Chapter 4

Announce-Listen

In this chapter, we explore Announce-Listen as a scalable form of group communication. We present the core Announce-Listen algorithm, as well as an enhanced version with caching. We describe the algorithm's salient parameters and several important metrics for gauging its performance: consistency, convergence time, messaging overhead and memory usage. We focus on the derivation of a model for the consistency metric, providing analysis and simulation, then discuss the other metrics in terms of how they meet consistency requirements. This chapter concludes with an assessment of previous work on this topic, a summary of our findings, and a discussion of future directions for our research.

4.1 Core Algorithm

Each process participating in the Announce-Listen algorithm makes announcements at a fixed periodicity, T (Figure 4.1). An announcement from process p to process q experiences a transmission delay of Δ_{pq} and contains one piece of essential information, a key-value pair that is stored in a table at the recipient. The key is an identifier for the announcing process and the value is the state that the process disseminates. For example, a process might announce its location, its load, or even the time it expects to send the next announcement. Each process also listens for announcements from other processes. We call this table a registry and the collection of registries at each listener forms a distributed directory.

Announcements are sent to a multicast group address where they are disseminated simultaneously to all participating processes (Figure 4.2). Due to a combination of transmission delay and loss in the network, it may take multiple announcement periods for a particular piece of data originating at the sender to reach all of the receivers.

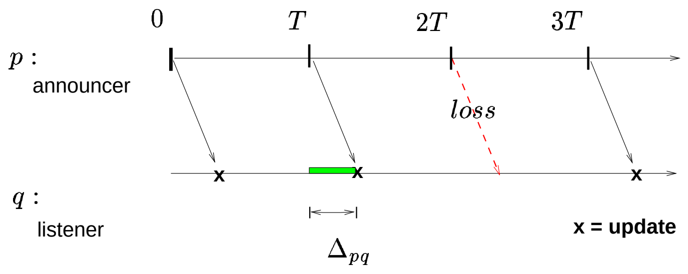

Figure 4.1: **Periodic Announcements and Updates.**

The basic algorithm is shown in Program 4.1. Each process acts as an announcer and sends its state information to the multicast address every T units of time. It accomplishes this by sending an announcement, then setting an announce timer to expire T units of time in the future. When the timer expires, the process sends the next announcement. Concurrently, each process acts as a listener on the same address, listening for announcements from other processes. On the receipt of an announcement, process state information is stored in the listener's directory, where it is stored indefinitely. Specifically, the entry in the registry indexed by *msg.key* is updated to store the value *msg.value*. In our studies we consider every process to be simultaneously both an announcer and a listener.[1]

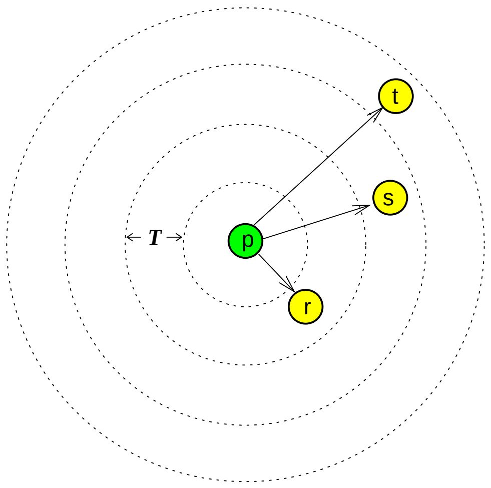

Figure 4.2: **Announcing to Multiple Processes.**

A more realistic representation of the AL algorithm is that a process caches entries for a limited time. Thus, we introduce a modified version of the algorithm in Program 4.2 that ages state information and removes it from the cache upon final expiration. We use this algorithm as the basis for the analysis in this chapter.

[1]Although we use the same address for announcing and listening, an implementation might use separate group addresses for each of these tasks. The effect of this would be that the process could be an "announce-only" process for one address, and a "listen-only" process for the other address.

```
ANNOUNCE-LISTEN ($T$)
send_announcement ()
set_announce_timer ($T$)
do
    if receive_announcement ($msg$) then
        update_registry_entry ($msg.key$, $msg.value$)
    if announce_timer_expired () then
        send_announcement ()
        set_announce_timer ($T$)
```

Program 4.1: **Announce-Listen Algorithm.**

```
ANNOUNCE-LISTEN-WITH-CACHING ($T_A$, $T_L$, $max\_age$)
send_announcement ()
set_announce_timer ($T_A$)
do
    if receive_announcement ($msg$) then
        update_registry_entry ($msg.key$, $msg.value$)
        set_listen_timer ($msg.key$, $T_L$)
    if $i$ = listen_timer_expired () then
        $age$ = age_entry ($i$)
        if $age > max\_age$ then
            remove_registry_entry ($i$)
        else set_listen_timer ($i$, $T_L$)
    if announce_timer_expired () then
        send_announcement ()
        set_announce_timer ($T_A$)
```

Program 4.2: **Announce-Listen Algorithm with Caching.**

The new algorithm relies on multiple timer values: T_A, the announcement periodicity, and T_L, a cache entry renewal periodicity. T_A behaves identically to T in the original program, and is used as the bound on the announce timer that reminds an announcer to send the next announcement. T_L is the bound on the wake-up timer that reminds a listener either to expire or to renew an entry in the registry.

A process begins the algorithm by sending an announcement message, then setting the announce timer to expire in T_A units of time. The process waits for one of three events to occur:

1. If the process receives an announcement, it caches a key-value pair in its registry, as before; the entry *msg.key* is updated to *msg.value*. However, now *each* registry entry is associated with an expiration timer. Thus, the listener resets the listen timer for entry *msg.key* to expire T_L units of time into the future.

2. If the listen timer expires, the process did not hear an update from the announcer associated with the i^{th} entry within the last T_L units of time. The process ages the registry entry, incrementing its age by one. The variable `age` depicts how many consecutive expirations have occurred for an entry at a given point in time. The expiration process allows *max_age* such expirations, before removing the data permanently. Whenever an announcement arrives for a given entry, the `age` variable is reset to 0 inside `update_registry_entry()`.

3. If the announce timer expires, the process issues its next announcement and resets the announce timer.

Although there exist variations of AL that send announcements using adaptive announce timers [55], for simplicity we examine the behavior of the algorithm when all entries use the same fixed announcement periodicity. Adaptive timers have been used to handle an influx of group members while keeping the overhead of communication fixed. If the membership grows large, the idea is to reduce the frequency of messages by increasing the periodicity, T_A, resulting in fewer messages. Likewise when the group size shrinks, the frequency of messaging increases.

There are also implementations that use progressively larger listen timer expiration values between each round of expiration, but we assume processes use the same fixed listen timer value, not only between each other, but also between rounds of aging cache data (e.g., step 13 in Program 4.2).

Here, we make the simplifying assumption that $T_A = T_L = T$. However, we assume `max_age` is set high enough to allow data to transit the network (e.g., $max\_age \geq 2$) and to accommodate the level of message loss in the network. In effect, announcements are issued at regular intervals of T_A and their state is cached for a duration of up to $T_L = kT_A$, where $k =$ `max_age`, typically a small integer value.

4.1.1 Scalability

Announce-Listen (AL) is an algorithm that uses periodic messaging to propagate local state information to remote processes. The idea of using announcements, with no form of feedback, is in contrast to traditional acknowledgment-based messaging, where the receipt of a message triggers an explicit acknowledgment (ACK) or where the detection of a missed message triggers a negative acknowledgment (NACK). One motivation behind AL is that an acknowledgment-style handshake between processes works well between pairs of processes, but can be problematic among groups of processes. As stated earlier, when a message is multicast to a large group of processes, there is the potential that implosion may result from simultaneous ACKs. Even when only NACKs are used to indicate negative vs. positive receipt of a message, implosion may still result when losses are correlated and simultaneous NACKs might be generated within a whole branch of the multicast routing tree. Therefore, AL offers groups of processes the potential to avoid implosion and to scale more gracefully than traditional ACK/NACK messaging.

