


References


6 Discussion

As the demand for powerful computers grows faster than the technology to develop new processors, the need for highly parallel multiprocessors increases. However, in order to fully utilize highly parallel multiprocessors, convenient paradigms for sharing memory between processes must be developed. These paradigms should allow the user to enjoy the same simplistic model of the world as in uniprocessors, without sacrificing the performance of the whole system. Hybrid consistency is an attractive consistency condition that tries to bridge the gap between these two contradictory goals. The weak semantics provided by the definition of hybrid consistency for weak operations, allows to implement them efficiently. On the other hand, the strong semantics provided for strong operations makes hybrid consistency a convenient and expressive consistency condition to work with. Furthermore, recent results [8] indicate that it may be possible to automatically transform programs written for sequential consistency into correct programs for hybrid consistency.

Until now, the study of hybrid consistency has largely neglected the issues concerning high-level synchronization operations. As these operations simplify the process of writing concurrent programs, and since they increase the power of concurrent programming languages, they become more common in parallel computers. Thus, it is important to show that the definition of hybrid consistency allows efficient implementations that can execute these operations.

In this paper, we have shown lower bounds on the time required to execute some of the most common synchronization operations. We have presented two efficient implementations for hybrid consistency, that support high-level synchronization operations. The time complexity achieved by these implementations is within a constant factor of the lower bounds.

At the moment we are not aware of any formal verified distributed implementation for hybrid consistency that supports read-modify-write operations and allows a single object to be accessed by both weak reads and weak writes and by strong operations, and executes weak operations faster than the network delay. An interesting open problem is to find such an implementation or to prove that such an implementation is impossible.

Both the programming example in [9] and the result in [8], that shows how to transform any given non-cooperative solution for the mutual exclusion problem under sequential consistency into a correct and efficient solution under hybrid consistency, exploit the fact that weak read operations can access the same objects that are being accessed by strong operations. However, there is no similar use for weak write operations. Thus, an important question is whether it is possible to write “meaningful” programs that can benefit from being able to access the same objects by weak writes and strong operations.

Acknowledgements: I would like to thank Hagit Attiya for many helpful comments and discussions. Thanks also to Ken Birman, Martha Kosa and Gil Neiger.

---

7 One of the results in [8] assumes that weak write operations can access the same objects that are accessed by strong read operations. However, this result relies on several assumptions that make it impractical.
Theorem 5.14 Every execution generated by the algorithm is hybrid.

Proof: For an execution $\sigma$, we have constructed a set of sequences $\{\tau_j\}_{j=1}^n$. We now show why this set satisfies the requirements of the definition of hybrid consistency:

- By Lemma 5.13, every sequence $\tau_j$ is legal.
- By construction, every sequence $\tau_j$ is a permutation of $\text{ops}(\sigma)$, which satisfies Condition 1 in the definition of hybrid consistency.
- By Lemma 5.12, for every pair of operations $op_1^1$ and $op_1^2$ such that either $op_1^1$ or $op_1^2$ is strong, $op_1^1 \xrightarrow{\tau_j} op_1^2$ if and only if $op_1^1 \xrightarrow{\sigma} op_1^2$, which satisfies Condition 2 in the definition of hybrid consistency.
- By Lemma 5.6, there exists a linearization $\rho$ of all operations in $\text{ops}(\sigma)$ such that for every pair of strong operations $sop_i$ and $sop_k$, $sop_i \xrightarrow{\tau_j} sop_k$ if and only if $sop_i \xrightarrow{\rho} sop_k$, which satisfies Condition 3 in the definition of hybrid consistency.
- By Lemma 5.10, for every sequence $\tau_j$, $\tau_j|j = \sigma|j$, which satisfies Condition 4 in the definition of hybrid consistency.

5.3 Complexity Analysis

The exact complexity of the implementation depends on the complexity of the atomic broadcast mechanism. By using the same assumptions as in Section 4 regarding the time and message complexity of the atomic broadcast we get the following results:

Weak read operations are executed instantaneously and cause no messages to be sent. Weak write operations are executed instantaneously and cause $3n + 1$ physical messages.

Strong read and strong read-modify-write operations, may broadcast a dummy message, followed by a broadcast of the strong-read or strong-rmw message. Each of these messages, causes the receiver to send an ack message. Thus, strong read and strong read-modify-write operations cause $6n + 2$ physical messages.

Strong write operations broadcast only a strong-write message, causing $3n + 1$ physical messages.

Strong operations may have to wait for the ack messages of previously invoked weak writes or previously sent dummy messages. Then, they broadcast a strong-read, strong-write or strong-rmw message to every process and must wait for an acknowledge. Hence, a strong operation may have to wait for at most 4 rounds of messages. Thus, the time required to execute a strong operation is $8d$. 

39
5.11 and since the insertion of weak reads does not change the order of other operations in $\tau''_j$, the claim holds if neither $op^1_k$ nor $op^2_k$ is a weak read. Moreover, since weak reads are inserted after the previous strong operation by the same process, the only case in which $op^1_k$ may be ordered in $\tau''_j$ before $op^2_k$ is that $op^2_k$ is a weak read, $op^1_k$ is a strong operation, $k \neq j$, and the operation $op^2_k$ reads from is ordered after $op^1_k$ in $\tau''_j$.

Assume, without loss of generality, that $op^2_k$ reads from $wo_i(x, v)$. Since all (weak and strong) write operations and all strong read-modify-write operations are ordered before their corresponding delivery event, $wo_i(x, v) \rightarrow_{\tau''_j} del_j(wo_i(x, v))$. If $op^1_k$ is a strong write, then it is ordered in $\tau''_j$ immediately before $del_j(op^1_k)$. Thus, $del_j(op^1_k) \rightarrow_{\tau''_j} w(i, x, v) \rightarrow_{\tau''_j} del_j(wo_i(x, v))$. If, on the other hand, $op^1_k$ is a strong read or a strong read-modify-write operation, then by Lemma 5.9, $op^1_k$ is ordered in $\tau''_j$ after $del_j(\text{rel}(op^1_k))$. Thus, $del_j(\text{rel}(op^1_k)) \rightarrow_{\tau''_j} w(i, x, v) \rightarrow_{\tau''_j} del_j(wo_i(x, v))$. In either case, by the use of atomic broadcast, the update, strong-write or $\text{strong-rmw}$ message of $wo_i(x, v)$ is delivered at $p_k$ after the execution of $op^2_k$. A contradiction to the assumption that $op^2_k$ reads from $wo_i(x, v)$.

![Proof](https://via.placeholder.com/150)

**Lemma 5.13** Every read is legal in every sequence $\tau''_j$.

**Proof:** By Lemma 5.8, all strong read and strong read-modify-write operations are legal in all of the sequences $\tau''_j$. Thus, assume, by way of contradiction, that there exists a sequence $\tau''_j$ and a weak read $r_i(x, v)$ such that $r_i(x, v)$ is not legal in $\tau''_j$. By Lemma 5.1, $i \neq j$. Denote by $wo_i(x, v)$ the operation $r_i(x, v)$ reads from and by $sop_k$ the previous strong operation by $p_k$. Thus, by the rules for inserting weak reads, there exists an obliterating write $wo_k(x, u)$ such that $wo_i(x, v) \rightarrow_{\tau''_j} w_k(x, u) \rightarrow_{\tau''_j} sop_k$. If there are several such operations, then let $wo_k(x, u)$ be the last one of them in $\tau''_j$.

If $wo_i(x, v)$ is a weak write, then so is $wo_k(x, u)$. Hence, either the $\text{update}$ message of $wo_k(x, u)$ or the $\text{update}$ message of another write to $x$ by $p_i$ is delivered at $p_k$ after the execution of $wo_i(x, v)$ and before the delivery of all $\text{ack}$ messages for $sop_k$. Thus, either $wo_k(x, u)$ or another write to $x$ is executed in $p_k$ between the execution of $wo_i(x, v)$ and the invocation of $r_i(x, v)$. A contradiction to the assumption that $r_i(x, v)$ reads from $wo_i(x, v)$.

If, on the other hand, $wo_i(x, v)$ is either a strong write or a strong read-modify-write, then $x$ is not accessed by weak writes, and both $wo_i(x, v)$ and $wo_k(x, u)$ are ordered immediately before their corresponding $\text{strong-write}$ or $\text{strong-rmw}$ messages. Since $sop_k$ is ordered before $del_j(sop_k)$, it follows that $wo_k(x, u)$ is executed by $p_k$ after the execution of $wo_i(x, v)$ and before the execution of $sop_k$. Thus, $wo_k(x, u)$ is executed by $p_k$ between the execution of $wo_i(x, v)$ and the invocation of $r_i(x, v)$. A contradiction to the assumption that $r_i(x, v)$ reads from $wo_i(x, v)$.

Finally, from each process $p_j$, create a new sequence of operations $\tau_j$ by removing all delivery events from the sequence $\tau''_j$.
Proof: By Lemma 5.10, the claim holds for \( k = j \). Thus, we assume for the rest of the proof that \( k \neq j \). Assume, by way of contradiction, that for some sequence \( \tau_{j'}^1 \) and two operations \( op_1^k \) and \( op_2^k \) such that either \( op_1^k \) or \( op_2^k \) is strong, \( op_1^k \xrightarrow{\tau_{j'}^1} op_2^k \) but \( op_2^k \xrightarrow{\sigma} op_1^k \). By Lemma 5.5 and Lemma 5.6 and the construction of \( \tau_{j''}^i \), this can only happen in one of the following cases:

Case 1: \( op_1^k \) is either a strong read or a strong read-modify-write operation that accesses a weak write object and \( op_2^k \) is a weak write. By construction of \( \tau_{j'}^1 \), \( op_2^k \) is ordered before \( del_j(op_1^k) \). In particular, \( op_2^k \) is ordered in \( \tau_{j'}^1 \) before \( del_j(rel(op_1^k)) \). By Lemma 5.9, \( op_1^k \) is ordered in \( \tau_{j''}^1 \) after \( del_j(rel(op_1^k)) \). Thus, \( op_2^k \xrightarrow{\tau_{j'}^1} op_1^k \). A contradiction.

Case 2: \( op_2^k \) is either a strong read or a strong read-modify-write operation that accesses a weak write object and \( op_1^k \) is a weak write that does not influence \( p_i \). By definition, \( op_1^k \) is ordered before \( del_k(op_1^k) \) in every sequence \( \tau_{j''}^1 \). Assume, without loss of generality, that \( op_1^k \) writes to \( x \), and let \( w_i(x, w) \) be the last write that influences \( p_i \) before the delivery of the update message of \( op_1^k \). Note that by the code for executing weak writes, \( w_i(x, w) \) is not a strong read-modify-write operation. Thus, \( op_1^k \) is invoked before the delivery of the update or strong-write message of \( w_i(x, w) \) in \( p_i \). Moreover, \( op_1^k \) is ordered in \( \tau_{j'}^1 \) before \( op_2^k \) because \( w_i(x, v) \) is ordered in \( \tau_{j''}^1 \) before \( op_2^k \). Since \( op_2^k \) does not return before all of its ack messages are delivered, and since every ack message is sent using sendabc, the ack message for \( op_2^k \) is delivered in \( p_i \) after the update or strong-write message of \( w_i(x, w) \). Thus, \( op_1^k \) is invoked after the delivery of the update or strong-write message of \( w_i(x, w) \). A contradiction.

We now insert the weak reads. For each sequence \( \tau_{j''}^1 \), construct \( \tau_{j''}' \) by inserting into \( \tau_{j''}^1 \) all the weak reads by other processes, one after the other according to any arbitrary order. Each weak read \( r_k(x, v) \), \( k \neq j \), is inserted in the first possible place such that it will be ordered after the previous strong operation by \( p_k \) (if there is one) and after the operation it reads from (if there is one). Note that this is well defined: if there is no previous strong operation and the weak read returns the initial value of the object, then the weak read is inserted at the beginning of the sequence.

The next two lemmas show that after inserting the missing weak reads, the set of sequences \( \{\tau_j\}_j \) obeys the requirements in the definition of hybrid consistency, except for including the delivery events. The proofs of these lemmas are simpler than the corresponding proofs in [9]. This is due to the fact that, by the assumptions of the algorithm here, if a weak read reads from a weak write, then the object they access is never accessed by strong operations.

Lemma 5.12 For each sequence of operations \( \tau_{j''}^n \) and every pair of operations \( op_1^k \) and \( op_2^k \) such that either \( op_1^k \) or \( op_2^k \) is strong, \( op_1^k \xrightarrow{\tau_{j''}^n} op_2^k \) if and only if \( op_2^k \xrightarrow{\sigma} op_1^k \).

Proof: Assume, by way of contradiction, that there exists a sequence \( \tau_{j''}^n \) and two operations \( op_1^k \) and \( op_2^k \) such that either \( op_1^k \) or \( op_2^k \) is strong, \( op_1^k \xrightarrow{\tau_{j''}^n} op_2^k \) but \( op_2^k \xrightarrow{\sigma} op_1^k \). By Lemma
sop$_q$ is ordered before del$_j$(rel(sro$_i$(x, v))) in $\tau''_j$ and after del$_i$(rel(sro$_q$(x, v))) in $\tau'_i$, then sop$_q$ is either a strong read or a strong read-modify-write that accesses a weak write object and the operation sop$_q$ reads from is ordered inside the potential execution interval of $w_i(x, u)$. Denote the operation sop$_q$ reads from by $w_m$. By the minimality of sop$_q$, $w_m$ is not a strong read-modify-write operation. By Lemma 5.7, $w_m$ influences every process and is therefore ordered in $\tau''_i$ immediately before del$_j$(w$_m$). Thus, $w_m$ is ordered after del$_j$(rel(sro$_i$(x, v))) in $\tau''_j$. By Lemma 5.8, sop$_q$ is ordered after $w_m$ in $\tau''_j$, and therefore after del$_j$(rel(sro$_i$(x, v))). A contradiction.

