load distribution shows perfect round-robin of half the population, and no placement on the other half.

This really highlights some of the difficulties of understanding *SelByload* decision making. Although these scenarios are artificial and unlikely to occur often, there may be occasions where instances similar to these failure modes occur, and it is impossible (or at least very hard) to diagnose why, as the server order is not visible to the operator.

To summarise, although the *SelByLoad* algorithm tends to a good distribution of requests over time, it has clear weaknesses when examined with some load score distributions.

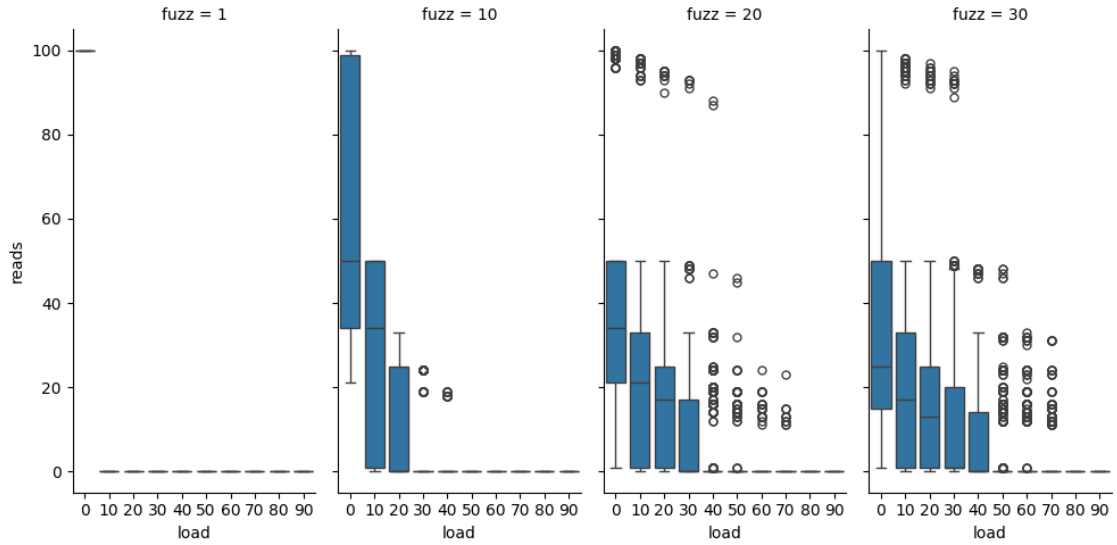
3 Tuning SelByLoad behaviour with refReset and fuzz

It may be possible to tune out the problematic behaviours with the available loadbalancing tuning variables, namely *fuzz* and *refReset*.

3.1 Fuzz

We start by looking at the many iterations of the previous randomised load orders with varying fuzz. In this example the loads are always `0,10,20,..,90`, but the order is shuffled, same as the above shuffled examples.

[92]:
```
loads = [0,10,20,30,40,50,60,70,80,90]
results = ""

for fuzz in [1, 10, 20, 30]:
    for rand in range(1,1000):
        cluster = Cluster_original(config = f"fuzz {fuzz}", fuzz = fuzz,␣
 ↪maxload = 80, disklinger = 0)
        for i, load in enumerate(random.sample(loads, len(loads))):
            cluster.addServer(f"{i}", load, 0, 0)
        test = TestRun(cluster = cluster)
        results += "\n" + test.run(duration = 1, refReset = 600, readspersec =␣
 ↪100, writespersec = 0, progress = False)


df = pd.read_csv(StringIO(results), header = None, names = test.headers())

sns.catplot(data=df, x="load", y="reads", kind='box', col="fuzz", aspect=0.5);
```



Predictably, when the fuzz is less than the server load spacing (10), the only server ever to be

chosen is the best scoring server. As the fuzz increases, the chance of the algorithm 'walking up' to higher loaded servers increases.

Reducing the fuzz may be a viable strategy for reducing this behaviour, but has two drawbacks:

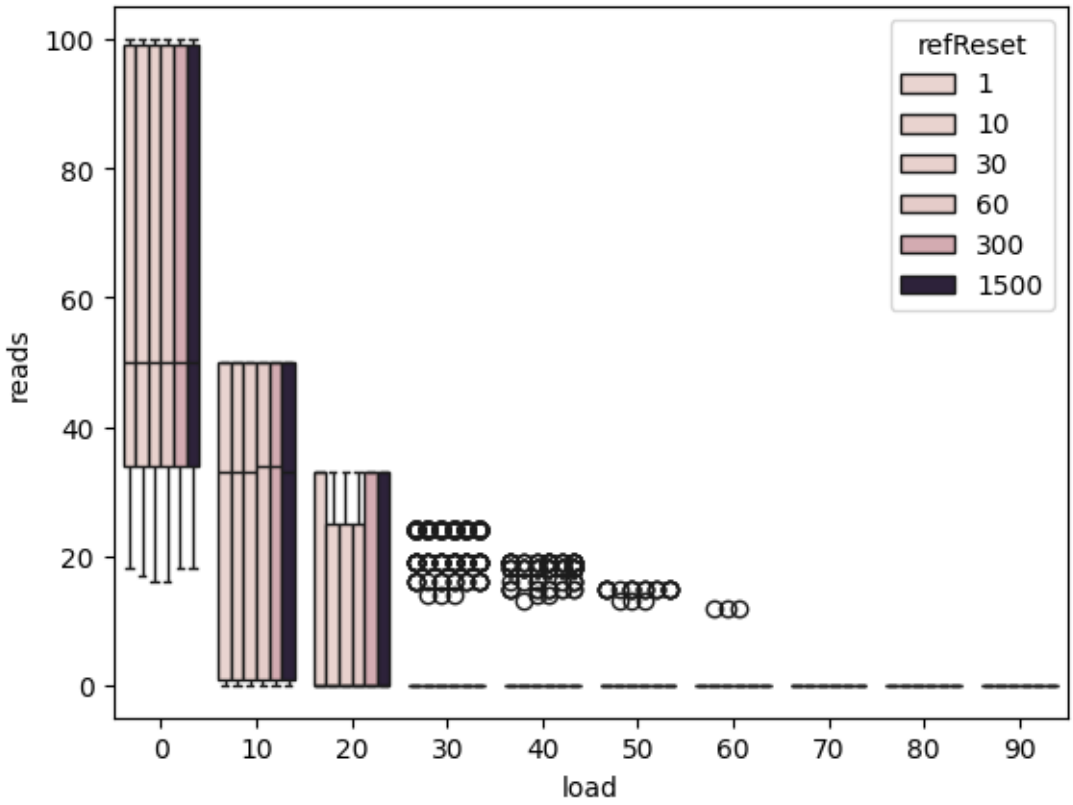
1) The fuzz controls the round robin load balancing mechanism, so a sufficiently small fuzz will reduce the population that gets transfers (in some cases). This increases the chance of oscillatory behaviour where the best scores get most transfers and then become overloaded.
2) The greater the number of servers in the set, the smaller fuzz will have to be to prevent the 'walk up' behaviour as the greater the chance of the servers falling into a 'load staircase' type configuration.