Another motivation for AL is to allow large groups of processes to be unencumbered by a feedback mechanism. Whereas Suppression attempts to reduce the number of responses and to spread them out over the Suppression interval, Announce-Listen attempts to eliminate explicit process interaction altogether. This is important in scenarios where time is of the essence. The ability to "not wait" for an acknowledgment removes not only a round-trip time (RTT) of delay, but also eliminates the considerable extra delay incurred when a message goes unacknowledged and must be retransmitted several times. AL can be thought of as decoupling the sending process from the receiver process(es), allowing each to behave asynchronously. Thus, AL not only generates fewer messages by removing feedback messages, but it also removes the delay associated them.

However, the removal of an explicit, tightly-coupled feedback mechanism has a direct impact on the way in which reliable data distribution can be implemented in such a system. An implicit form of feedback is possible, for example by piggy-backing feedback information in announcements themselves (see Section 4.9), but the information is now delayed until such time as an announcement is scheduled by the recipient (the reporter of feedback) in the direction back to the original sender.

Nonetheless, a compelling argument for AL is that there are applications in which reliable data delivery is not of critical importance. There are also applications where eventual data delivery is an acceptable level of service, and still others where we can forego reliability entirely. For instance, systems with a real-time element, or ones where data changes rapidly, may not require reliable delivery because by the time data is retransmitted it has changed anyway. An example that falls in this class of applications is a mobile device that announces location coordinates on a regular basis.

The reasoning behind why AL is a suffiicient alternative is that, through repeated announcements, a message eventually will reach its intended destination(s), and with repeated announcements the state at receivers will track the state of the announcer. Thus, for applications that can tolerate a more loose form of communication, guaranteed delivery of every message (in the form of acknowledgments or otherwise) becomes less necessary.

In short, Announce-Listen offers scalability to groups of communicating processes because it removes feedback messages thereby avoiding message implosion, and eliminates waiting for feedback messages. It is beneficial to use AL in lieu of acknowledged messaging in situations where there is little to no back-channel bandwidth or, even if there is a modicum of bandwidth, where feedback messages would overrun the sender. AL messaging is resilient to faults in the network and changes in group membership because state is continually replenished so processes can quickly learn the state of the system. However, scalability hinges on the fact that data changes happen at a high enough rate to warrant continual updates. Furthermore, AL is most appropriate for those applications where the eventual consistency of this loosely-coupled messaging model can be tolerated.

4.1.2 Model Parameters

In addition to T_A, the announcement periodicity, and T_L, the cache entry expiration timer, there are several other parameters used in our model: N, the number of processes participating in the algorithm; Δ, initially a fixed message transmission delay between processes; p, the probability of message loss; k, the number of times a cache will age an entry before expiring it entirely.

The update periodicity, the frequency with which the content of announcement messages changes (separate from process arrivals or departures) is given by the parameter r, where $r \geq T$. When $r = \infty$, the data value never changes, so each announcement contains the same data as the first announcement. When $r = T$, the data value changes with each announcement. Although the data may change more frequently than T, the announcement periodicity, there is no way to distribute it more quickly. Therefore, initially we assume $r \geq T$.

A process is considered to have *departed* from the system when it stops making announcements. Otherwise, a process is considered *alive.* $D(t)$ is the cumulative distribution function of the probability that a process departs t units of time after it arrives. $A(t)$ is the cumulative distribution function of the probability that a process arrives into the system at time t. The relationship between $D(t)$ and $A(t)$ is highly group specific. For instance, in a collaborative session that is scheduled to begin at a specific time, arrivals are likely to cluster around the beginning of the session, whereas departures are likely to cluster at the ending time. Membership is very stable during the middle of the session. In an application with mobile group members, arrivals and departures may be extremely dynamic.

These parameters are summarized in Table 4.1.

Parameter	*Description*
T_A	announcement periodicity
T_L	cache entry renewal interval
N	number of processes participating
Δ	transmission delay
p	message loss probability
k	maximum entry age
r	data change periodicity
$D(t)$	cumulative probability of process departure
$A(t)$	cumulative probability of process arrival

Table 4.1: **Announce-Listen Parameters.**

4.1.3 Metrics

There are several metrics by which we can evaluate Announce-Listen. A key metric is the level of consistency achieved among the N communicating processes as a function of time. Consistency is a measure of how closely information stored in caches at the listeners compares with the actual state of the announcers. Consistency is challenged whenever new information arrives into the system (through an announcement either updating old information or reporting the arrival of a new process) or old information departs (as might happen when data expires). A common goal for the Announce-Listen technique is to maximize consistency while minimizing the other metrics: convergence time, messaging overhead and memory usage.

Given consistency as a function of time, $C(t)$, we can determine the convergence time of the listeners, $t(c_o)$, i.e., the minimum time it takes to reach a particular level of consistency, c_o.

We derive a metric for messaging overhead, in terms of the amount of bandwidth consumed by the algorithm; we predict the number of messaging attempts that are necessary to attain a given level of consistency and discuss the fraction of those that are redundant due to the update periodicity (of information inside of announcements) or due to the nature of the multicast distribution.

Additionally, we present a metric for memory usage, the amount of memory consumed to achieve a certain level of consistency. This metric is particularly useful for properly tuning the cache expiration strategy for listeners, which must weigh the cost of storing information for too long against storing it for too short a time. While the optimal cache expiration strategy may be unimportant for memory-rich systems or ones with secondary storage, it may be critical for devices that are memory poor, such as hand-held devices or embedded sensors.

Whereas we focused on time and overhead metrics for the Suppression technique, for Announce-Listen we concentrate on consistency.

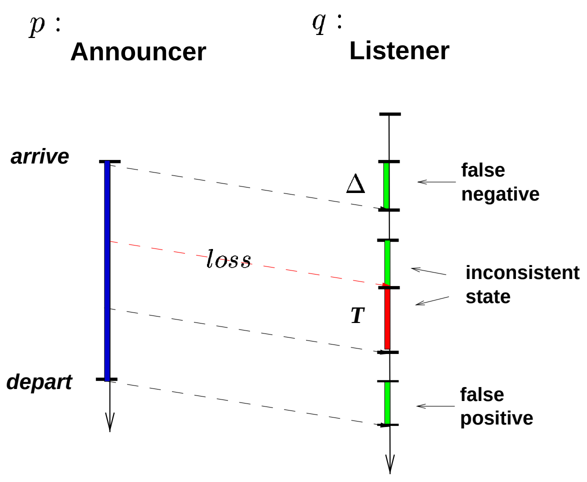

Figure 4.3: **Listener Errors.**

4.2 Consistency: Registry Cost

We can reformulate the issue of consistency by asking how good is the distributed registry built from processes using Announce-Listen? By *good* we mean how accurate is the registry. To answer this question, we define an objective function that attaches a cost to several types of errors: false negatives, false positives, and inconsistent state. We illustrate these errors using two communicating processes, an announcer and a listener process as shown in Figure 4.3.

- **false negative**: $p.announcing$ is true (process p is issuing announcement messages), but state information (a key-value pair) for p does not appear in the registry associated with the listener process q, $q.registry$. The cost of omitting information from the registry is $cost_n$.

- **false positive**: process p is a member of $q.registry$, but it is no longer issuing announcement messages. The cost of keeping expired information in the registry is $cost_p$.

- **inconsistent state**: process p is a member of $q.registry$ and is issuing announcement messages, but the information pertaining to p in $q.registry$ does not match the information for p in its own registry, $p.registry$. The cost of maintaining incorrect information in the registry is $cost_s$.

The severity of these three errors is not the same. While a false positive is misleading, in that it provides stale information about a process no longer participating in the registry, a false negative provides no information at all. Similarly, a piece of inconsistent information could be better than no information. For example, if the registry stores resource location information, a false positive is potentially useful in that it provides a hint of past history which may be used to track down the process in the future, if the process resumes announcements to the group at a later time. In this context, wrong state information also has potential benefit, even more so than false positives. Because the process is actually still participating, the information in the registry, while stale, may allow the process to track down the correct value through proxying or forwarding entities.

Thus for this example, $cost_n \geq cost_p \geq cost_s$ and the objective is to minimize the overall cost of inconsistencies. Other scenarios may differ as this is application dependent. For instance, if a theatre is announcing its movie schedule, a false positive may mean someone may make an unnecessary trip. On the other hand, a false negative may mean one simply schedules to see the movie later or elsewhere.

