

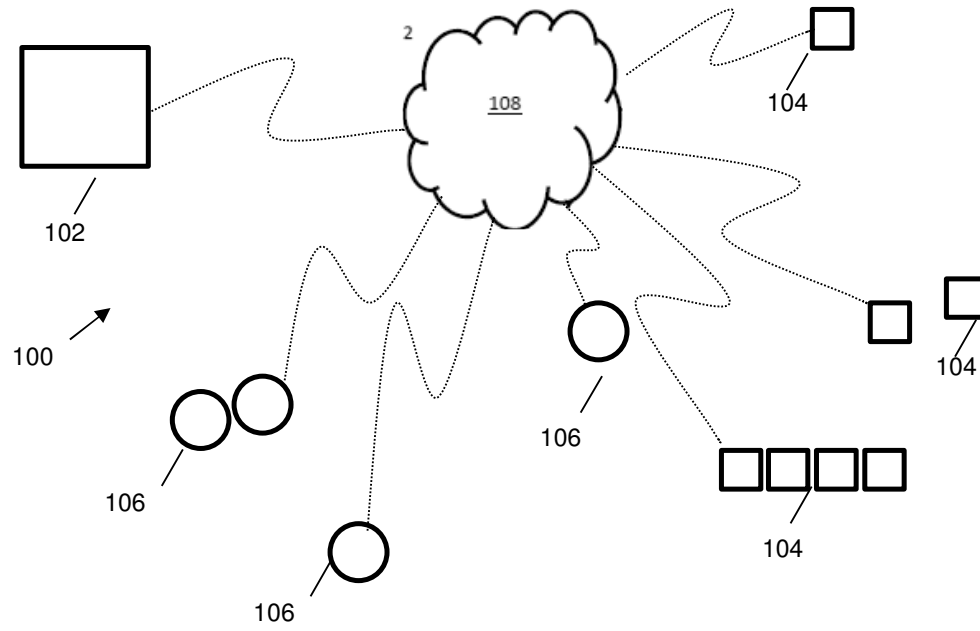


FIG. 1

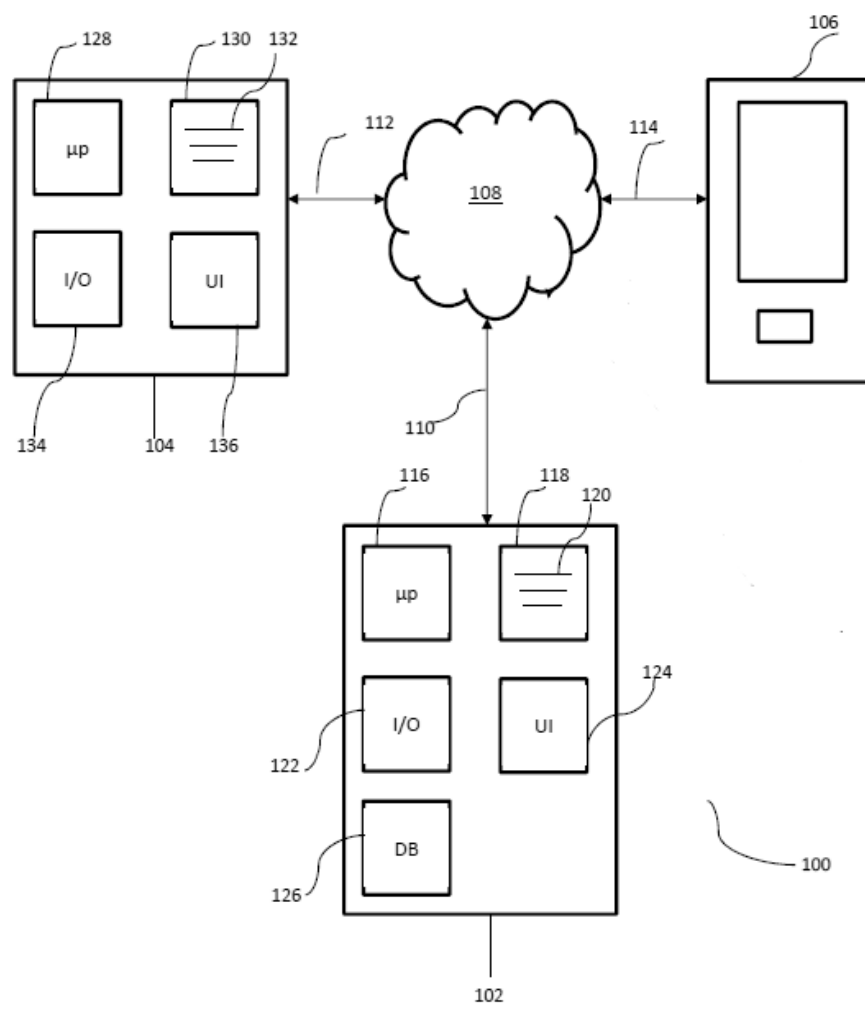


FIG. 2

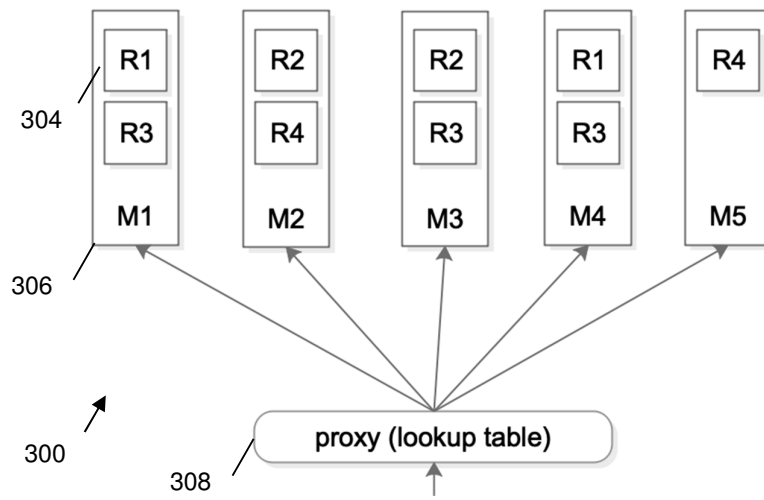


FIG. 3

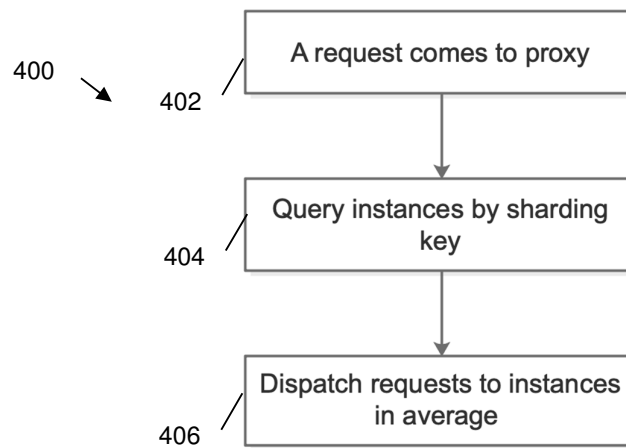


FIG. 4

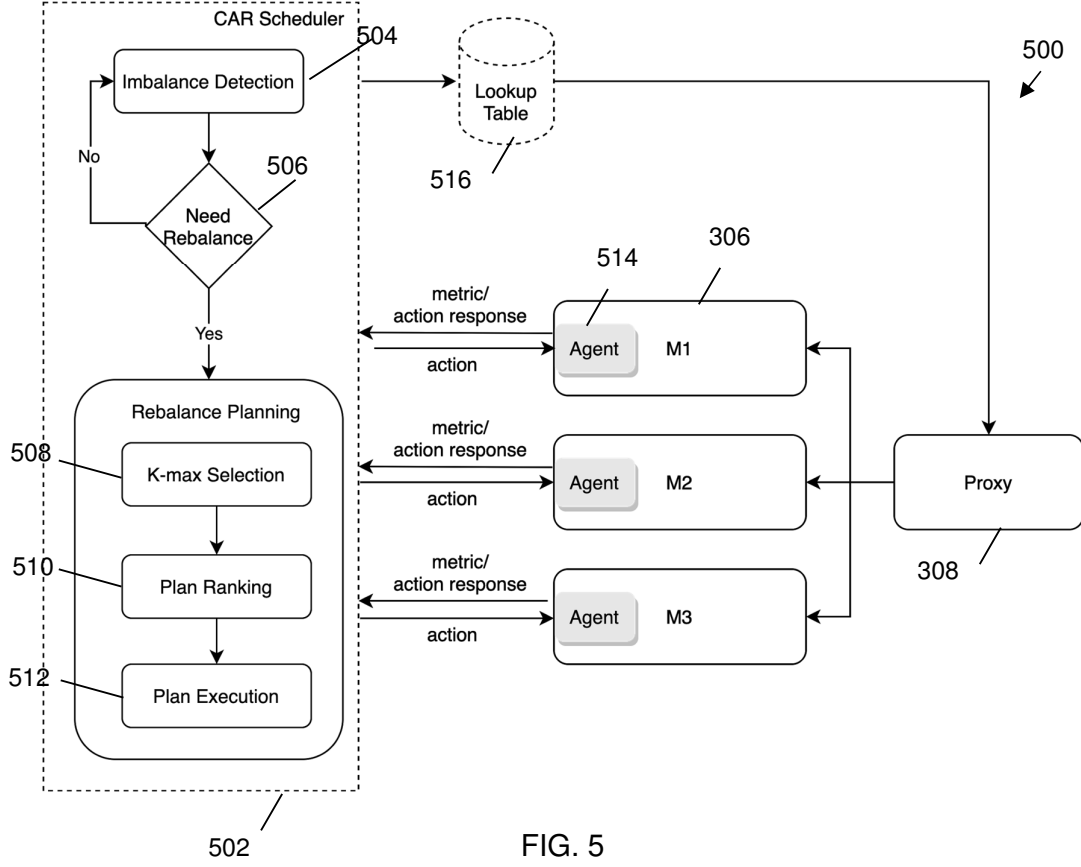


FIG. 5

Algorithm 1: Uniformity level of availability zone

Input: All the shards and replicas, the portfolio P of each replica

Output: Uniformity score

```

1  $score = 0$ 
  // iterate each shard in the system
2 for each shard  $S$  do
3   for each distinct pair of replica  $(r, r')$  in  $S$  do
4     if  $AZ(r) = AZ(r')$  then
5        $score \leftarrow score + 1$ ;
6 return  $score$ 
  
```

FIG. 6

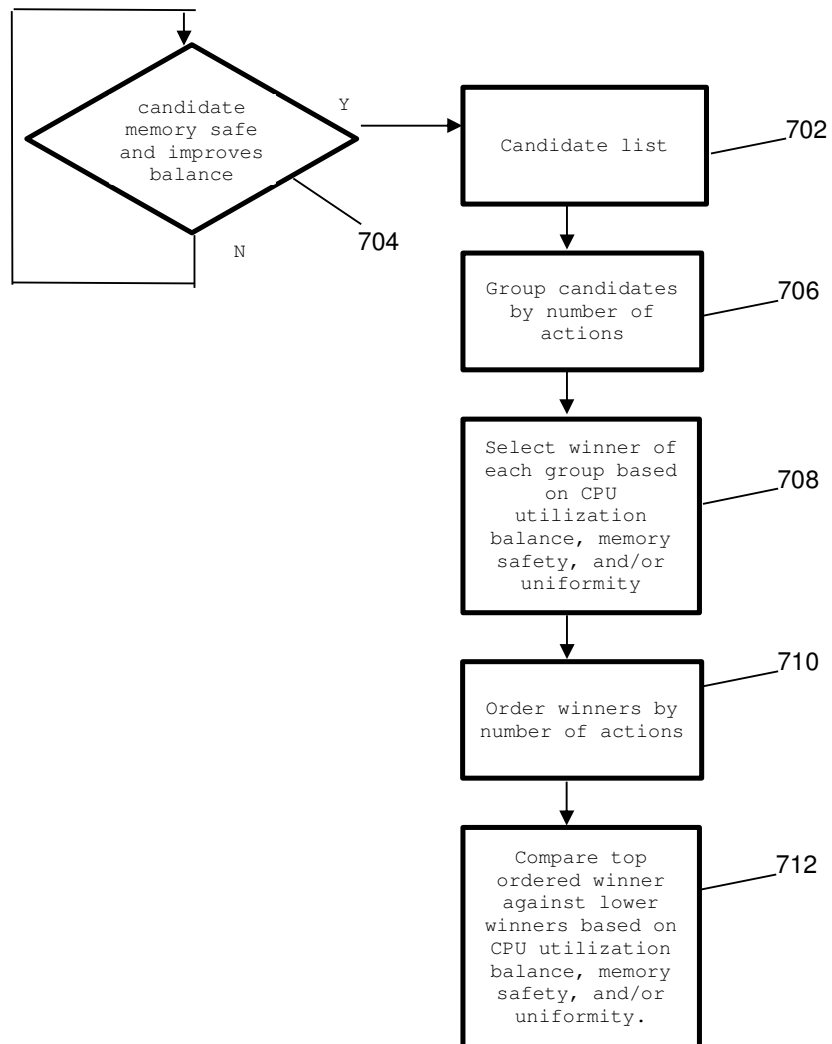


FIG. 7

Algorithm 2 KMax

Input: $\mathbf{S}$ =the current system state, $\mathbf{K}$ =expect maximal move actions,
 $\mathbf{R}$ =predefined maximal iterate times, α =predefined mostly strict memory safety level

Output: $\mathbf{Z}$ =the total plan candidates set

```

1: Initialize an empty set  $\mathbf{Z}$  of candidate plans
2: for  $k = \{1 \dots \mathbf{K}\}$  do
3:   Initialize an empty array  $L$ 
4:   for iterate  $k$  times do
5:     Select the heaviest machine  $m_{max}$  (with largest  $C(m)$ ) in  $\mathbf{S}$ 
6:     Select the heaviest replica  $r_{max}$  (with largest  $C(r)$ ) on  $m_{max}$ 
7:     Remove  $r_{max}$  on  $m_{max}$  and Store  $r_{max}$  in  $L$ 
8:      $C(m_{max}) \leftarrow C(m_{max}) - C(r_{max})$  // update state of  $m_{max}$ 
9:      $M(m_{max}) \leftarrow M(m_{max}) - M(r_{max})$ 
10:     $R(m_{max}) \leftarrow R(m_{max}) - \{r_{max}\}$ 
11:   end for
12:   for iterate  $\mathbf{R}$  times do
13:     Shuffle the replicas collected in  $L$  randomly.
14:     for each replica  $r$  in  $L$  do // assign each collected replica to the
        current lightest constraint machine
15:        $assigned = false$ 
16:       for each  $m$  in  $\mathbf{S}$  ordered by  $C(m)$  do // the first in order is
        lightest
17:         if  $m$  has no duplicated replica of  $r$  and  $m$  has enough
        memory where  $M(m) + M(r) < \alpha$  then
18:            $C(m) \leftarrow C(m) + C(r)$ 
19:            $M(m) \leftarrow M(m) + M(r)$ 
20:            $R(m) \leftarrow R(m) + \{r\}$ 
21:            $assigned \leftarrow true$ 
22:           break
23:         end if
24:       end for
25:       if not  $assigned$  then
26:         go back to Line 12 for next iterating
27:       end if
28:     end for
29:     Call GeneratePlan to generate plan given  $\mathbf{S}$  and  $S_{opt}$  where  $S_{opt}$ 
        is the optimal state after rebalance
30:     Store the plan to  $\mathbf{Z}$ 
31:   end for
32: end for

```

FIG. 8

Algorithm 3 GenUnsortedActions

Input: $\mathbf{S}$ =the input system state,**Input:** $\mathbf{S_p}$ =the optimal system state after rebalance,**Output:** $\mathbf{L}$ =the unsorted actions array

- 1: Generate positive set and negative set by the calculating the difference of $\mathbf{S}$ and $\mathbf{S_p}$
 - 2: Initialize an empty array $\mathbf{L}$
 - 3: **for** each two-tuple (mt,rt) in positive set **do**
 - 4: Find the first item (ms,rs) in negative set where $rs=rt$
 - 5: Generate an action (r,ms,mt) and store it to $\mathbf{L}$
 - 6: Remove (ms,rs) in negative set
 - 7: **end for**
 - 8: Return $\mathbf{L}$
-

FIG. 9

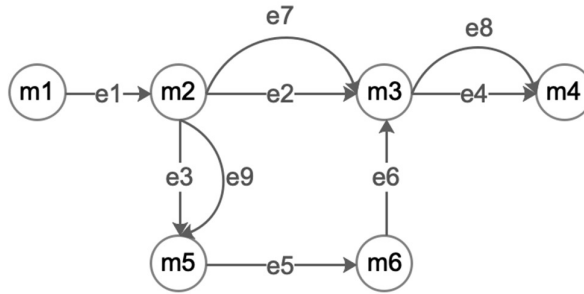


FIG. 10

Algorithm 4 SortActions

Input: **actions**=the disordered actions, **S**=the current system state,
 α =predefined mostly strict memory safety level

Output: **L**=the ordered actions

```

1: Build directed graph G from actions, initialize weight of each vertex as
   its relevant  $M(m)$  in S
2: Initialize an empty array L, the array index denotes the action seq.
3: for G has at least one edge do
4:   for randomly select a zero out-degree vertex  $v$  do
5:     for each incoming edge  $e$  do
6:       Remove  $e$  in G
7:       Put the relevant action on  $e$  into L
8:     end for
9:   remove  $v$  in G
10:  end for
11:  if G has at least one edge then
12:    Select an edge  $e$  in G whose source is not a zero in-degree vertex.
    Let the selected  $e$  with minimal  $em = M(tar) + M(r)$  and  $em < \alpha$  where
     $tar$  is the target vertex of  $e$ ,  $r$  is the relevant replica of  $e$ , and meaning
    of  $em$  is the estimate memory after  $r$  added to  $tar$ .
13:    if not valid  $e$  found then
14:      return FAIL
15:    end if
16:    update source machine state  $M(src) -= M(r)$  where  $src$  is the
    source vertex of found  $e$ 
17:    update target machine state  $M(tar) += M(r)$  where  $tar$  is the
    target vertex of found  $e$ 
18:    Remove  $e$  in G
19:    Put the action on  $e$  into L
20:  end if
21: end for
22: return L

```

FIG. 11