It is probably worth exploring smaller fuzz values from the current value used (15). The main thing this may do is give a higher chance of preventing the 'walk up' by allowing smaller gaps to act as firebreaks to separate the population.

3.2 RefReset

The *RefReset* duration controls how often (in seconds) the round robin counters are reset. In theory, resetting this more often should bias the placement decision towards the lower scores rather than trying to round robin over all available servers.

[93]:
```
loads = [0,10,20,30,40,50,60,70,80,90]
results = ""

for refReset in [1, 10, 30, 60, 300, 1500]:
    for rand in range(1,1000):
        cluster = Cluster_original(config = f"refReset {refReset}", fuzz = 15,␣
 ↪maxload = 80, disklinger = 0)
        for i, load in enumerate(random.sample(loads, len(loads))):
            cluster.addServer(f"{i}", load, 0, 0)
        test = TestRun(cluster = cluster)
        results += "\n" + test.run(duration = 1, refReset = refReset,␣
 ↪readspersec = 100, writespersec = 0, progress = False)

df = pd.read_csv(StringIO(results), header = None, names = test.headers())

sns.boxplot(data=df, x="load", y="reads", hue="refReset");
```



Changing *refReset* for workloads where there are many reads per second doesn't seem to have a meaningful impact. I believe this is because in the problematic orderings, the counters quickly fill up and the decision making ends up in the odd state we have seen (e.g. the peak style ordering demonstrated in the "More servers and varying loads" section).

4 Potential solutions

4.1 Sorted server list

As noted at the start, a descending load order sorted server list will always pick the lowest load server, or one within a fuzz. This is a predictable behaviour and would be an obvious way to 'fix' the algorithm.

The difficulty is that the server loads updates are not obviously triggered from within the Cluster class, and so I think this makes it hard to do a sensible "update all loads and sort" operation. The other option would be creating a sorted list on every decision (e.g. in *SelByLoad*), but this feels like it could add a large overhead, and it would be better to change the way the algorithm works instead.

4.2 Simple algorithmic changes

4.2.1 Best seen load comparison

Currently the *SelByLoad* algorithm compares the score of the potential server (*np*) against the currently selected server (*sp*). This enables it to walk up from a low score to a high score, and results in the low loaded servers at the end of the server list being favoured. A potential fix of this is keeping track of the best score seen and comparing against this when considering whether to select the node within a fuzz due to a better round robin score. This should will prevent the 'walk up' behaviour seen.

I haven't implemented the write decision path here for readability, and I have left in some instrumentation print statements to enable the flow to be followed.

[3]:
```
class Cluster_best_seen_load(Cluster):
    def SelByLoad(self, NeedSpace):
        #print("===== SelByLoad Start")
        sp = None
        best_load = 100
        for np in self.servers:
            #print(f"=== np = {np.name}, load = {np.load} best_load =␣
↪{best_load}")
            if np.load >= self.maxload:
                continue
            if np.load < best_load:
                #print(f"updating best load from {best_load} to {np.load}")
                best_load = np.load
            if (sp == None):
                sp = np
                #print("selecting this node as initial")
            else:
                if (sp.load - np.load) > self.fuzz:
                    #print("np is over a fuzz better than sp, pick np")
                    sp = np
                elif abs(best_load - np.load) <= self.fuzz:
                    #print("np within fuzz of best score, potential for RR␣
↪selection of np")
                    if (sp.RefR > np.RefR):
                        sp = np
                        #print ("sp has higher RefR than np, picking np")
                    #else:
                        #print ("sp has lower RefR than np, not picking np")
                #else:
                    #print("np has no redeeming qualities")
            #print(f"final state of run for np {np.name} ({np.load}): sp = {sp.
↪name} ({sp.load})")
        # return the selected node
        return sp
```

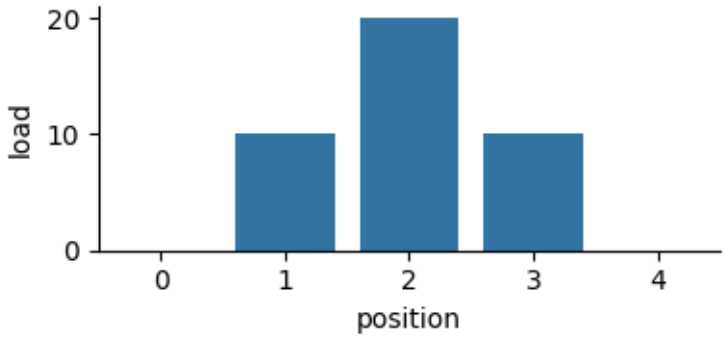
This follows the same flow as the original, with a single iteration over the unsorted list of servers, with subtly different tests:

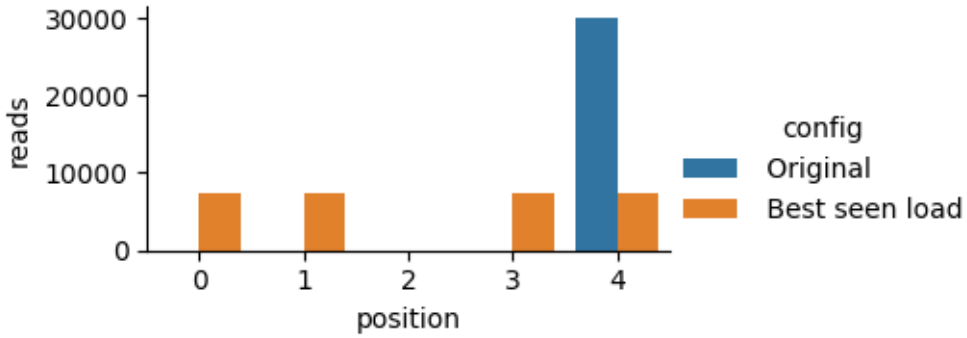
1. if the load of the next server is over a fuzz better than the currently selected, immediately select it
2. else, if the next server is within a fuzz of the best load score seen, select if it has a lower round robin score than the current selected

This algorithm will quickly pick a good scoring node, and then will only consider for round robin placement nodes that are within the fuzz of the best score. This still allows for a round robin placement over a population of nodes (determined by the fuzz), but without the 'walk up' behaviour seen with the original.

Looking at the pathological peak style distribution with this modification:

[95]:
```
loads = [0,10,20,10,0]

ral1 = Cluster_original(config = "Original", fuzz = 15, maxload = 80,␣
 ↪disklinger = 0)
for i, load in enumerate(loads):
    ral1.addServer(f"gw{i+1}", load, 0, 0)

ral2 = Cluster_best_seen_load(config = "Best seen load", fuzz = 15, maxload =␣
 ↪80, disklinger = 0)
for i, load in enumerate(loads):
    ral2.addServer(f"gw{i+1}", load, 0, 0)

test1 = TestRun(cluster = ral1)
test2 = TestRun(cluster = ral2)

results = test1.run(duration = 600, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)
results += "\n" + test2.run(duration = 600, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)

df = pd.read_csv(StringIO(results), header = None, names = test1.headers())

sns.catplot(data=df, kind='bar', x="position", y="load", aspect=2, height=2);
sns.catplot(data=df, kind='bar', x="position", y="reads", hue='config',␣
 ↪aspect=2, height=2);
#display(df[['name','load','reads']]);
```



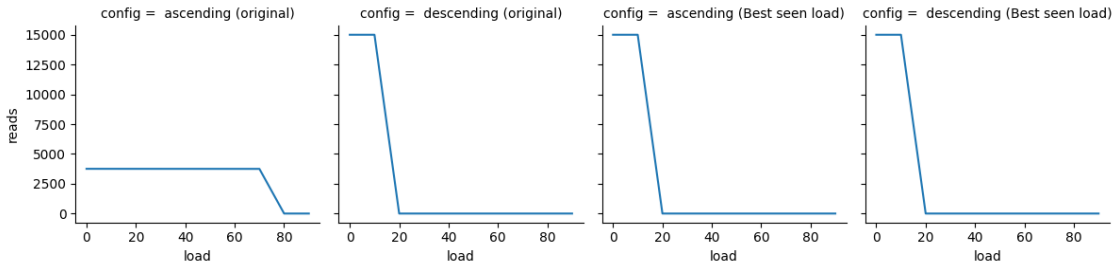


This is the 'expected' behaviour, with a round robin placement of requests across all servers within the fuzz of the best score. This occurs as the best load score is immediately set to 0 (the score of *gw1*), and so *gw3* can never be selected as it will fail the "within the fuzz of the best seen score" check, which in turn prevents the unconditional selection of the server that has a significantly better score (*gw5*).

Looking at the ascending and descending ordering comparisons, we can see the ideal behaviour in both cases with the new algorithm.

[96]:
```
loads = [0,10,20,30,40,50,60,70,80,90]


ral1 = Cluster_original(config = "ascending (original)", fuzz = 15, maxload =␣
 ↪80, disklinger = 0)
for i, load in enumerate(loads):
    ral1.addServer(f"gw{i}", load, 0, 0)
test1 = TestRun(cluster = ral1)

ral2 = Cluster_original(config = "descending (original)", fuzz = 15, maxload =␣
 ↪80, disklinger = 0)
for i, load in enumerate(loads[::-1]):
```

