Exactly-once Semantics in a Replicated Messaging System

Yongqiang Huang, Hector Garcia-Molina
Department of Computer Science
Stanford, CA 94305
{yhuang, hector}@cs.stanford.edu

Abstract

A wide-area 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). Alternatively, they may not be delivered at all, making the user miss important messages. In this paper we address the problem of exactly-once delivery to mobile clients when messages are replicated globally. 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. The servers are usually distributed geographically over a wide area. 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.

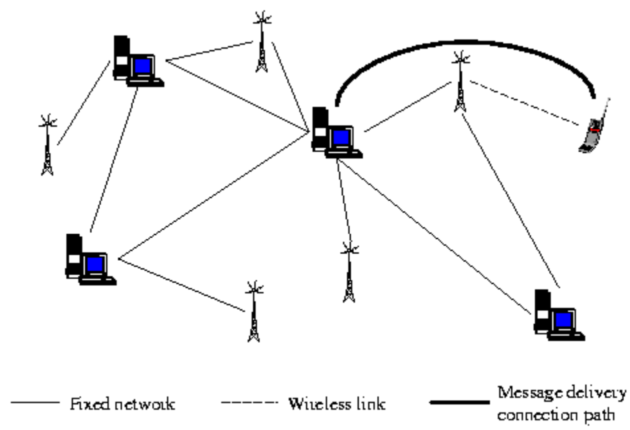

Figure 1. Replicated messaging system

Figure 1 illustrates such a replicated 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. 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.

Replication, of course, introduces the problem of duplicate delivery. A client can connect to one server and download note 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 *exactly-once* delivery semantics: a note should eventually be delivered to its target once, and only once. We first describe (Section 2) a set of exactly-once algorithms for a global replicated messaging system. Most of these algorithms exploit well-known ideas in other areas such as TCP/IP networking [6, 15, 5] and PCS messaging [1, 2]. Some, however, (e.g., Delay Reconnect) represent new variations. We also identify (Section 2.3) a "weaker" semantics called "relaxed exactly-once" which is satisfied by a subset of the algorithms. The new semantics is well suited for limited capability mobile devices. Finally, we propose (Section 3) a simple performance model suitable for comparing algorithms, and perform (Section 4) an analytical study of all the algorithms with regard to a common evaluation metric.

2 Exactly-once algorithms

In this section we describe algorithms that guarantee exactly-once delivery semantics. The reader is referred to a longer version of this paper [9] for implementation details. The algorithms are divided into two major groups: those (Section 2.2) that guarantee strict exactly-once semantics, and those (Section 2.3) that guarantee a "relaxed" version. 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.

Some of the algorithms are based on well known ideas such as sequence numbers and time stamps. Our contribution here is a comprehensive presentation of the algorithms, including some new variations, in a common framework that facilitates comparisons.

2.1 Delivery procedures

Exactly-once delivery algorithms 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. The predefined timeout guarantees that `Deliver(T,m)` will not block indefinitely in case of network partitions or disconnections, etc. 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. 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.

2.2 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 [6] with unbounded sequence numbers. However, our algorithm is a "multiple-stream" version of that protocol (because we now have multiple potential senders) 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, 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 is removed from K after it is successfully delivered to the client. Furthermore, the servers periodically send *synchronization* messages to each other to purge delivered notes. A synchronization message contains the sending server's K list (K_{sync}). The receiving server compares K_{sync} with its own K, and deletes from its own K any notes not in K_{sync} (that is, notes that have already been delivered by another server). See [9] 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. In this case B deletes $A.3$ from K_B. 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 ($R[A] + 1 = 3 + 1 = 4$). ∎

Due to space limitations, we defer correctness arguments for our algorithms to [9]. Informally, because notes in a stream are delivered sequentially, the client in effect remembers all the notes it has ever received in the past.

Strategy 1 assumes that sequence numbers never wrap around. Our next strategy extends Strategy 1 to deal with the sequence number wrap-around problem explicitly.

