A. Implementation 1

The condition WellRec(1), defined in Figure 11, requires us to provide a function Rec. Given the state of a node, \( \sigma \), and a node identifier, \( n \), Rec must return the number of updates that \( \sigma \) 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:

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

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 \( l' \) causally precedes another label \( l \), the clock of \( l \) is less than or equal to the clock of \( l' \) for every node. Further, if \( l' \) is a put label, the clock of \( l \) is strictly less than the clock of \( l' \) for the node identifier of \( l' \).

**Lemma 5 (Clock Monotonicity).**

\[
\forall p, hz, w_2, l, l', n, \:\text{where} \:\text{LNode}(l') = n \text{ and } LNode(l) = n \text{ and } l \text{ causally precedes } l'.
\]

\[
(W_2(p) \xrightarrow{hz} l') W_2 \land l' \leq_{hz} l \Rightarrow \text{clock}(\text{LPostState}(l'), n) \leq \text{clock}(\text{LPostState}(l), n)
\]

We prove this by first proving Lemma 4. The proof of Lemma 4 is a straightforward induction on the length of the causal order that holds by the above two inequalities. Thirdly, if the causal order holds by the node order, the guard condition holds by a gets-from relation from the put label. On a put step, the vector clock of the node for the node itself is equal to the clock of the node for the node itself. As mentioned above, the vector clock of a node in a node in a non-decreasing order. Therefore, as \( l' \) precedes \( l \), and they are by the same node \( n \), the vector clock of \( l' \) is less than or equal to the vector clock of \( l' \) 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 inequalities and inequalities of the induction hypotheses.

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

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

Let us see why the above lemma holds. At a high level, the vector clock is nondecreasing from \( l' \) to \( l \) because Lemma 5 implies that it is nondecreasing from \( l' \) to \( l' \) and the guard condition implies that it is nondecreasing from \( l' \) to \( l \). More precisely, an update label \( l \) with the prestate \( W_2 \) applies an update originating from the put label \( l' \), and another put label \( l' \) causally precedes \( l' \). Let \( n' \) be the node identifiers of \( l' \), \( l' \), and \( l' \) respectively. Let \( c' \) be

\[
c \triangleq \text{clock}'(W, n, n')
\]

**Lemma 6 (CauseCond).**

\[
\forall p, hz, w_2, l, l', n, \:\text{where} \:\text{LNode}(l') = n \text{ and } LNode(l) = n \text{ and } l \text{ causally precedes } l'.
\]

\[
(W_2(p) \xrightarrow{hz} l') W_2 \land l' \leq_{hz} l \Rightarrow \text{clock}(\text{LPostState}(l'), n) \leq \text{clock}(\text{LPostState}(l), n)
\]

Let us see how the above lemma holds. At a high level, the vector clock is nondecreasing from \( l' \) to \( l \) because Lemma 5 implies that it is nondecreasing from \( l' \) to \( l \) and the guard condition implies that it is nondecreasing from \( l' \) to \( l \). More precisely, an update label \( l \) with the prestate \( W_2 \) applies an update originating from the put label \( l' \), and another put label \( l' \) causally precedes \( l' \). Let \( n' \) be the node identifiers of \( l' \), \( l' \), and \( l' \) respectively. Let \( c' \) be

\[
c \triangleq \text{clock}'(W, n, n')
\]

We prove this by first proving Lemma 4. The proof of Lemma 4 is a straightforward induction on the length of the causal order that holds by 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 inequalities and inequalities of the induction hypotheses.

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

Let us see why the above lemma holds. At a high level, the vector clock is nondecreasing from \( l' \) to \( l \) because Lemma 5 implies that it is nondecreasing from \( l' \) to \( l' \) and the guard condition implies that it is nondecreasing from \( l' \) to \( l \). More precisely, an update label \( l \) with the prestate \( W_2 \) applies an update originating from the put label \( l' \), and another put label \( l' \) causally precedes \( l' \). Let \( n' \) be the node identifiers of \( l' \), \( l' \), and \( l' \) respectively. Let \( c' \) be

\[
c \triangleq \text{clock}'(W, n, n')
\]

We prove this by first proving Lemma 4. The proof of Lemma 4 is a straightforward induction on the length of the causal order that holds by 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 inequalities and inequalities of the induction hypotheses.
B. Implementation 2

The condition WellRec(1), 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

\[ \text{rec'}(W, n, n') \triangleq \text{let } (H[n \mapsto (\_\!, \_\!, \_\!)] \mapsto W \text{ in rec}(\sigma)(n') \]

We now prove the condition 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 CauseCond, we first state two important invariants of the implementation. The first invariant states the transitivity property explained above that if a label \( l_x \) is causally dependent on a put operation \( l'_x \), the identifier of \( l'_x \) is either directly or indirectly in the dependencies of \( l_x \).