The cost of q's registry, $Cost(q)$, is given by:

$$\begin{aligned}
&Cost(q) = \\
&\quad (\# \, p :: p.announcing \wedge (\forall v :: (p, v) \notin q.registry)) * cost_n + \\
&\quad (\# \, p :: \neg p.announcing \wedge (\exists v :: (p, v) \in q.registry)) * cost_p + \\
&\quad (\# \, p :: (\exists v :: (p, v) \in q.registry) \wedge (p, v) \notin p.registry)) * cost_s
\end{aligned}$$

$Cost(q)$ is a random variable that depends on which processes are in the system, and on which messages are lost by the network. Therefore, we evaluate AL by examining the average cost of a registry, $E[Cost(q)]$.

Below we derive the likelihood of false positives, false negatives, and inconsistent state. First we create a model of the listener state transition probabilities. Next we relate it to an estimate of any given registry entry being in error, then revise these estimates to accommodate process departures and discuss the impact of process arrivals.

Once we can approximate the degree of error on a per entry basis, we can deduce the overall consistency in a given registry, and from that we can extrapolate the overall consistency of a collection of distributed registries participating in the Announce-Listen algorithm.

4.2.1 Listener State Transition Probabilities

A model of the listener process is depicted in Figure 4.4, which displays listener state transition probabilities.

When a listener receives an announcement with probability $1 - p$ from a remote process, the process becomes newly `Alive`, i.e., the announcer's state is entered into the listener's registry. Once registered, the remote process refreshes its state periodically by sending renewal messages. These renewal messages are received with probability $1 - p$, and dropped with probability p. If dropped, the listener cycles through k `Not Sure` states before marking the announcing process as `Departed`, i.e., expiring the announcer's state from the registry. As the listener progresses from one `Not Sure`

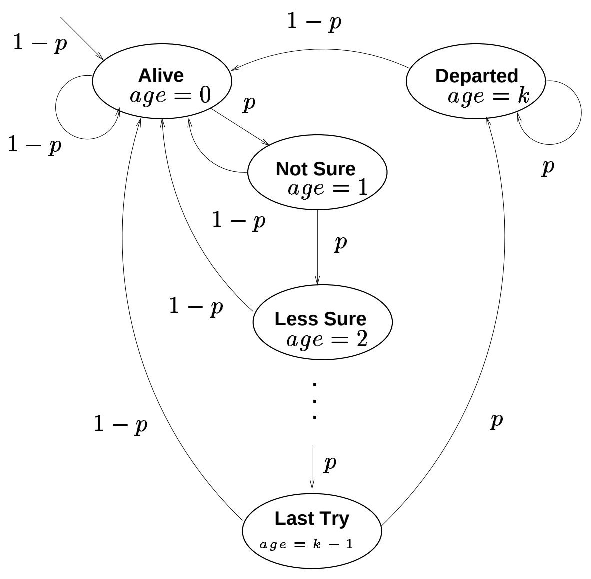

Figure 4.4: **State Transition Probabilities.**

state to the next, the listener becomes progressively less certain as to the state of the announcer. The state `Last Try` is the last state in which the listener maintains a valid registry entry for the announcer. If the listener regains communication with the announcer, the announcer is considered `Alive` again and its state is refreshed.

Each state is also associated with the *age* of the entry, which appears as a state name in Figure 4.4. For an entry to have $age = m$, m listen timers must have expired and the m^{th} timer expired at time mT_L (relative to when the entry was inserted into the registry). The age changes to $(m+1)$ at time $(m+1)T_L$ if no announcements were received in the interval $[mT_L, (m+1)T_L]$. That means all announcements sent in $[mT_L - \Delta, (m+1)T_L - \Delta]$ were lost. The number of announcements is:

$$a = \left\lfloor \frac{(m+1)T_L - \Delta}{T_A} \right\rfloor - \left\lfloor \frac{mT_L - \Delta}{T_A} \right\rfloor$$

Therefore, the state transition probability from $age = m$ to $age = m+1$ is p^a. When $T_L = T_A$, $a = 1$, giving the probabilities shown in Figure 4.4.

4.2.2 Error Model

Figure 4.5 associates listener actions with each state transition and depicts the listener's event-driven caching and aging strategy. We use the convention that state transitions are labelled with $\frac{A}{B}$, where A is an event that occurs, and B is the consequence of the event occurring. The diagram also offers another way to think about the listener model, by comparing it to the model maintained by the announcer.

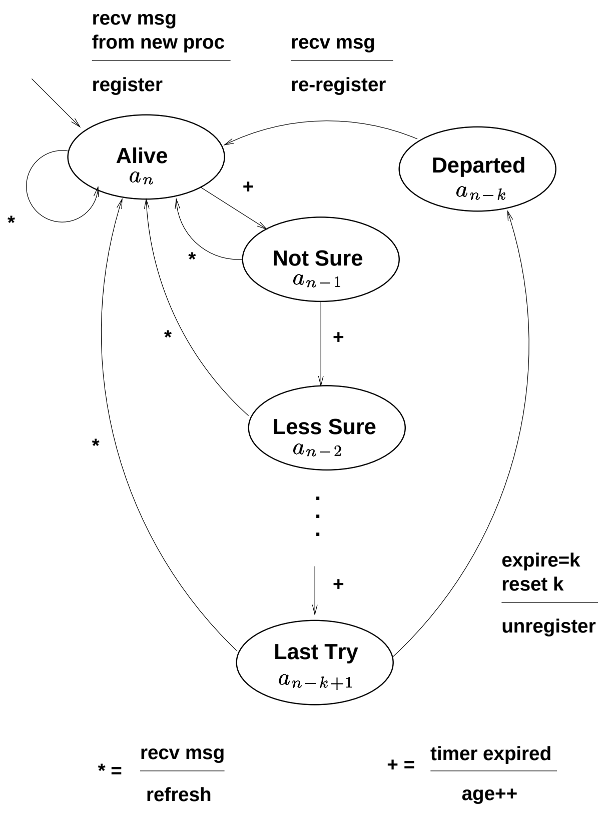

Figure 4.5: **Event-Driven Caching and Aging Strategy.**

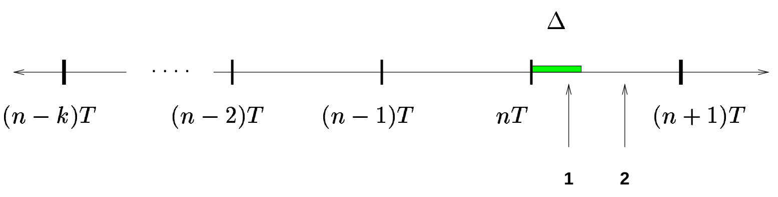

Figure 4.6: **Boundary Condition at Δ.**

When a listener receives an announcement from a process not currently in the registry, it registers the announcer as being `Alive`. The listener refreshes the announcer's registry entry with the receipt of each subsequence announcement message. If the listen timer T_L expires, the listener ages the announcer's state information and becomes `Not Sure` of the announcer's current state. The listener is willing to progress the announcer through $k-1$ stale states. If k expiration periods pass, the announcer is considered `Departed` from the system and its entry is removed from the registry. At such time as a new message is received from the announcer, the listener re-registers the announcer as being `Alive` once again.

If an announcer process begins at time 0, then the number of announcements sent to a listener process by time t is effectively $n = \lfloor t/T \rfloor$. If the listener stores what it thinks is the current state of the announcer, and the announcer has issued n announcements, then ideally the listener state is a_n, the data value sent in the announcer's n^{th} announcement. However if the n^{th} announcement is lost, the listener may contain a previous data value from an earlier announcement. The probability that the listener has cached old state from the m^{th} announcement, a_m, is the probability that the listener actually received the m^{th} announcement and lost all subsequent announcements between m and n, the current announcement:

$$\begin{aligned} Pr[\mathit{listener\ state\ is}\ a_m] &= Pr[\mathit{received}\ m^{th}\ \mathit{announcement}] * \\ &\quad Pr[\mathit{missed\ all\ states\ after}\ a_m] \\ &= (1-p) * p^{n-m} \end{aligned}$$

Now consider the fact that the listener ages the announcer's entry through k old states before removal from the registry: states a_n through a_{n-k+1}, which also appear as state names in diagram 4.5. The *cost* of being in one of these states is captured by its distance from the `Alive` state, a_n. If the listener state is a_s, then the *cost* is given by $n-s$. If the distance $n-s > k$, then the announcer is considered `Departed` and its state expired.

