# Exactly-once semantics in a replicated messaging system


## Abstract

A distributed message delivery system can use replication to improve performance and availability. However, without safeguards, replicated messages may be delivered to a mobile device more than once, making the device's user repeat actions (e.g., making unnecessary phone calls, firing weapons repeatedly). In this paper we address the problem of exactly-once delivery to mobile clients when messages are replicated. We define exactly-once semantics and propose algorithms to guarantee it. We also propose and define a relaxed version of exactly-once semantics which is appropriate for limited capability mobile devices. We study the relative performance of our algorithms compared to weaker at-least-once semantics, and find that the performance overhead of exactly-once can be minimized in most cases by careful design of the system.


## 1 Introduction

In a *replicated messaging system*, messages are replicated at various servers, pending delivery to mobile clients. When the client makes a connection to the network, it contacts any of the servers, and downloads its pending messages. If messages are not replicated, the client could not get its messages when its only server was down. Furthermore, the connection to its server may be slow, making it impossible to download all the messages in a reasonable time. Both of these problem are solved with replication: the client has a choice of servers, and can contact one that is "nearby" at the time of connection and that offers good response time.

Message replication is useful in critical applications where it is important to deliver messages as soon as a client makes any connection. For instance, in a military scenario, it is critical to get messages to units in the field, no matter at what point they connect, and no matter how short their connection is. Even for non-critical applications, replication can be very useful. For example, when an American traveler visits Europe, it would be much more convenient to access his or her emails through a local server, rather than relying on a slow trans-Atlantic connection.

Replication, of course, introduces the problem of duplicate delivery. A client can connect to one server and download message $M_1$, and later connect to a different server and receive $M_1$ a second time. Duplicate delivery can range from "disastrous" to simply "annoying." For example, a message "fire missile" or "buy stock" may cause an action to be unintentionally performed twice with clearly undesirable consequences. In other cases, a duplicate message to "call Fred back" can be simply confusing, since the recipient does not know whether Fred wants to talk to him/her a second time.

In this paper we study mechanisms for ensuring that messages are delivered *exactly once* (no delivery and duplicate delivery must be ruled out). While this problem has been extensively studied in the past (see our related work section), we believe it is important to revisit the problem in the context of a replicated message system for mobile clients. In particular, we will

- Study how servers with replicas can coordinate to eliminate or reduce the likelihood of duplicate delivery.
- Explore duplicate elimination mechanisms that may be appropriate for clients with no or limited stable storage.
- Study "weaker" exactly-once semantics that may be useful in a mobile environment.
- Present an evaluation metric for comparing exactly-once mechanisms, useful for highlighting their weaknesses or strengths in a mobile environment.

We stress that we are not discovering "new" mechanisms for exactly-once delivery. All of the ideas we present (e.g., sequence numbers to identify duplicates) are well known. Our goal is instead to adapt these well known "building blocks" into mechanisms that are well suited for a mobile environment, and to evaluate the tradeoffs.

We also stress that not all applications require strict exactly-once semantics. However, if we can find a mechanism that guarantees exactly-once delivery at a reasonable performance and complexity cost, then we may still want to use it. Thus, we believe that regardless of the application, it is important to understand how exactly-once delivery can be achieved with replicated messages, so that an informed decision can be made regarding message delivery options.

Finally, note that some applications may not require exactly-once delivery because they implement their own safeguards. For example, the message to fire a missile may specify which missile to fire, so that a duplicate message will not cause a second firing. However, these applications mechanisms (e.g., missile numbers, application sequence numbers) are identical to the mechanisms we describe here. Thus, whether exactly-once is guaranteed by the messaging system or by the application, the issues are the same, and the results we present here are relevant.

The structure of our paper is as follows. We start in Section 2 by defining more precisely our framework and the notion of exactly-once delivery. The algorithms for guaranteeing exactly-once semantics are given in Section 3. Section 4 describes our performance model, while Section 5 presents selected comparisons of the strategies. We discuss related work in Section 6 and conclude in Section 7.

## 2 Delivery semantics

Figure 1 illustrates a replicated messaging system. Servers are located at various locations around the globe. The servers are connected by a fixed wide area network, e.g., the Internet. These servers are responsible for storing and delivering messages to clients. We call these messages *notes* to distinguish them from other network traffic that does not require delivery guarantees (e.g., exactly-once semantics). When a new note is generated, it is replicated to all servers for delivery to the client. The replication happens under the guidance of a separate replication protocol, which we do not consider in detail here.

A client, which is typically a wireless information access device like a PDA, stays disconnected from the server network most of the time. It occasionally establishes a wireless connection into a network access point, which then lets it communicate with one of the servers to get its notes. Because client connections can be short lived, or may not provide good connectivity to all servers, it is important to replicate notes at multiple servers. This way, when a connection is made, note delivery can be made from the server that offers best service. However, without any safeguards, a note can be delivered more than once to a client by different servers.

The semantics of a note specify the required delivery properties, e.g., whether duplicates are allowed, or whether the note needs to be delivered by a deadline. In this paper, we focus on the following semantics:

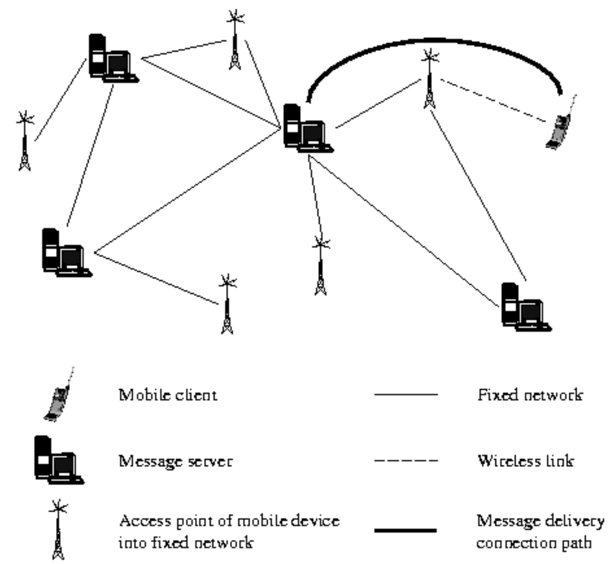



Figure 1: Replicated message delivery system.

- *Exactly-once*: a note should eventually be delivered to its target once, and only once.

As mentioned earlier, notes that lead to irrevocable changes in the physical world often need exactly-once guarantees. We focus on notes destined for a mobile, wireless device, for instance, asking someone on the shipping floor to send some goods, instructing a soldier in the field to initiate some action, asking a traveling manager to change the asking price for an acquisition, or telling a nurse in the emergency department to administer some drugs. The mobile device may have limited or no storage capacity to remember what notes have been received in the past, making it more challenging to guarantee exactly-once delivery. Also, as we will see, some mechanisms for duplicate elimination work better than others when the client is mobile and is only connected for short periods.

There are other possible delivery semantics, some of which are limited notions of exactly-once. For example, with *at-least-once* semantics, a note should be delivered at least once to the target, and duplicate delivery is not a concern. (For instance, a note stating that "the building is on fire" may require at-least-once semantics.) As another example, *limited lifetime* semantics may indicate that duplicate deliveries are not allowed up to a maximum note lifetime (expiration time), but that beyond that there is no guarantee (presumably because the note is of no significance after the expiration time). Although we do not study these limited semantics here, we believe that solutions for those semantics can be obtained by "relaxing" the exactly-once solutions. Thus, the exactly-once algorithms we will present here can form the basis for many other algorithms.

# 3 Exactly-once algorithms

In this section we describe algorithms that guarantee exactly-once delivery semantics. Additional implementation details (pseudo-code) for selected algorithms are given in Appendix A. The algorithms are divided into two major groups: those (Section 3.2) that guarantee strict exactly-once semantics, and those that guarantee a "relaxed" version which will be defined in Section 3.3. The former group assumes that the clients have some stable storage, while the latter group does not have such a restriction.

Our description of the algorithms assumes $N$ servers and one client for simplicity. The algorithms can easily be extended to service multiple clients.

## 3.1 Delivery procedures

Algorithms that guarantee exactly-once delivery use a procedure `Deliver(T,m)` to send a message $m$ to target node $T$ (server or client). `Deliver(T,m)` is guaranteed to return by a certain predefined time. If the call is successful, then $T$ executed a matching `Receive` call and successfully received $m$, and the sender received an acknowledgment confirming receipt of $m$. (Note that $T$ could have received $m$ more than once.) If `Deliver(T,m)` fails, the sender was unable to receive the acknowledgment from $T$ before timing out. Note that it is possible for a call to `Deliver(T,m)` to timeout, while the matching `Receive(m)` completes successfully, because the acknowledgment was not properly received by the sender.

## 3.2 Strict exactly-once algorithms

The strict exactly-once algorithms must cope with the different failures modes of `Deliver(T,m)`. The key in all algorithms is to assign some form of *identifier* (id) to each note, so that duplicate or missing notes can be handled. The algorithms vary in how they identify notes, how note ids can be recycled, and how long used note ids have to be remembered by servers and clients. These "small details" have significant impact on the potential performance and the usability of the algorithms.

**Strategy 1 (Sequenced Streams)** *Client remembers latest sequence number from each server. Sequence numbers are taken from an infinite domain.*

Our first algorithm resembles the well-known sliding window protocol [5] with unbounded sequence numbers. However, our algorithm is a "multiple-stream" version of that protocol with a sliding window width of one.

When a new note $m$ is generated, it is first stored in any one of the servers, for example, server $B$. Each server keeps a sequence number counter, which is incremented for each new note generated at that server. The globally unique id of note $m$ is the unique node id of server $B$ prepended to $m$'s sequence number at $B$, e.g., $B.5$. After this unique id is assigned, $m$ will be replicated to all the other servers, where it will have the same id.

In this algorithm, servers deliver (using procedure `Deliver`) all notes with the same server id sequentially. In other words, the notes are divided into $N$ separate "streams," by server id. To enforce sequentiality, the client $c$ keeps an array $R$ of sequence numbers, one for each server. Array $R$ is kept in stable storage so it survives client failures. The client will only accept note $B.n$ if $R[B] + 1 = n$. If the note is accepted, $R[B]$ is incremented; otherwise note $B.n$ is discarded. Notice that a note is not considered "delivered" (as far as exactly-once semantics is concerned) until it is accepted and processed by the client. Also note that the acceptance of a note and the corresponding counter increment should be an atomic operation at the client.

Each server $A$ maintains a list $K$ of to-be-delivered notes, sorted by increasing sequence number for each "stream." A note $m$ can be removed from $K$ after it is successfully delivered to the client. Furthermore, the servers periodically synchronize with each other to purge notes that have been delivered. (See Appendix A for more details.)

