Technical Report

UCAM-CL-TR-681
ISSN 1476-2986

Number 681

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Indirect channels: a bandwidth-saving technique for fault-tolerant protocols

Piotr Zieliński

April 2007

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Indirect channels: a bandwidth-saving technique for fault-tolerant protocols

Piotr Zieliński
piotr.zielinski@cl.cam.ac.uk
Cavendish Laboratory, University of Cambridge, UK

Abstract

Sending large messages known to the recipient is a waste of bandwidth. Nevertheless, many fault-tolerant agreement protocols send the same large message between each pair of participating processes. This practical problem has recently been addressed in the context of Atomic Broadcast by presenting a specialized algorithm.

This paper proposes a more general solution by providing virtual *indirect channel*s that physically transmit message ids instead of full messages if possible. Indirect channels are transparent to the application; they can be used with any distributed algorithm, even with unreliable channels or malicious participants. At the same time, they provide rigorous theoretical properties.

Indirect channels are conservative: they do not allow manipulating message ids if full messages are not known. This paper also investigates the consequences of relaxing this assumption on the latency and correctness of Consensus and Atomic Broadcast implementations: new algorithms and lower bounds are shown.

1 Introduction

Sending large network messages that are known to the receiver is a waste of bandwidth. Nevertheless, many fault-tolerant distributed protocols suffer from this problem. For example, in Consensus and Atomic Broadcast algorithms commonly perform an all-to-all message exchanges. Such an exchange involves $O(n^2)$ messages, and results in each of the n processes receiving the same message $O(n)$ times. The network bandwidth usage could be significantly reduced if large messages were sent only to processes that do not know them already.

This problem has been known for a long time, and many real-world protocols have taken steps to mitigate it. For example, in the Network News Transfer Protocol [11], a client can advertise the possession of a given article m with the IHAVE command. The server can then ask the client to send m or not; this avoids unnecessary communication if the server knows m already. Another example is *rsync*, a popular file synchronization program [1]. Before synchronizing, both end-points compute and compare hashes of parts of the file, so as to avoid transmitting parts already known to the other party. Both NNTP and rsync reduce bandwidth usage at the expense of latency: if the recipient does not know the message, three communication steps are needed instead of one ("do you have

m?", "no", "here is m"). Since in point-to-point protocols, one cannot generally predict whether the recipient knows the message or not, this increase in latency is impossible to avoid.

The situation is different in group-oriented protocols, such as Atomic Broadcast, which are multi-step by nature. This means that at the beginning of the second step each process can reasonably believe that all the others have received the original broadcast as well (Figure 2). From that point on, bandwidth can be saved by sending message identifiers instead of full messages [12, 24]. Despite its apparent simplicity, this technique is surprisingly difficult to implement correctly. Great care must be taken to avoid delivering *orphaned* message ids, for which no correct process knows the corresponding full message [10]. The first such an implementation has been recently proposed by Ekwall and Schiper [10] for Atomic Broadcast.

One can imagine that a similarly careful analysis can be carried out for other distributed protocols such as Generic Broadcast, Atomic Multicast, or just other implementations of Atomic Broadcast [8]. However, the method in [10] suggests that the amount of work involved is similar to actually designing a new protocol from scratch. It would be easier if one could take any existing protocol and perform such a transformation automatically.

This paper proposes a solution. Instead of solving the problem on the protocol level, it provides a new lightweight abstraction: *indirect channels*. In brief, if the channel believes the receiver already knows the transmitted message, it sends a short message identifier instead. If this belief turns to be wrong, the receiver requests the full message from the sender. As in [10], care must be taken to avoid orphaned message identifiers.

Indirect channels differ from the NNTP/rsync synchronization method in that the decision whether to send the full message or only the id is taken locally by the sender, without consulting the recipient. This optimistic approach saves latency if the guess is right (typical), at the expense of a higher latency in anomalous runs.

Indirect channels aim at bridging the gap between theory and practice. On the one hand, they provide precise reliability guarantees required by theoretical abstractions. On the other hand, their implementation gives room to system-specific tuning, which can affects only the performance but not correctness. Cache-like memory management is a good example: all possibly large data can be discarded at any time without breaking the algorithm, with the performance degradation being the only penalty for an unlucky removal.

Standard Atomic Broadcast [6] run over indirect channels exhibits a latency similar to that of the algorithm in [10]. The advantage of indirect channels lies in the fact that they can be used with any distributed protocol without modifications, even with those tolerating malicious participants. Depending on the underlying channels, indirect channels can provide several levels of reliability guarantees. All in all, indirect channels allow an algorithm designer to separate the large-messages issue from the main problem.

By design, indirect channels conservatively avoid orphaned ids altogether. Is it possible to improve the latency even further by tolerating orphaned ids in a controlled way? Atomic Broadcast with Indirect Consensus [10] can tolerate orphaned ids in the first two steps. This approach, however, is algorithm-specific and may require more processes for the same fault resilience. I show a general method of achieving a limited orphan-tolerance with any Atomic Broadcast without decreasing its resilience. This is done by separating

the agreement on id ordering from that on id-to-message mappings, and running them in parallel. Interestingly, no such improvement is possible with Consensus; this paper presents lower bounds on the maximum latency improvement resulting from treating small and large messages differently.

This paper is structured in the following way. Section 2 introduces indirect channels and presents an implementation that handles crash-failures and reliable communication. Section 2.2 extends the algorithm for unreliable communication and Byzantine failures. Section 2.3 presents an experimental evaluation. Section 3 investigates further improvements in latency of Consensus and Atomic Broadcast as a result of limited tolerance of orphaned ids.

2 Indirect channels

I assume that the system consists of n processes $p_1, \ldots, p_n$, which can fail only by crashing, and use asynchronous reliable channels for communication: all messages between correct processes *eventually* get delivered. Section 2.2 will relax these assumptions and consider eventually reliable channels and Byzantine (malicious) participants.

2.1 Implementation of indirect channels

The objective of this section is to implement channels that save the network bandwidth by physically transmitting a message only if the recipient does not known it. If the sender believes that the recipient knows the message, it sends the message id instead. In rare cases when this belief turns out to be wrong, the sender retransmits the full message upon a request from the recipient. To be able to fulfil that request, the sender must store its messages until the recipient acknowledges a successful reception or asks for retransmission. In the meanwhile, if the cache management demands removing this information from memory, the message must be preemptively sent to the recipient in full.

Figure 1 shows an implementation of the algorithm sketched above, which provides a reliable indirect channel using a reliable underlying channel. For clarity, I use "ind-send" and "ind-receive" for the indirect channel being implemented, and "send" and "receive" for the underlying physical channel.

I assume each message $m = (s, x)$ consists of a collection of small fields s, and a large object x, whose transmitting is to be avoided if possible. In order to achieve this, each such object x will be assigned an identifier id_x the first time it is transmitted. Subsequent transmissions will use id_x instead of x.