An application may have different requirements regarding the consistency of the registry. We would like to calculate the likelihood that the error in the registry is less than some tolerance e, denoted $Pr[Err \leq e]$, where $0 \leq e \leq k$.

Given the current time t, we know that $a_{\lfloor t/T \rfloor}$ is the last message sent by the announcer. This message takes time Δ before it can be received. We consider two cases: $t < \Delta + \lfloor \frac{t}{T} \rfloor T$ and $t \geq \Delta + \lfloor \frac{t}{T} \rfloor T$. In the former case, we can never be consistent with the announcer because of the network delay. The boundary condition that arises at the listener is displayed in Figure 4.6.

Therefore, $Pr[Err \leq e]$ is evaluated separately in each of these intervals. The probabilities of the listener process being in each of the states are shown in Tables 4.2 and 4.3. The tables are similar, except that when t is within Δ of the Announce message, the listener could not possibly have received the last message, a_n, regardless of whether or not the network loses the message.

State	*Name*	e	$P[Err = e]$
a_n	*Alive*	0	0
a_{n-1}	*Not Sure*	1	$1-p$
a_{n-2}	*Less Sure*	2	$(1-p)p$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_{n-k+1}	*Last Try*	$k-1$	$(1-p)p^{k-2}$
a_{n-k}	*Departed*	k	p^{k-1}

Table 4.2: **Inconsistent State:** $nT \leq t \leq nT + \Delta$.

State	*Name*	e	$P[Err = e]$
a_n	*Alive*	0	$1-p$
a_{n-1}	*Not Sure*	1	$(1-p)p$
a_{n-2}	*Less Sure*	2	$(1-p)p^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_{n-k+1}	*Last Try*	$k-1$	$(1-p)p^{k-1}$
a_{n-k}	*Departed*	k	p^k

Table 4.3: **Inconsistent State:** $nT + \Delta \leq t \leq (n+1)T$.

The probability within each interval that $Err \leq e$ is a sum of the probabilities that $Pr[Err = i]$ for i from 0 to e. The probability of the current time t falling within (versus after) Δ of the n^{th} announcement is proportional to the ratio of the transmission delay to the announcement periodicity Δ/T (versus $1 - \Delta/T$). By within Δ, we mean $t < \Delta + \lfloor \frac{t}{T} \rfloor T$. By after Δ, we mean $t \geq \Delta + \lfloor \frac{t}{T} \rfloor T$.

$$\begin{aligned} Pr[Err \leq e \mid within\ \Delta] &= \sum_{i=0}^{e} Pr[Err = i]_{within\ \Delta} \\ &= (1-p)(1 + p + p^2 + \cdots + p^e) \\ &= 1 - p^{e+1} \end{aligned}$$

$$Pr[Err \leq e \mid after\ \Delta] = \sum_{i=0}^{e} Pr[Err = i]_{after\ \Delta}$$

$$
\begin{aligned}
&= (1-p)(1+p+p^2+\cdots+p^{e-1}) \\
&= 1-p^e
\end{aligned}
$$

$$
\begin{aligned}
Pr[within\ \Delta] &= \frac{\Delta}{T} \\
Pr[after\ \Delta] &= \frac{T-\Delta}{T}
\end{aligned}
$$

$$
\begin{aligned}
Pr[Err \leq e] &= Pr[Err \leq e \mid within\ \Delta] * Pr[within\ \Delta] + Pr[Err \leq e \mid after\ \Delta] * Pr[after\ \Delta] \\
&= 1 - p^{e+1} - p^e(1-p)\left(\frac{\Delta}{T}\right)
\end{aligned}
$$

4.2.3 Departures

Tables 4.2 and 4.3 are based on the condition that the announcer process is alive and they assume that the announcer only leaves at the state `Departs`. In fact, the announcer may have departed earlier. Therefore, we need to refine the earlier tables and take the departure distribution probability, D(t), into account, focusing in particular on the probability of the announcer being alive at time t, $\overline{D}(t) = 1 - D(t)$.

Below we consider the current time t falling in the interval after Δ, as well as within Δ. In each of these intervals, we present examples of different states and explain that each state represents multiple possibilities, i.e., each leads to different types of errors that arise in the system (false negatives, false positives, and inconsistent state).

Appendix A contains Tables A.1 and A.2 summarizing the observations below. Although not in the tables, we discuss the impact of the rate of data change, r, on the analysis in Section 4.3.

After Δ. Correct states in the system only occur when the current state at the announcer and listener are identical; when the listener considers the announcer `Alive` and the announcer process is not dead, or when the listener thinks the announcer `Departed` and the announcer has departed. According to Figure 4.5, this happens in state a_n when the announcer is alive and the message was received successfully by the listener. This also happens in state a_{n-k} when the announcer has departed and the listener finally expires the announcer's state.

For example, here is how we can refine the `Departed` state by considering departure probabilities, when t occurs after Δ. When a process is not alive at time t, it could have departed at anytime in the interval $[0, t]$. Therefore, the probability of a process not being alive at time t, X_t, is the sum of the probability that the process died at time $(n-k)T$ (the earliest point at which the listener could have detected the departure), plus the probabilities that the process departs within any announcement interval between $(n-k)T$ and the current time t. If the process departs in an intermediate interval, it must also have been preceded by message loss for the listener to be in state a_{n-k}.

$$\begin{aligned}
X_t &= D((n-k)T) + \\
&\quad (D((n-k+1)T) - D((n-k)T)) \times p + \\
&\quad (D((n-k+2)T) - D((n-k+1)T)) \times p^2 + \\
&\quad \vdots \\
&\quad (D(t) - D(nT)) \times p^k
\end{aligned}$$

Another departure from Tables 4.2 and 4.3 arises when the listener process receives an announcement message. In that case, there is no falsely believing the announcer has departed. Thus, a false negative error is not possible. Likewise, a false positive error is not possible when the announcer is actually departed from the system, since we cannot falsely believe the announcer is alive.

Within Δ. When we consider the interval within Δ, there is only one correct state in which the listener and announcer match. This occurs in state a_{n-k} (Figure 4.5) when the announcer has departed and the listener detects it. In that state, it is not possible to have a false positive error where the listener falsely believes that the announcer is alive when it is not. However, it is possible for a false negative to occur if k messages have been lost and the announcer is actually still alive. Except for that case, a false negative error is not possible because a message is always received; therefore, the listener cannot falsely think the process has departed. Finally, it is not possible for the listener to be within Δ of the announcement and for its registry to contain the correct data value for the announcer, since the listener has not received the update message yet.

The calculation for the probability of a process not being alive at time t, Y_t, is similar to that of X_t. However, there is one fewer message to account for because a message could not have possibly reached the listener yet, due to network transmission delay, Δ.

$$\begin{aligned}
Y_t &= D((n-k)T) + \\
&\quad (D((n-k+1)T) - D((n-k)T)) \times p + \\
&\quad (D((n-k+2)T) - D((n-k+1)T)) \times p^2 + \\
&\quad \vdots \\
&\quad (D(t) - D((n-1)T)) \times p^{k-1}
\end{aligned}$$

4.2.4 Overall Registry Consistency

We have seen how to calculate the probability of false positives, false negatives and inconsistent state for one announcer's registry entry. Since each of these corresponds to an error in the registry state, the total probability of error per entry is given by:

$$Pr[Err] \quad = \quad Pr[False\ -] + Pr[False\ +] + Pr[Inconsistent\ State]$$

The expected cost per entry is given by:

$$E[cost] \quad = \quad Pr[False\ -] * cost_n + Pr[False\ +] * cost_p + Pr[Inconsistent\ State] * cost_s$$

At time t, $Pr[Err]$ is the probability of error for one registry entry given one announcer, and $Pr[C] = 1 - Pr[Err]$ is the likelihood of its consistency. We can use $Pr[C]$ to derive $Pr[C_N]$, the probability of consistency for one listener registry with N announcers.

$$Pr[C_N] \quad = \quad (1 - Pr[Err])^{N-1}$$

Note that the state for the local announcer is never inconsistent, thus the usage of $N - 1$. We also can differentiate between different types of inconsistency based on the number of announcements that have been missed, as shown in Appendix A.