**Example 1** The system consists of two servers, $A$ and $B$, plus a client $c$. Note $m$, with id $A.3$, is waiting to be delivered on both $A$ and $B$. In other words, $K_A = K_B = \{A.3\}$. Client $c$ first connects to server $A$ and gets $m$. At this point $R[A] = 3$. If $A$'s call to `Deliver(c,m)` returns success, $A$ will remove $A.3$ from $K_A$. $A$ may also synchronize with $B$ to inform it of $m$'s delivery. This synchronization can be done immediately after $m$'s delivery, or periodically.

Later when $c$ connects to either $A$ or $B$, the server may attempt to send note $m$ again, perhaps because

$A$'s call to `Deliver` failed, or because server synchronization has not been performed yet between $A$ and $B$. But $c$ will not process $m$ again because $m$'s sequence number (3) is not what is currently expected ($R[A] + 1 = 3 + 1 = 4$). ∎

It is easy to see why this algorithm guarantees exactly-once delivery. Because notes in a stream are delivered sequentially, and the client remembers the most recent sequence number, in effect the client remembers all the notes it has ever received in the past. Therefore, duplicate delivery can never happen. A note $m$ will eventually be delivered to $T$ because $T$ will eventually request notes from a server, and that server will have $m$ (servers only delete a note after the note has been delivered successfully to the client).

Strategy 1 assumes that sequence numbers never wrap around. One can either assume that counters are large enough so that wrap around is "never" a problem (just like Y2K wrap around was never supposed to happen), or one can extend the algorithm as we describe next.

**Strategy 2 (Sequenced Wrap-around Streams)** *Client remembers latest sequence number from each server. Sequence numbers wrap.*

The problem with reusing a particular sequence number is that a message bearing the first use of such a number may get delayed in the network and resurface during the number's second use, causing re-delivery of the original message. This problem has been studied extensively in the networking literature [15, 3, 12]. The solutions proposed are primarily based on making sure that enough time has elapsed between possible reuses of the same sequence number. In other words, let $T_e$ denote the maximum lifetime of a message (for example, $T_e$ can be derived from an IP packet's time-to-live field). Then a sequence number cannot be reused until at least $T_e$ time has passed since a message bearing this sequence number was sent out.

In our context, the problem is slightly more complex because a note may linger not just in the network, but also at some other server. This complication is best illustrated by an example.

**Example 2** In a two-server ($A$ and $B$) system, the client $c$ gets note $m_1$ (id = $A.5$) from $A$. After time $T_e$, server $A$ thinks it is safe to reuse sequence number 5 because any previous messages delivering $m_1$ will have expired in the network by now. Hence a new note $m_2$ is generated on $A$, also with id $A.5$.

Let's assume that at this time $R[A] = 4$ on the client. Moreover, assume that, because of some difficulty in server synchronization, server $B$ is still not aware that $m_1$ has been delivered by $A$. If $c$ connects to $B$ at this time, $B$ will attempt to deliver $m_1$ to $c$ again. Because $A.5$ is indeed the expected id, $c$ will accept $m_1$, thus resulting in duplicate delivery. Moreover, when $c$ connects to $A$ later, it will not accept $m_2$ because it thinks that it has seen the sequence number before, thus resulting in $m_2$'s non-delivery. ∎

To avoid problems, we cannot reuse id $x.s$ until all servers have removed $x.s$. Furthermore, we must also ensure that the following id, $x.(s+1)$ is not in use either! To see why, suppose in our example that $A.5$ is reused when $B$ has removed $A.5$ but not some message $m_3$ with id $A.6$. When the client receives the new $m_2$ with $A.5$ there is no problem, but $R[A]$ is advanced to 6. This makes it possible for $c$ to receive the old $m_3$ again, instead of a new $A.6$ that follows $m_2$. Thus, the following condition is the one that guarantees the correct reuse of sequence numbers in our replicated delivery system.

**Condition 1** *An id $x.s$ can only be reused (i.e., a new note generated on server $x$ with $s$ as its sequence number) $T_e$ time after server $x$ has received confirmation from all other servers that note $x.(s+1)$ has been removed from their respective $K$ lists.*

To implement this condition, when server $x$ synchronizes with server $y$, $y$ will reply with the current state of its $K$ list. If the reply says that $y$ has already removed note $x.(s+1)$ from $K$, $x$ has effectively received a promise from $y$ that it will not attempt to deliver note $x.(s+1)$ in the future.

It is easy to see how this synchronization avoids the problem illustrated in Example 2. If server $A$ does not know that $m_1$ with id $A.5$ has been removed from $B$, then it will not be able to reuse number 5. Only after $A$ knows that $m_1$ does not exist at any server, will it start counting the required $T_e$ seconds to make sure $m_1$ is also gone from the network.

Although Strategy 2 guarantees exactly-once semantics, it still has the following potential shortcomings:

1. The client needs a minimum amount of stable storage ($N$ times the size of a sequence number counter) to store its $R$ array. The strategy will not work if, for example, the client can only remember one sequence number in its limited stable storage space.

2. It requires that notes with the same server id be delivered in strict sequential order. The problem can be alleviated if the client is willing to remember $k$ sequence numbers per server id. This

allows up to $k$ consecutive notes in a stream to be delivered out-of-order. Unfortunately, the solution further increases the storage requirement on the client $k$-fold.

3. Since the size of $R$ is proportional to the total number of servers ($N$), the strategy does not scale well if there are potentially many servers in the system. As an example, assume a 16-bit integer is used for server ids, but not all possible ids are in use at the same time. The client either needs to remember $2^{16}$ sequence numbers, or it must be informed whenever servers join or leave the system, which significantly complicates the dynamic server reconfiguration protocols.

4. The strategy also does not scale well with many clients. Because not every note is destined for all the clients, a note must have one sequence number per client in order to keep the sequence numbers consecutive. To cope, a server must now have as many sequence number counters as there are clients, and the protocol is significantly more expensive as a result.

The following strategy is designed to address the above problems, although it has its own drawbacks, as we will see in the performance section.

**Strategy 3 (Id List)** *Client remembers individual note ids it has received.*

For this strategy we simply assume that notes have globally unique ids, but the ids need not form sequenced streams. For example, each note may be assigned a server/date/time id. The client keeps in stable storage an explicit id list $R$ of individual notes it has received. For each new note received, the client first checks to see if its note id appears in $R$. If so, the client simply ignores the note; otherwise, it processes the note and inserts the note id into $R$. As before, processing the note and updating the data structure that records the processing must be an atomic operation.

If the client had infinite storage space for $R$, such a strategy would trivially guard against duplicate delivery, since the client would be able to remember the ids of all the notes it has ever received. If the client has only a fixed number of slots, $q$, to store $R$, we need to purge entries from $R$, while still guaranteeing exactly-once semantics.

The client can purge a note id $m$ only after it is certain that no server will ever try to deliver $m$ in the future, and only when it is sure that $m$ is not lingering in the network itself. For the latter, we assume a maximum network message lifetime, and use it as in Strategy 2 (Sequenced Wrap-around Streams). Here we only focus on ensuring that $m$ is not re-delivered by a server.

A note $m$ can be in one of three possible delivery states on any server:

1. "Undelivered": server thinks that $m$ has not been delivered to its target yet;

2. "Delivered": server knows that $m$ has been delivered, but it cannot yet forget about $m$ completely because other servers might not know about this delivery yet;

3. "Purged": server knows that all other servers are aware that $m$ has been delivered.

To keep track, each server keeps the following two data structures:

- $K$: list of notes that have not been purged, including Undelivered and Delivered notes;

- $D$: the subset of notes in $K$ that are "Delivered."

When a note $m$ is created, it is entered into list $K$ on every server. When a server $B$ delivers $m$ (successful completion of `Deliver(c, m)`), $m$ is put into list $D$ on that server. Periodically or immediately upon $m$'s delivery, server $B$ will attempt to synchronize its state with the other servers, by exchanging the contents of their $K$ and $D$ lists. During the synchronization, other servers will realize that $m$ has been delivered, and thus will put $m$ into their respective $D$ as well. Moreover, once $B$ confirms that $m$ has been put into every server's $D$, it can authorize the client to purge $m$ from its $R$ list. The servers can also purge $m$ by removing $m$ from both $K$ and $D$ at the same time.

**Example 3** We have two servers $A$ and $B$ plus one client $c$, as before. Suppose that two notes, $m_1$ and $m_2$, are waiting to be delivered. Client $c$ first connects to server $A$ and gets note $m_1$. At this point, the data structures have the following values:

$$\begin{aligned} K_A = K_B &= \{m_1, m_2\} \\ D_A &= \{m_1\} \\ D_B &= \emptyset \\ R_c &= \{m_1\} \end{aligned}$$

At this point (or periodically), $A$ tries to synchronize with $B$. However, let us assume that $c$ initiates a connection to server $B$ before $A$ could successfully inform $B$ about $m_1$'s delivery. In this case, $B$ attempts to deliver note $m_1$ again, but $c$ will simply ignore it because $m_1$ is already in $R_c$.

Assume that $B$ proceeds to deliver $m_2$. The data structures are now as follow:

$$\begin{aligned} K_A = K_B &= \{m_1, m_2\} \\ D_A &= \{m_1\} \\ D_B &= \{m_1, m_2\} \\ R_c &= \{m_1, m_2\} \end{aligned}$$

When $B$ synchronizes with $A$, $A$ will also add $m_2$ to $D_A$. Moreover, after $B$ gets $A$'s state information, $B$ will realize that $D_A = D_B = \{m_1, m_2\}$. Hence it is safe to purge both $m_1$ and $m_2$. In particular, $B$ will authorize $c$ to remove these two note ids from $R_c$. They will also be purged from $K_B$ and $D_B$, and from $K_A$ and $D_A$ as well during the next synchronization. ∎

To see why this algorithm guarantees exactly-once semantics, we observe that when $c$ connects to a server $B$ at any point after it has received note $m$, there are only two possible states:

1. $m$ is still in $c$'s list $R$.
2. $m$ has been purged from $R$. Since $c$ only purges $m$ when a server tells it to, and a server can only authorize the purging after $m$ has been put in $D$ of *all* servers, $m$ must be marked either Delivered or Purged on all servers.

Because neither of the above cases will result in note $m$ being delivered again to client $c$, we can see that this strategy indeed guards against duplicate delivery.

When the client's $R$ list fills up, the client must refuse to process further notes (perhaps by not sending acknowledgments to the server), until it receives a purging authorization, which will free up one or more $R$ slots. Delays due to $R$ overflow can have a significant performance penalty, especially if the client has very limited stable storage. The impact will be studied in detail in Section 5.