Assume that sro$_i$(y, w) is dragged by sro$_q$(x, v) before del$_j$(rel(sro$_i$(y, w))) in $\tau''_j$. By the minimality of sro$_i$(x, v), sro$_i$(y, w) is ordered after del$_i$(rel(sro$_i$(y, w))) in every sequence $\tau''_i$ before the insertion of sro$_i$(x, v). Thus, del$_i$(sro$_i$(x, v)) and del$_i$(sro$_i$(y, w)) are ordered inside a potential execution interval of some weak write $w_i(x, u)$ that includes a strong operation sop$_q$ which is ordered in $\tau''_i$ before del$_j$(rel(sro$_i$(y, w))). If there is more than one such strong operation, then assume that sop$_q$ is the first one of them in $\tau''_i$. Since sop$_q$ is ordered before del$_j$(rel(sro$_i$(y, w))) in $\tau''_i$ and after del$_j$(rel(sro$_i$(y, w))) in $\tau''_i$, then sop$_q$ is either a strong read or a strong read-modify-write operation that access a weak write object and the write sop$_q$ reads from is ordered inside the potential execution interval of $w_i(x, u)$. Denote the operation sop$_q$ reads from by $w_m$. By the minimality of sop$_q$, $w_m$ is not a strong read-modify-write operation. By Lemma 5.7, $w_m$ influences every process and is therefore ordered in $\tau''_i$ immediately before del$_j$(w$_m$). Thus, $w_m$ is ordered after del$_j$(rel(sro$_i$(y, w))) in $\tau''_j$. By Lemma 5.8, sop$_q$ is ordered after $w_m$ in $\tau''_j$, and therefore after del$_j$(rel(sro$_i$(y, w))). A contradiction.

**Lemma 5.10** For every sequence $\tau''_j$, $\tau''_j \mid j = \sigma \mid j$.

**Proof:** Consider any two operations op$_i^1$ and op$_i^2$ such that op$_i^1 \longrightarrow op_i^2$. We show that op$_i^1 \tau'_i \longrightarrow op_i^2$ by a simple case analysis. If neither op$_i^1$ nor op$_i^2$ is neither a strong read nor a strong read-modify-write operation that accesses a weak write object, then the claim holds by Lemma 5.3 and since the insertion of strong read and strong read-modify-write operations that access weak write objects does not change the order of other operations. If both op$_i^1$ and op$_i^2$ are either a strong read or a strong read-modify-write operation that accesses a weak write object, then the claim holds by Lemma 5.6. If only op$_i^1$ is either a strong read or a strong read-modify-write operation that accesses a weak write object, then the claim holds since strong read and strong read-modify-write operations that access weak write objects are inserted before their corresponding delivery event. If, on the other hand, only op$_i^2$ is either a strong read or a strong read-modify-write operation that accesses a weak write object, then by Lemma 5.9, del$_j$(rel(op$_i^2$)) $\tau''_j \longrightarrow op_i^2$. By the construction of $\tau''_j$, op$_i^1 \longrightarrow del_j$(rel(op$_i^2$)). Thus, op$_i^1 \longrightarrow op_i^2$ and the claim holds.

**Lemma 5.11** For each sequence of operations $\tau''_j$ and every pair of operations op$_k^1$ and op$_k^2$ in $\tau''_j$ such that either op$_k^1$ or op$_k^2$ is strong, op$_k^1 \longrightarrow op_k^2$ if and only if op$_k^1 \longrightarrow op_k^2$. 

36
\[ \tau'' = \text{w}_{0k}(y, w) \ldots \text{w}_{i}(y, u) \ldots \text{w}_{m} \ldots \text{sop}_{q} \ldots \text{del}(\text{sro}_{i}(y, w)) \ldots \text{del}(\text{sro}_{i}(x, v)) \ldots \text{del}(\text{w}_{i}(y, u)) \]

\[ \tau'' = \text{sro}_{i}(y, w) \ldots \text{sro}_{i}(x, v) \ldots \text{sop}_{q} \ldots \text{w}_{0k}(y, w) \ldots \text{del}(\text{w}_{0k}(y, w)) \ldots \text{w}_{m} \ldots \text{del}(\text{w}_{0m}) \ldots \text{del}(\text{w}_{i}(y, u)). \]

Since \( \text{w}_{0k}(y, w) \xrightarrow{\tau''_i} \text{sop}_{q} \) and \( \text{sop}_{q} \xrightarrow{\tau''_i} \text{w}_{0k}(y, w) \) by Lemma 5.6, \( \text{w}_{0k}(y, w) \) is a weak write. Thus, by Lemma 5.7, \( \text{w}_{0k}(y, w) \) influences every process and is therefore ordered in every sequence \( \tau''_p \) immediately before \( \text{del}_p(\text{w}_{0k}(y, w)) \). Since \( \text{del}_p(\text{w}_{0k}(y, w)) \xrightarrow{\tau''_i} \text{sop}_{q} \) and \( \text{sop}_{q} \xrightarrow{\tau''_i} \text{del}_p(\text{w}_{0k}(y, w)) \), then \( \text{sop}_{q} \) is either a strong read or a strong read-modify-write operation that accesses a weak write object. Denote the operation \( \text{sop}_{q} \) reads from by \( \text{w}_{0m} \).

Thus, we have \( \text{w}_{i}(y, u) \xrightarrow{\tau''_i} \text{w}_{0m} \xrightarrow{\tau''_i} \text{sop}_{q} \). By the minimality of \( \text{sop}_{q} \) and Lemma 5.6, \( \text{w}_{0m} \) is a weak write. Hence, by Lemma 5.7 \( \text{w}_{0m} \) influences every process and is therefore ordered in every sequence \( \tau''_p \) immediately before \( \text{del}_p(\text{w}_{0m}) \). Thus, \( \text{sop}_{q} \xrightarrow{\tau''_i} \text{w}_{0k}(y, w) \xrightarrow{\tau''_i} \text{w}_{0m} \). A contradiction to the assumption that \( \text{sop}_{q} \) is ordered after the operation from which it reads (the assumption about the minimality of \( \text{sro}_{i}(x, v) \)).

**Lemma 5.9** Every strong read and every strong read-modify-write operation \( \text{sro}_{i}(x, v) \) is ordered after \( \text{del}_i(\text{rel}(\text{sro}_{i}(x, v))) \) in every sequence \( \tau''_i \).

**Proof:** It is clear from the construction of \( \{\tau''_j\}_{j=1}^n \) that the claim holds for strong read and strong read-modify-write operations that do not access weak write objects. We will show that the claim holds for strong read and strong read-modify-write operations that access weak write objects, too.

Assume, by way of contradiction, that there exists a strong read or a strong read-modify-write operation \( \text{sro}_{i}(x, v) \) that access a weak write object and is ordered in some sequence \( \tau''_i \) before \( \text{del}_i(\text{rel}(\text{sro}_{i}(x, v))) \). Assume, without loss of generality, that \( \text{sro}_{i}(x, v) \) is the first strong read or strong read-modify-write operation such that following the insertion of \( \text{sro}_{i}(x, v) \), there exists a strong read or a strong read-modify-write operation \( \text{sro}_{i}(y, w) \) which is ordered in some sequence \( \tau''_i \) before \( \text{del}_i(\text{rel}(\text{sro}_{i}(y, w))) \).

Thus, either \( \text{sro}_{i}(x, v) \) is ordered in \( \tau''_i \) before \( \text{del}_i(\text{rel}(\text{sro}_{i}(x, v))) \) (in which case \( \text{sro}_{i}(x, v) \) access a weak write object), or there exists another strong read or strong read-modify-write operation \( \text{sro}_{i}(y, w) \) which is dragged in \( \tau''_i \) before \( \text{del}_i(\text{rel}(\text{sro}_{i}(y, w))) \) (in which case \( \text{sro}_{i}(y, w) \) access a weak write object).

Assume that \( \text{sro}_{i}(x, v) \) is ordered in \( \tau''_i \) before \( \text{del}_i(\text{rel}(\text{sro}_{i}(x, v))) \). Thus, \( \text{del}_i(\text{rel}(\text{sro}_{i}(x, v))) \) is ordered in some sequence \( \tau''_i \) inside a potential execution interval of some weak write \( \text{w}_{i}(x, u) \) that includes a strong operation \( \text{sop}_{q} \) which is ordered in \( \tau''_i \) before \( \text{del}_i(\text{rel}(\text{sop}_{q})) \). If there is more than one such strong operation, then assume that \( \text{sop}_{q} \) is the first one of them in \( \tau''_i \). Since \( \text{sro}_{i}(x, v) \) does not start until all \text{ack} messages of \( \text{rel}(\text{sro}_{i}(x, v)) \) are delivered, \( \text{del}_i(\text{rel}(\text{sro}_{i}(x, v))) \) is ordered in \( \tau''_i \) before the potential execution interval of \( \text{w}_{i}(x, u) \). Since
Assume, by way of contradiction, that there exists a strong read or a strong read-modify-write operation that accesses a weak write object and is ordered in some sequence $\tau''_j$ before the operation from which it reads. Let $sro_i(x, v)$ be the first strong read or strong read-modify-write that access a weak write object such that following the insertion of $sro_i(x, v)$, there exists a strong read or a strong read-modify-write which access a weak write object and is ordered in some sequence $\tau''_j$ before the write from which it reads.

Thus, either $sro_i(x, v)$ is ordered in $\tau''_j$ before the operation from which it reads, or there exists another strong read or strong read-modify-write $sro_i(y, w)$ that access a weak write object and is dragged in $\tau''_j$ before the operation from which it reads.

Assume that $sro_i(x, v)$ is ordered in $\tau''_j$ before the operation from which it reads, and denote this operation by $wo_k(x, v)$. In particular, $sro_i(x, v)$ is ordered before $del_j(wo_k(x, v))$ in $\tau''_j$. Thus, there exists a sequence $\tau''_m$ such that $del_j(sro_i(x, v))$ is ordered in $\tau''_m$ inside the potential execution interval of some weak write $w_i(x, u)$ that includes a strong operation $sop_q$ which is ordered in $\tau''_m$ before $del_j(wo_k(x, v))$. If there are several such strong operations, then assume that $sop_q$ is the first one of them in $\tau''_m$. Thus, we have

$$\tau''_m = wo_k(x, v) \ldots w_i(x, u) \ldots wo_m \ldots sop_q \ldots del(sro_i(x, v)) \ldots del(w_i(x, u))$$

$$\tau''_j = sro_i(x, v) \ldots sop_q \ldots wo_k(x, v) \ldots del(wo_k(x, v)) \ldots wo_m \ldots del(wo_m) \ldots del(w_i(x, u)).$$

Since $wo_k(x, v) \xrightarrow{\tau''_m} sop_q$ and $sop_q \xrightarrow{\tau''_j} wo_k(x, v)$ by Lemma 5.6, $wo_k(x, v)$ is a weak write. Thus, by Lemma 5.7, $wo_k(x, v)$ influences every process and is therefore ordered in every sequence $\tau''_m$ immediately before $del_j(wo_k(x, v))$. Since $del_j(wo_k(x, v)) \xrightarrow{\tau''_m} sop_q$ and $sop_q \xrightarrow{\tau''_j} del_j(wo_k(x, v))$, then $sop_q$ is either a strong read or a strong read-modify-write operation that access a weak write object. Denote the operation $sop_q$ reads from by $wo_m$. Thus, we have $w_i(x, u) \xrightarrow{\tau''_m} wo_m \xrightarrow{\tau''_j} sop_q$. By the minimality of $sop_q$ and Lemma 5.6, $wo_m$ is a weak write. Hence, by Lemma 5.7 $wo_m$ influences every process and is therefore ordered in every sequence $\tau''_m$ immediately before $del_j(wo_m)$. Thus, $sop_q \xrightarrow{\tau''_j} wo_k(x, v) \xrightarrow{\tau''_m} wo_m$. A contradiction to the assumption that $sop_q$ is ordered after the operation from which it reads (the assumption about the minimality of $sro_i(x, v)$).

Assume that $sro_i(y, w)$ is dragged by $sro_i(x, v)$ before the operation from which it reads, and denote this operation by $wo_k(y, w)$. In particular, $sro_i(y, w)$ is dragged by $sro_i(x, v)$ before $del_j(sro_i(y, w))$ in $\tau''_m$. By the minimality of $sro_i(x, v)$, $sro_i(x, v)$ is ordered after $wo_k(y, w)$ in every sequence $\tau''_m$ before the insertion of $sro_i(x, v)$. Thus, there exists a sequence $\tau''_m$ such that both $del_i(sro_i(x, v))$ and $del_i(sro_i(y, w))$ are ordered in $\tau''_m$ inside the potential execution interval of some weak write $w_i(y, u)$ that includes a strong operation $sop_q$ which is ordered in $\tau''_m$ before $del_j(wo_k(y, w))$. If there are several such strong operations, then assume that $sop_q$ is the first one of them in $\tau''_m$. Thus, we have
Lemma 5.7 Every strong read and every strong read-modify-write operation that does not read from a read-modify-write operation, reads from a write that influences every process.

Proof: Let \( \text{src}_i \) be a strong read or a strong read-modify-write operation that reads from some write \( w_k \). If \( w_k \) is strong, then \( w_k \) influences every process by definition. Thus, assume that \( w_k \) is weak. If \( k = i \), then since \( \text{src}_k \) is a strong operation, it is not executed until the conditional execution interval of \( w_k \) is completed. Therefore, \( w_k \) is not overwritten and by Lemma 5.4, \( w_k \) influences every process. If \( k \neq i \), then \( w_k \) influences \( p_k \) and by Lemma 5.4, \( w_k \) influences every process.

Lemma 5.8 Every strong read and every strong read-modify-write is legal in every sequence \( \tau^\prime \).

Proof: By Lemma 5.1 and since the insertion of strong read and strong read-modify-write operations that access weak write objects does not change the order of other operations, every strong read or strong read-modify-write that do not access weak write objects is legal in every sequence \( \tau^\prime \). As for strong read and strong read-modify-write operations that access weak write objects, we prove that each of these operations is ordered in every sequence \( \tau^\prime \) after the operation it reads from. Since by definition, every strong read or strong read-modify-write operation that accesses a weak write object appears in every sequence \( \tau^\prime \) before any obliterating write operation, this will imply that the strong read or the strong read-modify-write operation is legal.
an extra care is needed in the proofs of these lemmas here, compared with the corresponding proofs in [9], due to the dual semantics (both read and write) of read-modify-write operations.

**Lemma 5.6** There exists a linearization $\rho$ of all the operations in $\sigma$ such that for every pair of strong operations $sop_k$ and $sop_l$, $sop_k \xrightarrow{\rho} sop_l$ if and only if $sop_k \xrightarrow{\tau_{i''}} sop_l$, for every $\tau_{i''}$.

**Proof:** We have to show that all the strong operations are ordered in all the sequences $\tau_{j''}$ in the same order, and that this order is a linearization of the strong operations in $\text{ops}(\sigma)$.