4.2.5 Arrivals

If we look at each registry entry and start time at $t = 0$, the tables are accurate (Tables 4.2 and 4.3, and Tables A.2 and A.1). However, when we look at global registry consistency, each entry may have a different start time. Thus, let us examine the impact of the arrival distribution on $Pr[Err(t)]$, the probability of error at time t.

Let $A(u)$ be the distribution for process arrivals into the Announce-Listen algorithm. If a process arrives into the system at time u, then for that process the new probability of error at time t is shifted in time becoming $Pr[E(t - u)]$.

Consider an example in the discrete realm. Process i $(0 \leq i \leq N)$ arrives at time i (relative to time 0 when the algorithm begins) and each stays through time A, the last arrival time.

$$\begin{aligned} Average\ Pr[Err(t)] \quad &= \quad Pr[Err(t)] \cdot u_0 + Pr[Err(t-1)] \cdot u_1 + \\ &\qquad Pr[Err(t-2)] \cdot u_2 + \ldots + Pr[Err(t-N-1)] \cdot u_N \end{aligned}$$

In the continuous domain, the average probability of error at time t becomes the integral

$$Average\ Pr[Err(t)] \quad = \quad \int_0^A A(u) \cdot Pr[Err(t-u)]du$$

4.3 Inconsistent State: Analysis and Simulation

In this section, we focus on the likelihood of inconsistent state. We discuss our intuition about the behavior of the system, then ground it in analysis for $r \geq T$, i.e., the periodicity of data changes is greater than or equal to the announcement periodicity. We derive expressions for the expected probability of single entry consistency, as well as overall registry consistency. We validate these results using simulation.

4.3.1 Analysis

Assuming T and Δ are identical across announcers, $Pr[\textit{one entry is inconsistent}]$ can be determined by summing the entries in the table in Appendix A that correspond to inconsistent state.

First, consider our intuition about the case where $D(t) = 0$, the numbers of late arrivals are 0, and $r = T$, meaning each announcement contains new data. Either an announcement is lost en route to the listener with probability p or it is received with probability $(1 - p)$. In the event that a message is received, the registry is inconsistent for that entry for a period of Δ/T in each announcement interval. Accounting for the fact that the message may not be received, we posit that

$$Pr[\textit{one entry is inconsistent}] \quad = \quad p + (1-p) \times \frac{\Delta}{T}$$

Now let us actually model the impact of $r \geq T$ and compare the results. There are *fewer* inconsistent state errors due to the fact that error only increases when we cross an "r boundary." In other words, a_n could be the first announcement in a cluster of r/T identical announcements. If we assume the cache is infinite, then the number of announcements that have arrived at the listener by time t will be $n = \lfloor t/T \rfloor$. As performed in Section 4.2.2, to arrive at $Pr[Err \leq e \mid \textit{within}\ \Delta]$ or $Pr[Err \leq e \mid \textit{after}\ \Delta]$, we just add the columns of the tables. Only now, the columns are infinitely long and lead to the results:

$$\begin{aligned} Pr[Err \leq e \mid \textit{within}\ \Delta] &= 1 - p^{\lfloor t/T \rfloor - 1} \\ Pr[Err \leq e \mid \textit{after}\ \Delta] &= p(1 - p^{\lfloor t/T \rfloor - 1}) \end{aligned}$$

If a_n is the second announcement in a series of r/T similar announcements, we can drop the first line in both columns and then sum the remaining columns. If a_n is the k^{th} announcement in a series of r/T similar announcements, $1 \leq k \leq r/T$, we drop the first $(k-1)$ lines. The sums become

$$\begin{aligned} Pr[Err \leq e \mid \textit{within}\ \Delta] &= p^{k-1} - p^{\lfloor t/T \rfloor - 1} \\ Pr[Err \leq e \mid \textit{after}\ \Delta] &= p\ (p^{k-1} - p^{\lfloor t/T \rfloor - 1}) \end{aligned}$$

Therefore,

$$\begin{aligned}
Pr[\textit{one entry is inconsistent, given } k] &= \left(\frac{\Delta}{T}\right)\left(p^{k-1} - p^{\lfloor t/T \rfloor - 1}\right) + \\
&\quad \left(1 - \frac{\Delta}{T}\right) p \, \left(p^{k-1} - p^{\lfloor t/T \rfloor - 1}\right) \\
&= \left(p^{k-1} - p^{\lfloor t/T \rfloor - 1}\right)\left(\frac{(1-p)\Delta}{T} + p\right)
\end{aligned}$$

We take the mean for k over $1 \ldots r/T$,

$$\begin{aligned}
Pr[\textit{one entry is inconsistent}] &= \frac{1}{r/T} \times \sum_{k=1}^{r/T} \left(p^{k-1} - p^{\lfloor t/T \rfloor - 1}\right)\left(\frac{(1-p)\Delta}{T} + p\right) \\
&= \frac{T}{r}\left(\frac{(1-p)\Delta}{T} + p\right)\left(\frac{1-p^{r/T}}{1-p} - \frac{r}{T}p^{\lfloor t/T \rfloor - 1}\right)
\end{aligned}$$

When we assume $t \gg T$, the expression becomes

$$\begin{aligned}
Pr[\textit{one entry is inconsistent}] &\simeq \frac{T}{r}\left(\frac{(1-p)\Delta}{T} + p\right)\left(\frac{1-p^{r/T}}{1-p}\right) \\
&= \frac{\Delta}{r}\left(1 - p^{r/T}\right) + \frac{T}{r}\frac{p}{1-p}\left(1 - p^{r/T}\right)
\end{aligned}$$

We can see from these results that when the data change rate, r, is large, the same message gets announced multiple times and the loss probability goes down exponentially. We also can check this formula against our intuition for $r = T$ and find that the results match.

$$\begin{aligned}
Pr[\textit{one entry is inconsistent}]_{r=T} &= \frac{\Delta}{r}(1-p) + \frac{T}{r}p \\
&= \frac{\Delta}{T}(1-p) + p
\end{aligned}$$

From $Pr[\textit{one entry is inconsistent}]$ we calculate the expected number of incorrect entries in the registry,

$$\begin{aligned}
E[\#\ \textit{incorrect entries}]_{r=T} &= \sum_{i=1}^{N-1} Pr[\textit{one entry is inconsistent}]_{r=T} \\
&= \left(p + (1-p) \times \frac{\Delta}{r}\right) \times (N-1)
\end{aligned}$$

as well as the expected fraction of inconsistent entries in the registry, $E[\%\ \textit{inconsistency}]_{r=T}$, and consistent entries in the registry, $E[\%\ \textit{consistency}]_{r=T}$.

$$E[\%\ \textit{inconsistency}]_{r=T} = (E[\#\ \textit{incorrect entries}]_{r=T})/N$$

$$= \left(p + (1-p) \times \frac{\Delta}{r}\right) \times \frac{(N-1)}{N}$$

$$\begin{aligned} E[\%\ \mathit{consistency}]_{r=T} &= 1 - E[\%\ \mathit{inconsistency}]_{r=T} \\ &= 1 - \left(p + (1-p) \times \frac{\Delta}{r}\right) \times \frac{(N-1)}{N} \end{aligned}$$

In the next section, we validate these results with simulation. We concentrate on the results for $r = T$, as they serve as a worst case scenario.

4.3.2 Simulation Results

Each simulation was run 50-100 times using the `ns` simulator [6]. As in the Suppression simulations, we used a star topology with participating processes at the edges of the star, resulting in fixed delays between all processes. We observed the behavior of AL with between $2 - 10$ nodes, in increments of one node, and between $20 - 100$ nodes, in increments of 10 nodes. We also simulated loss probabilities between 0 and 1 in increments of .1.