## 3.3 Relaxed exactly-once algorithms

A common theme of all the strategies presented so far is that the client has to carry stable state ($R$) across connections. Because in these algorithms the client ultimately decides whether a note should be processed, we name them "client-centric" approaches.

In some cases, a client-centric approach is not appropriate because the mobile client has no stable storage, or has limited storage, insufficient for an array of counters (one per potential server) or for a full queue of ids. In other cases, the client may have sufficient stable storage, but we may not wish to rely on it because the handheld device is vulnerable to theft, physical damage, or loss of power.

Therefore, in this subsection, we will develop a new suite of algorithms where servers assume the primary responsibility of remembering which notes have been delivered. The client need not remember state information, so the device it runs on can be safely turned off, or even replaced by a new device (e.g., if the previous one was stolen). When a client connects to a server, the server is responsible for guaranteeing that no duplicate note will be delivered. We will call these algorithms "server-centric", to contrast with algorithms in the previous subsection.[1]

### 3.3.1 Atomic delivery

It is not hard to see that exactly-once semantics cannot be achieved with procedure `Deliver(T, m)` of Section 3.1 when the client keeps no state. In particular, suppose that a `Deliver` call fails (timeouts) while the matching `Receive` returns success. If the client discards its state, all traces of that message's delivery will be gone from the system, potentially leading to duplicate delivery.

Thus, server-centric algorithms *cannot* achieve exactly-once semantics. Instead, we will strive to achieve a weaker notion of correctness. Intuitively, we recognize that the handoff of a note from a server to a client represent an unavoidable "window of vulnerability." That is, during this operation there will be some chance that the note is delivered without the server learning about it. If this happens, there is not much we can do. However, we would still like to have algorithms that do not "screw up" in other ways. For example, if the delivering server $A$ did find out that note $m_1$ was received by the client, then it would be unacceptable for some other server $B$ to try to deliver $m_1$.

Intuitively, our weaker notion of correctness says that the algorithm must guarantee exactly-once semantics, except for the unavoidable note handoff problems during the window of vulnerability. We capture this notion by assuming an atomic delivery procedure, denoted by `aDeliver(T,m)`. To define its properties, we first define a function $\rho(A, m)$.

[1] Incidentally, notice that even though the device on which the client application runs does not rely on stable storage, exactly-once delivery is still desired. Even though the device itself will not keep a record of an accepted note, the note will affect the real-world where the device runs, e.g., firing a missile. Accepting the same note again is unacceptable, even if the device does not know that the missile has been fired already.

Function $\rho(A, m)$ represents the number of times that node $A$ has received and processed the same message $m$[2]. Assume that before the sender $S$ calls the procedure `aDeliver(T,m)`, we have $\rho(T, m) = 0$.

Then an atomic delivery procedure guarantees the following: when `aDeliver(T,m)` returns success, $\rho(T, m) = 1$; and when `aDeliver(T,m)` returns error, $\rho(T, m) = 0$. Moreover, in the former case, $\rho(T, m)$ remains 1 unless `aDeliver(T,m)` is called again, in which case the outcome will be undefined. In other words, it is up to our exactly-once algorithms to guarantee that `aDeliver` is never called again if the first time it was successful. In the case where `aDeliver(T,m)` returns error, $\rho(T, m)$ will remain 0 until the next time `aDeliver(T,m)` is called. We further assume that the return value of `aDeliver(T,m)` is always logged in permanent storage. Consequently, even if the server crashes right at the moment when the function returns, it is still able to figure out later the outcome of the call from the log.

In summary, a server-centric algorithm has to perform "risky" operations when it delivers a note to a client. The `aDeliver` procedure encapsulates this risky operation. By saying that `aDeliver` exists, we are *conceptually* saying that there is no risk, simply so we can focus on other potentially dangerous operations, ones that *can* be avoided with a good algorithm. In other words, our algorithms should not perform any more "risky" operations than absolutely necessary. We will say that an algorithm guarantees *relaxed exactly-once* semantics when it guarantees exactly-once delivery under the assumption that `aDeliver` exists.

When we compare an algorithm that guarantees strict exactly-once semantics to one that offers relaxed semantics, we must keep in mind that the latter has some probability of failure. (This probability can be reduced by increasing the timeout period a server waits before declaring a delivery failure.) Of course, the relaxed algorithm has an important advantage in that it does not require stable storage.

### 3.3.2 Algorithms

The strategies presented in this section differ in the amount of participation required of the client. Notes can have any type of globally unique identifier.

**Strategy 4 (Server Synchronization)** *Client has no state.*

In this strategy the client is not required to remember anything when it is turned off. For a client with volatile memory, this may be the only option. The servers assume full responsibility for guaranteeing exactly-once semantics.

As before, each server keeps a list $K$ of undelivered and delivered but not-yet-purged notes, and a list $D$ of just the latter. Before a server can start delivering notes, it first gathers the list $D$ from all of the other servers to merge into its own. This step ensures that the server is aware of all the notes that have previously been delivered to the client, so that it will not deliver them again.

Exactly as in Strategy 3 (Id List), the servers periodically synchronize to reclaim space in their $K$ and $D$ lists. In particular, messages whose delivery is known to all servers are purged.

**Example 4** Server $A$ delivers note $m$. Hence $D_A$ contains $m$. Later when $B$ wants to deliver notes, it obtains $D_A$ and merges with $D_B$. This will guarantee that any previously delivered note, $m$ in particular, will be included in $D_B$, and thus will not be delivered again. ∎

**Strategy 5 (Delay Reconnect)** *Client does not reconnect within time $T_r$.*

In the two algorithms that follow, we use *time* to prevent duplicate deliveries. We assume a clock synchronization protocol, such as NTP [13]. We also assume a bounded maximum clock drift, which, if not zero, can be added to the appropriate parameters in our algorithms.

Suppose that the client promises to refrain from connecting to any server until time $T_r$ has passed since its last connection to any server[3] (How the selection of parameter $T_r$ affects the overall performance of the strategy will be investigated in Section 5). The reconnect delay can be achieved by the client remembering the timestamp of its last connection, or the client can use a timer that starts ticking at the end of a connection. Alternatively the human user of the client device may be trusted to discipline him or herself.

When $T_r$ has passed, and the client connects again, the new server checks to make sure that, for each of the other servers $s$, the most recent synchronization message received from $s$ is within $T_r$ time. If not, the server will attempt to synchronize with the problematic server(s) while letting the client wait. If all servers have synchronized within $T_r$ time, then it is

[2] As before, the successful reception and the acceptance of a note need to be atomic.

[3] To be more precise, $T_r$ is measured from the end of last connection (when the client sent out the last message) to the initiation of the next connection (when the client sends the first request message).

safe to deliver notes to the client, because the notes delivered to the client during its most recent connections are known to the current server.

To be more precise, each server maintains an array $TS[s]$, containing the timestamp of the most recently received synchronization message from each server $s$. The servers periodically send these synchronization messages to each other, which include the originating server's list $D$, as in Strategy 4 (Server Synchronization).

**Example 5** Client $c$ connects to server $A$ and receives note $m_1$. Now $D_A$ includes $m_1$. After time $T_r$ has passed, $c$ makes another connection with server $B$. The first thing $B$ does is to check when was the last synchronization message it received from $A$.

- Case (i): if the synchronization was sent within $T_r$ time, it must have been generated at server $A$ when $D_A$ contained $m_1$ [4]. Because $B$ incorporates $D_A$ into its own list $D$, $m_1$ must be in $D$ when $B$ starts delivery.
- Case (ii): if the $A$ synchronization was more than $T_r$ seconds old, $B$ is not allowed to start delivery until it has contacted $A$ and received an up-to-date $D_A$, which should contain $m_1$.

Thus, because client connections must be separated by $T_r$ time units, the union of all servers' synchronization messages less than $T_r$ old will include all notes the client has received. ∎

Compared to Strategy 4 (Server Synchronization), this one has a better start-up time because, if all goes well, a server only needs to check local information before it is allowed to deliver notes. However, when there is a long communication blackout between two servers, this scheme blocks delivery as does Strategy 4. The overhead of the Delay Reconnect strategy is the additional requirement on the client not to connect too frequently.

Another way of looking at this difference is that we have switched from a lazy propagation scheme to an eager one. In the Server Synchronization strategy, the list of delivered notes is propagated only when needed, while the client waits. In Strategy 5 (Delay Reconnect), the list of delivered notes is propagated eagerly by a server, potentially reducing the overhead when the client connects.

**Strategy 6 (Connect History)** *Client remembers complete connection history (when and to which server it has connected before).*

[4] Or, if $D_A$ didn't include $m_1$, that means $m_1$ must have been purged from $A$, which in turn implies that $m_1$'s delivery was known to all servers.

Each server maintains a table $TS$ of the most recent synchronization timestamps from other servers just as in Strategy 5 (Delay Reconnect). However, the client is no longer subject to the $T_r$ reconnection time limit. Moreover, the servers are no longer required to synchronize with each other periodically, but only when there is new information.

The client now needs to remember the last time it connected to each server. This is recorded in a history array $H$, where $H(A)$ is the time when the last connection to server $A$ ended. When the client makes a new connection, it first uploads to the server its $H$ array. The server must ensure that it has received synchronization messages from each other server that are fresher than the client's last connect time.

**Example 6** Note $m_1$ is replicated at servers $A$ and $B$. Server $A$ delivers $m_1$ to $c$ at time $t_1$. At a later time $t_2$, $c$ connects to $B$, giving it the value $H(A) = t_1$. Server $B$ makes sure it has synchronized with $A$ after time $t_1$, thus ensuring that $B$ knows about $m_1$'s delivery. ∎

Compared to Strategy 5 (Delay Reconnect), the advantages of the Connect History strategy are threefold. First, it eliminates the client restriction not to reconnect too soon. Second, it saves network bandwidth by avoiding unnecessary synchronizations. Last, unlike Strategies Server Synchronization and Delay Reconnect, Connect History will not block delivery because of an *irrelevant* network partition. For example, the client first connects to server $A$ then to $B$, but it has never connected to $C$. If $B$ cannot talk to $C$ (but can talk to $A$), it can still go ahead and deliver notes to the client. All these advantages come at the price of the client carrying a little more information than it is required to in the Delay Reconnect strategy.