We first show that all strong operations appear in the same order in all sequences $\tau_{j''}$. Assume, by way of contradiction, that there exist two strong operations $sop_k$ and $sop_l$ and two sequences $\tau_{i''}$ and $\tau_{i''}$ such that $sop_k \xrightarrow{\tau_{i''}} sop_l$ but $sop_k \xrightarrow{\tau_{i''}} sop_l$. By Lemma 5.2 and since the insertion of strong read and strong read-modify-write operations that access weak write objects do not change the order of other operations, this can only happen if at least one of $sop_k$ or $sop_l$ is either a strong read or a strong read-modify-write operation that accesses a weak write object.

If both $sop_k$ and $sop_l$ are either a strong read or a strong read-modify-write operation that accesses weak write objects, then let $sop_k = sro_i(x, v)$ and $sop_l = sro_i(y, w)$. Assume, without loss of generality, that $sro_i(x, v)$ appears before $sro_i(y, w)$ in the order of insertion. Thus, the strong-read or strong-rmw message of $sro_i(x, v)$ is delivered after the strong-read or strong-rmw message of $sro_i(y, w)$. Since we assumed that $sro_i(y, w) \xrightarrow{\tau_{i''}} sro_i(x, v)$, then either $sro_i(x, v) \in D(sro_i(y, w))$ or $sro_i(y, w)$ is dragged by another strong read before $sro_i(x, v)$. In either case, $sro_i(y, w)$ is ordered before $sro_i(x, v)$ in every sequence $\tau_{i''}$. A contradiction.

Otherwise, without loss of generality, $sop_k$ is either a strong read or a strong read-modify-write that accesses a weak write object but $sop_k$ is not. Let $sop_k = sro_i(x, v)$. If the strong-read or strong-rmw message of $sro_i(x, v)$ is delivered before the strong-write, strong-read or strong-rmw message of $sop_k$, then $sro_i(x, v)$ is ordered before $sop_k$ in every sequence $\tau_{i''}$. Thus, the strong-read or strong-rmw message of $sro_i(x, v)$ is delivered after the strong-write, strong-read or strong-rmw message of $sop_k$. Since $sro_i(x, v)$ is ordered before $sop_k$ in $\tau_{i''}$, then either $sop_k \in D(sro_i(x, v))$ or $sro_i(x, v)$ is dragged by another strong read or strong read-modify-write before $sop_k$. In any case, $sro_i(x, v)$ is ordered before $sop_k$ in every sequence $\tau_{i''}$. A contradiction.

We now show that the order in which strong operations appear in $\tau_{i''}$ preserves the order implied by $\rho$. Assume, by way of contradiction, that the order in which all strong operations appear in every sequence $\tau_{i''}$ is not a linearization. Note that strong read and strong read-modify-write operations are always inserted before their corresponding delivery event. By Lemma 5.2 and since the insertion of strong read and strong read-modify-write operations that access weak write objects does not change the order of other operations, the total order is not a linearization only if there exists a strong read or a strong read-modify-write $sro_i(x, v)$ that accesses a weak write object and another strong operation $sop_k$ such that $sop_k$ terminates

32
\[ \tau'' = \ldots sop_1 \ldots sop_k (y, u) \ldots del_i (sro_l (x, u)) \]

then \( sro_i (x, v) \) must be ordered before \( w_j (x, v) \) and \( sop_q \) in \( \tau'' \). Moreover, \( sro_i (x, v) \) must be ordered before \( sop_q \) in \( \tau'' \), too. Also, if \( sro_k (y, u) \) is a strong read or a strong read-modify-write that is dragged by \( sro_i (x, v) \) (we describe the rules for that later), then \( sro_k (y, u) \) must be ordered before \( sro_i (x, v) \) and \( sop_q \) both in \( \tau'' \) and \( \tau'' \).

Formally, to define the order of strong read and strong read-modify-write operations that access weak write objects, we introduce some definitions. For each such strong read or strong read-modify-write operation \( sro_i (x, v) \), and for each process \( p_j \), if the strong-read or strong-rmw message of \( sro_i (x, v) \) is delivered inside the potential execution interval of some weak write \( w_j (x, u) \), then let \( I_j (w_j (x, u), sro_i (x, v)) \) be the set of strong operations already ordered in \( \tau' \) between the invocation of \( w_j (x, u) \) and the delivery of the strong-read or strong-rmw message of \( sro_i (x, v) \). Define \( B_j (w_j (x, u), sro_i (x, v)) \) to be the set of strong read and strong read-modify-write operations \( \{ sro_q \} \) such that \( sro_q \in I_j (w_j (x, u), sro_i (x, v)) \) and the operation \( sro_q \) reads from is included in the potential execution interval of \( w_j (x, u) \). Let \( I_j (sro_i (x, v)) \) be the union of \( \{ I_j (w_j (x, u), sro_i (x, v)) \} \) for all such \( w_j (x, u) \) and \( B_j (sro_i (x, v)) \) be the union of the sets \( B_j (w_j (x, u), sro_i (x, v)) \) over all such \( w_j (x, u) \). Let \( B (sro_i (x, v)) = \bigcup_{j=1}^n B_j (sro_i (x, v)) \), \( I (sro_i (x, v)) = \bigcup_{j=1}^n I_j (sro_i (x, v)) \), let \( C (sro_i (x, v)) \) be the set of strong read and strong read-modify-write operations that access weak write objects and appear in \( I (sro_i (x, v)) \setminus B (sro_i (x, v)) \) and let \( D (sro_i (x, v)) = I (sro_i (x, v)) \setminus C (sro_i (x, v)) \).

Intuitively, \( I (sro_i (x, v)) \) is the set of strong operations that \( sop_k \) might be inserted before, although \( del_i (sro_i (x, v)) \) is ordered after them in some sequence \( \tau' \). \( D (sro_i (x, v)) \) is the set of strong operations that \( sro_i (x, v) \) is actually inserted before, although \( del_i (sro_i (x, v)) \) is ordered after them in some sequence \( \tau' \). \( C (sro_i (x, v)) \) is the set of strong read and strong read-modify-write operations that are dragged by \( sro_i (x, v) \) while \( B (sro_i (x, v)) \) is the set of strong read and strong read-modify-write operations that appear in \( I (sro_i (x, v)) \) and are not dragged by \( sro_i (x, v) \). \( B (sro_i (x, v)) \) is merely used to define \( C (sro_i (x, v)) \). For example, if we have

\[ \tau'' = \ldots w_j (x, v), sw_q, sro_k (y, u), w_z (z, w), sro_l (z, w), del_i (sro_i (x, u)), del_i (w_j (x, v)), \ldots \]

and this is the only potential execution interval of a weak write to \( x \) that includes the delivery event of \( sro_i (x, v) \), then \( I (sro_i (x, v)) = \{ sw_q, sro_k (y, u), sro_l (z, w) \} \), \( B (sro_i (x, v)) = \{ sro_l (z, w) \} \), \( C (sro_i (x, u)) = \{ sro_k (y, u) \} \) and \( D (sro_i (x, u)) = \{ sw_q, sro_l (z, w) \} \).

Add \( sro_i (x, v) \) to each \( \tau'' \) in the last possible place such that it will be ordered before every strong operation in \( D (sro_i (x, v)) \), before the delivery event of its strong-read or strong-rmw message and before any obliterating write operation. For every strong read or strong read-modify-write operation \( sro_q \) in \( C (sro_i (x, v)) \), if \( sro_q (x, v) \) is ordered before \( sro_i (x, v) \) in \( \tau'' \), then reorder \( sro_q \) immediately before \( sro_i (x, v) \) in \( \tau'' \). We say that \( sro_q \) is dragged by \( sro_i (x, v) \) in \( \tau'' \). If there is more than one strong read or strong read-modify-write operation that is dragged by \( sro_i (x, v) \), then break ties according to the abcast order of their corresponding strong-read and strong-rmw messages.

The next six lemmas prove that the set of sequences \( \{ \tau'' \}_{j=1}^n \) obeys the requirements in the definition of hybrid consistency, except for not including all operations in \( \sigma \). Note that
every process before $\textit{op}_i^2$. In particular, $w_i$ is delivered at $p_k$ before $\textit{op}_i^2$. Since $\textit{op}_i^2$ is a strong operation and since $\textit{op}_i^1$ is invoked after $\textit{op}_i^2$, $w_i$ is delivered at $p_k$ before $\textit{op}_i^1$ starts. Therefore, there must exist another write $w_i(x, w)$ that influences $p_j$ and is delivered at $p_j$ between $w_i$ and $\textit{op}_i^1$. Thus, $w_i(x, w)$ is ordered in $\tau_j^i$ between $w_i$ and $\textit{op}_i^1$. A contradiction to the assumption that $w_i$ is the first write to $x$ that influence $p_j$ and is ordered after $\textit{op}_i^1$ in $\tau_j^i$. If $i = j$ and $\textit{op}_i^1$ is delivered inside the conditional execution interval of $w_i$, then the $\textit{ack}$ message of $\textit{op}_i^1$ is delivered at $p_k$ after the $\text{update}$ message of $w_i$. By Lemma 5.4, $w_i$ influences $p_k$ and is delivered in $p_k$ before $\textit{op}_i^1$ starts. Therefore, there must exist another write $w_i(x, w)$ that influences $p_j$ and is delivered at $p_j$ between $w_i$ and $\textit{op}_i^1$. Thus, $w_i(x, w)$ is ordered in $\tau_j^i$ between $w_i$ and $\textit{op}_i^1$. A contradiction to the assumption that $w_i$ is the first write to $x$ that influence $p_j$ and is ordered after $\textit{op}_i^1$ in $\tau_j^i$. 

Thus, we have shown that the sequences $\{\tau_j^i\}_{i=1}^n$ obey all the requirements in the definition of hybrid consistency, except for including all the operations of $\text{ops}(\sigma)$. To finish the proof, we must insert the missing operations, i.e., the missing read and read-modify-write operations, to each $\tau_j^i$ without violating the requirements of the definition of hybrid consistency.

We start with the missing strong read and strong read-modify-write operations. For each $j$, create $\tau_j^{\prime\prime}$ by inserting the missing strong read and strong read-modify-write operations into $\tau_j^i$. Insert the operations according to the abcast order of their strong-read and strong-rmw messages. Unlike the other strong operations (i.e., strong writes and strong read or strong read-modify-write operations that do not access weak write objects), sometimes it is not possible to insert a strong read or a strong read-modify-write operation that accesses a weak write object immediately before its corresponding delivery event. This is because the strong-read or strong-rmw message can be delivered at another process inside a potential execution interval of a weak write to the same object. If this happens, then the last value written to the object is the value written by the local weak write, and not the value returned by the strong read or strong read-modify-write. Therefore, in order to maintain the legality of the sequence, the strong read or strong read-modify-write has to be inserted before the weak write. However, since hybrid consistency requires that the views of all processes agree on the order of strong operations, if a strong read or strong read-modify-write $\textit{sro}_i(x, v)$ is inserted in one sequence before another strong operation $\textit{sop}_k$, then $\textit{sro}_i(x, v)$ must be ordered before $\textit{sop}_k$ in every sequence $\tau_j^i$.

On the other hand, if $\textit{sop}_k$ is a strong read or a strong read-modify-write that is inserted long before its corresponding strong-read or strong-rmw message, then inserting $\textit{sro}_i(x, v)$ before $\textit{sop}_k$ could cause $\textit{sro}_i(x, v)$ to be ordered before a previous weak operation by $p_i$. In order to prevent such chain reactions, it is sometimes required to drag a previously inserted strong read or strong read-modify-write $\textit{sro}_i(y, u)$ before $\textit{sro}_i(x, v)$ instead of inserting $\textit{sro}_i(x, v)$ before $\textit{sro}_i(y, u)$.

For example, if we have:

$$
\tau_j^{\prime\prime} = \ldots w_j(x, v) \ldots \textit{sop}_i \ldots \textit{sro}_i(y, u) \ldots \text{del}_i(\textit{sro}_i(x, u)) \ldots \text{del}_i(w_j(x, v)) \quad \text{and}
$$
corresponding update message is delivered at every process. Moreover, this value is different for different weak writes.

Assume, by way of contradiction, that there exists some write that influences some processes but not all of them; let \( w_k(x, v) \) be the first such write, i.e., the one for which the value of \( lts \) during the delivery event of its update message is minimum. Assume that \( w_k(x, v) \) does not influence \( p_i \). Thus, \( w_k(x, v) \) is not executed by \( p_i \). The code of the algorithm for executing weak writes imply that there exists another write \( w_i(x, u) \) for which the following holds: (a) \( w_i(x, u) \) influences \( p_i \), (b) the update message of \( w_i(x, u) \) is delivered at \( p_i \) before the update message of \( w_k(x, v) \), and (c) the update message of \( w_i(x, u) \) is delivered at \( p_k \) after \( w_k(x, v) \) is invoked. (This is the condition which is checked whenever an update message is delivered.) Since \( w_k(x, v) \) is the first write that influences only some of the processes, \( w_i(x, u) \) influences \( p_k \) and therefore \( w_k(x, v) \) is overwritten.

Recall that atomic broadcast delivers all messages to all processes in the same order. Since \( w_k(x, v) \) is the first write that influences only some of the processes, \( w_i(x, u) \) influences all processes and its update or strong-write message is delivered before the update message of \( w_k(x, v) \) to all processes. Thus, by the assumptions about \( w_i(x, u) \), none of the processes decides to execute \( w_k(x, v) \). Hence, \( w_k(x, v) \) does not influence any process. This is a contradiction to the assumption that \( w_k(x, v) \) influences some processes.

\[ \text{Lemma 5.5} \quad \text{For each sequence of operations } \tau_i \text{ and every pair of operations } op_i^1 \text{ and } op_i^2 \text{ in } \tau_i \text{ such that either } op_i^1 \text{ or } op_i^2 \text{ is strong, } op_i^1 \overset{\tau_i}{\rightarrow} op_i^2 \text{ if and only if } op_i^1 \overset{\sigma}{\rightarrow} op_i^2. \]

**Proof:** By Lemma 5.3, the claim holds if \( k = j \). Thus, for the rest of the proof, we may assume that \( k \neq j \).