Lemma 7 (Update Dependency Transitivity).
\[
\forall p, h_Z, W_Z, l_x, l'_x : \\
(W_{20}(p) \xrightarrow{h_Z} \chi_Z(l_x)) \land LIsPut(l_x) \land LIsPut(l'_x) \land l'_x \in h_Z l_x \Rightarrow \\
\text{let } \_\!, \_\!, \_\! \triangleright put(\_\!, \_\!, \_\!) \Rightarrow \_\!, \_\!, \_\! = l_x \text{ in } \\
\text{(LNode}(l'_x), LClock(l'_x)) \in \text{udep}(u) \\
\lor (\exists l''_x : \text{LIsPut}(l''_x) \land l''_x \in h_Z l'_x \\
\land (\text{LNode}(l''_x), \text{LClock}(l''_x)) \in \text{udep}(u)))
\]

The above lemma states that for every put label \( l_x \) that emits the update \( u \) and every put label \( l'_x \) that causally precedes \( l_x \), either the timestamp of \( l'_x \) is already in \( \text{udep}(u) \) or there exists a put label \( l''_x \) that depends on \( l'_x \) and the timestamp of \( l''_x \) is in \( \text{udep}(u) \).

The second invariant states that, if a put label \( l_x \) depends on another put label \( l'_x \) and some node has received the update for \( l_x \), then it has received the update for \( l'_x \) as well.

Lemma 8.
\[
\forall p, h_Z, W_Z, l_x, l'_x, n : \\
(W_{20}(p) \xrightarrow{h_Z} \chi_Z(l_x)) \land LIsPut(l_x) \land LIsPut(l'_x) \land l'_x \in h_Z l_x \\
\land LClock(l_x) \leq \text{rec'}(W_Z, n, \text{LNode}(l_x)) \Rightarrow \\
\text{LClock}(l'_x) \leq \text{rec'}(W_Z, n, \text{LNode}(l'_x))
\]

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 \( l_x \) and that \( l_x \) is causally dependent on another put label \( l'_x \). We want to show that the update of \( l'_x \) is already received. By Lemma 7, we have two cases. Case 1: The identifier of \( l'_x \) is directly in \( \text{udep}(u) \). The guard method checks that its update is already received. Case 2: The identifier of \( l'_x \) is indirectly in \( \text{udep}(u) \); that is, there exists another label \( l''_x \) that is causally dependent on \( l'_x \), and the timestamp of \( l''_x \) is in \( \text{udep}(u) \). As the timestamp of \( l''_x \) is in \( \text{udep}(u) \), from the guard method checks, we have that the update of \( l''_x \) is already received. As \( l''_x \) is causally dependent on \( l'_x \), and the update of \( l''_x \) is already received, by the induction hypothesis, we have that the update of \( l'_x \) is already received as well.

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

Lemma 9 (CauseCond).
\[
\forall p, h_Z, W_Z, l_x, W'_Z, l'_x : \\
(W_{20}(p) \xrightarrow{h_Z} \chi_Z(l_x)) \land LIsUpdate(l_x) \\
\land \text{let } \_\!, \_\!, \_\! \triangleright update(\_\!, \_\!, \_\!) \Rightarrow \_\!, \_\!, \_\! = l_x \\
\land LIsUpdate(l'_x) \land \text{LIsPut}(l''_x) \land l''_x \in h_Z l'_x \\
\land (\text{LNode}(l''_x), \text{LClock}(l''_x)) \in \text{udep}(u)) \\
\Rightarrow \\
\text{let } n', \_\! \triangleright put(\_\!, \_\!, \_\! : \_\!) \Rightarrow \_\!, \_\! = l'_x \text{ in } \\
\text{c''} \leq \text{rec'}(W_Z, n, n')
\]

If an update is being received that is originated by the put label \( l'_x \), and another put label \( l''_x \) causally precedes \( l'_x \), then the update of \( l''_x \) 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 \( l''_x \) is directly or indirectly in the dependencies of the update from \( l'_x \).

