A. Implementation 1

The condition $\mathsf{WellRec}(\mathbb{I})$, defined in Figure 11, requires us to provide a function Rec. Given the state of a node, σ, and a node identifier, n, Rec must return the number of updates that σ has received from n. In this implementation, $clock$ stores the number of updates received from other nodes. Therefore, we define Rec to be $clock$. Let us define the function $clock'$ that mirrors the function Rec' in Figure 11 as follows

$$clock'(W, n', n) \triangleq let\ (H[n' \mapsto (_, \sigma, _)], _) = W\ in\ clock(\sigma)(n)$$

We first prove the main condition of WellRec, CauseCond. Then, we sketch the straightforward proof of the other three conditions.

To prove the CauseCond condition, we need to prove the following monotonicity property for vector clocks. We refer to the vector clock of the poststate of a label as the vector clock of that label. If a label $l_{\mathcal{I}}$ causally precedes another label $l'_{\mathcal{I}}$, the clock of $l_{\mathcal{I}}$ is less than or equal to the clock of $l'_{\mathcal{I}}$ for every node. Further, if $l'_{\mathcal{I}}$ is a put label, the clock of $l_{\mathcal{I}}$ is strictly less than the clock of $l'_{\mathcal{I}}$ for the node identifier of $l'_{\mathcal{I}}$.

Lemma 5 (Clock Monotonicity).

$$\begin{array}{l}\forall p, h_{\mathcal{I}}, W_{\mathcal{I}}, l_{\mathcal{I}}, l'_{\mathcal{I}}, n: \\ (W_{\mathcal{I}0}(p) \xrightarrow{h_{\mathcal{I}}}{}^{*}_{\mathcal{I}(\mathbb{I}_1)} W_{\mathcal{I}} \wedge l_{\mathcal{I}} \curvearrowright_{h_{\mathcal{I}}} l'_{\mathcal{I}}) \Rightarrow \\ (clock(\mathsf{LPostState}(l_{\mathcal{I}}), n) \leq clock(\mathsf{LPostState}(l'_{\mathcal{I}}), n) \\ \quad \wedge\ (\mathsf{LIsPut}(l'_{\mathcal{I}}) \wedge n = LNode(l'_{\mathcal{I}})) \Rightarrow \\ \qquad clock(\mathsf{LPostState}(l_{\mathcal{I}}), n) < clock(\mathsf{LPostState}(l'_{\mathcal{I}}), n))\end{array}$$

Let us see why the above lemma holds. By the definitions of Figure 12, the causal order holds by either the node order, gets-from relation, or transitivity. Firstly, if it holds by the node order, the conclusion is immediate by noting the following facts. On every step, the mapping of the vector clock for every node is nondecreasing. On a put step, the vector clock of the node for the node itself is incremented. Secondly, we consider the case that the causal relation holds by a gets-from relation from the put label $l_{\mathcal{I}}$ to the get label $l'_{\mathcal{I}}$. Let n and n' be the node identifiers of $l_{\mathcal{I}}$ and $l'_{\mathcal{I}}$ respectively. The get label $l'_{\mathcal{I}}$ can get the value put by the put label $l_{\mathcal{I}}$ only if there exists an update label $l''_{\mathcal{I}}$ by n' before $l'_{\mathcal{I}}$ that receives the update of $l_{\mathcal{I}}$. When the update is being received, the guard function checks that the vector clock value of $l_{\mathcal{I}}$ for every node n'' other than n is less than or equal to the current vector clock value for n''. Then the update function remaps the current vector clock value for n to the vector clock value of $l_{\mathcal{I}}$ for n. Thus, the vector clock of $l_{\mathcal{I}}$ is less than or equal to the vector clock of $l''_{\mathcal{I}}$ for every node. As mentioned above, the vector clock of a node in nondecreasing. Therefore, as $l''_{\mathcal{I}}$ precedes $l'_{\mathcal{I}}$, and they are by the same node n', the vector clock of $l''_{\mathcal{I}}$ is less than or equal to the vector clock of $l'_{\mathcal{I}}$ for every node. The inequality of the conclusion is immediate from the transitivity of the above two inequalities. Thirdly, if the causal order holds by the transitivity of other causal orders, the conclusion is immediate from the transitivity of the equalities and inequalities of the induction hypotheses.