Assume, by way of contradiction, that for some sequence \( \tau_i \) and two operations \( op_i^1 \) and \( op_i^2 \) such that either \( op_i^1 \) or \( op_i^2 \) is strong, \( op_i^1 \overset{\tau_i}{\rightarrow} op_i^2 \) but \( op_i^1 \overset{\sigma}{\rightarrow} op_i^1 \). We claim that \( op_i^1 \) is a weak write that does not influence \( p_i \) and \( op_i^2 \) is strong. Strong operations and weak writes by other processes that influence \( p_j \) are executed according to the abcast order. This order is consistent with the order by which operations are invoked at each process. Therefore, the only operations that may not be ordered correctly in \( \tau_i \) with respect to the previous and next strong operations by the same process are weak writes that do not influence \( p_i \). Moreover, since strong operations are ordered in \( \tau_i \) immediately before their corresponding delivery events, and since weak writes are always ordered before their corresponding delivery event, a weak write that does not influence \( p_i \) may not be ordered after a later strong operation by the same process.

Assume, without loss of generality, that \( op_i^1 \) writes to \( x \). Let \( w_i \) be the first write to \( x \) that influences \( p_i \) and is ordered after \( op_i^1 \) in \( \tau_i \). The existence of such a write is guaranteed by the assumption that \( op_i^1 \) does not influence \( p_i \) and the definition of \( \tau_i \). The code of the algorithm also implies that \( w_i \) is delivered at \( p_i \) before \( op_i^2 \).

If \( i \neq j \), or if \( i = j \) and \( op_i^2 \) is delivered after the completion of the conditional execution interval of \( w_i \) in \( p_j \), then by Lemma 5.4, \( w_i \) influences every process. Thus, \( w_i \) is delivered at
two operations \( op^1_k \) and \( op^2_l \), \( del_j(op^1_k) \xrightarrow{\tau_j^i} del_j(op^2_l) \) if and only if \( del_i(op^1_k) \xrightarrow{\tau_i^j} del_i(op^2_l) \), for every \( i \) and \( j \).

The next five lemmas prove that the set of sequences \( \{\tau'_j\}^n_{j=1} \) obeys the requirements in the definition of hybrid consistency, except for not including all operations. These lemmas and their proofs are (almost) the same as in the proof of RWHC [9].

**Lemma 5.1** Every sequence \( \tau'_j \) is legal.

**Proof:** Recall that the sequences \( \{\tau'_j\}^n_{j=1} \) include only strong read and strong read-modify-write operations that do not access weak write objects. Moreover, the only weak reads that are included in each \( \tau'_j \) are weak reads by \( p_j \), and they are ordered according to the order in which they occur in \( p_j \). Thus, every weak read, strong read or strong read-modify-write operation is ordered in each sequence after the write it reads from and before any obliterating write. ■

**Lemma 5.2** There exists a linearization \( \rho \) of all the operations in \( \sigma \) such that for every pair of strong operations \( sop_k \) and \( sop_l \) in \( \tau'_j \), \( sop_k \xrightarrow{\rho} sop_l \) if and only if \( sop_k \xrightarrow{\tau'_j} sop_l \), for every \( \tau'_j \).

**Proof:** The strong operations appear in every \( \tau'_j \) in an order which is consistent with the abcast order. Since a strong operation does not return before all its ack messages return, the real time of its delivery is always between the real time of its invocation and the real time of its termination. Thus, the order in which all strong operations appear in every \( \tau'_j \) is a linearization. ■

**Lemma 5.3** For every sequence \( \tau'_j \), \( \tau'_j[j] = \sigma[j] \).

**Proof:** Atomic broadcast delivers all messages sent by the same process in the order they were sent. Also, strong operations do not return until all ack messages are delivered. Thus, since weak operations are ordered according to their invocation and strong operations are ordered according to their delivery, for every sequence \( \tau'_j \), \( \tau'_j[j] = \sigma[j] \). ■

The following lemma is crucial in the rest of the proof.

**Lemma 5.4** If a write influences one process, then it influences every other process.

**Proof:** Recall that all messages are delivered to every process in the abcast order. Furthermore, the variable \( lts \) is incremented each time a message that was sent using abc is delivered by the atomic broadcast. Thus, for every weak write, the value of \( lts \) is the same when the
Potential Execution Interval
\[ \text{T} \]
Conditional Execution Interval
\[ \text{T} \]

\[ \sigma \]

\[ \text{inv}(w_i(x, v)) \]
\[ \text{del}(w_k(x, v)) \]
\[ \text{del}(w_i(x, v)) \]

Figure 8: Conditional/potential execution intervals

\( j \neq k \), if it updates the copy of \( x \) in \( p_j \). A weak write \( w_k(x, v) \) influences process \( p_k \) if it is not overwritten. A strong write influences every process. Intuitively, a weak write influences its invoking process if it is not overwritten, and influences another process if it is executed by that process. Strong writes are never overwritten and are executed by every process.

Next, define the notion of an operation being executed by a process as follows:

- A strong operation \( sop_k \) is executed by \( p_j \) when the corresponding strong-write, strong-read or strong-rmw message is delivered at \( p_j \).
- A weak operation \( op_j \) is executed by \( p_j \) when the appropriate call event occurs.
- A weak write \( w_k \) that influences \( p_j \), \( k \neq j \), is executed by \( p_j \) when the corresponding update message is delivered at \( p_j \).

Define the broadcast order to be the order by which all messages sent with \textbf{abc} are delivered. Note that this order is unique.

We say that an object which is accessed by weak write operations is a weak write object.

We now turn to the details of the proof; for the rest of this section, fix some execution \( \sigma \) of the algorithm.

For each process \( p_j \), create the sequence of operations and delivery events \( \tau_j' \) as follows: Order all delivery events in \( p_j \) according to the order they occur in \( p_j \). Next, add all weak operations by \( p_j \) according to their order of invocation in \( p_j \) (with respect to themselves and to the delivery events in \( p_j \)). Next, add all strong writes, all strong read and read-modify-write operations that do not access weak write objects and all weak writes that are executed by \( p_j \) and are invoked by other processes immediately before the delivery events of their corresponding strong-write, strong-read, strong-rmw or update messages. Finally, add every weak write \( w_k \) that is not executed by \( p_j \) immediately before the last write to the same location that is executed in \( p_j \) before \( w_k \), breaking further ties in any arbitrary order. Note that if \( w_k \) is not executed, then the code of the algorithm guarantees the existence of such a write. Note that all delivery events are ordered in the same order in all the sequences \( \tau_j' \). That is, for every
write objects already at this stage. The creation of \( \{j_i^{\mu}\}_{i=1}^n \) is also similar. However, here we have to use extra care in the construction in order to handle the read-modify-write operations. The creation of \( \{j_i^{\mu}\}_{i=1}^n \) is, again, similar and the deletion of the delivery events is, clearly, the same.

We now turn to the formal proof, and start with several definitions and some notation, some of which are generalizations of the ones that appear in Section 4. For every operation that causes a message to be broadcast, i.e., weak write, strong write, strong read or strong read-modify-write, we denote by \( \text{del}_j(op) \) the delivery event of the corresponding message (update, strong-write, strong-read or strong-rmw, accordingly) in \( p_j \). Given a strong read or a strong read-modify-write operation \( sro_i \), we denote by \( \text{rel}(sro_i) \) the last (strong-read, strong-write, strong-rmw, update or dummy) message broadcast by \( p_i \) before the strong-read or strong-rmw message of \( sro_i \). We slightly abuse the notation and denote by \( \text{del}_j(\text{rel}(sro_i)) \) the delivery event of \( \text{rel}(sro_i) \) in \( p_j \).

Given two operations \( wo_i^1(x, v) \) and \( re_i^2(x, v) \), we say that \( re_i^2 \) reads from \( wo_i^1 \) if \( wo_i^1(x, v) \) is the last operation that updates \( p_i \)'s copy of \( x \) before \( re_i^2(x, v) \) reads \( x \). We say that a strong read or a strong read-modify-write operation reads from \( x \) when it copies the value of \( x \) into \( val \). We say that a read or a read-modify-write operation \( re_i(x, v) \) is legal in a sequence of operations \( \tau \) if either there exists a write or a read-modify-write operation \( wo_i(x, v) \) such that \( wo_i(x, v) \xrightarrow{\tau} ri(x, v) \) and there does not exist another write or read-modify-write operation \( wo_k(x, u) \) such that \( wo_k(x, u) \xrightarrow{\tau} wo_k(x, u) \xrightarrow{\tau} ri(x, v) \), or \( v \) is the initial value of \( x \) and there does not exist a write or a read-modify-write operation \( wo_i(x, u) \) such that \( wo_i(x, u) \xrightarrow{\tau} ri(x, v) \); otherwise, the read or read-modify-write is illegal in \( \tau \).

Given two operations \( re_i^1(x, v) \) and \( wo_k^2(x, v) \) such that \( re_i^1(x, v) \) reads from \( wo_k^2(x, v) \) and a sequence of operation \( \tau \), we say that \( wo_k^2(x, v) \) is an obliterator write for \( re_i^1(x, v) \) in \( \tau \) if \( wo_k^2(x, v) \xrightarrow{\tau} wo_k^2(x, u) \).

A conditional execution interval of a weak write \( w_k \) is the interval of events in \( \sigma \) between the invocation of \( w_k \) and the delivery of the update message of \( w_k \) by the atomic broadcast at \( p_k \). A weak write \( w_k(x, v) \) is overwritten by another (weak or strong) write \( w_j(x, u) \) if \( w_j(x, u) \) is executed in \( p_j \) during the conditional execution interval of \( w_k(x, v) \). A potential execution interval of a weak write \( w_k(x, v) \) is the largest interval inside the conditional execution interval of \( w_k \), starting with the invocation of \( w_k(x, v) \), in which no other write \( w_j(x, u) \), \( j \neq k \), is executed by \( p_k \). Note, that a weak write must update the local copy of the object and return immediately. Thus, a potential execution interval captures the interval in which a weak write that updates the local copy of the object may still be overwritten. For example, Figure 8 demonstrate the potential and conditional execution intervals of a weak write \( w_i(x, v) \) that is overwritten by another write \( w_k(x, u) \).

Define the notion of influence as follows. A weak write \( w_k(x, v) \) influences process \( p_j \),

\footnote{The definition of a conditional execution interval is the same as the definition of a partial execution interval in Section 4. However, we have chosen to use two different terms since the role of each term is different in each of the algorithms.}
received <dummy> from \( p_i \):
\[
\text{send}_\text{abc} < \text{ack}, j > \\
lts := lts + 1
\]

received <ack, j> from \( p_i \):
\[
\text{missingacks} := \text{missingacks} - 1 \\
\text{if } (\text{missingacks} = 0) \text{ and there is a pending strong operation then} \\
\text{if the strong operation is a write then} \\
\begin{align*}
generate \text{SAck}(x) \\
\end{align*}
\text{else if it is a read then} \\
\begin{align*}
generate \text{SReturn}(x, val) \\
\end{align*}
\text{else} \\
\begin{align*}
generate \text{Return\&Ack}(x, val) \\
\end{align*}
\text{endif}
\text{endif}
\]

Figure 7: The mixed algorithm — continuation of code for process \( p_i \).

Given an execution \( \sigma \), we explicitly show how to build the set of sequences \( T = \{\tau_j\}_{j=1}^n \), as required in the definition of hybrid consistency.

Very informally, the construction goes as follows. For each process \( p_i \), we first build the sequence \( \tau_j \), consisting of all delivery events and all operations except for weak reads by other processes and strong read and read-modify-write operations that access objects which are accessed by weak writes. The operations are ordered according to the order they occur in \( p_i \). The delivery events serve as markers which are useful for the rest of the proof. They are ordered according to the order they occur, such that each delivery event is ordered after the corresponding operation. We show that \( T' = \{\tau_j\}_{j=1}^n \) obeys all the requirements in the definition of hybrid consistency except for not including all operations in \( \text{ops}(\sigma) \). Following this, we insert the missing strong read and strong read-modify-write operations into \( T' \), creating a new set of sequences \( T'' = \{\tau''_j\}_{j=1}^n \), such that all reads and all read-modify-write operations are legal and \( T'' \) obeys all the requirements in the definition of hybrid consistency, except for not including all operations in \( \text{ops}(\sigma) \). Next, for each sequence \( \tau_j'' \), we insert all weak reads by other processes, creating a new set of sequences \( T''' = \{\tau'''_j\}_{j=1}^n \) such that all reads in each \( \tau'''_j \) are legal and \( T''' \) obeys all the requirements in the definition of hybrid consistency (except for having the delivery events). Finally, we construct the set of sequences \( T \) by removing all delivery events from \( T''' \).

Note that the construction of the set of sequences \( \{\tau_j\}_{j=1}^n \) is similar to what is done in the proof of RWHC [9]. In particular, the creation of \( \{\tau_j\}_{j=1}^n \), is almost the same, except that here we can insert strong read and strong read-modify-write operations that do not access weak
received < update, x, v, s > from p_j:
    send_abc < ack, j >
    lts := lts + 1
    if ((s >= last_mess[x]) or (last_id[x] = j)) then
        last_mess[x] := lts
        last_id[x] := j
    if i ≠ j then
        mem[x] := v
        weak_overwritten[x] := weak_started[x]
    endif
    endif
    if (i = j) then
        weak_ended[x] := weak_ended[x] + 1
    endif

received < strong-write, x, v > from p_j:
    send_abc < ack, j >
    lts := lts + 1
    last_mess[x] := lts
    mem[x] := v
    last_id[x] := j
    weak_overwritten[x] := weak_started[x]

received < strong-read, x > from p_j:
    send_abc < ack, j >
    lts := lts + 1
    if (j = i) then val := mem[x] endif

received < strong-rmw, x, f > from p_j:
    send_abc < ack, j >
    lts := lts + 1
    if (weak_started[x] = weak_ended[x]) or (weak_started[x] = weak_overwritten[x]) then
        if (j = i) then
            val := mem[x]
        endif
        mem[x] := f(mem[x])
    endif

Figure 6: The mixed algorithm — continuation of code for process p_i.
Read($x$):
  generate Return($x, \text{mem}[x])$

Write($x,v$):
  $\text{mem}[x] := v$
  $\text{weak}_{\text{started}}[x] := \text{weak}_{\text{started}}[x] + 1$
  abc < update, $x,v, lls$
  missing_{acks} := missing_{acks} + n
  generate Ack($x$)

SRead($x$):
  if the last previous operation by $p_i$ was a weak read then
    abc < dummy
    missing_{acks} := missing_{acks} + n
  endif
  wait until missing_{acks} = 0 /* This line is not atomic */
  abc < strong-read, $x$
  missing_{acks} := missing_{acks} + n

SWrite($x,v$):
  wait until missing_{acks} = 0 /* This line is not atomic */
  abc < strong-write, $x,v$
  missing_{acks} := missing_{acks} + n

RMW($x,f$):
  if the last previous operation by $p_i$ was a weak read then
    abc < dummy
    missing_{acks} := missing_{acks} + n
  endif
  wait until missing_{acks} = 0 /* This line is not atomic */
  abc < strong-rmw, $x,f$
  missing_{acks} := missing_{acks} + n

Figure 5: The mixed algorithm — code for process $p_i$.  