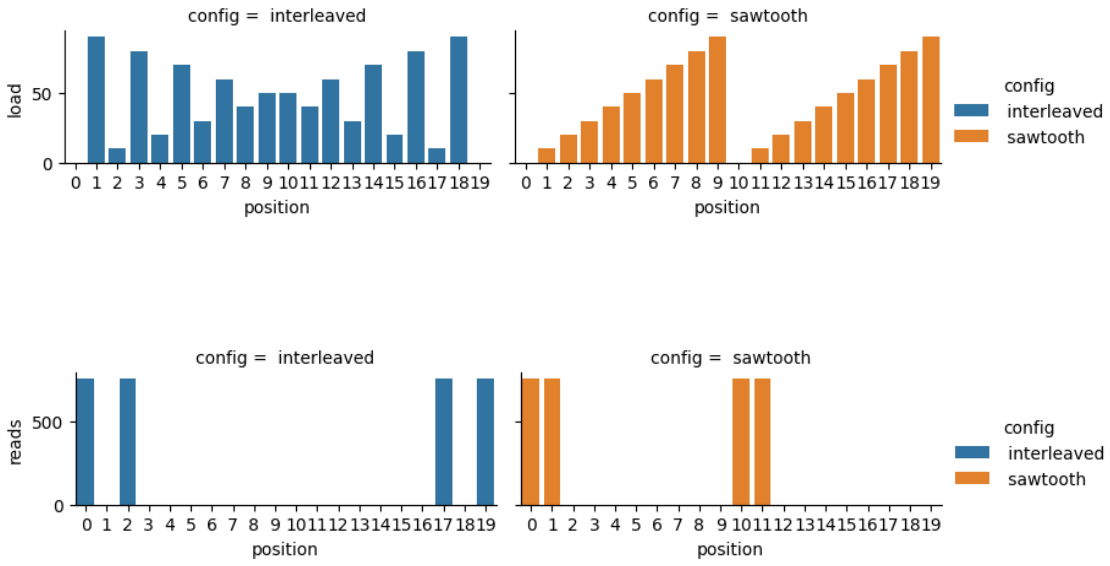
```
    ral2.addServer(f"gw{i}", load, 0, 0)
test2 = TestRun(cluster = ral2)

ral3 = Cluster_best_seen_load(config = "ascending (Best seen load)", fuzz = 15,␣
 ↪maxload = 80, disklinger = 0)
for i, load in enumerate(loads):
    ral3.addServer(f"gw{i}", load, 0, 0)
test3 = TestRun(cluster = ral3)

ral4 = Cluster_best_seen_load(config = "descending (Best seen load)", fuzz =␣
 ↪15, maxload = 80, disklinger = 0)
for i, load in enumerate(loads[::-1]):
    ral4.addServer(f"gw{i}", load, 0, 0)
test4 = TestRun(cluster = ral4)


results = test1.run(duration = 600, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)
results += "\n" + test2.run(duration = 600, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)
results += "\n" + test3.run(duration = 600, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)
results += "\n" + test4.run(duration = 600, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)


df = pd.read_csv(StringIO(results), header = None, names = test1.headers())

sns.relplot(data=df, x="load", y="reads", kind='line', col="config", height=3,␣
 ↪aspect=1);
```



And in the 'interesting edge cases', we can see perfect load balancing onto servers with the correct scores, with no more preferential treatment of nodes later in the list.

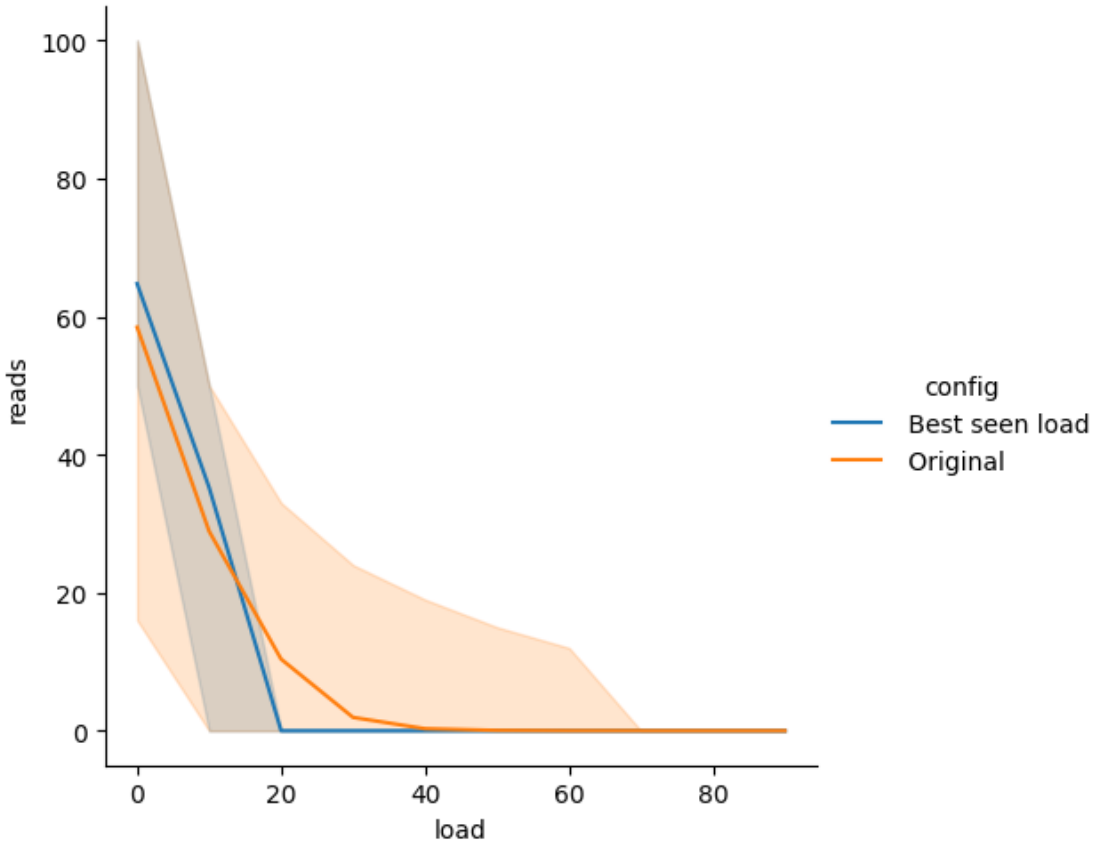
[97]:
```
ral1 = Cluster_best_seen_load(config = "interleaved", fuzz = 15, maxload = 80,␣
 ↪disklinger = 0)
```

```
for i, load in␣
 ↪enumerate([0,90,10,80,20,70,30,60,40,50,50,40,60,30,70,20,80,10,90,0]):
    ral1.addServer(f"{i}", load, 0, 0)

ral2 = Cluster_best_seen_load(config = "sawtooth", fuzz = 15, maxload = 80,␣
 ↪disklinger = 0)
for i, load in␣
 ↪enumerate([0,10,20,30,40,50,60,70,80,90,0,10,20,30,40,50,60,70,80,90]):
    ral2.addServer(f"{i}", load, 0, 0)

test1 = TestRun(cluster = ral1)
test2 = TestRun(cluster = ral2)

results = test1.run(duration = 60, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)
results += "\n" + test2.run(duration = 60, refReset = 600, readspersec = 50,␣
 ↪writespersec = 0, progress = False)

df = pd.read_csv(StringIO(results), header = None, names = test1.headers())

sns.catplot(df, kind="bar", x="position", y="load", col="config", hue='config',␣
 ↪height=2, aspect=2);
sns.catplot(df, kind="bar", x="position", y="reads", col="config",␣
 ↪hue='config', height=2, aspect=2);
```



And finally looking at the many random runs example, we can see significantly more consistent placement decisions, with nodes outside a fuzz of the best score never receiving reads.