We experimented with a fixed announce timer $T = T_A$, as well as with T adjusted by mean-zero white-noise; with fixed and random sample intervals for when to observe the simulation; and with snapshot and event-driven simulations (assess the state of the system at random points in time versus only when significant events occurred). The results for these configurations were qualitatively similar.

Therefore, for our experiments, we chose to use an announce timer T that was adjusted in each interval by mean-zero white-noise; the announce timer value was selected from the uniform interval $[.5T, 1.5T]$, rather than being kept rigidly "fixed"[28]. We did this to avoid bursts of congestion when all processes announced simultaneously. We calculated the state of the system by using event-driven simulations, where error in a listener registry was evaluated with the receipt of each message.

The exact content of each announcement message is shown below in Table 4.4. The `key` identifies the process making the announcement and the `value` is state shared by the process. The `seqno` provides a sequence number (message count) for the announcements from the process, allowing the listener to detect losses in the data stream, whereas the `version` tracks the number of times the content has changed since the process began announcing. We send both `seqno` and `version` to more accurately determine wrong state; it may happen that a message is lost but the data in an announcement does not change between the last and next message received. An announcement message also contains the time it was sent, from which the listener can determine round-trip time delay (see Section 4.9), as well as the expected time of the next announcement, from which the listener can adapt its cache entry expiration strategy to variable timer periods.

However, in the simulations discussed below, every announcement constituted a data change

Field	Description
`key`	process identifier
`value`	process state
`seqno`	sequence number
`version`	version of data
`send_time`	time message sent
`next_time`	anticipated next transmission

Table 4.4: **Announcement Message Content.**

($r = T$) and data was never expired from the cache ($k = \infty$). The actual state stored by each listener about each announcer was extremely simple; a key-value pair consisting of a unique source identifier and the data version number < *source id, data version#* >. We were not interested so much in the state information sent, as we were interested in the version of the state.

The registry was examined during "steady state," i.e., those times when N (the membership) was constant. In particular, the system was examined at a point after all processes arrived (within a very short time of $t = 0$) and before they departed (within a short time of an agreed-upon completion time). Thus, any inconsistencies in the system were due to transmission delays or packet loss rather than $A(t)$ or $D(t)$. We studied this behavior because it emerges in several existing applications. For instance rendezvous style meetings that are slated to begin (or end) at a set time and systems where nodes simultaneously reboot at initialization (or leave concurrently when there is a failure). They have the property that arrivals (and departures) cluster around specified times. In these systems, group membership remains fairly static in between session initiation and completion. We show some representative results below.

In Figure 4.7, we show a comparison of the simulation vs. the analysis, with $T = 1$, $\Delta = .1$, the data change rate $r = 1$ and $p = 0$; the simulation validates the analysis. This experiment was repeated by varying $2 \leq N \leq 100$, $.001 \leq \Delta/T \leq .5$, $0 \leq r \leq 10$, and the results matched under these circumstances as well.

Figure 4.8 displays the impact of increasing Δ/T (the network delay relative to the announcement period) on the level of simulated consistency, $E[c]$. As Δ/T increases, consistency decreases. This can also be seen in Figure 4.9, which depicts the change of Δ/T over several orders of magnitude. Holding Δ/T constant, Figures 4.10, 4.11 and 4.12 show that as r increases (i.e., the rate at which data change slows relative to the announcement periodicity T), consistency increases. When data changes with every announcement, $r = T$, and there is no loss in the system, $p = 0$, the probability of an inconsistent entry reduces to Δ/T. This analysis is validated in the simulations in Figures 4.10, 4.11 and 4.12, which are scaled versions of each other. The graphs show the effects of decreasing Δ/T on the expected level of consistency.