Instantiating Rec' with the function $clock'$, the statement of the CauseCond condition for this implementation is as follows:

Lemma 6 (CauseCond).

$$\begin{array}{l}\forall p, h_{\mathcal{I}}, W_{\mathcal{I}}, l_{\mathcal{I}}, W'_{\mathcal{I}}, l''_{\mathcal{I}}: \\ \quad (W_{\mathcal{I}0}(p) \xrightarrow{h_{\mathcal{I}}}{}^{*}_{\mathcal{I}(\mathbb{I}_1)} W_{\mathcal{I}} \wedge W_{\mathcal{I}} \xrightarrow{l_{\mathcal{I}}}_{\mathcal{I}(\mathbb{I}_1)} W'_{\mathcal{I}} \\ \quad\ \wedge\ \mathsf{LIsUpdate}(l_{\mathcal{I}}) \wedge \\ \quad\ let\ _, _, n \triangleright update(_, _, _, m) \colon _ = l_{\mathcal{I}} \\ \qquad (_, _, _, _, _, _, l'_{\mathcal{I}}) = m\ in \\ \quad\ \mathsf{LIsPut}(l''_{\mathcal{I}}) \wedge l''_{\mathcal{I}} \curvearrowright_{h_{\mathcal{I}}} l'_{\mathcal{I}}) \Rightarrow \\ \quad let\ n'', c'' \triangleright put(_, _, _) \colon _, _ = l''_{\mathcal{I}}\ in \\ \quad c'' \leq clock'(W_{\mathcal{I}}, n, n'')\end{array}$$

Let us see how the above lemma holds. At a high level, the vector clock is nondecreasing from $l''_{\mathcal{I}}$ to $l_{\mathcal{I}}$ because Lemma 5 implies that it is nondecreasing from $l''_{\mathcal{I}}$ to $l'_{\mathcal{I}}$ and the guard condition implies that it is nondecreasing from $l'_{\mathcal{I}}$ to $l_{\mathcal{I}}$. More precisely, an update label $l_{\mathcal{I}}$ with the prestate $W_{\mathcal{I}}$ applies an update originating from the put label $l'_{\mathcal{I}}$, and another put label $l''_{\mathcal{I}}$ causally precedes $l'_{\mathcal{I}}$. Let n, n', and n'' be the node identifiers of $l_{\mathcal{I}}$, $l'_{\mathcal{I}}$, and $l''_{\mathcal{I}}$ respectively. Let c and c' be

$$\begin{array}{rcl} c & \triangleq & clock'(W_{\mathcal{I}}, n, n'') \\ c' & \triangleq & clock(\mathsf{LPostState}(l'_{\mathcal{I}}))(n'') \end{array}$$

Let c'' be the clock of $l''_{\mathcal{I}}$. We want to show that $c'' \leq c$.

We first show that

$$c'' = clock(\mathsf{LPostState}(l''_{\mathcal{I}}))(n'')$$

We have a put label $l''_{\mathcal{I}}$ by node n'' with clock c''. Therefore, (1) the clock of the operational semantics for n'' in the poststate of $l''_{\mathcal{I}}$ is c''. (2) The vector clock of n'' in the poststate of $l''_{\mathcal{I}}$ is $clock(\mathsf{LPostState}(l''_{\mathcal{I}}))$. We also have that (3) in every state, the clock of the operational semantics for a node and the vector-clock value of the node for itself are equal. This is because they are zero in the initial state and are concurrently incremented only in put steps by the node. The above three equalities imply the conclusion equality.

As $l''_{\mathcal{I}}$ causally precedes $l'_{\mathcal{I}}$, and $l'_{\mathcal{I}}$ is a put label, by Lemma 5, we have (1) $c'' \leq c'$ and (2) $(n'' = n') \Rightarrow (c'' < c')$. Furthermore, by the conditions of the guard function, we have (3) $(n'' \neq n') \Rightarrow (c' \leq c)$ and (4) $(n'' = n') \Rightarrow (c' = c+1)$. We consider two cases. Case $n'' \neq n'$: by 1 and 3 above, we have $c'' \leq c$. Case $n'' = n'$: by 2 and 4 above, we have $c'' \leq c$.