It is important to note that, although some of the server-centric algorithms require the client to store some information (e.g. its connection history), there is a fundamental difference between this information and that of the client-centric algorithms. In the client-centric approaches, the client information ultimately determines if a note delivery can result in duplicate delivery or not. In the server-centric schemes, the client information is used solely for improving performance rather than for correctness. In the event that the client loses its information, the servers can always perform a full synchronization among themselves, thereby still guaranteeing relaxed exactly-once semantics.

# 4 Evaluation

The relative merits and shortcomings of our algorithms can be evaluated along several dimensions:

- Semantic guarantees. In particular, server-centric strategies provide a less strict delivery guarantee than the client-centric ones.
- Reliability. For instance, how likely is it that a client connection may be broken by network problems?
- Availability. For example, what is the likelihood that the client will be able to connect to a server and get notes when it wants to.
- Performance. For example, what is the overall network traffic incurred, or how fast can notes be delivered to the client?

We have studied several metrics that quantify the reliability, availability or performance of the algorithms. Due to space limitations, in this paper we only present one metric, which we have found to be especially useful. Our metric is the *expected throughput*, defined as the expected number of notes that a client can successfully receive during a connection that lasts $T_c$ seconds.

We have chosen this metric because it captures a number of factors. The metric captures the synchronization overhead of the algorithms, because algorithms that require more synchronization before or while notes are delivered will have less overall throughput. The metric also captures the benefits of replicating notes at servers: As we add more servers, clients will find closer or better connected servers for download, achieving better delivery throughput. The expected throughput is also a good indirect measure of system reliability, because a higher throughput means that the client can get the same number of notes faster, and is therefore less prone to disconnects. Moreover, our metric focuses on the connection-time performance as observed by the client, which we feel is the most critical component of the overall system performance in a wireless communication scenario.

Having selected a metric, the biggest challenge is finding a good system model. The model should be rich enough to capture the main tradeoffs we want to study, while not being so detailed that we are overwhelmed with parameters and components. Our goal is *not* to predict exact performance of one algorithm versus another in a particular scenario. Rather, our goal is to clearly demonstrate and quantify the important features of the various strategies, so we strive for the simplest model that allows us to do so.

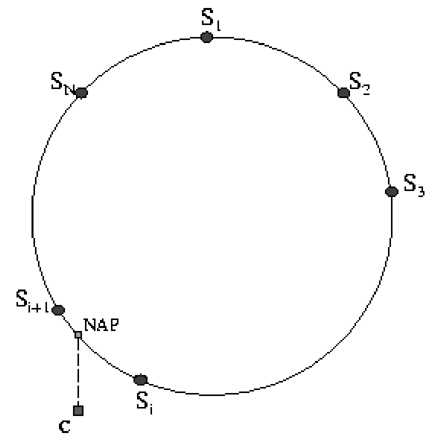



Figure 2: System model.

The global network is represented by a circle, as shown in Figure 2. Each node (e.g., a server or a mobile access point) of the network resides on the circle. The network distance between any two nodes, measured in network hops, is proportional to their distance on the circumference of the circle. For instance, if the circumference is 30 network hops, then the distance between two nodes directly across from each other is 15 hops. In Figure 2, $S_i$ is the $i$'th server. The mobile client is represented by $c$, and the Network Access Point ($NAP$) is its entry into the global network. A wireless line (dotted line) connects $c$ and $NAP$.

We choose a circular representation of the network because it allows us to easily model the fundamental characteristics of a server network. For example, the effect of placing more servers around the globe to increase access locality is reflected by the reduced distance to the nearest server from a random point on the circle. (Of course, a sphere may have been a more accurate model for the Earth's networks, but we do not believe we can gain more insights from such a substantially more complex model.)

We assume that the servers are uniformly distributed on the circle. Consequently, the more servers there are, the shorter the distance between neighboring servers. For example, with 3 servers and 30 total hops, neighboring servers are separated by $\frac{30}{3} = 10$ hops. If the number of servers is increased to 10, that distance is only $\frac{30}{10} = 3$ hops.

When a client $c$ wants to get its notes, it first connects to a random access point ($NAP$) on the circle. In real life, this access point could be an ISP for mobile units, or a LAN segment which takes in the traveling client as a guest. From this access point, the client figures out where the nearest server is on the

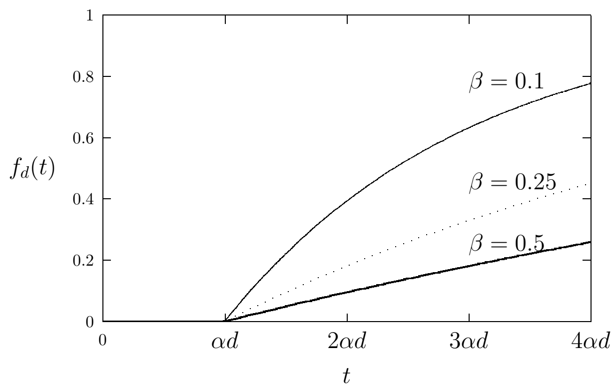



Figure 3: $f_d(t)$, $\alpha = 0.05$

circle ($S_{i+1}$ in the figure), and connects to it. In an actual implementation, technologies such as dynamic DNS translation and service redirection [9, 4, 1] can be used to identify the nearest server. Once connected, one of the algorithms described in Section 3 will be used to guarantee exactly-once delivery of notes.

We model the characteristics of the fixed network by a distribution function $f_d(t)$ (Figure 3), which gives the probability that node $A$ can deliver a message to another node $B$ distance $d$ away on the circle within time $t$. To simplify our analysis in this paper, we assume that all messages (including notes) are relatively short, so that the major component in message delay is the length-invariant part. Thus, for measuring delays we assume that all messages are of the same size. Function $f_d(t)$ is given by the following formula:

$$f_d(t) = \begin{cases} 0 & \text{if } t < \alpha d \\ 1 - e^{-\frac{t-\alpha d}{\beta d}} & \text{otherwise} \end{cases}$$

Here $\alpha$ and $\beta$ are scaling constants. Intuitively, $\alpha$ gives the minimum time per hop it takes to deliver a message[5]. Thus, $\alpha$ captures the maximum speed of the network subject to the physical limitation of the networking medium and the routing infrastructure. For example, $\alpha = 0.05$ means that the shortest time it takes to deliver a message over one hop is 50ms. However, due to unreliability and fluctuations in the network, not all messages are delivered in the minimum possible time. We assume that, after time $\alpha d$, the probability of a message still not delivered by time $t$ decays exponentially with $t$. We use $\beta$ to measure how fast this probability changes with $t$. In a sense, $\beta$ reflects the "faultiness" of the network by measuring the percentage of messages that are delivered by a certain time period. For instance, $\beta = 0.25$ says that $1 - e^{-\frac{\alpha}{\beta}} \approx 18\%$ of messages are delivered after twice the minimum time it takes to deliver such a message over the same distance. Examples of $f_d(t)$ with $\alpha = 0.05$ and different values of $\beta$ are given in Figure 3.

Therefore, our formulation of function $f_d(t)$ is capable of representing a large variety of networks, from slow and lossy networks to fast and reliable ones. Since it is not tied to the values of a specific network, it allows us to vary the parameters and observe how the performance of various algorithms changes as the underlying network characteristics change.

Function $f$ models message delivery in the fixed network (i.e., the circle). For the link between the client ($c$) and its network access point ($NAP$), we assume a constant message delay $t_{cap}$. We do not model the "faultiness" of this link because we are going to evaluate expected throughput only when the client is connected for $T_c$ seconds. That is, our metric will measure how many notes can be delivered, given that the client has connected and remains connected for $T_c$ seconds.

Table 1 summarizes our model parameters. The table also lists the *base* values used for obtaining the results of Section 5. We have experimented with many parameter values, before settling on these as good initial values. As part of our experiments, we have of course studied the sensitivity of each parameter, as it is varied from its base value.

Although one can easily argue that our base values should be larger or smaller than what we have chosen, we believe that our base values are still quite "representative." For example, it seems reasonable to say that a message can go "around the world" in 30 hops ($D = 30$). We have also selected numbers for $\alpha$ and $\beta$ ($\alpha = 0.05$, $\beta = 0.25$) that we believe are reasonable for a typical wide area network.

# 5 Results

Our expected throughput numbers are obtained analytically from our performance model. The derivations are not presented in the main body of the paper, but are given in Appendix B.

Our performance numbers are plotted as the ratio of our exactly-once strategies over a base strategy, Strategy 0, that provides the weaker at-least-once semantics. Thus, if $p_i$ is the expected throughput of Strategy $i$, and $p_0$ is that of Strategy 0, then we plot $\frac{p_i}{p_0}$. Concretely, when $\frac{p_i}{p_0} = 0.5$, a client is only ex-

[5]The assumption that the minimum time is proportional to the network distance is approximately correct for relatively short messages.

| Symbol | Meaning | Unit | Base value |
|---|---|---|---|
| $p_i$ | expected throughput of Strategy i | notes | |
| $D$ | total number of hops in the global network | hops | 30 |
| $N$ | total number of servers in the global network | servers | 10 |
| $T_c$ | connection duration | seconds | 15 |
| $\alpha$ | parameter of distribution function $f$ | sec/hop | 0.05 |
| $\beta$ | parameter of distribution function $f$ | sec/hop | 0.25 |
| $\lambda$ | shorthand for $\alpha + \beta$ | sec/hop | 0.3 |
| $t_{cap}$ | delivery time between client and network access point | seconds | 0.2 |
| $q$ | number of slots to hold list $R$ in Strategy 3 | entries | 20 |
| $T_r$ | reconnect limit in Strategy 5 | seconds | 100 |
| $T_u$ | periodic server synchronization interval in Strategy 5 | seconds | 300 |
| $d'$ | number of hops to previous connection server in Strategy 6 | hops | 10 |
| $t'$ | time since previous connection in Strategy 6 | seconds | 10 |

Table 1: Parameters used in analysis

pected to receive half as many notes with Strategy $i$ as with Strategy 0 for a connection of the same duration. Analogously, $\frac{p_i}{p_0} = 1$ implies that Strategy $i$ provides exactly-once semantics with no observable overhead to the client in terms of expected throughput.

We have selected only one representative graph for each strategy due to space limitations. In these graphs, we plot the expected throughput ratio against a parameter that we believe is the most critical or interesting for the strategy in question. For example, for Strategy Id List we have chosen to plot $\frac{p_3}{p_0}$ versus $q$, the number of $R$ slots on the client, because $q$ is unique to this strategy, and $q$ has a strong impact on performance. We do not present graphs for Strategies 1 and 2, since they achieve the same expected throughput as the base strategy (Appendix B).