---
mem: copy of every object, each initially equal to its initial value
last_msg: array of integers with one entry for each object, all initially 0
last_id: array of ids with one entry for each object, all initially 0
weak_started: array of integers with one entry for each object, all initially 0
weak_end: array of integers with one entry for each object, all initially 0
weak_overwritten: array of integers with one entry for each object, all initially 0
lts: integer, initially 0 (serial number of the last message delivered)
missing_acks: integer, initially 0 (counter for acknowledgements)
val: an object

Figure 4: The mixed algorithm — state of pi.

The mixed algorithm consists of a sequence of operations: 

strong-rmw message is delivered between the invocation of a weak write to the same object and the delivery of its update message, is not executed, unless the weak write is already overwritten by another write. The read-modify-write is executed by applying its function to the value of the local copy of the object, and updating the object with the result. The invoking process should also copy the value of the object into val before changing its value. The contents of val is returned when the strong read-modify-write operation returns.

After executing a strong operation, an ack message is sent back to the initiator of the operation. A strong operation does not return until all ack messages have been received.

It is necessary to broadcast a dummy message between the invocation of a weak read and the invocation of a strong read or a strong read-modify-write because the correctness of the algorithm depends on the fact that the invocation of a strong read or a strong read-modify-write is separated by at least a certain amount of logical time from the invocation of the last previous weak operation by the same process. This is due to the reordering of strong read and strong read-modify-write operations, which is done in the proof of correctness, with respect to the delivery events of their corresponding strong-read and strong-rmw messages. Note that there is no need to send a dummy message after a weak write since weak writes broadcast an update message; this already guarantees that there is enough logical time between the invocation of the weak write and the strong operation.

The formal description of the code appears in Figure 4, Figure 5, Figure 6 and Figure 7.

5.2 Proof of Correctness

The correctness proof for this algorithm follows the same outline as the proof for RWHC [9]. The similarities as well as the differences between the two proofs are pointed out in appropriate places throughout the text.
For each object, \textit{weak\_started}, \textit{weak\_ended}, and \textit{weak\_overwritten} are used to decide whether to perform the update part of a read-modify-write operation that was invoked by a different process, as described below.

The last variable, \textit{val}, is used to temporarily store a value that a pending strong read or a strong read-modify-write should return, as described below.

Weak operations are executed on the local copy of the memory and return immediately. A weak write increments the appropriate entry in \textit{weak\_started} and causes an \textit{update} message to be broadcast to all processes. This message contains the object to be updated, the new value for the object and a logical timestamp which is the value of \textit{llts} when the operation is invoked. Whenever an \textit{update} message is received, an \textit{ack} message is sent back to the initiator of the operation. Following this, the invoking process increments the appropriate entry of \textit{weak\_ended}. In other processes, the write is then executed on the local copy of the object if either (a) the previous operation that updated the same object was invoked by the same process, or (b) the value of \textit{last\_mess} was smaller than the value of \textit{llts} that was included in the \textit{update} message of the weak write. These conditions guarantee that all writes to the same object are executed by all processes in the same order. This order corresponds to the order implied by the logical timestamp of the writes. In other words, it corresponds to the value of the variable \textit{llts} at the point of their invocation. Ties among all writes with the same logical timestamp are broken by ordering the first delivered weak write after all other weak writes with the same logical timestamp. The variable \textit{last\_id} is required to preserve the order of writes by the same process, since two consecutive weak writes by the same process may have the same logical timestamp. After a process that did not invoke the write executes it, the appropriate entry of \textit{weak\_started} is copied into the appropriate entry of \textit{weak\_overwritten}. (This is done in order to signal that the last weak write by the local process to this location has been overwritten.)

Whenever a strong operation is invoked, if the strong operation is not a write and the last local operation was a weak read, then a \textit{dummy} message is broadcast to all processes. Every process that receives a \textit{dummy} message, returns an \textit{ack} message to the initiator using \textit{send\_abc}. Next, the process waits until all \textit{ack} messages of all previous operations and \textit{dummy} messages return. Then, a \textit{strong\_write}, a \textit{strong\_read} or a \textit{strong\_rmw} message is broadcast to every process, according to the operation’s type. The operation is executed by every process whenever the appropriate message is delivered:

- A strong write is executed by updating the local copy of the object, and by copying the appropriate entry of \textit{weak\_started} into the appropriate entry of \textit{weak\_overwritten}.
- A strong read is executed at the invoking process by copying the value of the local copy of the object to \textit{val}. This value is returned when the operation returns. A strong read is executed by other processes by doing nothing.
- A strong read-modify-write is executed if the appropriate entry of \textit{weak\_started} is equal to the appropriate entry of \textit{weak\_ended} or if the appropriate entry of \textit{weak\_started} is equal to the appropriate entry of \textit{weak\_overwritten}. By this condition, a read-modify-write whose
are read, write, and read-modify-write. Weak operations are executed instantaneously, while the
time required to execute a strong operation is proportional to the network delay.

The algorithm presented in this section is similar to the algorithm that was developed by
Attiya and Friedman and implements hybrid consistency for read/write objects only [9]. We
denote the latter algorithm by RWHC for the rest of this section. Since RWHC does not support
read-modify-write operations and since these operations have both read and write semantics,
several additions were needed in the mechanism that decides how and when to execute an
operation. The basic idea is similar to the one in Section 4. However, the rules for executing
operations in this algorithm were modified in order to allow weak write operations and strong
operations to access the same objects. Strong write and strong read operations are executed,
as before, when their corresponding messages are delivered. Weak writes, on the other hand,
are executed under the following conditions: A weak write that is not concurrent\(^5\) with any
other (weak or strong) write to the same object is always executed when its corresponding
message is delivered. If there are two concurrent weak writes to the same object, then the
first one is executed by every process and the second one is ignored by every process. Also,
since weak and strong operations may access the same objects, when we build the views of the
processes, it is sometimes required to reorder strong operations like we do with weak writes in
Section 4. To guarantee that this reordering will not violate the requirements in the definition
of hybrid consistency, processes send acknowledgments after executing weak writes and strong
operations do not start until all acknowledgments for previous (weak) operations are delivered.

A more detailed description of the algorithm and its pseudocode in Section 5.1 are followed
by a correctness proof in Section 5.2 and a complexity analysis in Section 5.3.

5.1 The Algorithm

We assume a system of \(n\) processes connected by some interconnection network. We assume,
also, an atomic broadcast mechanism like the one used in the algorithm in Section 4.

Every process maintains a local copy of the entire memory \(\text{mem}\), an array of integers
\(\text{last mess}\), an array of ids \(\text{last id}\), an array of integers \(\text{weak started}\), an array of integers
\(\text{weak ended}\), an array of integers \(\text{weak overwritten}\), an integer counter \(\text{lts}\) and an additional
object \(\text{val}\).

The counter \(\text{lts}\) is used to count the messages delivered by the atomic broadcast. It serves
as a logical time counter and its value is added to messages as a logical timestamp. It is
initiated to 0 and incremented each time a message that was sent with \(\text{abc}\) is delivered.

For each object, \(\text{last mess}\) and \(\text{last id}\) uniquely identify the last operation that updated
the object. That is, whenever a (weak or strong) operation updates an object, \(\text{last mess}\) is
assigned to the current value of the variable \(\text{lts}\) and \(\text{last id}\) is assigned to the id of the process
that invoked the operation.

\(^5\)Concurrent is with respect to the global logical time. In other words, two weak writes are concurrent if
the same number of messages have been delivered at the invoking processes before they were invoked.
weak read that reads from a strong operation and \( k \neq j \). Let \( swo_i \) be the operation \( \text{op}^1_k \) reads from. Thus, \( swo_i \xrightarrow{r} \text{op}^1_k \xrightarrow{r} \text{op}^2_k \). By Lemma 4.1, \( swo_i \xrightarrow{r} \text{op}^2_k \). Hence, by the rules for inserting weak reads, \( \text{op}^1_k \xrightarrow{r} \text{op}^2_k \).

We have shown that all the conditions in the definition of hybrid consistency are satisfied by any execution generated by the algorithm, even if weak read operations are allowed to access the same objects that are accessed by strong read, strong write and strong read-modify-write operations. Thus, the algorithm is a correct implementation for hybrid consistency under these assumptions.

**Remark:** It can be argued that this result holds for queue and stack operations as well. In this case, a read operation serves as a peek operation, i.e., it returns the head of the queue or the stack, respectively. The formal proof is omitted since it is almost the same as the proof for objects that support read, write and read-modify-write operations.

### 4.4 Complexity Analysis

The complexity of the implementation depends on the complexity of the atomic broadcast mechanism. Implementations for atomic broadcast in which the delivery time for every message is within \( 2d \) have been developed [9, 10]. Thus, we assume that the time required to deliver a message in our implementation is \( 2d \).

Weak operations are executed instantaneously. A strong operation broadcasts a \texttt{strong-op} message to all processes and then waits for the acknowledgements. Thus, there are two rounds of messages for each strong operation. Therefore, the time required to execute a strong operation is \( 4d \).

The atomic broadcast mechanism used in [9] has a message complexity of \( n + 1 \) physical messages for \texttt{abc} and 2 physical messages for \texttt{send.abc}. If we use these figures, then the message complexity of the above implementation is 0 physical messages for weak reads, \( n + 1 \) physical messages for weak writes and 3\( n + 1 \) physical messages for strong operations.\(^4\)

### 5 A Mixed Implementation for Hybrid Consistency

In this section, we present an asynchronous implementation for hybrid consistency in which every object that is accessed by strong operations can be also accessed by either weak write operations or weak read operations, but not by both. The weak operations supported by this implementation are read and write. The strong operations supported by this implementation

\(^4\)The atomic broadcast implementation used in [9] is centralized and is not fault tolerant. Other implementations of atomic broadcast could result in a higher message complexity.
Lemma 4.6 Every sequence $\tau_j$ is legal.

Proof: Assume, by way of contradiction, that there exists a sequence $\tau_j$ which is not legal. By Lemma 4.3 and the way weak reads are inserted, this can only happen if there exists a weak read $r_i(x,v)$ that reads from a strong operation, $i \neq j$, and $r_i(x,v)$ is ordered in $\tau_j$ after an obliterating write. (By the construction of $\tau_j$, the other cases are already covered by Lemma 4.3.) Let $sw_{o_k}(x,u)$ be the first obliterating write for $r_i(x,v)$ in $\tau_j$. Denote the previous strong operation of $p_k$ by $sop_k$ and the operation $r_i(x,v)$ reads from by $sw_{o_i}(x,v)$. Thus, $sw_{o_i}(x,v) \rightarrow sw_{o_k}(x,u) \rightarrow sop_k \rightarrow r_i(x,v)$. Recall that strong operations are ordered in every sequence $\tau_m$ according to the order their strong-op message is delivered and that this order is unique. Hence, the strong-op message of $sw_{o_i}(x,u)$ is delivered in $p_k$ after the strong-op message of $sw_{o_i}(x,v)$ and before the strong-op message of $sop_k$. Hence, $sw_{o_k}(x,u)$ is executed by $p_k$ between the execution of $sw_{o_i}(x,v)$ and the execution of $r_i(x,v)$. A contradiction to the assumption that $r_i(x,v)$ reads from $sw_{o_i}(x,v)$.

The next lemma replaces Lemma 4.4 of Section 4.2.

Lemma 4.7 For each sequence of operations $\tau_j$ and every pair of operations $op_k^1$ and $op_k^2$ such that either $op_k^1$ or $op_k^2$ is strong, if $op_k^1 \rightarrow op_k^2$, then $op_k^1 \rightarrow op_k^2$. 

Proof: By Lemma 4.4 and the way weak read operations are inserted, we know that the claim holds if $op_k^2$ is not a strong operation or if $op_k^1$ is a weak read that reads from a strong operation and $k \neq j$. Hence, we are left with the case that $op_k^2$ is a strong operation, $op_k^1$ is a
• By Lemma 4.4, every operation in each sequence \( \tau_j \) is ordered between its previous and next strong operations by the same process, which satisfies Condition 2 in the definition of hybrid consistency.

• By Lemma 4.1, there exists a linearization \( \rho \) of all operations in \( \sigma \) such that every sequence \( \tau_j \) agrees with \( \rho \) on the order of strong operations, which satisfies Condition 3 in the definition of hybrid consistency.

• By Lemma 4.2, for every sequence \( \tau_j \), \( \tau_j |j = \sigma|j \), which satisfies Condition 4 in the definition of hybrid consistency.

### 4.3 Accessing Strong Objects with Weak Reads

In this section we show that for objects that support read, write and read-modify-write operations, the algorithm in Figure 2 implements hybrid consistency even if weak read operations and strong operations are allowed to access the same objects. We start by generalizing several definitions that were given in Section 4.2.

A (weak or strong) read operation \( r_i(x, v) \) reads from a write or a read-modify-write operation \( wo_k(x, v) \) if \( wo_k(x, v) \) is the last operation that updates \( p_i \)'s copy of \( x \) before \( r_i(x, v) \) reads \( x \). We say that a read or a read-modify-write \( r_i(x, v) \) is legal in a sequence of operations \( \tau \) if

1. there exists a write or a read-modify-write operation \( wo_k(x, v) \) such that \( wo_k(x, v) \xrightarrow{r} r_i(x, v) \) and there does not exist another write or read-modify-write operation \( wo_l(x, u) \), \( u \neq v \), such that \( wo_k(x, v) \xrightarrow{r} wo_l(x, u) \xrightarrow{r} r_i(x, v) \), or

2. \( v \) is the initial value of the object and there does not exist another write or read-modify-write \( wo_l(x, u) \) such that \( wo_l(x, u) \xrightarrow{r} r_i(x, v) \).

Otherwise, the read or the read-modify-write is illegal in \( \tau \). Note that if every read and every read-modify-write is legal in a sequence of operations (that includes only read, write and read-modify-write operations), then the sequence is legal.