We now prove the other three conditions of well-reception. The condition InitCond is immediate from the fact that the init function returns a constant zero function as the initial vector clock. The three cases of StepCond follow from the following facts: (1) The put function only increments the mapping of the vector clock for the current node $self$. (2) The get function keeps the vector clock unchanged. (3) Let $clock_1$ and $clock_2$ be the vector clocks of the current node in the prestate and the poststate of update label, c' be the clock value of the update label, n' be the sender node, and $clock'$ be the sender vector clock. We have to show that $clock_1(n') + 1 = c'$ and $clock_2(n') = clock_1(n') + 1$ and also $\forall n'' \colon n'' \neq n' \Rightarrow clock_2(n'') = clock_1(n'')$. By a simple induction, it can be shown that for every node, the clock value for the node that the instrumented semantics maintains is equal to the vector-clock value of the node for itself that the implementation maintains. Therefore, it can be shown that (3.1) $c' = clock'(n')$. The guard function checks that (3.2) $clock'(n') = clock_1(n') + 1$. The update function updates only the mapping of the vector clock for n' to $clock'(n')$. Therefore, we have that (3.3) $clock_2 = clock_1[n' \mapsto clock'(n')]$. The conclusions follow from the above three equalities. The condition SeqCond is proved by a simple induction with the invariant that for every node, the $store$ map of the implementation is equal to the abstract map.

B. Implementation 2

The condition $\mathsf{WellRec}(\mathbb{I})$, defined in Figure 11, requires us to provide function Rec. In this implementation, the function rec stores the number of updates received from other nodes. Therefore, we choose function Rec to be rec. Let us define the function rec' that mirrors the definition of Rec' from Figure 11 as follows

$$rec'(W, n, n') \quad \triangleq \quad let\ (H[n \mapsto (_, \sigma, _)], _) = W\ in \\ rec(\sigma)(n')$$

We now prove the condition $\mathsf{CauseCond}$ of well-reception. The other three conditions of well-reception for this implementation can be proved similar to the previous implementation. To prove the condition $\mathsf{CauseCond}$, we first state two important invariants of the implementation. The first invariant states the transitivity property explained above that if a label $l_{\mathcal{I}}$ is causally dependent on a put operation $l'_{\mathcal{I}}$, the identifier of $l'_{\mathcal{I}}$ is either directly or indirectly in the dependencies of $l_{\mathcal{I}}$.

Lemma 7 (Update Dependency Transitivity).

$$\begin{array}{l} \forall p, h_{\mathcal{I}}, W_{\mathcal{I}}, l_{\mathcal{I}}, l'_{\mathcal{I}}: \\ (W_{\mathcal{I}0}(p) \xrightarrow{h_{\mathcal{I}}}{}^{*}_{\mathcal{I}(\mathbb{I}_2)} W_{\mathcal{I}} \\ \quad \wedge \mathsf{LIsPut}(l_{\mathcal{I}}) \wedge \mathsf{LIsPut}(l'_{\mathcal{I}}) \wedge l'_{\mathcal{I}} \curvearrowright_{h_{\mathcal{I}}} l_{\mathcal{I}}) \Rightarrow \\ let\ _, _ \triangleright put(_, _, _) \colon _, u = l_{\mathcal{I}}\ in \\ ((\mathsf{LNode}(l'_{\mathcal{I}}), \mathsf{LClock}(l'_{\mathcal{I}})) \in udep(u) \\ \quad \vee (\exists l''_{\mathcal{I}} \colon \mathsf{LIsPut}(l''_{\mathcal{I}}) \wedge l'_{\mathcal{I}} \curvearrowright_{h_{\mathcal{I}}} l''_{\mathcal{I}} \\ \qquad \wedge (\mathsf{LNode}(l''_{\mathcal{I}}), \mathsf{LClock}(l''_{\mathcal{I}})) \in udep(u))) \end{array}$$

The above lemma states that for every put label $l_{\mathcal{I}}$ that emits the update u and every put label $l'_{\mathcal{I}}$ that causally precedes $l_{\mathcal{I}}$, either the timestamp of $l'_{\mathcal{I}}$ is directly in $udep(u)$ or there exists a put label $l''_{\mathcal{I}}$ that depends on $l'_{\mathcal{I}}$ and the timestamp of $l''_{\mathcal{I}}$ is in $udep(u)$.