Each process p_i maintains two sequence numbers: idm_i and idx_i, both initially 0. Number idm_i is the next free identifier id_m assigned to the whole message $m = (s, x)$. Number idx_i is the next free id_x to be assigned to a large object x. Several messages m containing the same object x get different id_m's but the same id_x. Identifiers id_m are unique locally to the issuing process, id_x's are unique globally.

Each process also maintains the set $mappings_i$ of all known mappings (id_x, x) from object ids id_x to actual objects x. For global uniqueness, each id_x is of the form (p_i, idx_i), where p_i is the process that created the mapping (id_x, x), called the *originator* of x, and idx_i is the object sequence number assigned by p_i to x.

```
1   idm_i ← 0; idx_i ← 0; mappings_i ← ∅; messages_i ← ∅
2   when p_i ind-sends "m = (s, x)" to {q_1, ..., q_k} do
3     id_m ← idm_i; increment idm_i
4     if (id_x, x) ∉ mappings_i for any id_x then
5       id_x ← (p_i, idx_i); increment idx_i
6       insert (id_x, x) into mappings_i
7       broadcast ⟨MAP id_x, x⟩
8     send ⟨SHORT id_m, s, id_x⟩ to {q_1, ..., q_k}
9     for all p ∈ {q_1, ..., q_k} do
10      insert (id_m, s, id_x, p) into messages_i
11  when p_i removes (id_m, s, id_x, p) from messages_i do
12    let x be such that (id_x, x) ∈ mappings_i
13    send ⟨FULL id_m, s, x⟩ to p
14  when p_i removes (id_x, x) from mappings_i do
15    remove all (∗, ∗, id_x, ∗) from messages_i                {11–13}
16  when p_i receives ⟨ACK id_m⟩ or ⟨NACK id_m⟩ from p do
17    remove all (id_m, ∗, ∗, p) from messages_i
18      if ⟨ACK id_m⟩ then without triggering lines 11–13
19  when p_i receives ⟨MAP id_x, x⟩ do
20    insert (id_x, x) into mappings_i
21  when p_i receives ⟨SHORT id_m, s, id_x⟩ from p do
22    wait until (id_x, x) ∈ mappings_i for some x
23                  or timeout elapsed
24        if (id_x, x) ∈ mappings_i then
25          ind-receive "m = (s, x)"
26          send ⟨ACK id_m⟩ to p                    {at the first opportunity}
27        else
28          send ⟨NACK id_m⟩ to p
29  when p_i receives ⟨FULL id_m, s, x⟩ do
30    ind-receive "m = (s, x)"
```

Figure 1: Indirect channels

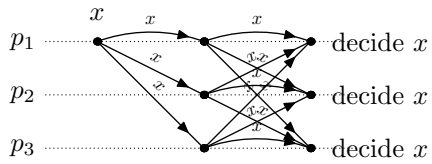

Figure 2: A round of MR Consensus [18]

When a process p_i wants to ind-send a message $m = (s, x)$ to a set of processes $q_1, \ldots, q_k$, it first generates a new locally unique id_m for m. Then, it checks whether $mappings_i$ contains (id_x, x) for some id_x. If not, p_i generates a new globally unique $id_x = (p_i, idx_i)$, adds the new mapping (id_x, x) to $mappings_i$, and broadcasts it to other processes. By doing so, p_i becomes the originator of x. When a process p_j receives this mapping, it adds (id_x, x) to $mappings_j$ (lines 19–20).

After ensuring that $mappings_i$ contains (id_x, x), process p_i sends $\langle \text{SHORT } id_m, s, id_x \rangle$ to processes $q_1, \ldots, q_k$. It also adds (id_m, s, id_x, p) to the set $messages_i$ for all recipients $p \in \{q_1, \ldots, q_k\}$.

Both sets $messages_i$ and $mappings_i$ are caches: due to their potentially large size, individual entries can be removed at any time by the cache management system. Set $messages_i$ consists of entries (id_m, s, id_x, p) stating that p_i has not received any acknowledgement from p for $\langle \text{SHORT } id_m, s, id_x \rangle$ sent to it in line 8. To avoid message loss, removing such an entry requires p_i to send the full message to p (lines 11–13). When a mapping (id_x, x) is removed from $mappings_i$, all entries in $messages_i$ containing id_x are removed as well, and the corresponding unacknowledged messages are sent in full (line 13).

When a process p_i receives $\langle \text{SHORT } id_m, s, id_x \rangle$ (line 21), it waits until it knows the mapping (id_x, x), at which point it delivers $m = (s, x)$ and sends $\langle \text{ACK } id_m \rangle$ to the sender. If this does not happen within the timeout period, it sends $\langle \text{NACK } id_m \rangle$ instead. When the sender receives one of those acknowledgement (lines 16–18), it removes the corresponding entry from $messages_i$, if it has not been removed by the cache before. If the acknowledgement is $\langle \text{NACK } id_m \rangle$, this removal triggers lines 11–13, which send the corresponding full message to the recipient. Each such message is delivered directly to the application (lines 29–30).

Behaviour in typical runs. Figure 2 shows an execution of the Mostefaoui-Raynal (MR) Consensus algorithm [18] in a typical run. In the first step, the coordinator p_1 broadcasts $m = (s, x)$, where x is its proposal and s is some short bookkeeping information. In the second step, each process p_i broadcasts $m_i = (s_i, x)$ where s_i is short. With normal channels, the potentially large proposal x is sent $O(n^2)$ times.

With indirect channels, p_1 first broadcast $\langle \text{MAP } id_x, x \rangle$ and $\langle \text{SHORT } id_m, s, id_x \rangle$ to all processes (lines 7 and 8). Each process p_i adds (id_x, x) to its $mappings_i$ (lines 19–20). In the second step, when p_i broadcasts $m_i = (s_i, x)$, it notices $(id_x, x) \in mappings_i$ (line 4), so it broadcasts only $\langle \text{SHORT } id_{m_i}, s_i, id_x \rangle$. The recipients will all eventually receive $\langle \text{MAP } id_x, x \rangle$ from p_1, reconstruct x, and deliver $m_i = (s_i, x)$. Messages $\langle \text{ACK } id_{m_i} \rangle$ will let p_i know that all processes successfully reconstructed x. Process p_i will remove the corresponding entries from $messages_i$, which will prevent broadcasting $\langle \text{FULL } id_{m_i}, s_i, x \rangle$ in the future (line 13).

As a result, in typical runs of Consensus, the full proposal x is broadcast only once. Note that, since each process must receive x somehow, this is the best one can achieve. This property of indirect channels, independent of the algorithm running on top, can be formalized as follows.

Theorem 2.1 (One-reception, see A.4). *If all message delays between correct processes are shorter than $\frac{1}{2}$timeout, and the caches never forget information, then each correct process physically receives any large object x, originated at a correct process, at most once.*

For performance reasons, $\langle \text{MAP } id_x, x\rangle$ and $\langle \text{SHORT } id_m, s, id_x\rangle$ in lines 7 and 8 are sent in a single message broadcast to all processes. If $\{q_1, \ldots, q_k\} \subsetneqq \{p_1, \ldots, p_n\}$, processes $p_j \notin \{q_1, \ldots, q_k\}$ just ignore $\langle \text{SHORT}\rangle$. Since $\langle \text{MAP } id_x, x\rangle$ is a large message, piggybacking $\langle \text{SHORT } id_m, s, id_x\rangle$ is cheap.

Behaviour in anomalous runs. In runs with message with delays longer than $\frac{1}{2}$*timeout* or caches forgetting their entries too quickly, indirect channels no longer guarantee one-reception and can send the same object multiple times. In the worst case, when entries are removed from $messages_i$ and $mappings_i$ almost immediately, each message $m = (s, x)$ will be transmitted in full in line 13. This is in addition to messages sent in lines 7 and 8, so both latency and bandwidth usage is higher in comparison to just using the underlying channels. Nevertheless, reliability of indirect channels is guaranteed in any run:

Theorem 2.2 (Reliability, see A.2). *If the underlying channel is reliable, then each message ind-sent by a correct process to another correct process will eventually be ind-received.*

Duplicate messages. If caches forget entries too quickly, indirect channels can deliver the same message twice: first in line 25 after reconstructing it from $\langle \text{SHORT}\rangle$ and later in line 29 after receiving $\langle \text{FULL}\rangle$ sent in line 13. In theory, such duplicates can be easily avoided by processes remembering id_m's of received messages. Nevertheless, I believe that duplicate-suppression is usually better done at the application level, for the following reasons.

Remembering received ids requires memory that, in the worst case, grows linearly with the number of received messages. Methods of dealing with this problem are application-specific and include imposing a near-FIFO order by using a sliding window as in TCP [23], employing Bloom filters [5], common in anonymity systems [13], and others. Algorithms that proceed in rounds or epochs will usually discard all messages from the previous ones, so storing duplicate-information can be limited to current round. Most Consensus and Atomic Broadcast algorithms are not affected by duplicates so this problem can be ignored. Finally, Section 2.2 will show that duplicates can be eliminated altogether when one is willing to accept some message loss.

Delaying acknowledgements. Sending an $\langle \text{ACK}\rangle$ immediately after receiving every message might be expensive. To mitigate this problem, a process can buffer $\langle \text{ACK}\rangle$s and piggyback them on other messages; most agreement protocols reply to received messages (almost) immediately (Figure 2). In general, the $\langle \text{ACK}\rangle$ buffering delay should be small

enough to avoid spontaneous cache entry removal at the sender (lines 11–13) and resending the message in full. The more frequently messages are sent, the sooner the cache entries get removed, and the shorter the acceptable ⟨ACK⟩ delay is. At the same time, high message frequency means a short waiting time for the opportunity for piggybacking an ⟨ACK⟩.

Consider a very simple model of this situation. Let d be the message delay, and f the frequency of each process broadcasting messages. With immediate ⟨ACK⟩s, each process's cache needs to be able to store each object for $2d$ time, which means storing $2df$ objects at any given time. When ⟨ACK⟩s are piggybacked on the next broadcast message ($1/f$ wait on average), the storage requirement grows from $2df$ to $(2d + 1/f) \cdot f = 2df + 1$. This means that using delayed ⟨ACK⟩s requires the cache to be able to store only 1 more object on average. This additional amount of required space (1 object) is constant and independent on message delay or frequency.

Cache issues. As opposed to [10], where all mappings are kept forever, entries in $mappings_i$ and $messages_i$ in the algorithm in Figure 1 can be removed at any time. The strategy for entry removal is beyond the scope of this paper; the possibilities include general strategies such as LRU or LFU [22] and application-specific techniques from distributed garbage collection [2]. If the *sender* removes an entry from $mappings_i$ too early, it might need to send more messages in lines 7 and 13. If the *receiver* removes a mapping too early, it may have to send ⟨NACK⟩s, which will also increase the latency and force the sender to broadcast full objects in lines 16–18. Removing mapping entries too late increases memory requirements for the cache.

To sum up, using a cache increases the complexity of the algorithm, however, it makes it more practical for real systems with limited memory. Most importantly, the flexibility of the cache removal strategy allows for free system-specific tuning without the fear of breaking the algorithm.

2.2 System extensions

So far I have assumed the crash-stop failure model and reliable channels. This section will relax those assumptions and present variants of the algorithm from Figure 1 for Byzantine settings and eventually reliable channels. Interestingly, operating with the latter model results in a significant simplification of the indirect channel implementation.

Eventually reliable channels. The algorithm in Figure 1 implements reliable channels, provided that the underlying channels are reliable as well. However, typical network connections are lossy, and implementing reliable channels on top of unreliable ones requires the sender to store all messages whose reception has not been confirmed. As opposed to the entries of $mappings_i$ and $messages_i$ in the indirect channel algorithm, this information cannot be freely discarded. As a result, a faulty recipient can always force the sender to use an unbounded amount of memory [3].

Fortunately, many distributed algorithms [8, 17] require channels that are only *eventually* reliable: the channels can be lossy until in a certain (unknown) time, after which they are permanently reliable. In practice, this period permanent reliability need not be infinite, but just long enough for the algorithm to do some useful work (in the order

of several message delays) [15]. Therefore, it can be assumed that standard lossy networks satisfy this definition; in the worst case, the algorithm will stall until the network is reliable for a sufficiently long period of time.

Can the algorithm in Figure 1 be used with eventually reliable channels? Yes, Corollary A.3 shows that in if the underlying channels are eventually reliable, then the indirect channels are eventually reliable as well. Moreover, significant simplifications can be made, if all entries in $mappings_i$ are kept for at least twice a message delay plus *timeout* after last access. In this case, messages $\langle$ACK$\rangle$ and $\langle$NACK$\rangle$ will always remove elements from $messages_i$ (lines 16–18) before the cache has a chance to by executing lines 14–15. We can therefore assume that then lines 11–13 need to be executed only as a result of receiving a $\langle$NACK$\rangle$ in line 15. With this observation, one can remove the set $messages_i$ entirely along with the code that references it (lines 9–15). This will require changing lines 16–18 to

```
15  on receive ⟨NACK id_m, s, id_x⟩ from p do
16    if (id_x, x) ∈ mappings_i for some x then
17      send ⟨FULL id_m, s, x⟩ to p
```

A failure of the test in line 16 means that the information about x had been removed from $mappings_i$: no retransmission is possible and the message $m = (s, x)$ will be lost.

Note that messages $\langle$NACK$\rangle$ contain now more information, so line 27 needs to be changed to

```
27          send ⟨NACK id_m, s, id_x⟩
```

One of the consequences of sending $\langle$FULL$\rangle$ only in response to an explicit $\langle$NACK$\rangle$ (line 17) is that no duplicate messages can be received. Also, since no action is taken after receiving an $\langle$ACK$\rangle$, those messages do not need to be sent at all, and line 25 can be removed. These two observations eliminate the need for explicit message duplication and $\langle$ACK$\rangle$ piggybacking discussed earlier. This considerable simplification of the algorithm assumes that all entries in $mappings_i$ are kept long enough; if they are not, messages may get lost.

Byzantine processes and hash identifiers. Since channels are point-to-point abstractions that offer no guarantees if either endpoint is malicious, the algorithm in Figure 1 handles malicious participants with almost no modifications. The only problem is posed by processes external to the sender-receiver system, namely the originator p_i of an object, because it can convince two honest processes to map the same id_x into different objects. This problem can be solved by changing the form of id_x from (p_i, idx_i) to $H(x)$, where H is a secure hash function. When a process receives a mapping (id_x, x) in line 19, it verifies whether $id_x = H(x)$, which prevents malicious processes from spreading false mappings. In fact, the new form of id_x eliminates the concept of an explicit originator from the algorithm entirely.

Hash-based message identifiers $id_x = H(x)$ can also be useful in crash-stop settings. As opposed to $id_x = (p_i, idx_i)$, the same object x sent by different processes gets the same id_x. For example, if all processes propose the same x to a Consensus instance, processes can reconstruct x from received id_x without having to wait for the possible high-latency message $\langle$MAP$\rangle$ broadcast in line 7. The downside of hash-based identifiers might be their size, computational cost, and the probabilistic guarantees they offer.

Dynamic groups. The broadcast in line 7 assumes a system consisting of a fixed set of n processes. In more dynamic settings, the mapping $\langle \text{MAP } id_x, x \rangle$ should be broadcast only to the intended recipients of x, direct or indirect, as determined by the application. The originator of x includes this set R of the recipients in id_x. Later, when some process re-sends x to a process $p \notin R$, for example if p joined the group just after the originator created id_x, then x is sent directly using $\langle \text{FULL} \rangle$. Such situations are atypical and should not have much impact on the overall performance.

Note on failure detectors. The algorithm in Figure 1 uses timeouts. I believe that failure detectors [6] are inadequate for implementing indirect channels because (i) they too powerful but (ii) not powerful enough.

First, classic failure detectors cannot be implemented in Byzantine settings [9]. Second, failure detectors cannot be used to implement indirect channels over eventually reliable channels. This is because per-process failure detection offered by failure detectors is not fine-grained enough to deal with individual messages being lost between correct processes (Theorem B.2). Failure detectors can therefore only be used with reliable channels.

To implement reliable indirect channels, one cannot use any of the simplifications described in earlier in this section: the original algorithm in Figure 1 must be used. This algorithm uses timeouts to ensure that processes do not wait indefinitely for messages that might never arrive (line 23). The downside of this approach is that if the timeout is too short, then unnecessary $\langle \text{NACK} \rangle$ and $\langle \text{FULL} \rangle$ messages are sent. This decreases the performance, but crucially, it does *not* affect the correctness of the algorithm (safety or liveness). In other words, indirect channels can be implemented correctly in purely asynchronous settings.

Now, failure detectors are too powerful to be implemented a purely asynchronous system. This is because they need to satisfy two families of properties: completeness (eventually all faulty processes suspected) and accuracy (eventually no correct processes suspected). Typical problems solved with failure detectors, such as Consensus or Atomic Broadcast, require both properties to achieve correctness. On the other hand, indirect channels require only a timeout equivalent of completeness, not accuracy. Thus, using failure detectors is overkill, and would make a wrong impression about the solvability of the problem itself.

2.3 Applications

Indirect channels can be used to improve throughput and latency of any distributed algorithm that tend to transmit the same object often, in particular Consensus and Atomic Broadcast protocols. For the latter case, Ekwall and Schiper [10] designed a specialized protocol and showed theoretically and experimentally that it improves both latency and throughput.

Following Ekwall and Schiper [10], I conducted an experimental evaluation of indirect channels using Neko [24], a Java framework for prototyping and evaluating distributed algorithms. Three Atomic Broadcast protocols were tested, with message payloads ranging from 100 to 10,000 bytes. Each protocol-payload combination was run for 60 seconds with three Ethernet-connected computers, one of which was constantly abcasting 50 messages with the given payload per second. The average latencies are shown in Figure 3.

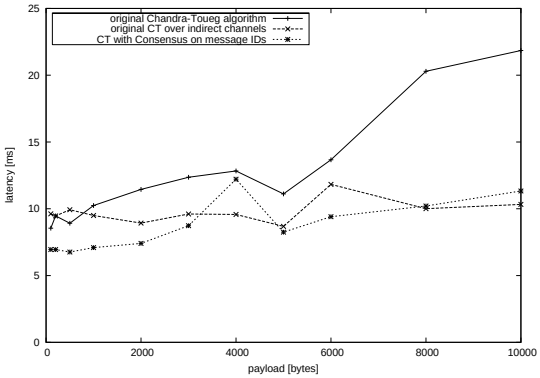

Figure 3: Experimental evaluation of three Atomic Broadcast protocols

The first Atomic Broadcast algorithm is the standard Chandra and Toueg (CT) protocol [6] which embeds the payload in all its messages. The second protocol is also CT, but run over indirect channels, broadcasting each payload only once. The Neko implementation of [10] is not publicly available yet, so instead I tested a simplified but unsafe version of it, essentially CT with Consensus on message ids instead of full messages. The simplified version cannot be slower than the full algorithm [10], which is sufficient for this comparison.

Results. For small payload sizes, the latencies of all three protocols were similar. For the payload size over 5kB, the latency of CT started growing, whether that of the other two protocols remained relatively constant. This is because CT embeds the large payload in every message, whereas the other two algorithm broadcast the full payload only once. As a result, their latencies are dominated by small messages and are not significantly affected by the payload size.

3 Further improvements in latency

The indirect channel implementation in Figure 1 improves network usage by transmitting ids instead of full messages. As shown in [10] and here, this technique reduces both bandwidth usage and latency. Note that no further reduction in bandwidth is possible because each recipient must receive the full message at least once, and this is what indirect channels guarantee in typical runs (one-reception property).

The optimality of the resulting latency is not that clear. Indirect channels are conservative in that they do not allow the application to manipulate the message id before the full message is known. Without this assumption, failures in the system could force the application to decide on an *orphaned id* without a corresponding message [10]. This section will investigate whether the latency of Consensus and Atomic Broadcast can be further improved by processing short and long messages in parallel.

In the standard latency model, in which all messages have the same transmission delay d, the latency of those two abstractions is fairly well studied. Asynchronous Consensus requires at least two communication steps ($2d$) [7, 16] and several algorithm achieve that bound [6, 18]. Similarly, asynchronous Atomic Broadcast requires three steps ($3d$) [25], again with many matching algorithms [6].

I am not aware of any previous theoretical study of the impact of different size-dependent message delays on the latency of agreement protocols. This section attempts to shed some light on the problem by considering the following simple model. Assume there are only two kinds of messages: small with a transmission delay d and large with a longer delay of $D > d$. I will investigate the latency of Consensus and Atomic Broadcast in this model, by presenting relevant algorithms and lower bounds.

3.1 Consensus

In the Consensus problem, all processes propose and then agree on one of the proposals [14]. The following conditions should be met: (i) only proposals can become decisions, (ii) no two processes decide differently, (iii) all correct processes eventually decide.

Even in failure-free runs, Consensus requires two communication steps [7, 16] (Figure 2). If proposals are "large", then so are all used messages, and the resulting latency is $2D$. If run over indirect channels, the second step of a Consensus algorithm [6, 18] uses short ids instead of full messages, thereby reducing the overall latency to $d + D$. This is an upper bound.

As for a lower bound, two communication steps are needed in any case, so the latency cannot be lower than $2d$. At the same time, all processes must receive the winning proposal, which gives us a lower bound of $\max\{2d, D\}$.

The discrepancy between those bounds ($\max\{2d, D\} < d + D$) poses a question whether there is a Consensus algorithm that cleverly manipulates the message ids while waiting for the full messages, and then immediately decides. The answer is "no". Theorem B.3 states that in any agreement problem, the decision value must be known, but not committed, one communication step (d units of time) in advance. Since large messages are known only after D units of time, the total latency cannot be reduced below $D + d$.

Indirect Consensus [10] and Consensus over indirect channels achieve the same latency, however, the latter approach works with any Consensus algorithm, without modifications. In particular, it tolerates the same number of faulty participants as in the original version. In comparison, the indirect version of MR Consensus [10] requires less than a third processes faulty ($n > 3f$), despite the original tolerating less than a half ($n > 2f$) [18]. The reason for this discrepancy is that, with indirect channels, ids without corresponding full messages do not count towards the $n - f$ quorum in the second phase of MR, preventing orphaned ids.

3.2 Atomic Broadcast

In Atomic Broadcast, processes broadcast messages, and then deliver all of them in the same order [8]. Formally, we require the following properties: (i) if a correct process broadcast a message m, then all correct processes will eventually deliver m; (ii) if a process delivers a message m, then all correct processes eventually deliver m; (iii) for any message m, every process delivers m at most once, and only if m was previously abcast; and (iv) if some process delivers message m' after message m, then every process delivers m' only after it has delivered m.

Standard Atomic Broadcast protocols require three communication steps, with large messages used in all three steps (latency $3D$). Running such protocols over indirect

channels limits the large messages to the first step only, thereby reducing the latency to $D + 2d$. This is an upper bound.

As for a lower bound, we can use the same reasoning as in Section 3.1. Three communication steps are needed ($3d$), and the messages must be known one step before delivery ($D + d$) (Theorem B.3). The resulting lower bound $\max\{3d, D + d\} < D + 2d$ leaves us with the same question as before: is there an algorithm that attains it?

This time the answer is "yes" and [10] provide an example. In the second step of their Atomic Broadcast algorithm, processes can propose ids to Indirect Consensus without knowing the full messages. Achieving this with indirect channels is not straightforward because they do not allow orphaned id manipulation. One might try to modify some indirect-channel-based Consensus algorithm to allow such manipulations, but the result would be algorithm-specific.

Instead, this section shows how the latency of any three-step Atomic Broadcast algorithm can be reduced to $\max\{3d, D + d\}$, without affecting its fault-tolerance (unlike MR-based [10]). My approach is to split the Atomic Broadcast problem into two independent problems:

1. Agreement on the order of delivered message ids, which can be solved by Atomic Broadcast on message ids (latency $3d$).

2. Agreement on the mapping from message ids to full messages, which can be solved by Generic Broadcast [4, 21] over indirect channels (latency $D + d$).

Assuming the implementations of the two above broadcast abstractions, as well as two failure detectors: Ω and $\Diamond S$ [6], I will now present an Atomic Broadcast algorithm with the latency of $\max\{3d, D + d\}$ in typical runs.

3.2.1 Algorithm

The details of the algorithm are shown in Figure 4. Each process p_i maintains three variables: the next message sequence number idm_i, and two sets of mappings from ids to messages: the set $mymappings_i$ of mappings proposed by p_i and the set $mappings_i$ of globally accepted mappings. Both sets $mymappings_i$ and $mappings_i$ are initially empty.

To broadcast a large message m, process p_i first generates a new unique $id_m = (p_i, idm_i)$ and adds the mapping (id_m, m) to $mymappings_i$. Process p_i then broadcasts $\langle \text{ID } id_m \rangle$ using Atomic Broadcast, and the mapping $\langle \text{MAP } id_m, m \rangle$ using Generic Broadcast. Generic Broadcast [4, 21] is a version of Atomic Broadcast that is especially fast when messages do not *conflict*. In our case, two messages $\langle \text{MAP } id_m, m \rangle$ conflict if they map the same id_m into different messages m. In typical runs, such conflicts do not occur because the id_m's assigned by processes are unique. In runs with failures, multiple mappings with the same id_m may be sent, as we will see later.

Whenever a process p_i receives $\langle \text{MAP } id_m, m \rangle$, it adds the mapping (id_m, m) to $mappings_i$, unless $mappings_i$ contains a mapping for id_m already. Since all processes receive multiple mappings for a given id_m in the same order, Generic Broadcast ensures that all processes end up mapping a given id_m to the same message.

The delivery task at each process p_i consists of an infinite loop. In each iteration, p_i first waits for the id_m of a message, atomically broadcast in line 5. Then, it waits for the mapping for id_m to become known (by executing line 9). If, during this waiting,

```
1   idm_i ← 0; mappings_i ← ∅; mymappings_i ← ∅
2   when p_i abcasts m do
3     generate new id_m = (p_i, idm_i); increment idm_i
4     insert (id_m, m) into mymappings_i
5     broadcast ⟨ID id_m⟩ with Atomic Broadcast
6     broadcast ⟨MAP id_m, m⟩ with Generic Broadcast
7   when p_i gbdelivers ⟨MAP id_m, m⟩ do
8     if (id_m, m') ∉ mappings_i for no m' then
9       insert (id_m, m) into mappings_i
10  task delivery at p_i is
11    loop forever
12      wait until ⟨ID id_m⟩ is abdelivered from some p
13      execute                          {interrupt lines 13–16 when until holds}
14        wait until p_i is the Ω-leader and suspects p
15        broadcast ⟨MAP id_m, ⊥⟩ using Generic Broadcast
16      until (id_m, m) ∈ mappings_i for some m
17      if m ≠ ⊥ then
18        deliver m
19      else
20        send ⟨RETRY id_m⟩ back to p
21  when p_i receives ⟨RETRY id_m⟩ do
22    if (id_m, m) ∈ mymappings_i for some m then
23      remove (id_m, m) from mymappings_i
24      abcast(m)                                          {lines 2–6}
```

Figure 4: Atomic Broadcast

p_i becomes the Ω-elected leader and suspects the sender of id_m, it broadcasts an empty mapping $\langle \text{MAP } id_m, \bot \rangle$. This ensures that id_m will be mapped to some value ($\bot$ or m), and the potential failure of the sender of id_m will not prevent the system from delivering other messages.

When the mapping for id_m becomes finally known, p_i checks whether it is empty or not. If it is not empty, then the message m is atomically delivered. If not, the process sends a request to the sender to retry broadcasting the message (with a new id_m). If the failure detector is $\Diamond P$, then each correct sender will eventually not be suspected, and its broadcasts will succeed.

With weaker failure detectors ($\Diamond S$), this is not guaranteed. In this case, the sender should respond to $\langle \text{RETRY } id_m \rangle$ by abcasting $\langle \text{FULL } m \rangle$ using the underlying Atomic Broadcast directly, instead of executing lines 2–6. If in line 12 a process receives $\langle \text{FULL } m \rangle$ instead of $\langle \text{ID } id_m \rangle$, it should deliver m straightaway and start a new iteration of the **loop**.

```
1   when process p_i gbcasts m do
2     broadcast m using Reliable Broadcast
3   when p_i receives m do
4     broadcast ⟨CONFLICTS m, conf_m⟩ where conf_m is
5     the sequence of previously received m' conflicting with m
6   when p_i receives n − f msgs ⟨CONFLICTS m, conf_m⟩ do
7     if m is regular and conf_m is empty then
8       deliver m unless delivered before
9     else
10      for all regular m' in all conf_m do
11        FIFO-abcast m'
12      FIFO-abcast m
13  when p_i abdelivers m do
14    deliver m unless delivered before
```

Figure 5: Two-Class Generic Broadcast

3.2.2 Latency analysis

Consider the latency of the algorithm in runs in which no process fails during abcasting its messages. In such runs, Atomic Broadcast of $\langle \text{ID } id_m \rangle$ takes $3d$ units of time, and Generic Broadcast of $\langle \text{MAP } id_m, m \rangle$ over indirect channels takes $D + d$ because the only possible conflicting mapping $\langle \text{MAP } id_m, \perp \rangle$ is never sent. As a result, all messages m are atomically delivered in $\max\{3d, D + d\}$ time. This is a lower bound, as shown before.

One problem with the above analysis is that Generic Broadcast can be implemented with a two-step latency only if less than a third of processes are faulty ($n > 3f$) [20]. Ekwall and Schiper [10] report the same requirement in order to use a two-step MR Consensus algorithm [18]. Is the condition $n > 3f$ necessary for achieving the $\max\{3d, D + d\}$ latency?

No. This section shows a variant of Generic Broadcast that delivers messages in two steps while requiring only $n > 2f$. Messages $\langle \text{MAP} \rangle$ in Figure 4 can be divided into two classes: *regular* $\langle \text{MAP } id_m, m \rangle$ and *special* $\langle \text{MAP } id_m, \perp \rangle$. Note that: (i) regular messages never conflict with each other, and (ii) only regular messages need to be delivered quickly. Another example of regular and special messages can be read and write requests in systems in which writes are rare.

Figure 5 shows an implementation of Two-Class Generic Broadcast. To broadcast a message m, a process sends it using reliable broadcast [15]. Upon receiving m, each process rebroadcasts it along with the sequence $conf_m$ of previously received messages conflicting with m. When a process receives $n - f$ such messages, it first checks whether m is regular and all $conf_m$'s are empty, and delivers m if so. Otherwise, all regular messages in all $conf_m$'s are broadcast using an underlying FIFO Atomic Broadcast protocol, followed by m itself. All duplicates are explicitly removed on delivery.

Two messages m_1 and m_2 can conflict only if one of them is special. They can therefore be delivered in different orders at different processes only if (i) exactly one of them, say

m_1, is regular and is delivered in line 8, and (ii) some process abcasts m_2 in line 12 without abcasting m_1 in line 11. The first means that $n - f$ processes received m_1 before m_2, the second that $n - f$ processes received m_2 before m_1. This contradicts the assumption that $n > 2f$. Note that no Atomic Broadcast is used in the absence of conflicting messages.

Using Two-Class Broadcast over indirect channels, allows one to reduce the latency of any three-step Atomic Broadcast implementation to $\max\{3d, D + d\}$ without reducing its fault-tolerance. This should be contrasted with the indirect version of MR-based Atomic Broadcast, which reduces fault-tolerance from $n > 2f$ to $n > 3f$ [10].

3.3 Other broadcast abstractions

Optimistic Atomic Broadcast [19] and Generic Broadcast [4, 21] require two communication steps ($2D$) in runs with spontaneous order and no conflicts, respectively. Using indirect channels, the latency can be reduced to $D + d$, which is also a lower bound by Theorem B.3.

4 Conclusion

Excessive bandwidth usage caused sending the same large messages several times can be reduced by transmitting their ids whenever possible . Although this method has been widely used in many real-world point-to-point protocols [1, 11], applying it to fault-tolerant group protocols is surprisingly difficult [10]. For both families, the existing solutions are problem-specific.

This paper proposed a more general solution by providing virtual *indirect channels* that physically send a full message only during its first transmission; all other transmission use the message id only. Indirect channels are transparent to the application and have no latency overhead in typical runs. As a result, they can be used with any distributed algorithm, without modifications, even with unreliable channels and malicious participants. While providing rigorous theoretical properties at all times, the implementation lends itself to system-specific tuning through cache configuration, making it attractive for practical systems.

Running Consensus algorithms over indirect channels produces protocols with optimum latency and fault-tolerance. In the case of Atomic Broadcast, the latency of any implementing protocol can be further improved by handling message ordering and id mapping separately. The resulting latency and fault-tolerance are lower bounds, however, the problem whether this can be achieved just by using more sophisticated indirect channels is an open question.

References

[1] RSync. `http://samba.anu.edu.au/rsync/`.

[2] S. E. Abdullahi and G. A. Ringwood. Garbage collecting the Internet: A survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[3] Yehuda Afek, Hagit Attiya, Alan Fekete, Michael Fischer, Nancy Lynch, Yishay Mansour, Dai-Wei Wang, and Lenore Zuck. Reliable communication over unreliable channels. *Journal of the ACM*, 41(6):1267–1297, 1994.

[4] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Thrifty Generic Broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 268–282, Toledo, Spain, 2000.

[5] B. H. Bloom. Space-time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), July 1970.

[6] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[7] Bernadette Charron-Bost and André Schiper. Uniform Consensus is harder than Consensus. Technical Report DSC/2000/028, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, May 2000.

[8] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[9] Assia Doudou, Benoît Garbinato, and Rachid Guerraoui. Encapsulating failure detection: From crash to Byzantine failures. In *Proceedings of the 7th International Conference on Reliable Software Technologies*, pages 24–50, June 2002.

[10] Richard Ekwall and André Schiper. Solving atomic broadcast with indirect consensus. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006)*, pages 156–165. IEEE Computer Society, 2006. ISBN 0-7695-2607-1.

[11] C. Feather. Network News Transfer Protocol (NNTP). RFC 3977 (Proposed Standard), October 2006. URL `http://www.ietf.org/rfc/rfc3977.txt`.

[12] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998.

[13] GNUnet. Gnunet: decentralized anonymous and censorship-resistant p2p framework, 2006. URL `http://gnunet.org/`.

[14] Rachid Guerraoui, Michel Hurfin, Achour Mostéfaoui, R. Oliveira, Michel Raynal, and André Schiper. Consensus in asynchronous distributed systems: A concise guided tour. In S. Shrivastava and S. Krakowiak, editors, *Advances in Distributed Systems*, number 1752 in Lecture Notes in Computer Science, pages 33–47. Springer, 2000.

[15] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.

[16] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant Consensus when there are no faults. *ACM SIGACT News*, 32, 2001.

[17] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[18] Achour Mostéfaoui and Michel Raynal. Solving Consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 49–63, London, UK, 1999. Springer-Verlag.

[19] Fernando Pedone and André Schiper. Optimistic Atomic Broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79–101, 2003.

[20] Fernando Pedone and André Schiper. On the inherent cost of Generic Broadcast. Technical Report IC/2004/46, Swiss Federal Institute of Technology (EPFL), May 2004.

[21] Fernando Pedone and André Schiper. Generic Broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 94–108, 1999.

[22] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, 2003. ISSN 0360-0300.

[23] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001. URL `http://www.ietf.org/rfc/rfc3168.txt`.

[24] Péter Urbán, Xavier Défago, and André Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *Proceedings of the 15th Int'l Conf. on Information Networking (ICOIN-15)*, Beppu City, Japan, 2001.

[25] Piotr Zieliński. Low-latency Atomic Broadcast in the presence of contention. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, September 2006.

A Correctness proofs

A.1 Indirect channels

Lemma A.1. *If* $(id_m, s, id_x, p) \in messages_i$ *then* $(id_x, m) \in mappings_i$ *for some* m*.*

Proof. True at the beginning, we need to prove that every action preserves this property. Removing (id_x, x) from $mappings_i$ removes all $(*, *, id_x, *)$ from $messages_i$ (line 15). Adding (id_m, s, id_x, p) to $messages_i$ can only happen after ensuring that $(id_x, x) \in mappings_i$ (line 10). □

Theorem A.2 (Reliability). *If the underlying channel is reliable, then each message* $m = (s, x)$ *ind-sent by a correct process* p_i *to another correct process* p_j *will eventually be ind-received.*

Proof. Executing ind-send will add (id_m, s, id_x, p_j) to $messages_i$, and send $\langle \textsc{short}\ id_m, s, id_x \rangle$ to p_j, which p_j will eventually receive. Because of the timeout involved, **wait** in lines 22–23 will eventually terminate. If $(id_x, x) \in mappings_j$, then $m = (s, x)$ will be delivered. Otherwise, $\langle \textsc{nack}\ id_m \rangle$ will be sent back to p_i, which will remove (id_m, s, id_x, p_j) from $messages_i$ if it is there. Therefore, (id_m, s, id_x, p_j) is at some point removed from $messages_i$, which involves sending $\langle \textsc{full}\ id_m, s, x \rangle$ to p_j (line 13), so p_j will ind-receive it. If lines 11–13 were not triggered, then p_i received $\langle \textsc{ack}\ id_m \rangle$ from p_j (line 18), which means that p_j ind-received $m = (s, x)$. □

Corollary A.3 (Eventual reliability). *If the underlying channel is* eventually *reliable,* eventually *each message* m *indirect-sent by a correct process to another correct process will eventually be ind-received.*

Theorem A.4 (One-reception). *If all message delays* d *between correct processes satisfy* $2d < timeout$*, and the caches never forget information, then each correct process physically receives any large object* x*, originated at a correct process, at most once.*

Proof. It is sufficient to to show that the only time object x is physically sent is by the originator p_k in line 7, as part of $\langle \textsc{map}\ id_x, x \rangle$. This mapping will be in $mappings_k$ forever. No other process p_i can send x without indirect-receiving it first in line 26, which requires $(id_x, x) \in mappings_i$. Therefore, any subsequent execution of ind-send will encounter $(id_x, x) \in mappings_i$ and line 7 will not be executed.

Assume the originator broadcasts the initial $\langle \textsc{map}\ id_x, x \rangle$ at time 0. All correct processes receive this mapping by time d. Any id_x sent in line 8 at time $t \geq 0$, will arrive at the recipient p at a time t' between t and $t + d$. Since $t' + timeout > t + 2d > d$, lines 22–23 will never time out. Similarly, $\max\{t', d\} \leq t + d$, so the $\langle \textsc{ack}\ id_x \rangle$ will arrive back at the sender by time $t + 2d < t + timeout$, and (id_m, s, id_x, p) will be removed from mappings without triggering lines 11–13. No $\langle \textsc{nack} \rangle$s will ever be sent and caches hold information long enough, so no removal attempt will be made before. As a result, line 13 will never be executed. □

Theorem A.5 (Byzantine validity). *If an honest process* p_i *ind-received* $m = (s, x)$*, and the sender is honest, then it has indirect-sent* $m = (s, x)$*.*

Proof. The sender must have sent $\langle \text{SHORT } id_m, s, id_x \rangle$ such that $(id_x, x) \in mappings_i$. An honest process puts (id_x, x) in *mappings* either in line 6 or in line 20, both cases ensuring that $id_x = H(x)$. Assuming no hash collisions, two honest processes cannot resolve the same id_x to a different message. If an honest process ind-receives $m = (s, x)$ from another honest process, then the sender must have sent $\langle \text{SHORT } id_m, s, id_x \rangle$ in line 8, after executing ind-send m in line 2. □

A.2 Atomic Broadcast

Lemma A.6. *If $(id_m, m_i) \in mappings_i$ and $(id_m, m_j) \in mappings_j$, then $m_i = m_j$.*

Proof. If $m_i \neq m_j$, then the assumption implies that p_i gbdelivered (id_m, m_i) before (id_m, m_j), but p_j gbdelivered (id_m, m_j) before (id_m, m_i). Since (id_m, m_j) conflicts with (id_m, m_i), this contradicts the Uniform Partial Order property of the underlying Generic Broadcast. □

Lemma A.7. *Lines 13–16 will eventually terminate at any correct process.*

Proof. The Total Order property of the underlying Atomic Broadcast implies that all processes abdeliver id_m's in line 12 in the same order. For the sake of contradiction, consider the first id_m for which lines 13–16 do not terminate at some correct process p_i.

If the sender p_j is correct, then it has gbcast $\langle \text{MAP } id_m, m \rangle$ in line 6, so eventually $(id_m, m') \in mappings_i$ for some m', which implies the assertion. Therefore, assume p_j is faulty. By the choice of id_m and Uniform Agreement of Atomic Broadcast, all correct processes executed line 12 for id_m. By the choice of id_m and Uniform Agreement of Generic Broadcast, lines 13–16 will terminate for id_m at no correct process. Therefore, eventually a correct leader will gbcast $\langle \text{MAP } id_m, \perp \rangle$ in line 15. By Validity of Generic Broadcast, lines 13–16 will terminate at all correct processes, which proves the assertion. □

Theorem A.8 (Validity). *If a correct process broadcast a message m, then all correct processes will eventually deliver m.*

Proof. Validity of the underlying Atomic Broadcast implies that all correct processes will eventually abdeliver id_m in line 12. Lemma A.7 ensures that lines 13–16 will eventually terminate. If no process suspects the sender, then $\langle \text{MAP } id_m, \perp \rangle$ has never been gbcast in line 15, and Integrity of Generic Broadcast implies that $\langle \text{MAP } id_m, m \rangle$ gbcast in line 6 is the only possible in line 7. Therefore $(id_m, m) \in mapping_i$ in line 17, and m is delivered.

If the sender is suspected, then the condition in line 17 might not hold. In this case, the sender will receive $\langle \text{RETRY } id_m \rangle$ in lines 21–24 and repeat the abcasting procedure in lines 2–6, with a new id_m. Eventual Accuracy of $\Diamond P$ ensures that a correct sender will eventually not be suspected, so some repetition of lines 2–6 will eventually succeed in atomically delivering m in line 18. □

Theorem A.9 (Uniform Agreement). *If a process p_i delivers a message m, then all correct processes eventually deliver m.*

Proof. If p_i abdelivers $id_{m_1}, id_{m_2}, \ldots, id_m$ in line 12, then Lemma A.7 and Uniform Agreement of the underlying Atomic Broadcast ensure that all correct processes will do so as well. Since $(id_m, m) \in mappings_i$, Uniform Agreement of Generic Broadcast and Lemma A.6 ensure that eventually $(id_m, m) \in mappings_j$ at all correct processes p_j. This implies the assertion. □

Theorem A.10 (Uniform Integrity). *For any message m, every process delivers m at most once, and only if m was previously abcast.*

Proof. Let p be the abcaster of m and $id_1, ...$ be ids assigned to m by p in line 4. Any id_{k+1} is assigned to m as a result of p receiving "retry id_m", which is sent only if $(id_k, \perp) \in mappings_i$ at some process p_i. By Lemma A.6 (id_k, m) cannot hold at any process p_i. The only id_k for which $(id_k, m) \in mappings_i$ can hold is the last in the sequence $id_1, ...$ (if it exist). This proves the first part of the assertion.

For the second part, if m is delivered (line 18), then $(id_m, m) \in mappings_i$ for some id_m. This means that $\langle \text{MAP } id_m, m \rangle$ was gbcast in line 6, which implies the conclusion.□

Theorem A.11 (Uniform Total Order). *If some process p_i delivers message m' after message m, then every process p_j delivers m' only after it has delivered m.*

Proof. Let $id_{m_1}, ..., id_m, ..., id_{m'}, ...$ be the sequence of ids delivered by process p_i in line 12. Process p_j delivered $id_{m'}$, so Uniform Total Order of the underlying Atomic Broadcast implies that p_j must have delivered id_m, which then passed the test $(id_m, m'') \in mappings_j$ in line 16 for some m''. Since $(id_m, m) \in mappings_i$, Lemma A.6 implies $m'' = m$, which implies the conclusion. □

A.3 Two-class Broadcast

Lemma A.12. *If all $n - f$ correct processes receive m in line 3, then all correct processes will eventually deliver m.*

Proof. Those processes broadcast $\langle \text{CONFLICT } m, conf_m \rangle$, so all correct processes will execute lines 6–12. If m is regular and all $conf_m$ at all correct processes are empty, then all of them will deliver m in line 8. Otherwise, some correct process will abcast m in line 12, which will eventually be delivered by all correct processes. □

Theorem A.13 (Validity). *If a correct process broadcasts a message m, then all correct processes will eventually deliver m.*

Proof. The assumption implies that all $n - f$ correct processes will receive m; the conclusion follows from Lemma A.12. □

Theorem A.14 (Uniform Agreement). *If a process delivers a message m, then all correct processes eventually deliver m.*

Proof. If the delivery occurs in line 14, then the assertion follows from Uniform Agreement of the underlying Atomic Broadcast. If the delivery occurs in line 8, then $n - f$ processes broadcast $\langle \text{CONFLICT } m, conf_m \rangle$, at least one of them correct ($n > 2f$). Reliable broadcast used in line 2 ensures that eventually all $n - f$ correct processes will receive m. Lemma A.12 implies the assertion. □

Theorem A.15 (Uniform Integrity). *For any message m, every process delivers m at most once, and only if m was previously abcast.*

Proof. Explicit duplicate elimination ensures the first part of the assertion. For the second part, if m is delivered, some process must have received $\langle \text{CONFLICTS } m, conf_m \rangle$ in line 6, so some process must have received m in line 3, which implies the conclusion. □

Theorem A.16 (Uniform Partial Order). *If some process delivers message m_2 after a* conflicting *message m_1, then every process delivers m_2 only after it has delivered m_1.*

Proof. If none of those messages is delivered in line 8, then the conclusion follows from Uniform Total Order of the underlying Atomic Broadcast. If one of the messages (m) is delivered in line 8, then it is regular, and the other one (m') must be special (because regular messages do not conflict). Moreover, at least $n - f$ processes must have received m before m' (empty $conf_m$), so each process abcasting m' has at least one $m \in conf_{m'}$, so it abcasts m before. The FIFO property implies that all processes deliver m before m'.□

B Impossibility results

Definition B.1 (One-reception for failure detectors). *If all processes are correct, there are no suspicions, and the caches never forget information, then each correct process physically receives any large object x at most once.*

Theorem B.2. *If the underlying channel can lose messages, then one-reception cannot be achieved with standard failure detectors.*

Proof. Assume all caches hold all information forever, all processes are correct, and none are ever suspected. Consider a run in one process p ind-broadcasts a large message m every second or so. In this run, the actual message m, which should be physically broadcast at most once, arrives at all other processes at time t. Call this run $r(t)$.

Now consider another run r' in which this single broadcast is lost; note that r' does not depend on t. To guarantee eventual reliability, all processes will eventually, say at time t', have to start delivering messages m broadcast by p. Therefore, all processes will have to receive m by time t'. Consider any run $r(t)$ with $t > t'$. These runs are indistinguishable until time t, so they both receive m at time t', but $r(t)$ also receives m at time $t > t'$. This violates the one-reception property. □

Theorem B.3. *Time $D + d$ is required for asynchronous agreement protocols.*

Proof. To obtain contradiction, consider an algorithm that decides before $D + d$ in every execution in which all processes are correct and there is no message loss. With broadcast protocols, assume that process p_1 broadcasts m_1 at time 0, and no other broadcast takes place. For Consensus, assume each process p_i proposes a different value m_i at time 0; without loss of generality we can assume that all processes decide on m_1. For both, broadcasts and Consensus, the decision (delivery) must happen before time $D + d$.

Consider a scenario in which p_1 is faulty, and all large messages sent by p_1 to other processes are lost (but no others). This scenario is indistinguishable from the previous one

to processes other than p_1 until time D, and to process p_1 until time $D + d$. As a result, process p_1 decides on m_1 before time $D + d$, and say crashes at time $D + d$. No other process will ever receive m_1 but it will be required to decide on it by Uniform Agreement. This is a contradiction. □