[98]:
```
loads = [0,10,20,30,40,50,60,70,80,90]
results = ""

for rand in range(1,10000):
    cluster = Cluster_best_seen_load(config = f"Best seen load", fuzz = 15,␣
 ↪maxload = 80, disklinger = 0)
    for i, load in enumerate(random.sample(loads, len(loads))):
        cluster.addServer(f"{i}", load, 0, 0)
    test = TestRun(cluster = cluster)
    results += "\n" + test.run(duration = 1, refReset = 600, readspersec = 100,␣
 ↪writespersec = 0, progress = False)

for rand in range(1,10000):
    cluster = Cluster_original(config = f"Original", fuzz = 15, maxload = 80,␣
 ↪disklinger = 0)
    for i, load in enumerate(random.sample(loads, len(loads))):
        cluster.addServer(f"{i}", load, 0, 0)
    test = TestRun(cluster = cluster)
    results += "\n" + test.run(duration = 1, refReset = 600, readspersec = 100,␣
 ↪writespersec = 0, progress = False)

df = pd.read_csv(StringIO(results), header = None, names = test.headers())

sns.relplot(data=df, x="load", y="reads", kind='line', hue='config',␣
 ↪errorbar=('pi', 100));
```



However, there is still a bimodal behaviour with the new algorithm in this example, we can see sometimes the best scoring node is getting all 100 of the reads, and sometimes the reads are shared equally between the two best scoring nodes (0 and 10).

[99]:
```
loads = [0,10,20,30,40,50,60,70,80,90]
results = ""

for rand in range(1,1000):
    cluster = Cluster_best_seen_load(config = f"run {rand}", fuzz = 15, maxload␣
 ↪= 80, disklinger = 0)
    for i, load in enumerate(random.sample(loads, len(loads))):
        cluster.addServer(f"{i}", load, 0, 0)
    test = TestRun(cluster = cluster)
    results += "\n" + test.run(duration = 1, refReset = 600, readspersec = 100,␣
 ↪writespersec = 0, progress = False)

df = pd.read_csv(StringIO(results), header = None, names = test.headers())

runs_with_uneven_reads = df[df.reads > 60].config
display(runs_with_uneven_reads.head(1))
```

```
first_uneven_run = runs_with_uneven_reads.head(1)
display(df[df['config'].
 ↪isin(runs_with_uneven_reads)][['config','load','reads']])
```

```
94     run 10
Name: config, dtype: object
```

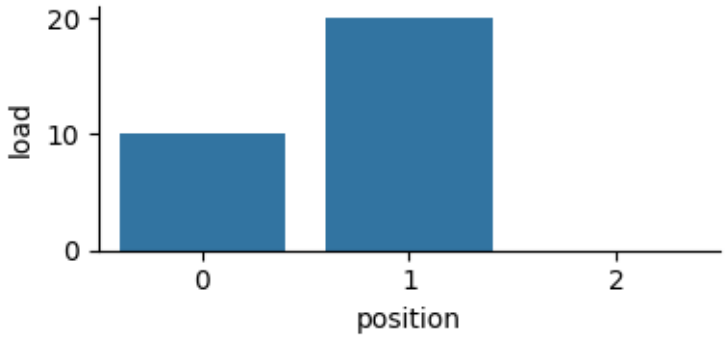
	config	load	reads
90	run 10	90	0
91	run 10	20	0
92	run 10	10	0
93	run 10	30	0
94	run 10	0	100
…	…	…	…
9885	run 989	20	0
9886	run 989	60	0
9887	run 989	0	99
9888	run 989	90	0
9889	run 989	30	0

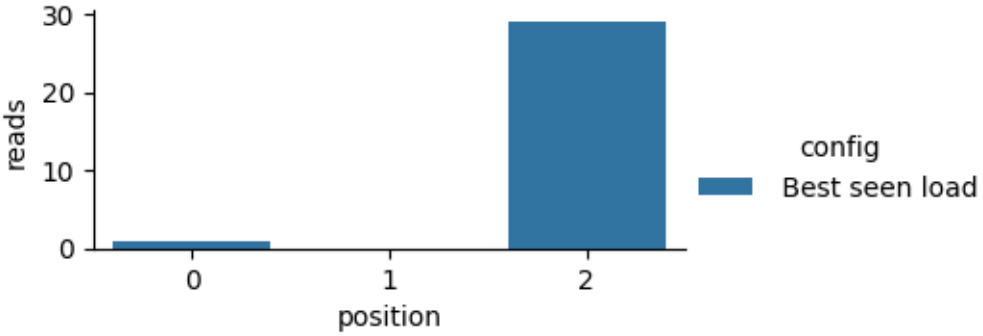
```
[3100 rows x 3 columns]
```

Investigation into these revealed two types of failure mode:

Failure mode 1 - A limited walk up If the best score near the start is within a fuzz of the ultimate best score, but is not the ultimate best score, the algorithm may walk up to a score that is outside the fuzz of the ultimate best score and therefore have to drop down to the best score, therefore we will not place reads on early good scoring nodes.