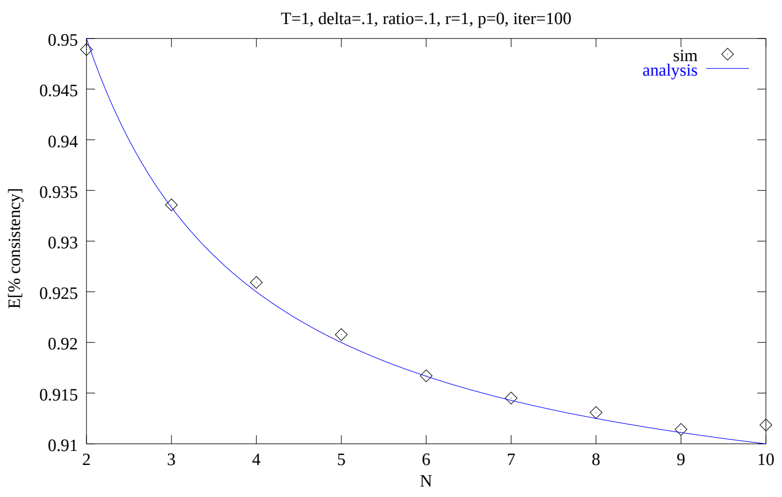

Figure 4.7: **Simulation vs. Analysis: Inconsistent State.**

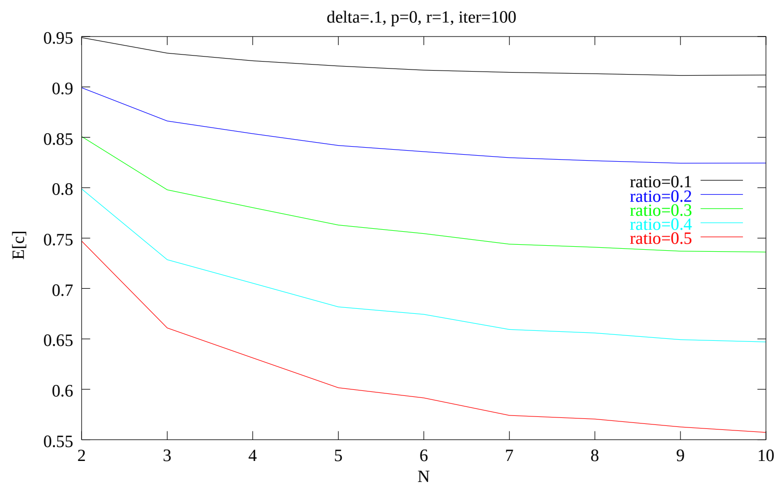

Figure 4.8: **Simulation: Inconsistent State (Large $ratio = \Delta/T$).**

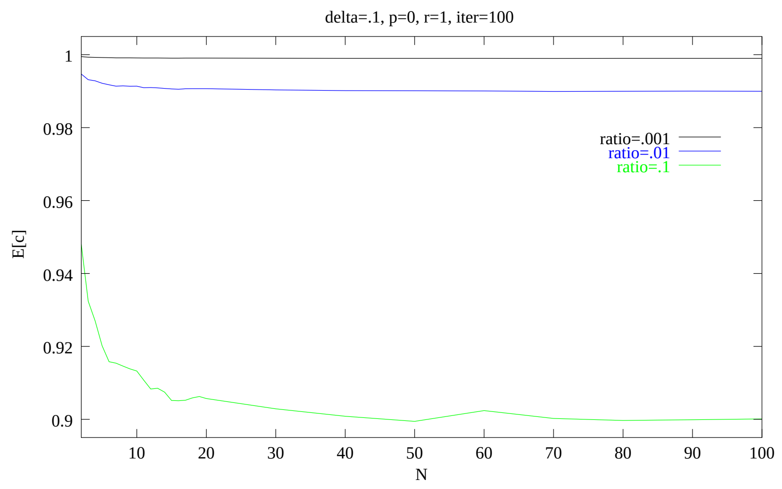

Figure 4.9: **Simulation: Inconsistent State (Small $ratio = \Delta/T$).**

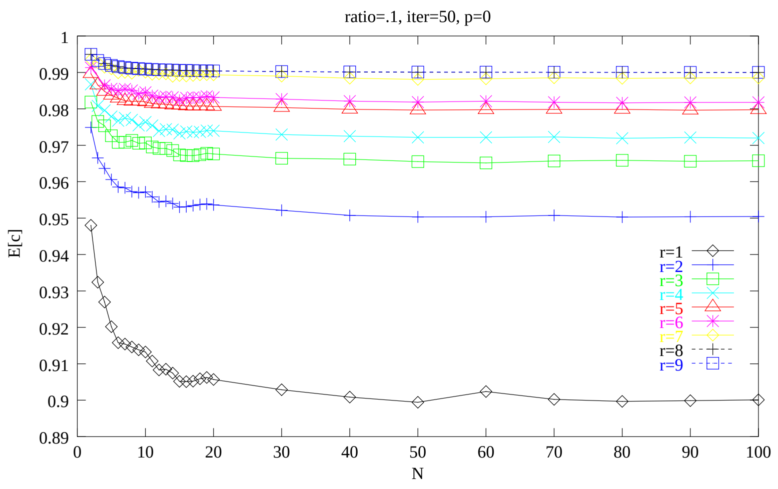

Figure 4.10: **Simulation: Data Change Rate ($\Delta/T = .1$).**

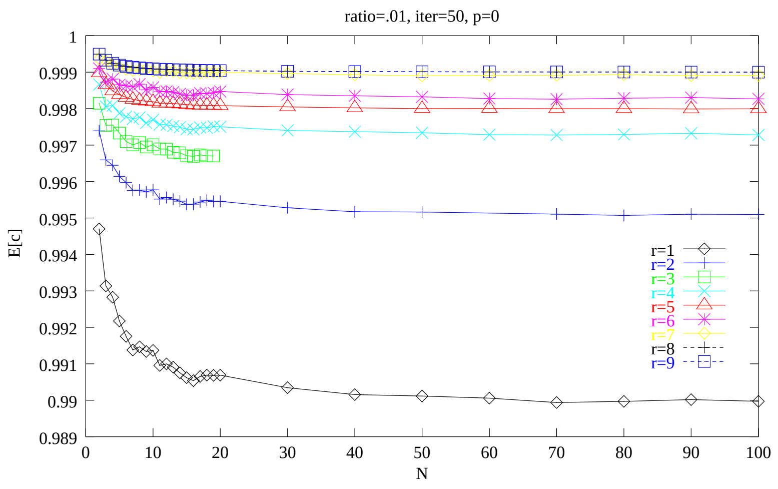

Figure 4.11: **Simulation: Data Change Rate ($\Delta/T = .01$).**

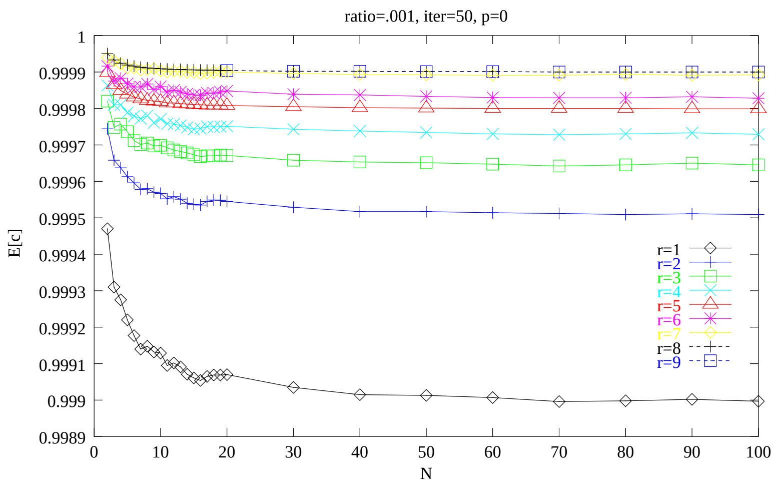

Figure 4.12: **Simulation: Data Change Rate ($\Delta/T = .001$).**

4.4 Convergence Time

Convergence time is defined as the minimum time before which the Announce-Listen algorithm reaches a particular level of consistency. It is a function of group size, announcement periodicity, transmission delay and packet loss.

Consider the lossless case. When there is no loss and transmission delay is a fixed amount, Δ, announcements are trivially calculated to take Δ units of time to reach all listeners. Each announcement reaches destination registries simultaneously. When transmission delay varies between pairs of processes, the time it takes for an announcement to propagate to all registries is bounded by the maximum Δ_{ij}, where Δ_{ij} is the pairwise delay between the announcer process i and listener process j $(\forall\ i, j :: 0 \leq i, j < N)$.

When loss is introduced, the calculation for convergence time depends on knowing the number of attempts (retransmissions) it takes to send an announcement successfully. In this regard there is some resemblance to the Suppression metric for $E[\#\ messages]$. For Suppression, we ask how many messages does it take to suppress a group of N processes. In the case of Announce-Listen, we are interested in knowing how many attempts, $E[\#\ attempts]$, are needed for a given announcement to reach all N listeners.

Below, we provide a rough estimate of propagation time to provide some intuition about the metric. On average, each announcement reaches only a subset of the listeners of size $(1-p)N$. So we can determine the number of iterations i or rounds of repeat announcements it takes to cover the whole set. On average, the 1^{st} message reaches $N(1-p)$ processes, the 2^{nd} reaches $N(1-p)p$ additional processes, the 3^{rd} reaches $N(1-p)p^2$ and so forth. Therefore, after i attempts, the number of listeners that the message has reached is $N(1-p^i) = N(1-p) + N(1-p)p + \ldots + N(1-p)p^{i-1}$, whereas the number of listeners the message has not reached is Np^i.

Let us define ε as the acceptable fraction of the membership N that goes unreached; ε therefore defines a target level for inconsistency. The goal for convergence is for the fraction of unreached membership to be less than the acceptable level of inconsistency, or $Np^i < \varepsilon N$. When we solve for i, and note that $0 \leq p \leq 1$ and $0 \leq \varepsilon \leq 1$, the number of iterations it takes for Np^i to get sufficiently small is

$$E[\#\ attempts] \quad > \quad \frac{log(\varepsilon)}{log(p)}$$

For instance, if the goal is for all processes to hear a given announcement, i.e., that the number of unreached processes should be less than one $(Np^i < 1 = \varepsilon N)$, then $\varepsilon = 1/N$, and the number of attempts must be larger than

$$E[\#\ attempts] \quad > \quad \frac{log(\frac{1}{N})}{log(p)}$$

When it is acceptable for a percentage of processes not to receive the announcement, e.g., $\varepsilon = .1$,

$$E[\#\ attempts] \quad > \quad \frac{log(.1)}{log(p)}$$

Once $E[\#\ attempts]$ is parameterized properly, the time it takes for the registry to reach a consistent state, $E[convergence\ time]$, is bounded in the worst case by the time between the first and last announcement, plus the one-way transmission delay for the last message to arrive at the furthest listener (Figure 4.13):

$$\begin{aligned} E[convergence\ time] &= (E[\#\ attempts] - 1) \times T_A + \Delta_{max} \\ &= \left(\frac{log(\varepsilon)}{log(p)} - 1\right) \times T_A + \Delta_{max} \end{aligned}$$

This is an oversimplification because we have assumed that the content of the message does not change between announcements, which may not be the case. If there is a premium on having the registry attain a certain level of consistency by a given time (i.e., having announcements reach a certain percentage of listeners), but each announcement contains new content ($r = T_A$), then it may not be possible to reach a certain level of consistency given p and Δ. In that case, it will be necessary to adjust r or to use an alternate form of dissemination than AL (i.e., a more reliable transport protocol).

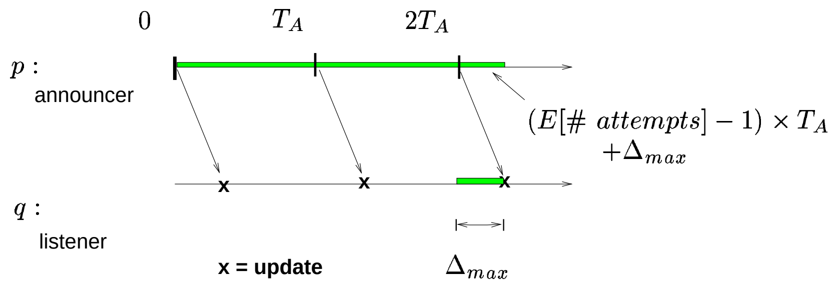

Figure 4.13: **Convergence Time.**

4.5 Messaging Overhead: Bandwidth

How does Announce-Listen consume network bandwidth? There are two aspects to answering this. First there is the question of how much bandwidth does AL require to achieve a certain level of consistency. Second, there is the question of how much of the bandwidth is wasted, i.e., is consumed by messages that are not necessary or that are redundant.