Given a read operation \( r^1_i(x, v) \), a write or a read-modify-write operation \( wo^2_k(x, v) \) such that \( r^1_i(x, v) \) reads from \( wo^2_k(x, v) \) and a sequence of operation \( \tau \), we say that \( wo^2_k(x, v) \) is an obliterating write for \( r^1_i(x, v) \) in \( \tau \) if \( wo^2_k(x, v) \xrightarrow{r} wo^3_k(x, u) \).

For each process \( p_j \), we build a sequence of operations \( \tau_j \) as in Section 4.2, with a small modification in the way weak reads are inserted into \( \tau_j \). Specifically, let \( \tau_j \) be the sequence of all operations in \( \sigma \) that are executed by \( p_j \), ordered by the order they are executed in \( \tau_j \). For each \( \tau_j \), add every weak write \( w_k \) that is not executed by \( p_j \) immediately before the weak write by \( p_j \) to the same object such that the update message of \( w_k \) is delivered inside the partial execution.
Figure 3: Proving the legality of $\tau_j$ in the generic algorithm
4. (Figure 3(d).) If neither \( w_i(x, v) \) nor \( w_i(x, u) \) are executed by \( p_j \), then there exists a write \( w_j(x, w) \) whose update message is delivered in \( p_j \) after the update messages of \( w_i(x, v) \) and \( w_i(x, u) \), and is invoked before the delivery of the strong-cp of \( sop_k \). Hence, the update message of \( w_j(x, w) \) is delivered in \( p_k \) between the delivery of the update messages of \( w_i(x, v) \) and \( w_i(x, u) \) and the ack message of \( sop_k \). Thus, either \( w_j(x, w) \) or some other write to \( x \) is executed by \( p_k \) between the delivery of the update messages of \( w_i(x, v) \) and \( w_i(x, u) \) and the invocation of \( r_k(x, v) \). A contradiction to the assumption that \( r_k(x, v) \) reads from \( w_i(x, v) \).

\[ \text{Lemma 4.4} \]

For each sequence of operations \( \tau_j \) and every pair of operations \( op_k^1 \) and \( op_k^2 \) such that either \( op_k^1 \) or \( op_k^2 \) is strong, \( op_k^1 \xrightarrow{\tau_j} op_k^2 \) if and only if \( op_k^1 \xrightarrow{\sigma} op_k^2 \).

\[ \text{Proof:} \] By Lemma 4.2, the claim holds if \( k = j \). Thus, for the rest of the proof, we may assume that \( k \neq j \). Recall that weak reads are placed between the previous and next strong operations of their invoking process as part of the construction of \( \tau_j \). Weak writes that influence \( p_j \) are ordered according to the order their update messages are delivered at \( p_j \) and are therefore ordered between the previous and next strong operations by the same process. A weak write that does not influence \( p_j \) is ordered before any next strong operation by the same process as part of the construction of \( \tau_j \) and due to the use of \( \text{abc} \) for both update and strong-cp messages.

Therefore, we are left with the case that \( op_k^2 \) is a weak write that does not influence \( p_j \) and \( op_k^1 \) is a strong operation. Recall that \( op_k^2 \) does not influence \( p_j \) only if its update message is delivered inside the partial execution interval of some weak write \( w_j \) to the same location. In this case, \( op_k^2 \) is ordered in \( \tau_j \) immediately before the invocation of \( w_j \). Since the update message of \( op_k^2 \) is delivered inside the partial execution interval of \( w_j \), the update message of \( w_j \) is delivered in \( p_k \) after the ack message that \( p_j \) sent for \( op_k^1 \). Hence, the strong-cp message of \( op_k^1 \) is delivered in \( p_j \) before the invocation of \( w_j \). Therefore, \( op_k^2 \) is ordered in \( \tau_j \) after \( op_k^1 \) and the claim follows.

\[ \text{Theorem 4.5} \]

Every execution generated by the algorithm described in this section is hybrid consistent.

\[ \text{Proof:} \] For an execution \( \sigma \), we have constructed a set of sequences \( \{\tau_j\}_{j=1}^n \). We now show why this set satisfies the requirements of the definition of hybrid consistency:

- By Lemma 4.3, each sequence \( \tau_j \) is legal.
- The construction of \( \{\tau_j\}_{j=1}^n \) guarantees that each \( \tau_j \) is a permutation of \( \text{ops}(\sigma) \), which satisfies Condition 1 in the definition of hybrid consistency.
Proof: By Lemma 4.1, the strong operations are ordered according to the order in which they are executed by every process. Thus, for every strong object \( x \), \( \tau_j x \) is included in the sequential specification of \( x \). By the construction of \( \tau_j \), every weak read by \( p_j \) is ordered in \( \tau_j \) after the write it reads from, with no other write to the same object in between them. Thus, it is left to be shown that weak reads by other processes are legal.

By construction of \( \tau_j \), a weak read \( r_k(x, v), k \neq j \), may not be legal only if one of the following holds:

- there exists a previous strong operation by \( p_k \) and the write \( r_k(x, v) \) reads from is ordered before the previous strong operation by \( p_k \), or
- there exists a next strong operation by \( p_k \) and the write \( r_k(x, v) \) reads from is ordered after the next strong operation by \( p_k \).

Assume, by way of contradiction, that there exists an illegal weak read \( r_k(x, v) \), and assume that \( r_k(x, v) \) reads from some weak write \( w_i(x, v) \). If there exists a next strong operation \( sop_k^2 \) and \( w_i(x, v) \) is ordered in \( \tau_j \) after \( sop_k^2 \), then the \texttt{update} message of \( w_i(x, v) \) is delivered after the execution of \( r_k(x, v) \) in \( p_k \). A contradiction to the assumption that \( r_k(x, v) \) reads from \( w_i(x, v) \).

Hence, we may assume that there exists a previous strong operation \( sop_k^1 \) and that there exists another weak write \( w_i(x, u), u \neq v \), such that \( w_i(x, v) \xrightarrow{\tau_j} w_i(x, u) \xrightarrow{\tau_j} sop_k^1 \xrightarrow{\tau_j} r_k(x, v) \). We complete the proof by examining the following four possible cases.

1. (Figure 3(a).) If both \( w_i(x, v) \) and \( w_i(x, u) \) are executed by \( p_j \), then the \texttt{update} messages of \( w_i(x, v) \) and \( w_i(x, u) \) are delivered in \( p_k \) in the order they appear in \( \tau_j \), and before the \texttt{strong-op} message of \( sop_k^1 \). Thus, either \( w_i(x, u) \) or some other write to \( x \) is executed by \( p_k \) between the delivery of the \texttt{update} message of \( w_i(x, v) \) and the invocation of \( r_k(x, v) \). A contradiction to the assumption that \( r_k(x, v) \) reads from \( w_i(x, v) \).

2. (Figure 3(b).) If only \( w_i(x, v) \) is executed by \( p_j \), then since \( w_i(x, u) \) is ordered before \( sop_k^1 \), there exists another write \( w_j(x, w) \) that is invoked between the delivery of the \texttt{update} message of \( w_i(x, v) \) and the \texttt{strong-op} message of \( sop_k^1 \). Thus, either \( w_j(x, w) \) or some other write to \( x \) is executed by \( p_k \) between the delivery of the \texttt{update} message of \( w_i(x, v) \) and the invocation of \( r_k(x, v) \). A contradiction to the assumption that \( r_k(x, v) \) reads from \( w_i(x, v) \).

3. (Figure 3(c).) If only \( w_i(x, u) \) is executed by \( p_j \), then there exists a write \( w_j(x, w) \) whose \texttt{update} message is delivered at \( p_j \) between the delivery of the \texttt{update} message of \( w_i(x, v) \) and the delivery of the \texttt{update} message of \( w_i(x, u) \). (It is possible that \( w_j(x, w) = w_i(x, u) \).) Hence, the \texttt{update} message of \( w_j(x, w) \) is delivered in \( p_k \) between the delivery of the \texttt{update} message of \( w_i(x, v) \) and the delivery of the \texttt{ack} message for \( sop_k^1 \). Thus, either \( w_j(x, w) \) or some other write to \( x \) is executed by \( p_k \) between the delivery of the \texttt{update} message of \( w_i(x, v) \) and the invocation of \( r_k(x, v) \). A contradiction to the assumption that \( r_k(x, v) \) reads from \( w_i(x, v) \).
Next, define the notion of an operation being executed by a process as follows: A strong operation \( sop_k \) is executed by \( p_j \) when the corresponding strong-op message is delivered at \( p_j \). A weak operation \( op_k \) is executed by \( p_j \) when the appropriate call event occurs. A weak write \( w_k \) that influences \( p_j \), \( k \neq j \), is executed by \( p_j \) when the corresponding update message is delivered at \( p_j \). Note that a weak write \( w_k \), \( k \neq j \), is not executed by \( p_j \) if and only if its update message is delivered inside the partial execution interval of some weak write by \( p_j \) to the same object. A strong operation is always executed by \( p_j \).

We now turn to the details of the proof.

For each process \( p_j \), let \( \tau_j \) be the sequence of all operations in \( \sigma \) that are executed by \( p_j \) (all strong operations, all operations by \( p_j \) and some of the weak write operations), in the order they are executed in \( p_j \). Note that if a weak write \( w_k \) is not executed by \( p_j \), then by the code of the algorithm, the update message of \( w_k \) is delivered inside a partial execution interval of another weak write by \( p_j \) to the same object. For each \( \tau_j \), add every weak write \( w_k \) that is not executed by \( p_j \) immediately before the weak write by \( p_j \) to the same object such that the update message of \( w_k \) is delivered inside the partial execution interval of \( w_j \). (If the update message of \( w_k \) is delivered inside the potential execution intervals of more than one weak write to the same object, then let \( w_j \) be the first one of them.) Finally, for every weak read \( r_k \), \( k \neq j \), insert \( r_k \) in the first place that is after the write it reads from (if exists) and the previous strong operation by \( p_k \) (if exists) but not after the next strong operation by \( p_k \) (if exists). (If there exists a next strong operation \( sop_k \) and the write \( r_k \) reads from is ordered after \( sop_k \), then \( r_k \) is ordered immediately before \( sop_k \).) Denote the resulting sequence by \( \tau_j \).

**Lemma 4.1** There exists a linearization \( \rho \) of all strong operations in \( \sigma \) such that for every pair of strong operations \( sop^1_k \) and \( sop^2_l \), \( sop^1_k \xrightarrow{\rho} sop^2_l \) if and only if \( sop^1_k \xrightarrow{\sigma} sop^2_l \), for every \( \tau_j \).

**Proof:** Recall that atomic broadcast delivers all messages to all processes in the same order. Since the strong operations are ordered in each \( \tau_j \) according to their order of delivery, they appear in every \( \tau_j \) in the same order. Since a strong operation does not return before all its \( \text{ack} \) messages return, the delivery of its strong-op message is always between its invocation and termination. Thus, the order in which all strong operations appear in every \( \tau_j \) extends the partial order implied by \( \sigma \) and is therefore a linearization.

**Lemma 4.2** For every sequence \( \tau_j, \tau_j|\sigma|j = \sigma|j \).

**Proof:** Atomic broadcast delivers all messages sent by the same process in the order they are invoked. Also, strong operations do not return until all \( \text{ack} \) messages are delivered. Thus, since weak operations are ordered according to their invocation and strong operations are ordered according to the delivery of their corresponding messages, for every sequence \( \tau_j, \tau_j|\sigma|j = \sigma|j \).

**Lemma 4.3** Every sequence \( \tau_j \) is legal.
Read($x$):
generate Return($x$, $mem[x]$)

Write($x$, $v$):
$mem[x] := v$
$local\_pending[x] := local\_pending[x] + 1$
$abc < update, $x$, $v$ >
generate Ack($x$)

Strong-Op($type$, $var\_list$):
$abc < strong\_op$, $type$, $variables\_list$ >
$missing\_acks := missing\_acks + n$

received < update, $x$, $v$ > from $p_j$:
if ($j = i$) then
$local\_pending[x] = local\_pending[x] - 1$
else if ($local\_pending[x] = 0$) then
$mem[x] := v$
endif

received < strong\_op, $t$, $v$ > from $p_j$:
send $abc < ack$, $j$ >
update all objects that needs to be updated according to $t$’s semantics and the variables’ list $v$
if $t$ was invoked by $p_i$ and has to return a value, then
store this value in $val$
endif

received < $ack$, $k$ > from $p_j$:
$missing\_acks := missing\_acks - 1$
if ($missing\_acks = 0$) then
if the pending strong operation must return a value then
generate SReturn($val$)
else
generate SAck()
endif
endif

Figure 2: The generic algorithm — code for process $p_i$. 
memory after receiving its update message if they have no pending weak writes to the same object, i.e., if the value of the corresponding entry of local_pending is 0. This check guarantees that the local copy of every object always holds the “newest” value, w.r.t. the global logical time, known to the local process. Hence, all writes to the same object are logically executed by all processes in the same order (the order in which their update messages are delivered).

A strong operation broadcasts a strong-op message to every process. The operation is executed by every process whenever the appropriate message is delivered. A strong operation that has write semantics, is executed by updating the local copy of the object. A strong operation that has read semantics, is executed at the invoking process by storing the value it should return in val, and by doing nothing otherwise. (Note, that some operations, e.g., read-modify-write and dequeue, have both read and write semantics. On the other hand, other operations, e.g., DEC-alpha’s MB operations [32], might not have a read or a write semantic at all; in this case, the operation is executed by doing nothing.) After executing a strong operation, an ack message is sent back to the initiator of the operation. A strong operation does not return until all ack messages have been received. Note that we can always execute a strong operation when its appropriate message is delivered since we assume that strong operations only access responsive objects. (Note that this is not always possible with objects that are not responsive. For example, a process may not be able to execute an await statement until another process issues an appropriate write.)

The formal description of the code for process $p_i$ appears in Figure 2.

4.2 Proof of Correctness

We prove the correctness of the implementation by showing that for every execution generated by the algorithm, we can build a set of sequences of operations $\{\tau_j\}_{j=1}^n$ as required by the definition of hybrid consistency.

We start with several definitions and some notation that will be used later.

A read operation $r_i(x,v)$ reads from a write operation $w_k(x,v)$ if $w_k(x,v)$ is the last write to $x$ that updates $p_i$’s copy of $x$ before $r_i(x,v)$ reads $x$. We say that a read $r_i(x,v)$ is legal in a sequence of operations $\tau$ if either there exists a write $w_k(x,v)$ such that $w_k(x,v) \xrightarrow{\tau} r_i(x,v)$ and there does not exist another write $w_i(x,u), u \neq v$, such that $w_k(x,v) \xrightarrow{\tau} w_i(x,u) \xrightarrow{\tau} r_i(x,v)$, or $v$ is the initial value of the object and there does not exist another write $w_i(x,u)$ such that $w_i(x,u) \xrightarrow{\tau} r_i(x,v)$; otherwise, the read is illegal in $\tau$.