[100]:
```
#loads = [50,10,70,90,20,40,30,0,80,60]
loads = [10,20,0]

ral1 = Cluster_best_seen_load(config = "Best seen load", fuzz = 15, maxload =␣
 ↪80, disklinger = 0)
for i, load in enumerate(loads):
    ral1.addServer(f"gw{i+1}", load, 0, 0)

test1 = TestRun(cluster = ral1)

results = test1.run(duration = 1, refReset = 600, readspersec = 30,␣
 ↪writespersec = 0, progress = False)

df = pd.read_csv(StringIO(results), header = None, names = test1.headers())

sns.catplot(data=df, kind='bar', x="position", y="load", aspect=2, height=2);
sns.catplot(data=df, kind='bar', x="position", y="reads", hue='config',␣
 ↪aspect=2, height=2);
```



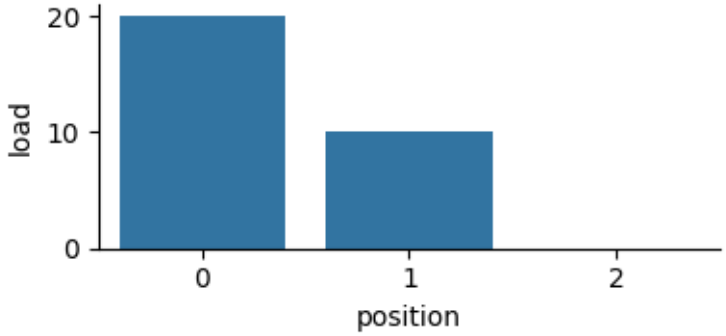


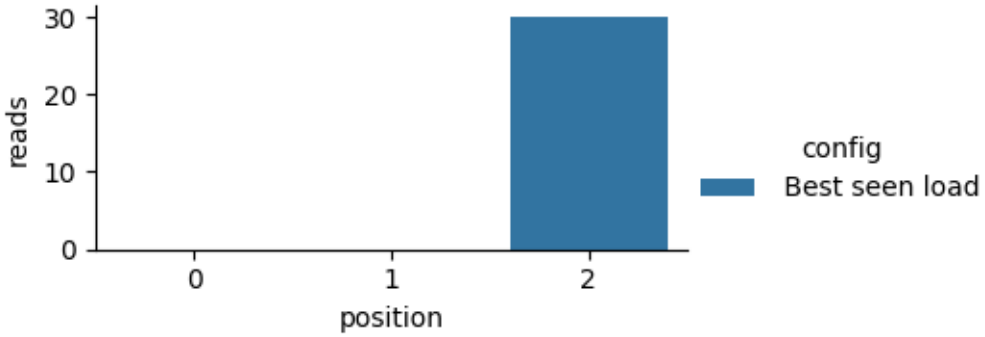
In the above example the gateway at position 1 (load 10) is chosen initially, and gets the first read. After that every read picks 1 again, then 2 (load 20) as the round robin selection suggests, and then finally picks position 3 (load 0) as the difference between the two is outside of a fuzz.

Failure mode 2 - Starting just outside the fuzz If an initially (or early) selected node is close to, but not within a fuzz of the best final score, we will skip over selecting nodes that are within a fuzz of the best final score because we are relying on round robin behaviour to make the switch.

[17]:
```
#loads = [20,10,60,40,90,50,0,30,70,80]
loads = [20,10,0]

ral1 = Cluster_best_seen_load(config = "Best seen load", fuzz = 15, maxload =␣
 ↪80, disklinger = 0)
for i, load in enumerate(loads):
    ral1.addServer(f"gw{i+1}", load, 0, 0)

test1 = TestRun(cluster = ral1)

results = test1.run(duration = 1, refReset = 600, readspersec = 30,␣
 ↪writespersec = 0, progress = False)
```

```
df = pd.read_csv(StringIO(results), header = None, names = test1.headers())

sns.catplot(data=df, kind='bar', x="position", y="load", aspect=2, height=2);
sns.catplot(data=df, kind='bar', x="position", y="reads", hue='config',␣
 ↪aspect=2, height=2);
```





Both of these weaknesses exist in the original algorithm, but were somewhat masked by the other failure modes.

Summary This algorithm change appears to prevent nodes outside the "best score plus a fuzz" range ever receiving reads, which is good for preventing already heavily loaded servers receiving even more transfers. However, there are simple circumstances that prevent nodes within a fuzz of the best score from receiving any reads, which is problematic as it will result in some nodes being underloaded and some nodes being overloaded.

I don't think there is an obvious way to get around these issues with this type of algorithm without knowing of the best viable score before the iteration starts. I think this is tricky to do as keeping track of the current best recorded score in the cluster may not be enough, as the best scoring node may be disqualified for other reasons (e.g. being offline or 'bad').

4.2.2 Two pass approach

We can consider the approach where we iterate over the nodes twice, the first pass to understand which nodes are eligible for the placement (filter out 'bad' nodes) and to find the best valid score (and the best node). The second pass can then focus on picking the eligible nodes within *best_score + fuzz* to ensure round robin distribution. Although we have two iterations over the server list, there are two things that make this less expensive and therefore may not double the time required for the algorithm to run for the same number of nodes:

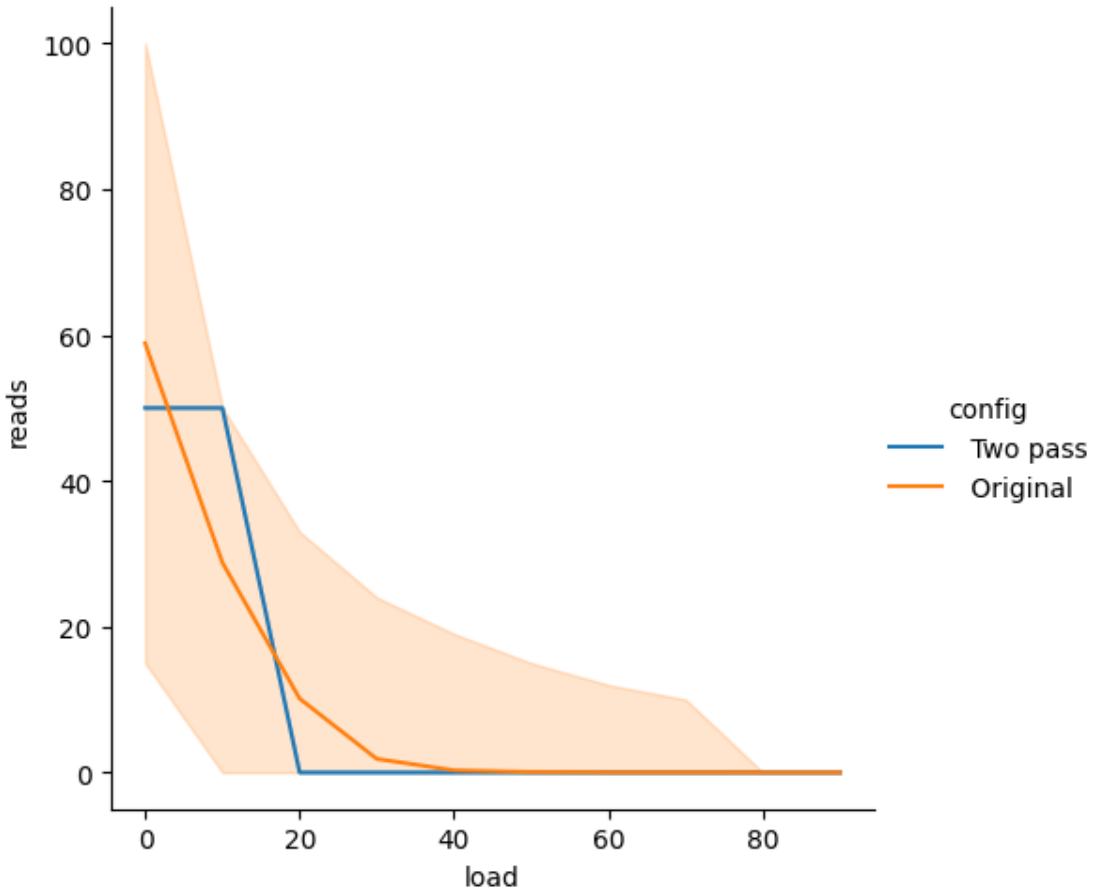
1) Only the first pass is over the entire server list, as the second pass can just focus on eligible servers
2) Both passes have simplified logic compared to the original

[15]:
```
class Cluster_two_pass(Cluster):
    def SelByLoad(self, NeedSpace):
        #print("===== SelByLoad Start")
        #print("=== first pass")
        best_load = 100
        best_pos = 0
        eligible_servers = []
        for i, np in enumerate(self.servers):
            #print(f"=== np = {np.name}, load = {np.load} best_load =␣
↪{best_load}")
            if np.load >= self.maxload:
                continue
            else:
                if np.load < best_load:
                    #print(f"updating best load from {best_load} to {np.load}")
                    best_load = np.load
                    sp = np # preselect the best eligible server
                eligible_servers.append(i)

        max_eligible_load = best_load + self.fuzz

        #print(f"sp = {sp.name}")
        #print(f"eligible_servers = {eligible_servers}")
        #print(f"best_load = {best_load}")
        #print(f"max_eligible_load = {max_eligible_load}")

        #print("=== second pass")
        for i in eligible_servers:
            #print(f"=== np = {np.name}, load = {np.load} best_load =␣
↪{best_load}")
            np = self.servers[i]

            if (np.load < max_eligible_load):
                #print("np is within a fuzz of sp, consider for RR")
                if (sp.RefR > np.RefR):
```

```
                    sp = np
                    #print ("sp has higher RefR than np, picking np")
                #else:
                    #print ("sp has lower RefR than np, not picking np")
            #else:
                #print("np is outside of a fuzz of sp, skip")

            #print(f"final state of run for np {np.name} ({np.load}): sp = {sp.
 ↪name} ({sp.load})")
        # return the selected node
        return sp
```

Looking at the many randomised load orders example, we now see a completely stable decision, with the correct round robin placement between the two nodes with scores between 0 and 15 in all load orderings

[48]:
```
loads = [0,10,20,30,40,50,60,70,80,90]
results = ""

for rand in range(1,10000):
    cluster = Cluster_two_pass(config = f"Two pass", fuzz = 15, maxload = 80,␣
 ↪disklinger = 0)
    for i, load in enumerate(random.sample(loads, len(loads))):
        cluster.addServer(f"{i}", load, 0, 0)
    test = TestRun(cluster = cluster)
    results += "\n" + test.run(duration = 1, refReset = 600, readspersec = 100,␣
 ↪writespersec = 0, progress = False)

for rand in range(1,10000):
    cluster = Cluster_original(config = f"Original", fuzz = 15, maxload = 80,␣
 ↪disklinger = 0)
    for i, load in enumerate(random.sample(loads, len(loads))):
        cluster.addServer(f"{i}", load, 0, 0)
    test = TestRun(cluster = cluster)
    results += "\n" + test.run(duration = 1, refReset = 600, readspersec = 100,␣
 ↪writespersec = 0, progress = False)

df = pd.read_csv(StringIO(results), header = None, names = test.headers())

sns.relplot(data=df, x="load", y="reads", kind='line', hue='config',␣
 ↪errorbar=('pi', 100));
```



Summary This algorithm appears to behave perfectly in all the problem scenarios identified whilst analysing the previous algorithms. It make have other downsides, such as increased run time.

4.2.3 Benchmarking algorithms

Although we cannot draw conclusions about the final performance from benchmarking these Python implementations, it will be informative to see if the proposed alternative algorithms are significantly slower.

Creating 3 clusters with the same 64 servers (with randomised loads), and using the ipython built in timing mechanism.