The second invariant states that, if a put label $l_{\mathcal{I}}$ depends on another put label $l'_{\mathcal{I}}$ and some node has received the update for $l_{\mathcal{I}}$, then it has received the update for $l'_{\mathcal{I}}$ as well.

Lemma 8.

$$\begin{array}{l} \forall p, h_{\mathcal{I}}, W_{\mathcal{I}}, l_{\mathcal{I}}, l'_{\mathcal{I}}, n: \\ (W_{\mathcal{I}0}(p) \xrightarrow{h_{\mathcal{I}}}{}^{*}_{\mathcal{I}(\mathbb{I}_2)} W_{\mathcal{I}} \\ \quad \wedge \mathsf{LIsPut}(l_{\mathcal{I}}) \wedge \mathsf{LIsPut}(l'_{\mathcal{I}}) \wedge l'_{\mathcal{I}} \curvearrowright_{h_{\mathcal{I}}} l_{\mathcal{I}} \\ \quad \wedge \mathsf{LClock}(l_{\mathcal{I}}) \leq rec'(W_{\mathcal{I}}, n, \mathsf{LNode}(l_{\mathcal{I}})) \Rightarrow \\ \mathsf{LClock}(l'_{\mathcal{I}}) \leq rec'(W_{\mathcal{I}}, n, \mathsf{LNode}(l'_{\mathcal{I}})) \end{array}$$

The lemma above can be proved by induction on step transitions. The interesting case is the update transition. Consider an update step that receives an update u that is originated from a put label $l_{\mathcal{I}}$ and that $l_{\mathcal{I}}$ is causally dependent on another put label $l'_{\mathcal{I}}$. We want to show that the update of $l'_{\mathcal{I}}$ is already received. By Lemma 7, we have two cases. Case 1: The identifier of $l'_{\mathcal{I}}$ is directly in $udep(u)$. The guard method checks that its update is already received. Case 2: The identifier of $l'_{\mathcal{I}}$ is indirectly in $udep(u)$; that is, there exists another label $l''_{\mathcal{I}}$ that is causally dependent on $l'_{\mathcal{I}}$, and the timestamp of $l''_{\mathcal{I}}$ is in $udep(u)$. As the timestamp of $l''_{\mathcal{I}}$ is in $udep(u)$, from the guard method checks, we have that the update of $l''_{\mathcal{I}}$ is already received. As $l''_{\mathcal{I}}$ is causally dependent on $l'_{\mathcal{I}}$, and the update of $l''_{\mathcal{I}}$ is already received, by the induction hypothesis, we have that the update of $l'_{\mathcal{I}}$ is already received as well.

Instantiating Rec' with the function rec', the statement of the $\mathsf{CauseCond}$ condition for this implementation is as follows:

Lemma 9 (CauseCond).

$$\begin{array}{l} \forall p, h_{\mathcal{I}}, W_{\mathcal{I}}, l_{\mathcal{I}}, W'_{\mathcal{I}}, l''_{\mathcal{I}}: \\ \quad (W_{\mathcal{I}0}(p) \xrightarrow{h_{\mathcal{I}}}{}^{*}_{\mathcal{I}(\mathbb{I}_1)} W_{\mathcal{I}} \ \wedge\ W_{\mathcal{I}} \xrightarrow{l_{\mathcal{I}}}_{\mathcal{I}(\mathbb{I}_1)} W'_{\mathcal{I}} \\ \quad\ \wedge\ \mathsf{LIsUpdate}(l_{\mathcal{I}}) \wedge \\ \quad\ let\ _, _, n \triangleright update(_, _, _, m) \colon _ = l_{\mathcal{I}} \\ \qquad (_, _, _, _, _, l'_{\mathcal{I}}) = m\ in \\ \quad\ \mathsf{LIsPut}(l''_{\mathcal{I}}) \ \wedge\ l''_{\mathcal{I}} \curvearrowright_{h_{\mathcal{I}}} l'_{\mathcal{I}}) \Rightarrow \\ \quad let\ n'', c'' \triangleright put(_, _, _) \colon _, _ = l''_{\mathcal{I}}\ in \\ \quad c'' \leq rec'(W_{\mathcal{I}}, n, n'') \end{array}$$

If an update is being received that is originated by the put label $l'_{\mathcal{I}}$, and another put label $l''_{\mathcal{I}}$ causally precedes $l'_{\mathcal{I}}$, then the update of $l''_{\mathcal{I}}$ is already received. Similar to the proof of Lemma 8, the proof is based on using Lemma 7 for the case analysis that the identifier of $l''_{\mathcal{I}}$ is directly or indirectly in the dependencies of the update from $l'_{\mathcal{I}}$. Then, the conclusion follows by the guard conditions and Lemma 8.

C. Implementation 3

Note that in Algorithm 2 presented in Figure 14, the map $clock$ keeps track of both the put operations that the node is dependent on and the put operations that it has received. Thus, every put operation that a node has received is regarded as a dependency of the node even if the node has not read the value that it has put. Tracking dependencies can be made more precise by having separate maps for dependencies and received put operations. In the algorithm presented in Figure 17, we have separate maps rec and dep to keep track of received put operations and the dependencies. For this algorithm, we have the following lemmas.

Lemma 10 (Clock Monotonicity).
$\forall p, h_\mathcal{I}, W_\mathcal{I}, l_\mathcal{I}, l'_\mathcal{I}, n$:
$(W_{\mathcal{I}0}(p) \xrightarrow{h_\mathcal{I}}{}^{*}_{\mathcal{I}(\mathbb{I}_3)} W_\mathcal{I} \ \wedge\ l_\mathcal{I} \curvearrowright_{h_\mathcal{I}} l'_\mathcal{I}) \Rightarrow$
$(dep(\mathsf{LPostState}(l_\mathcal{I}), n) \leq dep(\mathsf{LPostState}(l'_\mathcal{I}), n)$
$\wedge\ (\mathsf{LIsPut}(l'_\mathcal{I}) \wedge n = LNode(l'_\mathcal{I})) \Rightarrow$
$dep(\mathsf{LPostState}(l_\mathcal{I}), n) < dep(\mathsf{LPostState}(l'_\mathcal{I}), n))$

Lemma 11 (Dep not above Rec).
$\forall p, h_\mathcal{I}, W_\mathcal{I}, l_\mathcal{I}, l'_\mathcal{I}, n$:
$W_{\mathcal{I}0}(p) \xrightarrow{h_\mathcal{I}}{}^{*}_{\mathcal{I}(\mathbb{I}_3)} W_\mathcal{I} \ \Rightarrow\ dep(W_\mathcal{I}, n) \leq rec(W_\mathcal{I}, n)$

Let us define the function rec' as follows

$$rec'(W, n, n') \triangleq let\ (H[n \mapsto (_, \sigma, _)], _) = W\ in\ rec(\sigma)(n')$$

Using the function rec', we can state the following lemma.

Lemma 12.
$\forall p, h_\mathcal{I}, W_\mathcal{I}, l_\mathcal{I}, W'_\mathcal{I}, l''_\mathcal{I}$:
$let\ _, _, n \triangleright update(_, _, _, m) \colon _ = l_\mathcal{I}$
$(_, _, _, _, _, _, l'_\mathcal{I}) = m$
$n'', c'' \triangleright put(_, _, _) \colon _, _ = l''_\mathcal{I}\ in$
$(W_{\mathcal{I}0}(p) \xrightarrow{h_\mathcal{I}}{}^{*}_{\mathcal{I}(\mathbb{I}_3)} W_\mathcal{I} \ \wedge\ W_\mathcal{I} \xrightarrow{l_\mathcal{I}}_{\mathcal{I}(\mathbb{I}_3)} W'_\mathcal{I}$
$\wedge\ \mathsf{LIsUpdate}(l_\mathcal{I}) \ \wedge\ \mathsf{LIsPut}(l''_\mathcal{I}) \ \wedge\ l''_\mathcal{I} \curvearrowright_{h_\mathcal{I}} l'_\mathcal{I}) \Rightarrow$
$c'' \leq rec'(W_\mathcal{I}, n, n'')$

The condition $\mathsf{WellRec}(\mathbb{I})$ defined in Figure 11 requires the definition of the function Rec for the algorithm $\mathbb{I}$. In this algorithm, the map rec stores the number of updates received from the other nodes. Therefore, we define the function Rec to be rec. By this definition, Lemma 12 proved the main condition CauseCond of the WellRec conditions.