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, 5, 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, as illustrated in the next 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.*

Recall that server y synchronizes with server x by sending it K_y. If note $x.(s+1)$ has already been removed from K_y, x has effectively received a promise from y that y will not attempt to deliver note $x.(s+1)$ in the future.

Although Strategies 1 and 2 guarantee exactly-once semantics, they still have several potential shortcomings. The client needs a certain amount of stable storage to store its R array. Since the size of R is proportional to the total number of servers (N), the client either has to remember one sequence number for each potential server in the system, or it must be informed whenever servers join or leave the system. The strategy also does not scale well with many clients. Since not every note is destined for all the clients, a note must have a difference sequence number for each client in order to keep the sequence numbers consecutive. Consequently, a server must have as many sequence number counters as there are clients. Finally, notes with the same server id need to be delivered in strict sequential order unless the client is willing to remember more sequence numbers per server id. The following strategy is designed to address the above problems.

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. 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.

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 be able to purge entries from R. A note id m can only be purged from R after the client has received confirmation from all servers that m has been removed from their K lists. For details on the purging process please refer to [9]. Here we merely point out that, 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 one or more R entries are purged. 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 4.

2.3 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, however, a client-centric approach is not appropriate because the mobile client has no or very limited stable storage. 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.

A client-centric approach can also result in duplicate note transmission to the mobile device [1]. Although exactly-once semantics is not violated because the duplicate will be discarded by the client device before passing on to the user, duplicate delivery consumes valuable system resource such as the limited bandwidth between a wireless client and a server.

Therefore, in this subsection, we will develop a new suite of "server-centric" algorithms. 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 assumes the primary responsibility of remembering which notes have been delivered, and guaranteeing no duplicate delivery.

2.3.1 Atomic delivery

It is not hard to see that exactly-once semantics cannot be achieved with procedure `Deliver(T, m)` of Section 2.1 if 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 strict exactly-once semantics. Instead, we will strive to achieve a weaker notion of correctness. We recognize that the transmission of a note from a server to a client represents an unavoidable "window of vulnerability." That is, during this operation there will be some chance that the note is delivered but the acknowledgment is not known to the server. Our weaker semantics says that the algorithm must guarantee exactly-once semantics except for the unavoidable note delivery problems during the window of vulnerability.

We capture this weaker notion of correctness by assuming an atomic delivery procedure, denoted by `aDeliver(T,m)`. To define its properties, we first define a function $\rho(A, m)$. Function $\rho(A, m)$ represents the number of times that node A has received and processed the same message m[1]. 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 say that an algorithm guarantees *relaxed exactly-once* semantics when it guarantees exactly-once delivery under the assumption that `aDeliver` exists. The assumption that `aDeliver` exists allows us to concentrate on other parts of the algorithms, while acknowledging that server-centric algorithms are vulnerable to the "risky" note delivery operation.

2.3.2 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.*

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

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.

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.

Each server keeps a list D of notes that are "Delivered" but not yet "Purged." 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.

Example 3 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.[2] How the selection of parameter T_r affects the overall performance of the strategy will be investigated in Section 4. 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 with s occurred 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 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.

Example 4 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 [3]. 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. ▮

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. The tradeoff is the additional requirement on the client not to connect too frequently.

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

Each server remembers the most recent synchronization timestamps from each other server, 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 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 H to the server. 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 5 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 three-fold. First,

[2] 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).

[3] 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.

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, assume 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), B 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.

3 Evaluation

The relative merits and shortcomings of our algorithms can be evaluated along several dimensions. We have studied a number of 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 number of notes a client can expect to 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.

We model the global network 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.

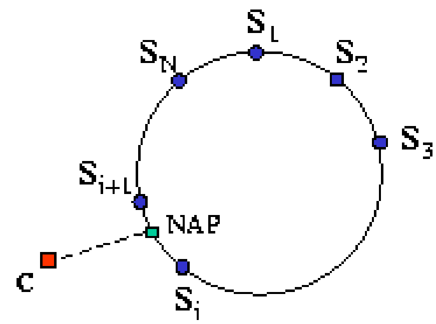

Figure 2. System model

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 circle (S_{i+1} in the figure), and connects to it. Once connected, the algorithms described in Section 2 will be used to guarantee exactly-once delivery of notes.

We model the characteristics of the fixed network by a distribution function $f_d(t)$, which gives the probability that node A can deliver a message to another node B distance d away on the circle within time t. Function $f_d(t)$ is given as:

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

To simplify our analysis, we assume that all messages (including notes) are relatively short, so that the message delay is length-invariant. Scaling constants α and β allow us to represent a large variety of networks, from slow and lossy networks to fast and reliable ones. Intuitively, α gives the minimum time per hop it takes to deliver a message.[4] Thus, α captures the maximum speed of the network subject

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

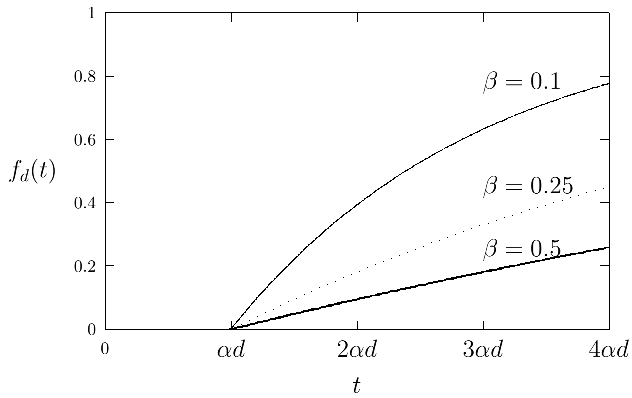

Figure 3. $f_d(t)$**,** $\alpha = 0.05$

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, since not all messages are delivered in the minimum possible time, we use β to reflect the "faultiness" of the network. We assume that, after time αd, the probability of a message still not delivered by time t decays exponentially with t. We use β to measure how fast this probability changes with t. 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 β are given in Figure 3.

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 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 4. 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.

We realize that there are other possible performance metrics besides the expected throughput which can be used to evaluate the algorithms. For example, we may study message delivery performance across several client connections instead of one connection. Consequently, we have to take into account additional system parameters such as the frequency and pattern of client reconnections and the message arrival rate, which may also be important factors in helping a system designer choose an appropriate strategy. In the future, we intend to study these alternative metrics and system parameters in more details as extension to this work.

Sym	Meaning	Unit	Base value
p_i	expected throughput of Strat. i	notes	
G	total # hops in global network	hops	30
N	total # servers	servers	10
T_c	connection duration	secs	15
α	param. of distribution func. f	sec/hop	0.05
β	param. of distribution func. f	sec/hop	0.25
λ	shorthand for $\alpha + \beta$	sec/hop	0.3
t_{cap}	delivery time btw c & NAP	secs	0.2
q	# slots to hold list R (Strat. 3)	entries	20
T_r	reconnect limit (Strat. 5)	secs	100
T_u	periodic sync interval (Strat. 5)	secs	300
d'	# hops to prev. server (Strat. 6)	hops	10
t'	time since prev. conn. (Strat. 6)	secs	10

Table 1. Parameters used in analysis

4 Results

Our expected throughput numbers are obtained analytically from our performance model. See [9] for derivations. In [9] we also propose a reference strategy, Strategy 0, as the basis of our performance comparisons. In Strategy 0, a server starts delivering notes as soon as the client is connected, without concerns for duplicate delivery. Thus, Strategy 0 only provides the weaker at-least-once semantics. Our performance numbers are plotted as the ratio of our exactly-once strategies over Strategy 0. 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 expected to receive half as many notes with Strategy i as with Strategy 0 for a connection of the same duration.

We have selected only a few representative graphs 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.

4.1 Strategy 3: Id List

Figure 4 gives the throughput ratio of Strategy Id List 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 correspond to different values of t_{cap}, the wireless link delay.

We observe that, in all cases, the ratio starts off low (high overhead) but improves as the client's stable storage increases. This is because when the client does not have many slots to hold its "received list" R, note delivery is suspended while the client waits for old note ids to be purged. In particular, there is a threshold above which Strategy Id List is able to perform as well as Strategy 0 ($\frac{p_3}{p_0} = 1$). As a rule of thumb, the minimum number of slots the client needs to avoid an R overflow is approximately equal to the

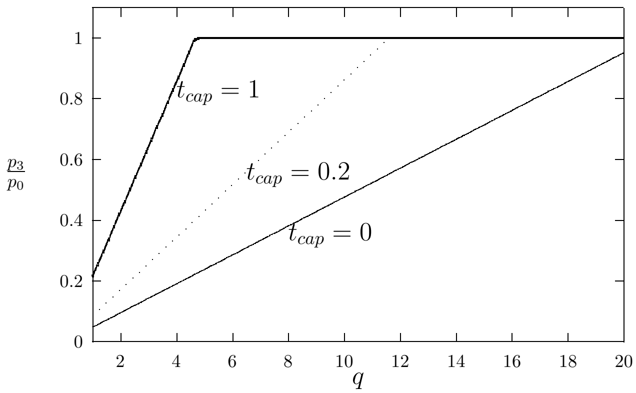

Figure 4. $\frac{p_3}{p_0}$ **versus** q

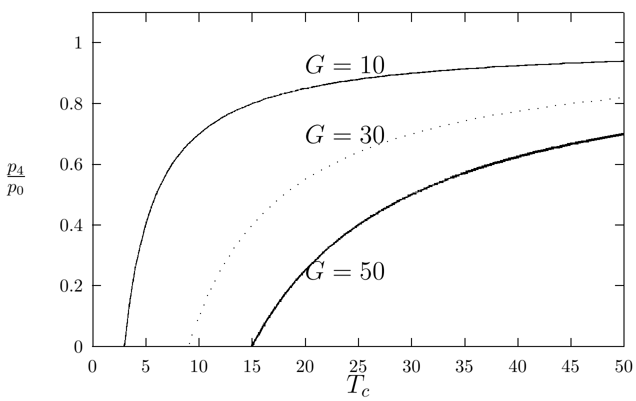

Figure 5. $\frac{p_4}{p_0}$ **versus** T_c

ratio of the time needed to purge a note id among the servers over the time to deliver a note. The graph shows 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) is sufficient. Thus, if the client device is designed to have stable storage at all, it should not be too expensive to provide enough storage capacity. Additionally, we see that the client needs much more stable storage with a very fast wireless link ($t_{cap} = 0$, zero delay) than with a slower link.

4.2 Strategy 4: Server Synchronization

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

As Figure 5 shows, short connections (small T_c) imply a high overhead because an initial server synchronization phase is needed. 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 is that performance decreases with increasing G. This is because, as the network grows larger (more hops around the globe), the cost of a server synchronization is greater.

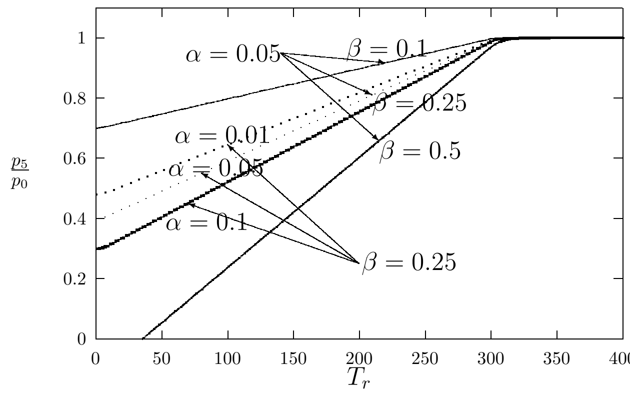

Figure 6. $\frac{p_5}{p_0}$ **versus** T_r

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

4.3 Strategy 5: Delay Reconnect

Figure 6 illustrates the effect of the reconnect limit T_r on Strategy Delay Reconnect. It also shows α and β variations to demonstrate the impact of network characteristics. In general, the longer the client is willing to wait in between connections (bigger T_r), the better throughput it will get. At $T_r = 0$, Delay Reconnect degenerates into Server Synchronization because the client can reconnect at any time. If the client is willing to wait at least one server synchronization period, T_u, the overhead is practically unnoticeable.

Figure 6 also shows that, with a slower (larger α) or a more "faulty" (larger β) network, performance suffers because server synchronizations take longer and are more likely to fail. Furthermore, we see that β has a relatively significant impact on performance.

4.4 Strategy 6: Connect History

Figure 7 has the same axes and general shape as Figure 5 (Server Synchronization). However, Connect History is effective in reducing synchronization overhead, as evidenced by the much smaller range of T_c plotted (0-5 sec compared to 0-50 sec in Figure 5).

The curves in Figure 7 represent different values of 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. If the client consistently connects to the same server ($d' = 0$), naturally Connect History

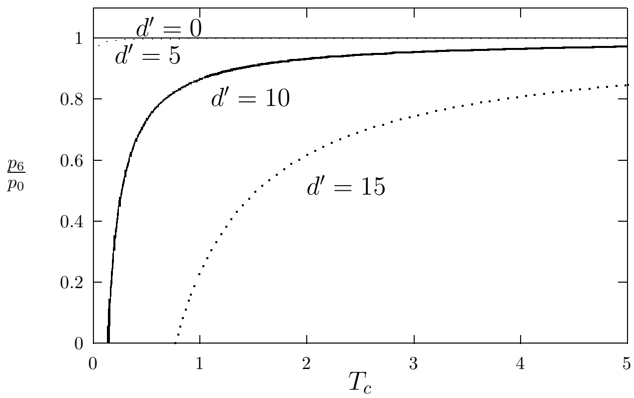

Figure 7. $\frac{p_6}{p_0}$ versus T_c

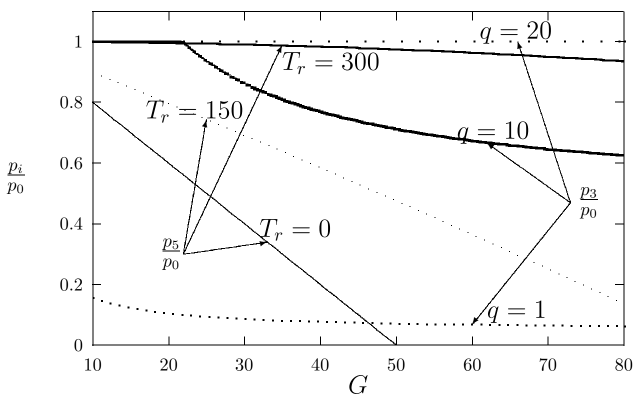

Figure 8. Comparison of $\frac{p_5}{p_0}$ and $\frac{p_3}{p_0}$ versus G

does not have any overhead ($\frac{p_6}{p_0} \equiv 1$). As d' increases, the performance overhead increases as well because there is a higher chance that a synchronization may be needed. Additionally, such a synchronization is also likely to take longer with a bigger d'. To summarize, Strategy Connect History works best when the client primarily makes infrequent connections to servers that are close together (small d').

4.5 Client- and server-centric comparison

Lastly, we plot a client-centric algorithm (Id List) and a server-centric one (Delay Reconnect) side by side to see how they scale with the size of the network (G). In Figure 8, each strategy is represented by three curves. The three Id List curves ($\frac{p_3}{p_0}$) differ in the amount of client stable storage (q), while the Delay Reconnect ones ($\frac{p_5}{p_0}$) in the amount of time the client is willing to wait between connections (T_r).

The important lesson to take away is that both strategies can perform fairly well under favorable conditions, and both can perform badly otherwise. For 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

Strategy	**1**	**2**	**3**	**4**	**5**	**6**
Strict exactly-once	√	√	√			
Relaxed exactly-once				√	√	√
Requires clnt stable storage	√	√	√			√[5]
Client storage ∝ # servers	√	√				[6]
1 uniq. id per clnt per note	√	√				
Requires in-order delivery	√	√				
Limit on client reconnects					√	
Performance sensitive to T_c				√	√	√
To pattern of connections						√
To size of client storage			√			
To freq. of periodic sync					√	

Table 2. Comparison of strategies

can be quite large even for small G. On the other hand, with enough storage, e.g. $q = 20$, the performance penalty can be avoided all together. Similarly, the key factor for Delay Reconnect seems to be how long the client is committed to wait between connections (T_r).

4.6 Summary of strategies

Table 2 compares 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. The third section deals with algorithm performance and looks at which parameters have an impact on the expected throughput for different strategies. The readers are referred to [9] for a more detailed explanation.

5 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) globally distributed servers, 2) limited or lacking stable storage on client, 3) wireless connectivity, 4) different notions of exactly-once, and 5) long-term semantics guarantees (spanning multiple connection sessions).

Algorithms to reliably multicast to mobile PCS users have been proposed in [1, 2]. These algorithms are often similar to some of the strategies presented in this paper. For example, [1] resembles a variation of Connect History.

[5] Strictly speaking, in Strategy 6 (Connect History), stable storage is needed for performance rather than required for correctness.

[6] 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.

However, our work differs from the mobile multicasting work in many significant ways. Their algorithms assume a strongly connected server network, while ours deal with a much more geographically distributed scenario. Hence, we use novel techniques such as delayed reconnections and timestamped histories to reduce server synchronization cost at connection establishment. While their algorithms are mostly server-centric, we point out the fundamental limitation of server-centric algorithms (they can only guarantee relaxed exactly-once), and present client-centric alternatives. We also perform comprehensive performance comparisons of all the algorithms (client- or server-centric).

The networking literature [4, 16] 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 [6, 15, 5] 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. Our algorithms have to worry about state synchronization among several servers over multiple connections. 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 wraparound problem has also been studied thoroughly [12, 5] in the context of window protocols.

There has been a great amount of work on replicated data management and replicated transactions [14, 7, 11, 8]. 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.

Another well-studied area in database research is transactional message queuing systems [3]. These message passing systems have been commercially available for a long time, such as IBM's MQSeries [10]. Such systems aim to provide transactional capabilities such as rollbacks and recovery to message delivery, while our work is concerned with delivery performance in a replicated environment.

6 Conclusion

In this paper we have studied the message delivery problem in mission-critical mobile scenarios where exactly-once semantics is imperative. We have proposed exactly-once algorithms, and developed a metric, expected note throughput, to compare them. We have found that the strategies differ in their performance, but almost every strategy can be tuned to minimize its overhead through careful selection of parameters. Overall, in many normal operating scenarios, the cost of exactly-once delivery is not much higher than that of at-least-once delivery. Thus, one can often select an appropriate strategy and operating parameters suitable for the intended operating environment such that exactly-once semantics is obtained at minimal or no extra cost.

References

[1] A. Acharya and B. R. Badrinath. A framework for delivering multicast messages in networks with mobile hosts. *Mobile Networks and Applications*, 1:199–219, 1996.

[2] V. Aravamudhan, K. Ratnam, and S. Rangarajan. An efficient multicast protocol for PCS networks. *Mobile Networks and Applications*, 2:333–344, 1997.

[3] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publisher, Inc., 1997.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] Y. Huang and H. Garcia-Molina. Exactly-once semantics in a replicated messaging system. Technical report, Stanford University, 2000. `http://dbpubs.stanford.edu/pub/2000-7`.

[10] IBM MQSeries. `http://www.ibm.com/software/ts/mqseries/`.

[11] 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.

[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.