[ ]:
```
server_count=64
loads = []

for i in range(0, server_count):
    loads.append(random.randint(0,100))
```

The test is performing 1000 reads a second for 100 (simulated) seconds, and we are running the test

several times and taking the average for a representative result using the built in iPython `timeit` wrapper

[66]:
```
cluster = Cluster_original(config = f"Original", fuzz = 15, maxload = 80,␣
 ↪disklinger = 0)
for i, load in enumerate(loads):
    cluster.addServer(f"{i}", load, 0, 0)
test = TestRun(cluster = cluster)
%timeit -n 3 test.run(duration = 100, refReset = 10, readspersec = 1000,␣
 ↪writespersec = 0, progress = False);

cluster = Cluster_best_seen_load(config = f"Best seen load", fuzz = 15, maxload␣
 ↪= 80, disklinger = 0)
for i, load in enumerate(loads):
    cluster.addServer(f"{i}", load, 0, 0)
test = TestRun(cluster = cluster)
%timeit -n 3 test.run(duration = 100, refReset = 10, readspersec = 1000,␣
 ↪writespersec = 0, progress = False);

cluster = Cluster_two_pass(config = f"Two pass", fuzz = 15, maxload = 80,␣
 ↪disklinger = 0)
for i, load in enumerate(loads):
    cluster.addServer(f"{i}", load, 0, 0)
test = TestRun(cluster = cluster)
%timeit -n 3 test.run(duration = 100, refReset = 10, readspersec = 1000,␣
 ↪writespersec = 0, progress = False);
```

```
908 ms ± 7.74 ms per loop (mean ± std. dev. of 7 runs, 3 loops each)
1.19 s ± 14.5 ms per loop (mean ± std. dev. of 7 runs, 3 loops each)
1.01 s ± 36.4 ms per loop (mean ± std. dev. of 7 runs, 3 loops each)
```

All of these complete in approximately one second on my machine, which gives a run time ranging from 9 to 12 microseconds per decision. The original implementation is fastest, with the 'two pass' implementation being ~10% slower and the 'best seen load' implementation being ~20% slower.

5 Conclusions

If my understanding of the XRootD C++ code is correct, and my Python reproduction of the code is representative, there are clear problems with the current CMSD *SelByLoad* implementation, which result in the issues seen. Namely:

1) Heavily loaded nodes receiving large amounts of transfers in some circumstances
2) Lightly loaded nodes within a fuzz of the best score not receiving any transfers in some circumstances
3) The circumstances that lead to 1 and 2 are unknown to the operator of the CMSD cluster

I believe that the tunables available for controlling *SelByLoad* are insufficient to prevent the behaviours seen (without causing other significant problems).

A potential alternative single pass algorithm 'best seen load' completely prevented problem 1 occurring, but did not address 2 (or 3). It may provide enough benefit with minimal changes to warrant further investigation and implementation into XRootD.

A potential two pass algorithm was also investigated, and seemed to perform well, with completely deterministic placement in all test cases irrespective of ordering in the server list. This addresses all issues above. There are concerns about increased calculation time affecting latency, but basic benchmarking showed a similarly fast execution time as the original implementation (~10% slower).

It may also be worth considering improvements that could be made to the placement algorithm (e.g. weighted placement within the fuzz, or a weighted placement across all eligible nodes). The two pass method could make it much easier to do this.

6 Appendix

6.1 Python framework implementation

[2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from io import StringIO
import random

class Server:
    def __init__(self, name, load, RefR, RefW):
        self.name = name
        self.load = load
        self.StartRefR = RefR
        self.StartRefW = RefW
        self.RefR = RefR
        self.RefW = RefW
        self.reads = 0
        self.writes = 0
    def RefCount(self, NeedSpace):
        if NeedSpace:
            self.RefW += 1
            self.writes += 1
        else:
            self.RefR += 1
            self.reads += 1
    def resetRefCount(self):
        self.RefR = 0
        self.RefW = 0
    def headers(self):
        return ["name", "load", "reads", "writes"]
    def __str__(self):
        return f"{self.name}, {self.load}, {self.reads}, {self.writes}"
```

```
class Cluster:
    def __init__(self, config, fuzz, maxload, disklinger):
        self.servers = []
        self.config = config
        self.fuzz = fuzz
        self.maxload = maxload
        self.disklinger = disklinger

    def headers(self):
        return ["config", "fuzz", "maxload", "disklinger", "position"]  + self.
 ↪servers[0].headers()

    def summary(self):
        summaryList = []
        for i,server in enumerate(self.servers):
            summaryList.append(f"{self.config}, {self.fuzz}, {self.maxload},␣
 ↪{self.disklinger}, {i}, {server}")
        return summaryList

    def __str__(self):
        return "\n".join(self.summary())

    def addServer(self, name, load, RefR, RefW):
        self.servers.append(Server(name, load, RefR, RefW))

    def RefReset(self):
        for server in self.servers:
            server.resetRefCount()

    def SelNode(self, NeedSpace):
        server = None
        server = self.SelByLoad(NeedSpace)

        if server is not None:
            server.RefCount(NeedSpace)
        else:
            print("no placement found")

class TestRun:
    def __init__(self, cluster):
        self.cluster = cluster
        self.timer = 0
        self.reads = 0
        self.writes = 0

    def run(self, duration, refReset, readspersec, writespersec, progress):
```

```
        output = []
        self.duration = duration
        self.refReset = refReset
        self.readspersec = readspersec
        self.writespersec = writespersec

        for timer_sec in range(0, duration):
            self.timer = timer_sec
            if (timer_sec % refReset == 0):
                self.cluster.RefReset()
            for i in range(0,readspersec):
                self.cluster.SelNode(False)
                self.reads += 1
            # reads happen before writes in this world, deal with it!
            for i in range(0,writespersec):
                selfcluster.SelNode(True)
                self.writes += 1
            if progress:
                output = output + self.summary()
        if progress:
            return "\n".join(output)
        else:
            return "\n".join(self.summary())

    def headers(self):
        return ["time", "total_reads", "total_writes", "duration", "refReset",␣
↪"writespersec", "readspersec"]  + self.cluster.headers()

    def summary(self):
        summaryList = []
        for server in self.cluster.summary():
            summaryList.append(f"{self.timer}, {self.reads}, {self.writes},␣
↪{self.duration}, {self.refReset}, {self.writespersec}, {self.readspersec}, "␣
↪+ server)
        return summaryList

    def __str__(self):
        return "\n".join(self.summary())
```

[ ]: