More precisely: for all positions p,

$$block[p].cns = \left|\{C \in Clients \mid p \in map_C[block[p].bid].psns \wedge map_C[block[p].bid].ts = block[p].ts\}\right| \quad (5)$$

The code maintains this invariant by linking any change to the local map with an operation on the *block[p].cns* value. These changes happen when the contents of the block is updated or deleted (by marking the block as *free*). The SyncPositons operation (see Algorithm 2) updates a client's local map to reflect changes performed by other clients and frees old data. The *WriteBlock* and *DuplicateBlock* set the *block[p].cns* value to 1 in order to trigger map changes in other clients.

As a corollary, we have the following:

INV-1′. When $block[p].cns$ has the maxim value ($|Clients|$), then every client's local map contains the latest information about position p. More precisely, for each p:

$$block[p].cns = |Clients| \Rightarrow \forall C \in clients,\ p \in map_C[block[p].bid].psns \wedge \wedge\ map_C[block[p].bid].ts = block[p].ts \quad (6)$$

INV-2. For each block, a valid position is always known to all CAOS clients.

More precisely, for any block id B, there is a position p such that

$$block[p].bid = B \quad \wedge \quad p \in \bigcap_{C \in Clients} map_C[B].psns$$

Before we prove this invariant, we provide some intuition. To maintain INV-2, we use the $map[B].vf$ set stored in the client's map. This set tracks which are the positions p of a block that a client has observed to have a maximum value for $block[p].cns$. In order to prevent data loss, we require that (1) at any time each client can only reassign a single position, and (2) that at least one position still remains if all the clients decide to reassign one position.

The second part of the requirement (2) is easily achieved by checking that the size of $map[B].vf$ is bigger than the number of clients. We address the first requirement (1) by requiring each client mark the $map[B].vf$ set as empty whenever they reassign a position from it. This will prevent the client to reassign any consolidated positions until it re-learns their location.

Proof. We prove it for the case that there are two clients, say C and $\mathcal{D}$; as will be seen, the proof generalises intuitively to more clients. Suppose INV-2 is not an invariant; then there is a transition from a state st_3 in which INV-2 holds for a block id B, to a state st_4 in which it does not hold for B. Suppose client C reallocates the crucial position p in st_3 which is lost in st_4. Since st_3 satisfies INV-2, in st_3 $p \in map_C[B].psns$ and $p \in map_{\mathcal{D}}[B].psns$, and since st_4 does not satisfy INV-2, in st_4 $map_C[B].psns \cap map_{\mathcal{D}}[B].psns = \emptyset$. Using Algorithms 2 and 5, we see that the transition for C from st_3 to st_4 required $|map_C[B].vf| > 2$ in st_3, so suppose that $map_C[B].vf \supseteq \{p, q, r\}$ in st_3. Since $vf \subseteq psns$ (Algorithms 2 and 5), we have $\{p, q, r\} \subseteq map_C[B].psns$ in st_3.

Let st_0 be the state immediately after the previous reallocation by C. Then $map_C[B].vf = \emptyset$ in st_0. Since $q, r \in map_C[B].vf$ in st_3, there was a state between st_0 and st_3 in which $q.cns = 2$, and one in which $r.cns = 2$. Let st_1 be the most recent of those states. Either q or r was reallocated between st_1 and st_3, and by definition of st_0, that reallocation was done by $\mathcal{D}$. Consider the most recent reallocation by $\mathcal{D}$ done before st_3, say in a state st_2. Now we have states $st_0, st_1, st_2, st_3, st_4$ in temporal order. Based on Agorithms 2 and 5, $|map_C[B].vf| > 2$ in st_2, so say $map_{\mathcal{D}}[B].vf \supseteq \{t_1, t_2, t_3\}$. Then, in st_2 we have: $block[t_1].cns = block[t_2].cns = block[t_3].cns = 2$, and by INV-1′, $\{t_1, t_2, t_3\} \subseteq map_C[B].psns \cap map_{\mathcal{D}}[B].psns$. In st_3, $\mathcal{D}$'s transition has removed one element from $map_{\mathcal{D}}[B].psns$, and in st_4, C's transition has removed one element from $map_C[B].psns$. Therefore, in st_4, $map_C[B].psns \cap map_{\mathcal{D}}[B].psns$ is non-empty, contradicting our hypothesis. □

E ALGORITHMS

Algorithm 4: Write a block to the store.

Input: block id, block, data
Output: block, status

```
1  function PrepareWrite (bid, block, data)
2      if (block.bid = bid or block.bid = free) then
3          block.bid ← bid;
4          block.data ← data;
5          block.cns ← 1;
6          block.ts ← current_time;
7          status ← block.data;
8      else
9          status ← null;
10     return (block, status);
```

Algorithm 5: Duplicate a store block to a new position.

Input: source block, destination block, destination position
Output: destination block, client map

```
1  function DuplicateBlock (sblock, dblock, p, map_c)
2      if ((dblock.cns = |clients| and
         |map_c[dblock.bid].vf| > |clients|) or
3        (dblock.cns = 1 and |map_c[dblock.bid].psns| > 1))
4      then
5          dblock.bid ← sblock.bid;
6          dblock.data ← sblock.data;
7          dblock.ts ← sblock.ts;
8          dblock.cns ← 1;
9          clear map_c[dblock.bid].vf;
10         move p from map_c[dblock.bid].psns to
             map_c[sblock.bid].psns;
11     return (dblock, map_c);
```

Algorithm 6: Update the local buffer data structure.

Input: block, position, buffer, obfuscation client map
Output: block, buffer

```
1  function UpdateBuffer(blk, p, buffer, map_oc)
2      if (buffer not full) then
3          add blk to buffer;
4      if (buffer is full) then
5          buf_blk ←R buffer;
6          (blk, map_oc) ←
             DuplicateBlock(buf_blk, blk, p, map_oc);
7          if (buf_blk = blk) then
8              remove buf_blk from buffer;
9      return (blk,buffer);
```