## 5.1 Strategy Id List

Figure 4 gives the throughput ratio of Strategy 3 over the base strategy ($\frac{p_3}{p_0}$) plotted against $q$, the number of slots in the client's stable storage. The three curves in the figure correspond to different values of $t_{cap}$, the wireless link delay. For example, the leftmost curve is for a one-second communication delay between the client and the network access point, while the rightmost curve assumes that this delay is zero.

We observe that the performance ratio starts off low (high overhead) but improves as the client stable storage increases. This is expected because when the client does not have many slots to hold its "received list" $R$, note delivery is delayed as the client waits for the servers to purge old note ids. In particular, we observe that there is a threshold above which Strategy 3 is able to perform as well as Strategy 0 ($\frac{p_3}{p_0} = 1$).

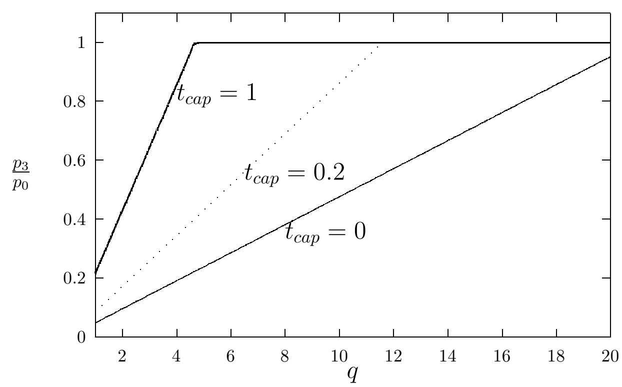



Figure 4: $\frac{p_3}{p_0}$ versus $q$

As a rule of thumb, the minimum number of slots the client needs to avoid an $R$ overflow is approximately equal to the ratio of the time needed to purge a note id among the servers over the time to deliver a note. From the figure, with $t_{cap} = 0.2$, the client needs about 12 slots to achieve a 100% performance ratio.

Figure 4 lets us study the impact of relative speeds of the wireless to the wireline networks. With a very fast wireless link ($t_{cap} = 0$, zero delay), the client needs much more stable storage than with a slower link. This is because a smaller $t_{cap}$ means that notes can be delivered faster, whereas note id purging is still limited by the speed of the fixed network.

Our results show that over a wide range of parameter values, a relatively modest amount of storage (e.g., $q$ on the order of tens or hundreds of note ids) makes the Id List strategy perform well. Thus, if the client device is designed to have stable storage at all, it should not be too expensive to provide enough storage capacity. On the other hand, if it is too ex-

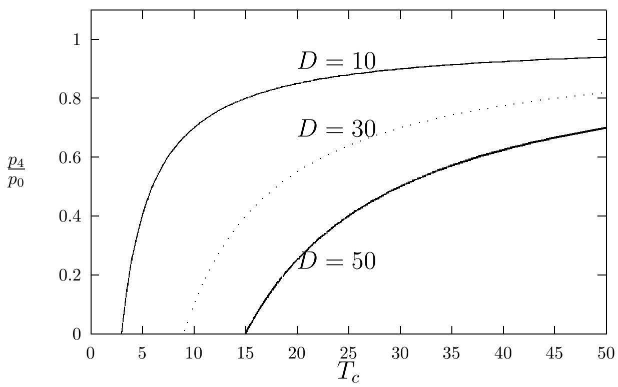



Figure 5: $\frac{p_4}{p_0}$ versus $T_c$

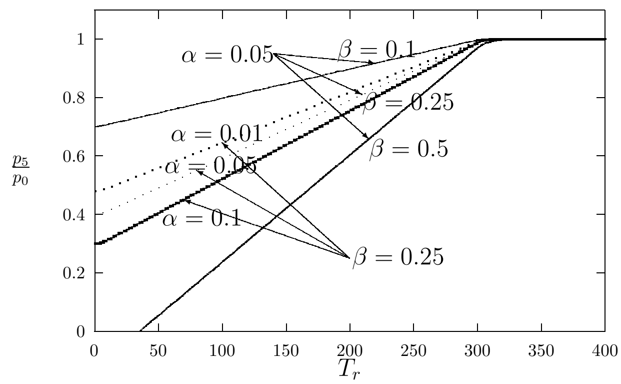



Figure 6: $\frac{p_5}{p_0}$ versus $T_r$

pensive to have stable storage (or if we cannot trust that storage, as discussed earlier), then the server-centric strategies should be selected.

## 5.2 Strategy Server Synchronization

In Figure 5, we study the throughput performance of Strategy 4. The x-axis is now the connection duration, $T_c$. Three curves represent different values of $D$, the total length of the circular global network.

For very short connections (small $T_c$), Strategy 4 incurs a very high overhead because an initial server synchronization phase is needed. No note can be delivered during this period and the throughput is practically zero. However, once the setup is finished, delivery proceeds at the same rate as in Strategy 0. Hence, for longer connections, the effect of the setup period is amortized, and the performance ratio gradually approaches 1 as $T_c$ increases.

Another trend we observe from Figure 5 is that performance decreases with increasing $D$. This is because, as the network grows larger (more hops around the globe), the cost of a server synchronization is greater. Consequently, the initial setup phase takes longer, and its impact on overall throughput is more marked.

In conclusion, Strategy 4 works well for longer connections or small networks. If neither of the above applies, Strategy 4 may still be selected because it does not rely on any client state information.

## 5.3 Strategy Delay Reconnect

Figure 6 illustrates the effect of the reconnect limit $T_r$ on Strategy 5. It also shows $\alpha$ and $\beta$ variations to demonstrate the impact of network characteristics. The middle three curves have a common $\beta$ (0.25), but different $\alpha$'s. The outer two curves and the middle one share the same $\alpha$ (0.05), but differ in their $\beta$ values. We remind the reader that $\alpha$ measures the overall speed of the network while $\beta$ reflects its "faultiness."

As a general principle, the longer the client is willing to wait in between connections (bigger $T_r$), the better throughput it will get. This is because a bigger $T_r$ gives the servers a longer time to perform a successful server synchronization, thus reducing the possibility of needing a synchronization during client connection setup. At $T_r = 0$, Strategy Delay Reconnect degenerates into the Server Synchronization strategy because the client can reconnect at any time. Then performance improves as $T_r$ increases, to the point where the overhead is practically unnoticeable when the client is willing to wait at least one server synchronization period, $T_u$ (which is 300 seconds in our setup). Intuitively, when $T_r > T_u$, at least one synchronization should have been attempted in between the client's two connections.

Looking at the various curves in Figure 6, we see that as we move towards a slower (larger $\alpha$) or a more "faulty" (larger $\beta$) network, the performance penalty becomes bigger. This is simply because in a less efficient network, server synchronizations take longer and are more likely to fail. Furthermore, we see from the figure that $\beta$ has a relatively significant impact on performance.

## 5.4 Strategy Connect History

Figure 7 has the same axes as Figure 5. However, the curves in the figure represent different values for $d'$, which measures the distance from the current server to the server of the previous connection. For example, $d' = 10$ implies that the client's previous connection was made to a server that is 10 hops away.

The graphs for Strategy 6 have the same general shape as Strategy 4 (Server Synchronization). In par-

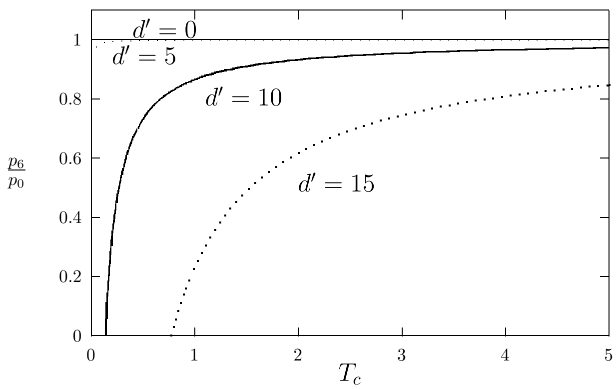



Figure 7: $\frac{p_6}{p_0}$ versus $T_c$

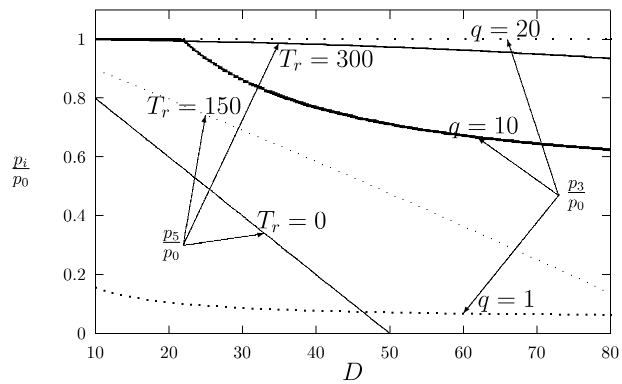



Figure 8: Comparison of $\frac{p_5}{p_0}$ and $\frac{p_3}{p_0}$ versus $D$

ticular, performance overhead is higher for shorter connections, but it drops as connection time gets longer. However, Strategy 6 is effective in reducing the overhead of Strategy 4. As an evidence, we notice that the range of $T_c$ plotted is much smaller in Figure 7 (0-5 sec) than in Figure 5 (0-50 sec).

If the client consistently connects to the same server ($d' = 0$), then naturally Strategy 6 does not have any overhead ($\frac{p_6}{p_0} \equiv 1$). As $d'$ increases, the performance overhead increases as well. The reasons are two-fold. Firstly, as the distance between the two servers gets larger, there is a higher chance that a synchronization may be needed because the current server has not been able to perform a successful synchronization since the previous connection. Secondly, such a synchronization is also likely to take longer because the previous server if farther away.

Another parameter not shown in the figure but which also has an impact on Strategy 6's performance is $t'$, the time since the previous connection. A bigger $t'$ means that the servers have had a longer time since the last connection to perform a successful synchronization. Consequently, the performance is also better because a makeup synchronization is less likely needed. To summarize, Strategy 6 works best when the client primarily makes infrequent connections (big $t'$) to servers that are close together (small $d'$).

## 5.5 Comparison of client- and server-centric algorithms

Lastly, we plot a client-centric algorithm (Strategy 3) and a server-centric one (Strategy 5) side by side to see how they scale with the size of the network ($D$). We have to keep in mind the fundamental differences between them. For example, one guarantees strict exactly-once semantics while the other only guarantees relaxed, and one requires stable storage on the client while the other does not. However, it is still informative to compare them side by side.

In Figure 8, each strategy is represented by three curves, which give the performance overhead when the size of the global network ($D$) is changed. The three $\frac{p_3}{p_0}$ curves differ in the amount of stable storage ($q$) allowed on the client, while the $\frac{p_5}{p_0}$ ones in the amount of time the client is willing to wait between connections ($T_r$).

The important lesson to take away from this figure is that both strategies can perform fairly well under favorable conditions, and both can perform badly otherwise. For Strategy Id List, it is important to have enough storage space on the client. When the storage is severely limited, e.g. $q = 1$, the performance overhead can be quite large even for small $D$. On the other hand, with enough storage, e.g. $q = 20$, the performance penalty can be avoided all together. Similarly, the key factor for Strategy Delay Reconnect seems to be how long the client is committed to wait between connections ($T_r$).

Although both strategies perform less well as $D$ increases, the downward slope is actually due to very different reasons. In the case of Strategy 3, a bigger $D$ means a longer time needed to purge a note id. Consequently, if the client does not have enough $R$ slots to avoid an overflow, it will eventually need to slow down to wait for the purging to catch up. On the other hand, a bigger $D$ increases both the server synchronization time and the likelihood of needing a synchronization for Strategy 5. The last point may also account for the fact that the curves for Strategy 5 tend to drop more steeply than those for Strategy 3.

### 5.6 Summary of strategies

In Table 2 we compare the features of all the strategies. The comparison is divided into three sections. The first section categorizes the strategies based on whether they provide strict or relaxed exactly-once semantics. The second section inspects the system assumptions made by the algorithms. Specifically, the first row looks at whether a strategy requires stable storage on the client. For those that do, the next row gives whether the minimum size of that storage is proportional to the number of potential servers in the system. The third row indicates whether each note has to potentially have a different unique id for each client. The fourth row states whether there is an inherent order imposed on note delivery. Finally, the fifth row looks at whether the client is free to connect into the system at any time it wishes. The third section deals with algorithm performance and looks at which parameters have an impact on the expected throughput for different strategies. The rows in this section should be self-explanatory.

## 6 Related work

This paper builds upon previous work on exactly-once semantics and on replication. However, our work is unique in that we consider all of the following factors: 1) wireless connectivity, 2) limited or lacking stable storage on client, and 3) different notions of exactly-once.

The networking literature has dealt extensively with the problem of "exactly-once" delivery of network packets. For example, the well known sliding window protocol and its many variants [5, 15, 3] are used to provide delivery guarantees when the underlying network is faulty and unreliable. Our notion of exactly-once, however, is different. The window protocols worry about exactly-once delivery only during one TCP session, not across longer periods of time. The protocols start over when the client disconnects from the server and reconnects later. Consequently, the window protocols only have to deal with one delivering server, whereas our algorithms have to worry about state synchronization among several servers. Moreover, most window protocols rely on the client remembering unique ids. Some of our algorithms, namely the server-centric ones, can cope with situations where the client has no stable storage.

The sequence number wrap-around problem has been studied thoroughly [12, 3] in the context of window protocols. However, as explained earlier in Section 3.2, the same problem is more involved in Strategy Sequenced Wrap-around Streams because a server has to take into account multiple servers.

Distributed systems research has studied exactly-once execution of remote procedure calls [2, 16]. Solutions also rely on the client's ability to remember unique ids. To the best of our knowledge, there has been no attempt to deal with exactly-once semantics with multiple servers or across multiple connections.

The Usenet News infrastructure [11, 8] avoids duplicate delivery by requiring a news reader to always connect to the same news server. The system solves the duplication problem among the servers (when messages are first replicated) using globally unique ids. Naturally, the challenge posed by limited capability wireless clients, which we address in this paper, does not come up in the Usenet system.

There has been a great amount of work on replicated data management and replicated transactions. For example, the work on cache consistency [14, 6, 10] is mostly concerned with multiple clients seeing a consistent picture as the data are being changed. As another example, Gray et al [7] looks at transaction serializability in a replicated environment. Our work also deals with the problems resulting from replication. However, our work addresses a mobile scenario, where, for instance, delivery throughput should be maximized in order to increase the number of notes delivered before losing a connection. Limited client storage is also a critical issue. We have considered different exactly-once notions to accommodate mobile devices with no storage.

Incidentally, the Server Synchronization strategy can be reformulated as a transactional update problem on message delivery status. Specifically, data replication algorithms can be used to make sure that each server sees an up-to-date version of the delivery state of each message. Although this approach may lead to additional algorithms, we believe that it will not cover the full range of solutions we have proposed here.

## 7 Conclusion

In this paper we have studied the message delivery problem in mission-critical mobile scenarios where exactly-once semantics is imperative. The algorithms we have proposed differ in the guarantees they make, in the assumptions they make about the system, in the requirements they place on the servers and the clients, and in their complexity and performance.

We have developed a metric, expected note throughput, that is useful in comparing delivery algorithms. An algorithm with a high throughput can deliver more notes in a shorter period of time, in-

| Strategy | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Strict exactly-once | √ | √ | √ | | | |
| Relaxed exactly-once | | | | √ | √ | √ |
| Requires client stable storage | √ | √ | √ | | | √† |
| Client storage requirement proportional to number of servers | √ | √ | | | | ‡ |
| One unique id per client for each note | √ | √ | | | | |
| Requires in-order delivery | √ | √ | | | | |
| Limit on client reconnects | | | | | √ | |
| Performance sensitive to connection duration $T_c$ | | | | √ | √ | √ |
| Sensitive to pattern of past connections | | | | | | √ |
| Sensitive to size of client storage | | | √ | | | |
| Sensitive to frequency of periodic server synchronizations | | | | | √ | |

Table 2: Comparison of strategies. †Strictly speaking, in Strategy Connect History, stable storage is needed for performance rather than required for correctness. ‡Although Strategy Connect History requires the client to remember timestamps in the history array $H$, the client only has to remember servers that it ever connects to. Moreover, there can also be protocols to allow the client to purge old entries.

creasing the chances of a successful download to a mobile device. We have found that the strategies differ in their performance, but every strategy can be tuned to minimize its overhead through careful selection of parameters. Overall, in many cases, the cost of exactly-once delivery is no higher than that of at-least-once delivery. Thus, it makes sense to implement exactly-once even if the application does not absolutely require it.

# References


[1] Akamai Technologies, Inc. `http://www.akamai.com`.

[2] K. P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications Co., 1996.

[3] G. M. Brown, M. G. Gouda, and R. E. Miller. Block acknowledgment: Redesigning the window protocol. *IEEE Transactions on Communications*, 39.4:524–532, 1991.

[4] V. Cardellini, M. Colajanni, and P. S. Yu. Redirection algorithms for load sharing in distributed web-server systems. In *The 19th IEEE International Conference on Distributed Computing Systems*, 1999.

[5] V. Cerf and R. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22:637–648, 1974.

[6] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[7] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD Conference*, pages 173–182, 1996.

[8] T. Gschwind and M. Hauswirth. A cache architecture for modernizing the Usenet infrastructure. In *Proceedings of the 32nd Annual Hawaii International Conference on Systems Sciences*, pages 1–9, 1999.

[9] E. Katz, M.Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computers Networks and ISDN Systems*, 27:155–164, 1994.

[10] R. Katz, S. Eggers, D. Wood, C. Perkins, and R. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276–283, 1985.

[11] D. Lawrence and H. Spencer. *Managing USENET*. O'Reilly & Associates, Incorporated, 1998.

[12] W. S. Lloyd and P. Kearns. Bounding sequence numbers in distributed systems: A general approach. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 312–319, 1990.

[13] D. L. Mills. Internet time synchronization: The network time protocol. *IEEE Transactions on Communications*, 39.10:1482–1493, 1991.

[14] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6.1:134–154, 1988.

[15] A. Shankar. Verified data-transfer protocols with variable flow control. *ACM Transactions on Computer Systems*, 7.3:281–316, 1989.

[16] A. Vaysburd and S. Yajnik. Exactly-once end-to-end semantics in CORBA invocations across heterogeneous fault-tolerant ORBs. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 296–297, 1999.

# A Algorithm details

The pseudo-code in this section uses the following conventions:

- The code enclosed in braces following a deliver or receive procedure call is executed when that call returns timeout. If the call returns success, execution continues from the next statement.
- If $m$ denotes a message that is sent by node $A$, then $m$.sender contains the id of $A$, and $m$.body is the content of the message. In some algorithms, $m$.timestamp will contain the local timestamp when $m$ was sent.
- For algorithms that assume a ServerID.SeqNo note id format, if $i$ is a note id, then $i$.server and $i$.seqno are its two parts, respectively.
- A call to `abort` will terminate execution of the current procedure.

Below we give the pseudo-code for the server and client connection routines of Strategy Sequenced Streams. The server contains a main message dispatching thread which executes the following loop forever:

```
loop forever {
  Receive(m); {continue;
               // we keep listening.}

  if (m.body == "Request my notes")
    fork(HandleClient(m.sender));
}

HandleClient(c) {
  foreach i in K {
    Deliver(c,
            "Here is note i: ...");
                          {abort;}
    K = K - {i};
  }
}
```

When a client wants to get its notes, it connects to a server and then executes the following procedure:

```
Deliver(s, "Request my notes"); {abort;}

loop {
  Receive(m); {abort;}
  if (m.body == "Here is note i: ..." &&
      i.seqno == R[i.server] + 1) {
    ProcessMessage(m);
    R[i.server] = i.seqno;
      // remember that we have
      //  received this message.
  }
}
```

Remark: The last two statements in the client's code, which process a message and remember its receipt, need to be atomic.

In Strategy Id List, the server's main loop now looks like the following:

```
loop forever {
  Receive(m); {continue;
               // we keep listening.}

  if (m.body == "Request my notes")
    fork(HandleClient(m.sender));

  if (m.body ==
      "Server synchronization: Ks,Ds") {
    P = K - Ks;
      // The other server has purged
      //  these already; we can too.
    K = K - P;
    D = D - P;

    P = Ks - K;
      // We have purged these already;
      //  the other server should too.
    Ks = Ks - P;
    Ds = Ds - P;

    D = D + Ds;
      // Merge in notes marked delivered
      //  on the other server.
    Deliver(m.sender,
            "Synchronization reply: K,D");
                              {continue;}
  }
}
```

Here we assume that server synchronizations are performed right after each message's delivery:

```
HandleClient(c) {
  // Start delivering undelivered messages
  U = K - D;
  foreach i in U {
    Deliver(c, "Here is note i: ...");
                              {abort;}
    D = D + {i};
    fork(ServerSynch(i,c));
  }
}
```

```
ServerSynch(i,c) {
  foreach s in S, in parallel {
      // S is the set of all other servers
    Deliver(s,
            "Server synchronization: K,D");
                                {continue;}

    Receive(m); {continue;}
      // get s's reply
    if (m.body ==
        "Synchronization reply: K[s],D[s]")
    {
      P = K - K[s];
        // The other server has purged
        //  these already; we can too.
      K = K - P;
      D = D - P;

      P = K[s] - K;
        // We have purged these already;
        //  the other server should too .
      K[s] = K[s] - P;
      D[s] = D[s] - P;

      D = D + D[s];
        // Merge in notes marked delivered
        //  on the other server.
    }
  }

  if (we have received reply from
      all servers in S) {
    // Purge those notes that everybody
    //  knows are delivered.
    P = D[1] intersect D[2] intersect ...
          intersect D[N];
    foreach i in P {
      Deliver(c,
              "Please purge i");
                    {continue;}
      K = K - {i};
      D = D - {i};
    }
  }
}
```

Finally, the client now accepts two kinds of messages: note delivery messages and purging authorizations.

```
Deliver(s, "Request my notes"); {abort;}

loop {
  Receive(m); {abort;}
  if (m.body == "Here is note i: ..." &&
      i not in R) {
    ProcessMessage(m);
    R = R + {i};
      // remember that we have
      //  received this message.
  }
  if (m.body == "Please purge i") {
    R = R - {i};
  }
}
```

Switching to the server-centric algorithm Server Synchronization, the server function becomes:

```
HandleClient(c) {
  foreach s in S, in parallel {
    Deliver(s,
            "Request for list D");
                          {abort;}
    Receive(m); {abort;}
    D[s] = m.body;
    D[s] = D[s] - (D[s] - K);
      // s may not know that some messages
      //  are already purged.
    D = D + D[s];
  }

  // Got response from each s.  Purge those
  //  messages that everybody knows about.
  P = D[1] intersect D[2] intersect ...
        intersect D[N];
  D = D - P;
  K = K - P;

  // Can start delivery now.
  U = K - D;
  foreach i in U {
    aDeliver(c,
             "Here is note i: ...");
                           {abort;}
    D = D + {i};
  }
}
```

Note that in order to simplify the code, we have the server abort as soon as it fails to contact one of the other servers. As a possible optimization, it may want to retry the problematic servers for a few times while letting the client hold.

We omit the rest of the server code and the client code for this strategy because they should be rather straightforward to construct.

Our next strategy, Delay Reconnect, also has a different HandleClient:

```
HandleClient(c) {
  foreach s in S, in parallel {
    if (TS[s] < GetCurrentTime() - Tr) {
        // < means earlier than
      Deliver(s,
              "Request for list D");
                           {abort;}
      Receive(m); {abort;}
      D[s] = m.body;
      TS[s] = m.timestamp;
      D[s] = D[s] - (D[s] - K);
      D = D + D[s];
    }
  }

  // We are safe. Can start delivery now.
  U = K - D;
  foreach i in U {
    aDeliver(c,
             "Here is note i: ...");
                           {abort;}
    D = D + {i};
  }
}
```

Lastly, Strategy Connect History looks like the following:

```
HandleClient(c) {
  // get connection history
  Receive(H); {abort;}

  foreach s in S, in parallel {
    if (TS[s] < H(s)) {
      Deliver(s,
              "Request for list D");
                           {abort;}
      Receive(m); {abort;}
      D[s] = m.body;
      TS[s] = m.timestamp;
      D[s] = D[s] - (D[s] - K);
      D = D + D[s];
    }
  }

  // We are safe. Can start delivery now.
  U = K - D;
  foreach i in U {
    aDeliver(c,
             "Here is note i: ...");
                           {abort;}
    D = D + {i};
  }
}
```

# B Expected throughput analysis

We will first derive a few constructs which will be needed by the analysis that follows. We refer the reader back to Figure 2. To begin, we give the formula for $E(d)$, the expected time it takes to deliver a message from node $A$ to node $B$ on the circle given the distance $d$ between them. Simply, $E(d)$ is just the expected value of delivery time $t$ based on the distribution $f_d(t)$:

$$
\begin{aligned}
E(d) &= \mathrm{E}_{f_d}[t] \\
&= \int_0^\infty t\, df_d(t) \\
&= \int_0^\infty t f_d'(t)\, dt \\
&= \int_{\alpha d}^\infty t \frac{1}{\beta d} e^{-\frac{t-\alpha d}{\beta d}}\, dt \\
&= -e^{-\frac{t-\alpha d}{\beta d}}(t+\beta d)\Big/_{\alpha d}^{\infty} \\
&= (\alpha+\beta)d \\
&= \lambda d
\end{aligned}
$$

Next we derive the communication delay between the client and its nearest server (between $c$ and $S_{i+1}$ in Figure 2). Because the circle has $D$ hops, and $N$ servers are spread evenly on it, the network distance between two adjacent servers, e.g., $S_i$ and $S_{i+1}$, is simply $\frac{D}{N}$.[6] Let $d_{sap}$ denote the average distance between the access point $NAP$ (a random point on the circle) and the nearest server $S_{i+1}$. This distance ranges from 0 (when the access point falls on a server itself) to half the distance between adjacent servers (when the point falls in the middle of two servers). Taking the average, we get $d_{sap} = \frac{0+\frac{1}{2}}{2}\frac{D}{N} = \frac{D}{4N}$. Finally, $t_{cs}$, the expected time it takes to deliver a message between the client and the nearest server, is simply the sum of the expected delay between $S_{i+1}$ and $NAP$ plus the constant delay between $NAP$ and $c$. That is,

$$
\begin{aligned}
t_{cs} &= E(d_{sap}) + t_{cap} \\
&= \lambda d_{sap} + t_{cap} \\
&= \frac{\lambda D}{4N} + t_{cap}
\end{aligned}
$$

Another quantity we will be using often is $t_{ss}$, the total expected time for a server to deliver in parallel

[6] For simplicity, we assume that the number of hops ($D$) is divisible by the number of servers ($N$).

one message each to the other $N-1$ servers in an $N$-server system. For simplicity's sake we assume that $t_{ss}$ is equal to the longest expected time among the $N-1$ messages[7]. Let $d_i$ denote the distance from a server to its $i$th nearest neighbor on a half circle. Then $d_i = \frac{D}{N}i$. Thus we get:

$$\begin{aligned} t_{ss} &= \max_{i=1\ldots\lfloor\frac{N}{2}\rfloor} E(d_i) \\ &= \max_{i=1\ldots\lfloor\frac{N}{2}\rfloor} \lambda\frac{D}{N}i \\ &= \lambda\frac{D}{N}\lfloor\frac{N}{2}\rfloor \end{aligned}$$

Lastly, we make two observations. First, the average time needed to deliver a message and receive an acknowledgment is simply twice the expected message delay. For example, it takes the server $2t_{cs} = 2(\frac{\lambda D}{4N} + t_{cap})$ time to deliver a note to the client and receive an acknowledgment. Second, assuming that notes are delivered sequentially, i.e., one after the acknowledgment of another, then the expected number of notes that can be delivered sequentially in time $T$ is simply $T$ divided by the time to deliver a single note. As an example, the expected number of notes a client can receive in time $T_c$ is $\frac{T_c}{2t_{cs}} = \frac{T_c}{2(\frac{\lambda D}{4N}+t_{cap})}$.

## B.1 Analysis

We derive the formulas for $p_i$, the expected number of notes received by the client in $T_c$ time under Strategy $i$. We propose a reference Strategy 0, which is an at-least-once algorithm that can potentially result in duplicate deliveries. Because of its relaxed semantics, we expect Strategy 0 to perform well in our throughput study. Hence we will use $p_0$ as our base of comparison.

In Strategy 0, because the server does not need to be concerned with exactly-once delivery, note delivery can start right after the client requests a connection. Therefore, the expected number of notes the client will receive is simply the total time of the connection ($T_c$) divided by the expected time it takes to deliver a single note ($2t_{cs}$). That is,

$$\begin{aligned} p_0 &= \frac{T_c}{2t_{cs}} \\ &= \frac{T_c}{2(\frac{\lambda D}{4N} + t_{cap})} \end{aligned}$$

[7]This is an optimistic estimate, because it is saying that delivering several messages in parallel takes only as long as delivering to the farthest one among them. This is clearly only an approximation.

Recall that in Strategy 1 (Sequenced Streams), notes are delivered in sequence number order, and the client remembers the latest received sequence numbers to prevent duplicate delivery. Strategy 2 (Sequenced Wrap-around Streams) is identical to Strategy 1 except that it provides the ability to reuse note ids. In both these strategies, once a client connects to a server, note delivery can begin. Consequently, except for the restriction on in-order delivery, these two strategies are capable of the same expected throughput as our base case, $p_0$. In other words,

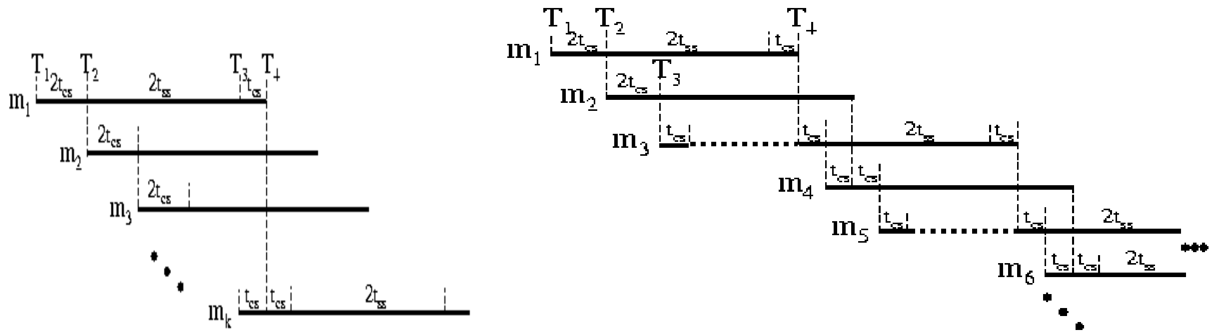
$$\begin{aligned} p_1 = p_2 &= p_0 \\ &= \frac{T_c}{2(\frac{\lambda D}{4N} + t_{cap})} \end{aligned}$$

In Strategy 3 (Id List), the client keeps a list $R$ of received note ids to guard against duplicates. As shown in the timing diagram (Figure 1(a)), the server sends a note to the client at time $T_1$. After receiving the acknowledgment ($T_2 = T_1 + 2t_{cs}$), the server immediately initiates the purging protocol[8], which requires it to synchronize with all the other servers ($T_3 = T_2 + 2t_{ss}$), and then inform the client ($T_4 = T_3 + t_{cs}$) to purge. However, the delivery of the second note $m_2$ need not wait for the purging of $m_1$, but can start as early as point $T_2$ in the figure.

From Figure 1(a), we see that a new note is delivered every $2t_{cs}$ time starting from time $T_1$. Moreover, after point $T_4$, entries in list $R$ are also purged at the same rate. Consequently, as long as the $R$ slots do not fill up by $T_4$, a steady state is reached where note ids are put in and taken out of $R$ at the same rate of once every $2t_{cs}$. As such, the expected throughput is simply $\frac{T_c}{2t_{cs}}$. To not fill up $R$ within time $T_4 - T_1 = 3t_{cs} + 2t_{ss}$, the client needs to have at least $\frac{2t_{cs}+2t_{ss}}{2t_{cs}} = 1 + \frac{t_{ss}}{t_{cs}}$ slots in its $R$. (The reason the numerator is $2t_{cs} + 2t_{ss}$ rather than $3t_{cs} + 2t_{ss}$ is that potentially the delivery of $m_k$ can overlap with the purging authorization of $m_1$ between $T_3$ and $T_4$ as shown in Figure 1(a).)

If, on the other hand, $R$ fills up before $T_4$, we have the more complicated situation depicted in Figure 1(b). Assume $q$, the number of slots in $R$, is 2. At time $T_3$, the server sends note $m_3$. However, because both slots in $R$ are used, the client cannot process $m_3$ until point $T_4$, when the slot for $m_1$ is freed. Similarly for later notes. Note that if the client chooses

[8]This "eager" mode of purging can potentially lead to heavy traffic between the servers because it requires a full server synchronization for each note delivered. A real implementation will likely optimize by batching several synchronizations into one, for example. However, the eager model gives the same throughput as the optimized versions under normal circumstances, and is a lot easier to study.




(a) Without $R$ overflow.

(b) With $R$ overflow ($q = 2$).

Figure B-1: Delivery timing diagram for Strategy 3.

to discard $m_3$ instead of holding it till $T_4$, additional delays may be needed to get the server to retransmit $m_3$.

A closer examination of Figure 1(b) tells us that the note delivery rate obtainable in a limited storage system is only $q$ (in this case, 2) notes per $2t_{cs} + 2t_{ss}$ time. That is to say, the expected number of notes received in $T_c$ time is $\frac{T_c}{2t_{cs}+2t_{ss}}q$. Putting the two cases together, we get:

$$
\begin{aligned}
p_3 &= \begin{cases} \frac{T_c}{2t_{cs}} & \text{if } q > 1 + \frac{t_{ss}}{t_{cs}} \\ \frac{T_c}{2t_{cs}+2t_{ss}}q & \text{otherwise} \end{cases} \\
&= \begin{cases} \frac{T_c}{2(\frac{\lambda D}{4N}+t_{cap})} & \text{if } q > 1 + \frac{t_{ss}}{t_{cs}} \\ \frac{T_c q}{2(\frac{\lambda D}{4N}+t_{cap}+\lambda\frac{D}{N}\lfloor\frac{N}{2}\rfloor)} & \text{otherwise} \end{cases}
\end{aligned}
$$

We now move on to the server-centric algorithms. In Strategy 4 (Server Synchronization), the server first needs to synchronize with each of the other servers in parallel. Because the synchronization takes $2t_{ss}$ time on average, we will deduct that time from the total time $T_c$. After the synchronization is complete, delivery starts and proceeds at the rate of $2t_{cs}$ time per note.

$$
\begin{aligned}
p_4 &= \frac{T_c - 2t_{ss}}{2t_{cs}} \\
&= \frac{T_c - 2\frac{\lambda D}{N}\lfloor\frac{N}{2}\rfloor}{2(\frac{\lambda D}{4N} + t_{cap})}
\end{aligned}
$$

In Strategy 5 (Delay Reconnect), the client promises not to reconnect into the system within time $T_r$. Each server initiates a synchronization with other servers every $T_u$ time. When a client connects to a server, the server first verifies that the last synchronization message received from each of the other servers was sent out at most $T_r$ ago. If so, actual note delivery can start right away. Otherwise, the server requires a setup period in which it needs to synchronize with the problematic server(s).

Let $p_s$ denote the probability that the latter is true, that is, a server synchronization period is needed before note delivery can start. Further assume that this synchronization takes an average of $t_s$ time. Hence on average the setup period takes $p_s t_s$ time, which we will deduct from the total connection time $T_c$. Thus, $p_5 = \frac{T_c - p_s t_s}{2t_{cs}}$.

As shown in Figure B-2, the client first connects to server $A$ at time $T_1$, and then to server $B$ at $T_4$. We assume that the distance between $A$ and $B$ is the farthest possible, i.e., $d_{AB} = \frac{D}{N}\lfloor\frac{N}{2}\rfloor$. During the period of $T_r$ time before the second connection (from $T_2$ to $T_4$), $A$ could potentially have attempted multiple synchronizations to $B$ ($s_2$ and $s_3$), the receipt of any of which by $B$ would have allowed $B$ to skip the setup with $A$. To first order of approximation, $p_s$ is equal to the probability that the first such synchronizations (i.e. $s_2$) did not make it to $B$. The message $s_2$ could have been sent at any time between point $T_2$ and point $T_3$ with equal probability. (Note that although our graph shows the situation where $T_r > T_u$, our derivation is valid for $Tr <= T_u$ as well.) If we look at an infinitesimal time segment $dt$ at time $t$ after point $T_2$, the probability that $s_2$ is sent during this period is $\frac{dt}{T_u}$. The probability that $s_2$ will reach $B$ by $T_4$ is simply $f_{d_{AB}}(T_r - t)$, which is

$$
\begin{cases} 0 & \text{if } T_r - t < \alpha d_{AB} \\ 1 - e^{-\frac{Tr - t - \alpha d_{AB}}{\beta d_{AB}}} & \text{otherwise} \end{cases}
$$

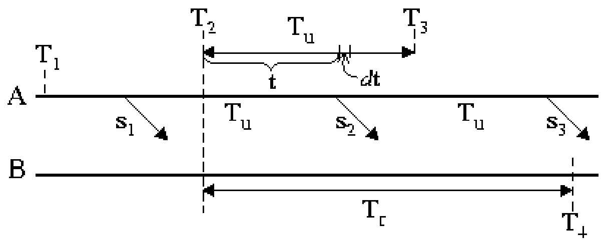



Figure B-2: Delivery timing diagram for Strategy 5.

Integrating with respect to $t$ gives us $p_s \approx 1 - \int_0^{T_u} \frac{dt}{T_u} f_{d_{AB}}(T_r - t)$. Since $f_{d_{AB}}(T_r - t)$ is zero when $t > T_r - \alpha d_{AB}$, simplifying further, we have

$$
\begin{aligned}
p_s &\approx 1 - \int_0^M \frac{1 - e^{-\frac{Tr - t - \alpha d_{AB}}{\beta d_{AB}}}}{Tu} dt \\
&= 1 - \frac{1}{T_u} \left(t - \beta d_{AB} e^{-\frac{Tr - t - \alpha d_{AB}}{\beta d_{AB}}}\right)\Big/_0^M \\
&= 1 - \frac{1}{T_u}\left(M - \beta d_{AB} e^{-\frac{Tr - \alpha d_{AB}}{\beta d_{AB}}} \left(e^{\frac{M}{\beta d_{AB}}} - 1\right)\right) \\
&= 1 - \frac{1}{T_u}\Big(M - \\
&\qquad \beta \frac{D}{N} \lfloor \frac{N}{2} \rfloor e^{-\frac{Tr - \alpha \frac{D}{N} \lfloor \frac{N}{2} \rfloor}{\beta \frac{D}{N} \lfloor \frac{N}{2} \rfloor}} \left(e^{\frac{M}{\beta \frac{D}{N} \lfloor \frac{N}{2} \rfloor}} - 1\right)\Big)
\end{aligned}
$$

where $M$ is a shorthand for $\max(0, \min(T_r - \alpha d_{AB}, Tu))$.

Similarly, we use the time it takes $B$ to synchronize with server $A$ to approximate $t_s$. Thus, synchronization time $t_s \approx 2t_{ss}$. Finally, we get:

$$
\begin{aligned}
p_5 &= \frac{T_c - p_s t_s}{2t_{cs}} \\
&\approx \frac{T_c - p_s 2t_{ss}}{2t_{cs}} \\
&= \frac{2N(T_c - 2\lambda \frac{D}{N} \lfloor \frac{N}{2} \rfloor p_s)}{\lambda D + 4N t_{cap}}
\end{aligned}
$$

Finally, we look at Strategy 6 (Connect History). Recall that this strategy is very similar to the previous one in that if the server has heard recently enough from the other servers, then note delivery can start right away. Otherwise, a synchronization period is needed. The difference, however, is that here the server has a list of the client's past connection history, and hence can perform a smarter checking.

We assume that when the client makes a connection to server $B$ at point $T_2$ (Figure B-3), there has only been one past connection, which was time $t'$ ago to server $A$. Right after the connection at point

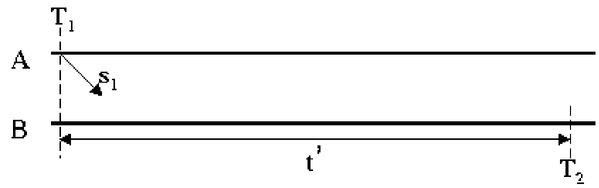



Figure B-3: Delivery timing diagram for Strategy 6.

$T_1$, server $A$ sent out a synchronization $s_1$. In other words, unlike the periodic synchronization of Strategy Delay Reconnect, here we assume that synchronizations are initiated at the end of each client connection. The distance between $A$ and $B$ is $d'$. The probability $p_s$ that $s_1$ has not arrived at $B$ by $T_2$ is $1 - f_{d'}(t')$. If that happens, the expected time $t_s$ it takes $B$ to perform a synchronization with $A$ is $2E(d')$.

Therefore, we have:

$$
\begin{aligned}
p_6 &= \frac{T_c - p_s t_s}{2t_{cs}} \\
&= \frac{T_c - (1 - f_{d'}(t'))2E(d')}{2t_{cs}} \\
&= \frac{T_c - e^{-\frac{t' - \alpha d'}{\beta d'}} 2E(d')}{2t_{cs}} \\
&= \frac{T_c - e^{-\frac{t' - \alpha d'}{\beta d'}} 2\lambda d'}{2(\frac{\lambda D}{4N} + t_{cap})} \\
&= \frac{2N(T_c - 2\lambda d' e^{-\frac{t' - \alpha d'}{\beta d'}})}{\lambda D + 4N t_{cap}}
\end{aligned}
$$