Theorem 5. $\mathsf{WellRec}(\mathbb{I}_3)$

From the above theorem and Theorem 2, we conclude that $\mathbb{I}_3$ is causally consistent. For more details and the proofs, please see our Coq development.

Corollary 3. $\mathsf{CauseConst}(\mathbb{I}_3)$

This algorithm can now be optimized by removing the line that updates the dependencies in the update function. We are working on the proof for the optimized algorithm.

```
𝕀3 (ALGORITHM 3)
State
  (store: Map[K, (V, N, C)],
   rec: Map[N, C],
   dep: Map[N, C])

Update
  (unode: N,
   udep: Map[N, C])
init
  ret (λk.(v0, n0, 0), λn.0, λn.0)

put (self, this)(k, v)
  (s, r, d) ← this;
  d' ← d[self ↦ r[self] + 1];
  r' ← r[self ↦ r[self] + 1];
  s' ← s[k ↦ (v, self, d'[self])];
  ret ((s', r', d'), (self, d'))

get (self, this)(k)
  (s, r, d) ← this;
  (v, n, c) ← s[k];
  d' ← d[n ↦ max(d(n), c)];
  ret (v, (s, r, d'))

guard (self, this)(k, v, u)
  (s, r, d) ← this;
  (n', d') ← u;
  ret forall (λn. n ≠ n' ⇒ d'[n] ≤ r[n]) N
      ∧ d'[n'] = r[n'] + 1

update (self, this)(k, v, u)
  (s, r, d) ← this;
  (n', d') ← u;
  r' ← r[n' ↦ d'[n']];
  d'' ← λn.max(d(n), d'(n))
  s' ← s[k ↦ (v, n', d'[n'])];
  ret (s', r', d'')
```

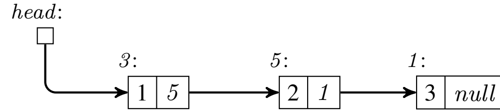
Figure 17. Causally Consistent Map 3

D. Linked-List Client Example

Program 3 (p_3): Linked list client

```
0 →                                              Construction
  put(2, null);                                  ▷ first link
  put(1, 3);
  put(head, 1);                                  ▷ update head
  put(6, 1);                                     ▷ second link
  put(5, 2);
  put(head, 5);                                  ▷ update head
  put(4, 5);                                     ▷ third link
  put(3, 1);
  put(head, 3)                                   ▷ update head
1 →                                              Traversal
  x1 ← get(head);
  if x1 ≠ v0 then
    i1 ← get(x1);                                ▷ item
    x2 ← get(x1 + 1);                            ▷ next-pointer
    if x2 ≠ null then
      i2 ← get(x2);                              ▷ item
      x3 ← get(x2 + 1);                          ▷ next-pointer
      assert(i1 < i2);
      if x3 ≠ null then
        i3 ← get(x3);                            ▷ item
        x4 ← get(x3 + 1);                        ▷ next-pointer
        assert(i2 < i3);
        assert(x4 = null)
```

We give a third, slightly more complex, example client program consisting of two nodes that construct and traverse a linked list. The first node initializes $head$ and adds three links by updating $head$ once each is constructed. When complete, the linked list has the following layout:



where boxes denote the contents of memory, and the annotations above the boxes specify their addresses.

Concurrently, another node reads $head$ and traverses the list. At any point in time, $head$ is either uninitialized (v_0) or else points to one of the three links. The traversing node checks that the stored items are in ascending order and that the length of the list is at most 3. Our automated verifier takes just under 8 minutes to check this program.