For the rest of the proof, fix some execution $\sigma$ of the algorithm.

A partial execution interval of a weak write $w_k$ is the interval of events between the invocation of $w_k$ and the delivery of the update message of $w_k$ by the atomic broadcast at $p_k$.

A weak write $w_k(x,v)$ influences process $p_j$ if it updates $p_j$’s copy of $x$.
The rest of this section is organized as follows. A more detailed description of the algorithm and the pseudocode can be found in Section 4.1. The proof of correctness for the generic case is given in Section 4.2 while in Section 4.3 we show that for objects that support read, write and read-modify-write operations, the algorithm is correct even if we allow weak read operations and strong operations to access the same objects. Finally, we conclude with a complexity analysis in Section 4.4.

4.1 The Algorithm

We assume a system of \( n \) processes, connected by some interconnection network, each holds a local copy of the entire memory. We assume an atomic broadcast mechanism capable of supporting two primitives: \( \text{abc} \) and \( \text{send}_\text{abc} \). The first primitive, \( \text{abc} \), accepts a message as a parameter, and broadcasts this message to all processes atomically. The second primitive, \( \text{send}_\text{abc} \), accepts two parameters: a message to send and a process id to whom the message is designated. A process may use \( \text{send}_\text{abc} \) to send a message to a single process when the interleaving of this message with other messages sent by it, using either \( \text{abc} \) or \( \text{send}_\text{abc} \), should reflect their relative order of invocation.

More precisely, denote by \( \text{deliver}(j) \) the sequence of all messages delivered at \( p_j \) (either from \( \text{abc} \) or \( \text{send}_\text{abc} \)), in the order they were delivered at \( p_j \); denote by \( \text{abc}\_\text{deliver}(j) \) the restriction of \( \text{deliver}(j) \) to messages sent with \( \text{abc} \). For every pair of processes \( p_i \) and \( p_j \), denote by \( \text{sent}(i, j) \) the sequence of messages that are sent from \( p_i \) to \( p_j \) (either with \( \text{abc} \) or \( \text{send}_\text{abc} \)), ordered in the order they were sent by \( p_i \); denote by \( \text{deliver}(j)|i \) the restriction of \( \text{deliver}(j) \) to messages sent from \( p_i \) to \( p_j \). We require that

1. \( \text{sent}(i, j) = \text{deliver}(j)|i \), and
2. \( \text{abc}\_\text{deliver}(i) = \text{abc}\_\text{deliver}(j) \).

We use \( \text{send}_\text{abc} \) instead of a regular send since it guarantees that the order of delivery of all messages at each process reflects the order in which they were sent. We use \( \text{send}_\text{abc} \) instead of \( \text{abc} \) because in some implementations of atomic broadcast, e.g., the one presented in [9], it has a lower message complexity.

Every process holds a local copy of the entire memory \( \text{mem} \), an array of integers \( \text{local}\_\text{pending} \), initially 0, an integer counter \( \text{missing}\_\text{acks} \) that counts the number of expected (missing) acknowledgements, initially 0, and an additional strong object \( \text{val} \) that is used to temporarily store a value that a pending strong operation should return, as described below.

Weak operations are executed on the local copy of the memory and return immediately. A weak write increments the corresponding entry of \( \text{local}\_\text{pending} \) and causes an \( \text{update} \) message to be broadcast to all processes. This message contains the object to be updated and the new value for the object. Upon receiving an \( \text{update} \) message, the invoking process decrements the corresponding entry of \( \text{local}\_\text{pending} \). Other processes execute the write on the local copy of the
4 A Generic Implementation for Hybrid Consistency

In this section, we present an asynchronous\(^3\) generic implementation for hybrid consistency that supports weak read/write operations and any type of strong operation that accesses responsive objects. This algorithm assumes that the set of objects accessed by weak operations is disjoint from the set of objects accessed by strong operations. However, when restricted to objects that support read, write, read-modify-write, enqueue, dequeue, push, and pop, weak reads can access the same objects as strong operations.

The basic motivation behind the algorithm is to allow weak operations to be executed instantaneously, while keeping the response time for strong operations linearly proportional to the network delay. Being able to execute weak operations fast is vital for the usefulness of this implementation, since it is the major advantage hybrid consistency has over sequential consistency. By allowing each process to hold a local copy of the entire memory, we can guarantee instantaneous execution of weak operations, provided that write operations will eventually update all copies.

The algorithm uses an atomic broadcast mechanism to send messages. Having an atomic broadcast as an underlying tool in the development of distributed systems is becoming quite common [11, 13, 29, 31]. Our decision to use atomic broadcast follows this trend and is justified by the immediate benefits that this mechanism provides. Assuming an atomic broadcast mechanism is more modular than implementing the communication protocol directly. Moreover, the atomic broadcast mechanism creates a local logical time, which simplifies the code of the algorithm and its proof of correctness. Several atomic broadcast algorithms with various degrees of fault tolerance and efficiency have been developed [6, 12, 14, 17]. Thus, the exact choice of an atomic broadcast algorithm can help reach the design goals of the implementation in terms of fault tolerant, message complexity, and time complexity.

Basically, the algorithm goes as follows. Weak operations are executed locally. Strong operations and weak write operations broadcast a message to all processes; the operation is executed by each process when the message is delivered at the process. Thus, the order of execution should reflect the order by which messages are delivered by the atomic broadcast mechanism.

Unfortunately, we have a problem since weak operations must return immediately and a weak read that immediately follows a weak write by the same process must return the value written by that write. This problem is solved using the following two rules: (a) a weak write updates the local copy of the memory immediately following its invocation, and (b) a process that invokes a weak write ignores every message that is delivered between the invocation of the weak write and the delivery of its corresponding message and contains a request to update the same object. This way each read returns the value of the “newest” write (according to the logical time) to the same object that is known to its invoking process when the read is invoked.

---

\(^3\)That is, processes run at different rates and the delay of messages is unknown.
Since no message is ever received in \( \sigma_1 \) and \( \sigma_2 \), the partial execution obtained from \( \sigma_1 \) by replacing \( p_i \)'s history with \( p_i \)'s history in \( \sigma_2 \) can be extended to a weakly consistent execution \( \sigma \), having no other accesses to \( x \). Then \( \text{ops}(\sigma) \) consists of the operation \([\text{RMW}_1(x, f_1), \text{Return\&Ack}_1(x, 0)]\) and \([\text{RMW}_2(x, f_2), \text{Return\&Ack}_2(x, 0)]\).

Since \( \sigma \) is weakly consistent, there exists a legal serialization \( \tau \) of all the operations in \( \sigma \). Thus, either \( r\text{mw}_1(x, f_1, 0) \xrightarrow{\tau} r\text{mw}_2(x, f_2, 0) \) or \( r\text{mw}_2(x, f_2, 0) \xrightarrow{\tau} r\text{mw}_1(x, f_1, 0) \). In both cases, since we assumed that \( f_1(0) \neq 0 \) and \( f_2(0) \neq 0 \), \( \tau \) is not legal. A contradiction.

Since all hybrid consistent executions and all sequentially consistent executions are weakly consistent too, the following two theorems follow immediately:

**Theorem 3.2** For any implementation of hybrid consistency, \( |\text{RMW}| \geq d \).

**Theorem 3.3** For any implementation of sequential consistency, \( |\text{RMW}| \geq d \).

### 3.2 Dequeue and Pop Operations

Attiya has shown that the time required to execute a dequeue or a pop operation in any implementation of sequential consistency is at least \( d \) [7]. These results rely on the requirement for a legal serialization of all operations in the definition of sequential consistency, but do not assume anything about the way operations are ordered in this serialization. Therefore, with few minor modifications, the proofs apply to weak consistency and hybrid consistency as well. (The details appear in [19].) Thus, we have:

**Theorem 3.4** For any weakly consistent implementation of a FIFO queue \( Q \), \( |\text{deq}(Q)| \geq d \).

**Theorem 3.5** For any weakly consistent implementation of a stack \( S \), \( |\text{pop}(S)| \geq d \).

Since every hybrid consistent execution is also weakly consistent, the next two theorems follow immediately:

**Theorem 3.6** For any hybrid consistent implementation of a FIFO queue \( Q \), \( |\text{deq}(Q)| \geq d \).

**Theorem 3.7** For any hybrid consistent implementation of a stack \( S \), \( |\text{pop}(S)| \geq d \).
We now introduce some notation we use in the rest of the paper. We denote by \( \text{op}_i \) an operation invoked by process \( p_i \) (weak or strong), and by \( \text{sop}_i \) we denote a strong operation invoked by process \( p_i \). We use superscripts, e.g., \( \text{op}_1^i, \text{op}_2^i, \ldots \), to distinguish among operations invoked by the same process. We sometimes use a shorthand notation for read, write and read-modify-write operations and denote by \( r_i(x,v) \) a (weak or strong) read operation invoked by process \( p_i \) returning \( v \) from \( x \); we denote by \( w_i(x,v) \) a (weak or strong) write operation invoked by process \( p_i \) writing \( v \) to \( x \); \( \text{rmw}_i(x,f,v) \) is a read-modify-write operation invoked by process \( p_i \) returning \( v \) from \( x \) and writing \( f(v) \) to \( x \); \( \text{ro}_i(x,v) \) is either a read or a read-modify-write operation invoked by process \( p_i \) returning \( v \) from \( x \); \( \text{wo}_i(x,v) \) is either a write or a read-modify-write operation invoked by process \( p_i \) writing \( v \) to \( x \); \( \text{rmw}_i(x,f,v) \) is a strong read-modify-write operation invoked by process \( p_i \) returning \( v \) from \( x \) and writing \( f(v) \) to \( x \); \( \text{sw}_i(x,v) \) is either a strong write or a strong read-modify-write operation invoked by process \( p_i \) writing \( v \) to \( x \); \( \text{sr}_i(x,v) \) is either a strong read or a strong read-modify-write operation invoked by process \( p_i \) returning \( v \) from \( x \); \( \text{sw}_i(x,v) \) is either a strong write or a strong read-modify-write operation invoked by process \( p_i \) writing \( v \) to \( x \).

3 Lower Bounds

3.1 Read-Modify-Write Operations

In this section, we prove a lower bound on the time required to execute a read-modify-write operation in implementations of weak consistency, hybrid consistency and sequential consistency. The proof technique that we use in this section was used before in [7, 9, 10, 28, 30].

Theorem 3.1 For any implementation of weak consistency, \(|\text{RMW}| \geq d\).

Proof: Assume, by way of contradiction, that there is an implementation of weak consistency for which \(|\text{RMW}| < d\). Let \( p_1 \) and \( p_2 \) be two processes that access an object \( x \), and let \( f_1 \) and \( f_2 \) be two functions such that \( f_1(0) \neq 0 \) and \( f_2(0) \neq 0 \). Without loss of generality, assume that \( x \) is initially \( 0 \).

By the specification of \( x \), there is some execution \( \sigma_1 \) such that \( \text{ops}(\sigma_1) \) is \([\text{RMW}_1(x,f_1),\text{Return}\&\text{Ack}_1(x,0)]\) and \( \text{RMW}_1(x,f_1) \) occurs at real time \( 0 \). The delay of all messages in \( \sigma_1 \) is exactly \( d \). By assumption, the real time at the end of \( \sigma_1 \) is less than \( d \). Hence, no message is received at any node during \( \sigma_1 \).

In a similar manner, by the specification of \( x \), there is some execution \( \sigma_2 \) such that \( \text{ops}(\sigma_2) \) is \([\text{RMW}_2(x,f_2),\text{Return}\&\text{Ack}_2(x,0)]\) and \( \text{RMW}_2(x,f_2) \) occurs at real time \( 0 \). The delay of all messages in \( \sigma_2 \) is exactly \( d \). By assumption, the real time at the end of \( \sigma_2 \) is less than \( d \). Hence, no message is received at any node during \( \sigma_2 \).
then \( o_{p^1} \xrightarrow{\tau} o_{p^2} \).

Let \( \tau \) be an execution. Denote by \( \tau \mid j \) the restriction of \( \tau \) to operations invoked by \( p_j \); similarly, denote by \( \tau \mid x \) the restriction of \( \tau \) to operations on object \( x \).

The semantics of an object can affect its possible implementations. In particular, some operations may not be able to terminate until the object they access holds a certain value or until some other operations are invoked. For example, the operation \( \text{await}(x = 1) \) cannot terminate until the value of \( x \) is 1. Hence, if the value of \( x \) is different than 1 when the \( \text{await} \) is invoked, then it cannot terminate until another write updates \( x \) with 1. On the other hand, the termination of operations like read, write, read-modify-write, push, and pop does not depend on an object having a certain value, or another operation being executed.

Formally, we assume that each call event is associated with a (possibly empty) list of input variables and every response event is associated with a (possibly empty) list of output variables. A typical input list would usually consist of an object to be accessed and a value or a function to be applied to the object; a typical output list would usually consist of the object that was accessed and a value returned from this object.

**Definition 2.1** An object \( x \) is responsive if for any initial value for \( x \), any operation type \( op \) supported by \( x \), any number of processes in the system and any input list \( \text{in}_\text{list} \), there exists an output list \( \text{out}_\text{list} \) such that \( op(\text{in}_\text{list}, \text{out}_\text{list}) \) is in the sequential specification of \( x \).

It is possible to mark some operations as strong; all other operations are called weak. In the case of objects that support read, write, and read-modify-write operations, this means that it is possible to use strong read, strong write, and strong read-modify-write operations. We denote the call events for strong operations by \( S\text{Write}(x, v) \), \( S\text{Read}(x) \), and \( S\text{RMW}(x, f) \) and the respective response events by \( S\text{Ack}(x) \), \( S\text{Return}(x, v) \), and \( S\text{Return\&Ack}(x, v) \).