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).
\[ \forall p, h, W, Z, l, l', n \colon (W_{Z0}(p) \xrightarrow{h} W_{Z}(l), W \land l \wedge h \triangleright W_{Z}(l')) \Rightarrow (dep(LPostState(lZ), n) \leq dep(LPostState(lZ'), n) \land (LlsPut(lZ') \triangleq n = LNode(lZ')) \Rightarrow \text{dep}(LPostState(lZ), n) < \text{dep}(LPostState(lZ'), n)) \]

Lemma 11 (Dep not above Rec).
\[ \forall p, h, W, Z, l, l', n \colon \text{rec}(W, Z, n) \Rightarrow \text{dep}(W, Z, n) \leq \text{rec}(W, Z, n) \]

Let us define the function rec' as follows:
\[ \text{rec}'(W, n, n') \triangleq \lambda \text{let } H[n \mapsto (\ast, n), n] = W \text{ in } \text{rec}(n'(n)) \]

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

Lemma 12.
\[ \forall p, h, W, Z, l, l', n \colon \text{let } \omega, n \triangleright \text{update}(\omega, \omega, m); \omega = lZ \]
\[ (\omega, n) \triangleright lZ \Rightarrow n'' \triangleright \text{put}(\omega, n); \omega = lZ'' \text{ in } \]
\[ (W_{Z0}(p) \xrightarrow{h} W, lZ \wedge W \xrightarrow{h} W_{Z}(lZ), W_{Z}(lZ')) \Rightarrow n'' \leq \text{rec}'(W, lZ, n'') \]

The condition WellRec(1) defined in Figure 11 requires the definition of the function Rec for the algorithm 1. 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. WellRec(1)

From the above theorem and Theorem 2, we conclude that I3 is causally consistent. For more details and the proofs, please see our Coq development.

Corollary 3. CauseConst(I3)

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.

<table>
<thead>
<tr>
<th>I3 (Algorithm 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
</tr>
<tr>
<td>(store: Map[K, (V, N, C)], rec: Map[N, C], dep: Map[N, C])</td>
</tr>
<tr>
<td>Update</td>
</tr>
<tr>
<td>(unode: N, udep: Map[N, C])</td>
</tr>
<tr>
<td>init</td>
</tr>
<tr>
<td>ret (λk. (v0, n0, 0), λn0, λn0)</td>
</tr>
<tr>
<td>put (self, this)(k, v)</td>
</tr>
<tr>
<td>(s, r, d) ← this;</td>
</tr>
<tr>
<td>d' ← d[\text{self} \mapsto r[\text{self}] + 1];</td>
</tr>
<tr>
<td>r' ← r[\text{self} \mapsto r[\text{self}] + 1];</td>
</tr>
<tr>
<td>s' ← s[k \mapsto (v, (v, d'[\text{self}]))];</td>
</tr>
<tr>
<td>ret ((s', r', d'), (self, d'))</td>
</tr>
<tr>
<td>get (self, this)(k)</td>
</tr>
<tr>
<td>(s, r, d) ← this;</td>
</tr>
<tr>
<td>(v, n, c) ← s[k];</td>
</tr>
<tr>
<td>d' ← d[n \mapsto \max(d(n), c)];</td>
</tr>
<tr>
<td>ret (v, (s, r, d'))</td>
</tr>
<tr>
<td>guard (self, this)(k, v, u)</td>
</tr>
<tr>
<td>(s, r, d) ← this;</td>
</tr>
<tr>
<td>(a', d') ← u;</td>
</tr>
<tr>
<td>ret forall (λn. n \neq n' \Rightarrow d'[n] \leq r[n]) N</td>
</tr>
<tr>
<td>∧ d'[n] = r[n] + 1</td>
</tr>
<tr>
<td>update (self, this)(k, v, u)</td>
</tr>
<tr>
<td>(s, r, d) ← this;</td>
</tr>
<tr>
<td>(a', d') ← u;</td>
</tr>
<tr>
<td>r' ← r[n'] \mapsto d'[n'];</td>
</tr>
<tr>
<td>d' ← \lambda n. \max(d(n), d'[n]);</td>
</tr>
<tr>
<td>s' ← s[k \mapsto (v, n', d'[n'])];</td>
</tr>
<tr>
<td>ret ((s', r', d'))</td>
</tr>
</tbody>
</table>

Figure 17. Causally Consistent Map 3
D. Linked-List Client Example

Program 3 (p3): Linked list client

0 → put(2, null);
put(1, 3);
put(head, 1);
put(6, 1);
put(5, 2);
put(head, 5);
put(4, 5);
put(3, 1);
put(head, 3)
1 →
x₁ ← get(head);
if x₁ ≠ v₀ then
    i₁ ← get(x₁);
x₂ ← get(x₁ + 1);
if x₂ ≠ null then
    i₂ ← get(x₂);
x₃ ← get(x₂ + 1);
assert(i₁ < i₂);
if x₃ ≠ null then
    i₃ ← get(x₃);
x₄ ← get(x₃ + 1);
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 (v₀) 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.