For completeness, we repeat the definitions of hybrid consistency and weak consistency:

**Definition 2.2 (Hybrid consistency)** An execution \( \sigma \) is hybrid consistent if there exists a linearization \( \rho \) of \( \sigma \) such that for each process \( p_j \), there exists a legal sequence of operations \( \tau_j \) with the following properties:

1. \( \tau_j \) is a permutation of \( \text{ops}(\sigma) \).
2. if \( o_{p^1} \xrightarrow{\rho} o_{p^2} \) and at least one of \( o_{p^1} \) and \( o_{p^2} \) is strong, then \( o_{p^1} \xrightarrow{\tau_j} o_{p^2} \), for any \( i \).
3. if \( o_{p^1} \xrightarrow{\rho} o_{p^2} \) and \( o_{p^1} \) and \( o_{p^2} \) are strong, then \( o_{p^1} \xrightarrow{\tau_j} o_{p^2} \), for any \( i \) and \( k \).
4. \( \tau_j \mid j = \sigma \mid j \).

**Definition 2.3 (Weak consistency)** An execution \( \sigma \) is weakly consistent if for every process \( p_j \) there exists a legal serialization \( \tau_j \) of \( \sigma \) such that \( \tau_j \mid j = \sigma \mid j \).
1. For every $i$ and $j$, every message in $\sigma$ from $p_i$ to $p_j$ has its delay in the range $[0, d]$, for a fixed nonnegative integer $d$. (This is a restriction on the network.)

2. For every $i$, at most one call at $p_i$ is pending at a time. (This is a restriction on the application program.)

From now on, we assume all executions are admissible. A history $h'$ is a partial history of a history $h$ if there exists a time $t_0$ such that $h'(t) = h(t)$ for all times $t < t_0$, $h'(t_0)$ is a prefix of $h(t_0)$ and $h'(t)$ is empty for all times $t > t_0$. A partial execution is a collection $E$ of partial histories, one for every process, in which there is a one-to-one correspondence between the messages received by $p_i$ from $p_j$ in $E$ and the messages sent by $p_i$ to $p_j$ in $E$, for every two processes $p_i$ and $p_j$.

Each pair of a call event and a subsequent matching response event forms an operation. The call event marks the start of the operation, while the response event marks the end of that operation. An operation $op$ is invoked when the application process issues the appropriate call for $op$; $op$ terminates when the mcs process issues the appropriate response for $op$. An operation is pending if it was invoked and did not terminate. Note that since we assumed that all executions are admissible, there can be at most one pending operation per process, i.e., an application program does not invoke a new operation before the previous one has terminated. Given a particular mcs, an object $x$ implemented by it, and an operation type $op$ on $x$, we denote by $|op(x)|$ the maximum time taken from the invocation to the termination of an operation of type $op$ on $x$ in any execution. We denote by $|op|$ the maximum of $|op(x)|$ over all objects $x$ implemented by the mcs.

Every object is assumed to have a sequential specification (cf. [24]) defining a set of operations, which are ordered pairs of call and response events, and a set of operation sequences, which are the allowable sequences of operations on that object. For example, in the case of an object that supports read, write, and read-modify-write operations, the ordered pair of events $[\text{Read}_i(x), \text{Return}_i(x, v)]$ forms an operation for any $p_i$, $x$, $v$ and $f$, as does $[\text{Write}_i(x, v), \text{Ack}_i(x)]$ and $[\text{RMW}_i(x, f), \text{Return&Ack}_i(x, v)]$. The set of operation sequences consists of all sequences in which every read or read-modify-write operation returns the value that is written by the latest preceding write or read-modify-write operation. Note that the value written by a read-modify-write operation is the result of applying its function on the value returned by it.

A sequence $\tau$ of operations for a collection of processes and objects is legal if, for every object $x$, the restriction of $\tau$ to operations of $x$ is in the sequential specification of $x$.

Given an execution $\sigma$, let $ops(\sigma)$ be the sequence of call and response events appearing in $\sigma$ in real-time order, breaking ties by ordering all events of the same process in the order they appear in that process and then using process ids. An execution $\sigma$ implies a partial order, $\sigma$, on the operations which appear in $\sigma$; $op^1 \sigma op^2$ if the response event of $op_1$ appears in $ops(\sigma)$ before the call event of $op_2$.

Given an execution $\sigma$, a sequence of operations $\tau$ is a serialization of $\sigma$ if it is a permutation of $ops(\sigma)$. A serialization $\tau$ of $\sigma$ is a linearization if it extends $\sigma$; that is, if $op^1 \sigma op^2$
modify-write operations, the call events are Read\(_i\)(\(X\)), Write\(_i\)(\(X,v\)), and RMW\(_i\)(\(X,f\)) for all objects \(X\), values \(v\), and functions \(f\).

2. **Response events**: The response of the mc\(s\) to operations initiated by the application program, according to their specification. For example, in the case of objects that support read, write, and read-modify-write operations, the response events are Return\(_i\)(\(X,v\)), Ack\(_i\)(\(X\)), and Return\&Ack\(_i\)(\(X,v\)) for all objects \(X\), and values \(v\).

3. **Message receive events**: receive\((i,m,j)\) for all messages \(m\) and nodes \(i\) and \(j\): the mc\(s\) process on node \(i\) receives message \(m\) from the mc\(s\) process on node \(j\).

4. **Message send events**: send\((i,m,j)\) for all messages \(m\) and mc\(s\) processes \(p_i\) and \(p_j\): the mc\(s\) process on node \(i\) sends message \(m\) to the mc\(s\) process on node \(j\).

The call and message-receive events are interrupt events.

An mc\(s\) process (or simply process) is an automaton with a (possibly infinite) set of states, including an initial state, and a transition function. Each interrupt event causes an application of the transition function. The transition function is a function from states and interrupt events to states, sets of response events and sets of message-send events. That is, the transition function takes as input the current state and an interrupt event, and produces a new state, a set of response events for the application process, and a set of messages to be sent. A step of a process \(p\) is a tuple \((s,i,s',R,M)\), where \(s\) and \(s'\) are states, \(i\) is an interrupt event, \(R\) is a set of response events, \(M\) is a set of message-send events, and \(s', R\) and \(M\) are the result of \(p\)'s transition function acting on \(s\) and \(i\). A history of a process \(p\) is a mapping \(h\) from \(\mathbb{R}\) (real time) to finite sequences of steps such that

1. for each real time \(t\), there is only a finite number of times \(t' < t\) such that the corresponding sequence of steps \(h(t')\) is nonempty (thus the concatenation of all the sequences in real-time order is a sequence);

2. the old state in the first step is \(p\)'s initial state; and

3. the old state of each subsequent step is the new state of the previous step.

An execution of an mc\(s\) is a set of histories, one for each process in \(P\), in which there is a one-to-one correspondence from the messages received by \(p_i\) from \(p_j\) onto the messages sent by \(p_j\) to \(p_i\), for any processes \(p_i\) and \(p_j\). An infinite execution is an execution in which every history is infinite. We use the message correspondence to define the delay of any message in an execution to be the real time of receipt minus the real time of sending. (The network is not explicitly modeled, although the constraints on executions, defined below, imply that the network reliably delivers all messages sent.)

Execution \(\sigma\) is admissible if the following conditions hold:

4
The rest of the paper is organized as follows: Section 2 includes the definitions of the system. The proofs of the lower bounds can be found in Section 3. The generic implementation of hybrid consistency appears in Section 4, while the specific implementation that supports strong read-modify-write operations is given in Section 5. We conclude with a discussion in Section 6.

2 The System

The model used here is very similar to the one used in [9, 10]. We repeat the definitions for completeness.

We consider a collection of application programs running concurrently and communicating via virtual shared memory, which consists of a collection of objects. We assume a system consisting of a collection of nodes $P$ connected via a communication network, such that each application program runs on a different node. The shared memory abstraction is implemented by a memory consistency system (mcs), which uses local memory and some protocol executed by the mcs processes (one at each node). A correctness condition is defined at the interface between the application programs (written by the user) and the mcs processes (supplied by the system). Thus, the mcs must provide the proper semantics when the values of the responses to calls are considered, throughout the network. An illustration of the system architecture is given in Figure 1.

In more detail, the following events may occur at the mcs process on node $i$:

1. Call events: The initiation of operations by the application program according to their specification. For example, in the case of objects that support read, write, and read-
large class of synchronization operations, each of which is uniquely defined by the function used in the modify part of the operation. Specific examples include test&set, fetch&add, and compare&swap.

Specifically, our first contribution is a lower bound on the time required to implement read-modify-write operations. We show that for any implementation of hybrid consistency, the execution time of a (weak or strong) read-modify-write operation is at least \( d \), where \( d \) is the network delay. Similarly, we show that the execution time of a (weak or strong) dequeue or a (weak or strong) pop operation on any implementation of hybrid consistency is at least \( d \). These lower bounds, combined with the implementations presented in [7, 10] for sequential consistency, suggest that there are no performance gains in providing weak semantics for read-modify-write, dequeue or pop operations. Note that labeling an operation as weak or strong is unrelated to its semantics. For example, it is possible to implement weak read-modify-write, weak pop and weak dequeue operations. Moreover, programs that use weak read-modify-write, weak pop or weak dequeue operations do make sense. However, our result indicates that the performance of such programs do not benefit significantly from this.

Our second contribution is an efficient generic implementation for hybrid consistency that supports most types of strong operations combined with weak read and weak write operations. The correctness of this implementation depends on the assumption that weak and strong operations never access the same object. Weak operations are executed instantaneously while the time required to execute strong operations is \( O(d) \). Thus, the execution time achieved by this implementation is within a constant factor of the lower bounds for most common synchronization operations. We then show that for objects that support read, write, read-modify-write, enqueue, dequeue, push, and pop, this implementation is a correct implementation for hybrid consistency even if weak reads and strong operations are allowed to access the same objects. A simple example that demonstrates the potential performance gains of this feature using read/write objects is given in [9]. To the best of our knowledge, this is the first formally verified distributed implementation for hybrid consistency, or any similar condition, e.g., [1, 2, 18, 22], that supports read-modify-write operations and allows weak read operations and strong operations to access the same objects. (Note that this algorithm is not correct if weak writes are allowed to access the same objects as strong operations.)

Our third contribution is an efficient implementation for hybrid consistency in which each object that is accessed by strong operations can be also accessed by either weak write operations or weak read operations (but not by both). The types of operations supported by this implementation are weak read, weak write, strong read, strong write, and strong read-modify-write. In this implementation, weak operations are executed instantaneously, while the time required to execute a strong operation is \( O(d) \).

---

1 In fact, we show the lower bounds for implementations of weak consistency. However, since hybrid consistency is strictly stronger than weak consistency, the result holds for hybrid consistency, as well.

2 The definitions of release consistency in [21, 22] are given using an automaton that allows weak and strong operations to access the same locations. However, this automaton is part of the definition, and is not a distributed implementation of the consistency condition.
1 Introduction

Shared memory is a convenient paradigm for communication between processes. It is more high level than message passing and it is a natural extension of serial programming. This has led to a wide study of concurrent programming with shared memory, and many problems have already been solved using shared memory. Thus, supporting shared memory on distributed machines is a desired goal.

Unfortunately, implementing shared memory on a distributed machine is much more complicated than on a serial machine. This is due to the higher degree of parallelism and the lack of synchronization that is inherent in the distributed architecture. Thus, the exact semantics of the shared memory must be explicitly defined. We call the definition of the memory's semantics a consistency condition.

Sequential consistency [27] is considered an attractive consistency condition, since it gives an illusion of real shared memory. The main drawback of sequential consistency, as reflected in recent results [10, 26, 28], is that it cannot be implemented efficiently. On the other hand, the semantics provided by weaker conditions, e.g., [3, 5, 23, 25, 28], that can be implemented efficiently, is too weak to solve some of the common problems of concurrent programming [9]. These results agree with recent trends in the architecture community for implementing distributed shared memory. The memory model of many parallel computers provide weak read and write operations combined with various kinds of strong synchronization operations [4, 16, 20]. Thus, the more frequently used weak read and write operations can be executed efficiently, resulting in greater performance, while the strong synchronization operations provide the same expressiveness that can be found in sequential consistency.

Hybrid consistency is an efficient and expressive consistency condition [9]; it captures some of the essential features of several other consistency conditions appearing in the literature [1, 15, 18]. Very informally, it guarantees two major properties:

1. Strong operations appear to be executed in some sequential order.
2. If two operations are invoked by the same process and one of them is strong, then they appear to be executed in the order they were invoked.

In particular, the second property guarantees that a strong operation appears to be executed after any (weak or strong) operation invoked before it by the same process, and before any (weak or strong) operation invoked after it by the same process.

The definition of hybrid consistency in [9] applies to any collection of objects for which a sequential specification is provided (cf. [24]). However, the algorithm and lower bounds presented in [9] consider only read and write operations. Our work extends the work of [9] by studying possible implementations of hybrid consistency in which other synchronization operations are supported. Our study concentrates on three types of operations: read-modify-write operations, FIFO queue operations, and stack operations. Read-modify-write unifies a
Abstract

In recent years, there is a growing tendency to support high-level synchronization operations, such as read-modify-write, FIFO queues and stacks, as part of the programmer's shared memory model. This paper examines the problem of implementing hybrid consistency with high-level synchronization operations. Lower bounds on the time required to implement several common synchronization operations are presented. They are followed by several efficient implementations for hybrid consistency that support various synchronization operations.

Specifically, it is shown that for any implementation of weak consistency, the time required to execute a read-modify-write, a dequeue or a pop operation is $\Omega(d)$, where $d$ is the network delay.

An efficient implementation for hybrid consistency that supports most types of high-level synchronization operations and weak read and write operations is presented. This implementation assumes that weak and strong operations never access the same objects. Weak read and write operations are executed instantaneously, while the time required to execute strong operations is $O(d)$. This is within a constant factor of the lower bounds for most of the commonly used types of operations. For objects that support read, write and read-modify-write, dequeue, enqueue, push, and pop, it is shown that this implementation is a correct implementation for hybrid consistency even if weak reads and strong operations are allowed to access the same objects. This implementation, however, is not correct if weak writes and strong operations access the same objects.

Following this, another efficient implementation for hybrid consistency that supports read, write and read-modify-write operations is presented. In this implementation, each object that is accessed by strong operations can be also accessed by either weak write operations or weak read operations, but not by both. Weak operations are executed instantaneously, while the time required to execute strong operations is $O(d)$. 
Implementing Hybrid Consistency with High-Level Synchronization Operations*

Roy Friedman
Department of Computer Science
The Technion
Haifa 32000, Israel

July 11, 1994

*Partially supported by Technion V.P.R. – Argentinian Research Fund and the US-Israel Binational Science Foundation. An extended abstract of this work appears in proceedings of the 13th ACM Symposium on Principles of Distributed Computing, August 1993, pp. 229–240.