Let us consider the operational requirements given the periodicity of data changes, r. In Section 4.4, we derived $E[\# \ attempts]$, the number of messages required by AL to achieve a level of consistency of $1 - \varepsilon$. We use that knowledge to determine $E[\# \ wasted]$, the number of messages unnecessarily sent.

- **No Change.** As stated in Section 4.2.2, if an announcer process begins at time 0, then by time t the process has sent $n = \lfloor t/T_A \rfloor$ messages. When there is no change in the announcements, $E[\# \ wasted] = \lfloor t/T_A \rfloor - E[\# \ attempts]$.

- **Each Announcement is an Update.** If each announcement is an update of the state information ($r = T_A$), then all announcements are necessary and no bandwidth is considered wasted; $E[\# \ wasted] = 0$. The caveat is that if p is high, there may be a permanent level of inconsistency in the registry, because pN of the processes never receive the announcement.

- **Data Changes Slower than T_A.** When $r > T_A$ (and r is an integer multiple of T_A), there will be r announcements that repeat the same data before the content in the announcement changes. If $r > E[\# \ attempts]$, then there will exist $E[\# \ wasted] = E[\# \ attempts] - r$ rounds of redundant announcements.

So far, we have assumed no new arrivals into the system. Static announcements are always useful for newcomers in the case where $r > T$. The implications are (1) the $E[\# \ wasted]$ calculations above should be adjusted to reflect the number of new members during each announcement period, and (2) after $E[\# \ attempts]$ messages are sent, an announcer should adjust r to be a function of $A(t)$. One way to accomplish this is to actually adjust T_A instead. Of course, convergence time would now be a function of any new announcement periodicity.

Now let us discuss the impact of multicast on the level of redundancy of each announcement. An announcement is deemed entirely unnecessary and a waste of bandwidth if it does not update the registry entry for *any* listeners. Unnecessary messages are akin to extra messages in the Suppression algorithm. However, it is more complicated to determine which messages are actually necessary, due to the multicast nature of announcements. Due to message loss, some listeners will receive an announcement, while others do not. If the same announcement is re-issued, even though the announcement provides a useful update for those listeners who did not receive the announcement earlier, it will be redundant for others in the group who have received it already. Therefore, bandwidth can be wasted not only as a result of the lack of data change in announcements, but also due to the multicast nature of the distribution tree. Therefore, $E[\# \ attempts] - 1$ of the required messages for consistency are only partially necessary: in other words, each time a message is sent, on average, the message reaches only $(1 - p)N$ of the nodes and does not reach pN of the nodes. The first message is necessary, but the subsequent messages are potentially redundant for any nodes that have already received it. Although we do not present an exact result here, we raise the issue for comparison with the non-multicast case.

On the other hand, if a process that receives an announcement unsubscribes from the group address immediately afterwards, then there is no redundancy.

4.6 Memory Overhead: Expiration Strategy

The cost of the registry depends in part on the data-aging model, which directly affects memory usage. The data expiration strategy can age data slowly (when k is large) or quickly (when k is small). Increasing k by adding more `Not Sure` states to the algorithm leads to:

- **Fewer False Negatives.** As a consequence of a longer expiration period, a listener waits longer before it declares an announcer `Departed`. This helps those processes that are actually alive, but whose announcement(s) might be experiencing packet loss or delay due to congestion.
- **More False Positives.** If an announcer stops announcing (because it has departed), then the listener holds the information longer. If false positives are costly to store or a listener needs to hear about departures in a more timely fashion (needing to know before k rounds of timeouts), then an explicit `Departed` announcement can offset the delay [13] [54].

Another way to look at the k expiration states is to think of them simply as providing a longer expiration timer. The aggregate expiration time (the total time an entry is cached, kT_L) should be long enough that an entry is not falsely removed because of delay or loss in the network. Therefore, to maintain a high level of consistency, an entry should remain in the cache at least as long as $E[\textit{convergence time}]$. Thus a policy to maintain high consistency sets the cost function to be small for recently stale data (within this bound), but increases it beyond this point; a consequence is that entries are expired and removed, but not before knowing definitively that an agreed-upon value has been reached.

4.7 Related Work

The idea of examining Announce-Listen probabilistically first appeared in [53]. At that time, the phrase Announce-Listen was coined to describe the class of protocols that rely on sender processes to periodically announce data and receiver processes to passively listen for updates. In this thesis, we specifically extend that work to more formally consider the range of errors that can arise in distributed registries built from AL. In addition, we incorporate other key parameters into the analysis that contribute to the operation of the algorithm. The goal has been to expose the principal variables affecting the system, in order to predict how they will behave under different operational conditions.

Raman et al. [49] develop an analytic model for the AL primitive based on classed queueing networks [7]. The focus of their research was to understand when adding feedback to Announce-Listen is beneficial for consistency. Although the model may be effective in capturing certain system phenomenon, it is entirely separate from the simulation results presented. Thus, although their model may be accurate, there are no simulations to validate it. In addition, as pointed out in their paper [49], the model is not analytically tractable when extended to perform two-level scheduling, which they propose based on the simulations.

One goal of our work is to conclusively validate our model and analysis through simulation. Once we can trust the model, it has predictive value. Given various parameters, we know how the system will behave. Given a particular performance metric to optimize, we know how to tune the system parameters to reach that optimization and the kind of impact those settings will have on other performance metrics.

There are other differences with the approach in [49]. We examine steady state, where the number of processes in the system is stable though the data in the announcements may change. Raman et al. do assign lifetimes to data, which translate to processes coming and going in our system. They characterize the system in terms of job rates and average behaviors, e.g., the rate at which new information arrives into, or information leaves the system, as well as the average probability that data is lost by one or more processes. We focus instead on individual announcements and individualized parameters, e.g., N, Δ, T_A, T_L, p, r, k. As a result, we directly expose the parameters that affect the performance of the algorithm, and we can differentiate between individual announcer's parameters and hence between policies toward individual announcements. For example, the expiration time for each entry is presented as a given at the outset of their model. By exposing p, k and T_L parameters in our model, we can understand how long it is appropriate to cache a value before removal. This allows us to correlate memory and consistency requirements. In general, by creating a model with parameters exposed, then we can adapt them as need be.

To combat the problem that some announcements are repeated ($r > T$) and will consume bandwidth unnecessarily, they discuss creating two separate queues. One to announce new data and another to announce old data, i.e., already announced at least once in the past. In their system, data migrates from new to old after the very first time it is sent. Afterwards, old data is only retransmitted upon request. Their research explores the appropriate ratio of bandwidth to allot to the two types of data.

Our analysis technique suggests that the appropriate point for the new-to-old transition should occur after $E[\#\ attempts]$ announcements, or beyond $E[convergence\ time]$, the point in time that all N listeners have received the data. This would save the retransmission delay of the NACK to be received and queueing delay to switch the data from old to new queue.

Both studies are interested in the consistency metric as a driving force behind AL. However, the metrics for latency slightly differ: they define an average latency from the instant a new or updated key-value pair is introduced into the system to the first time it is received correctly; we define the convergence time as the average time beyond which an updated key-value pair has successfully reached all participating processes. This difference results because we are considering an N process system, whereas they consider the behavior of two processes.

Sharma et al.[55] focus on adapting the periodicity of announcement messages, in order to keep bandwidth usage below a fixed threshold. They propose techniques for receivers to estimate senders' update intervals, both of which effectively reduce bandwidth usage over the lifetime of announcers. As there is typically a correlation between announce timers and listen timers, the techniques to adapt announce timers are coupled with modifications to the expiration strategies of receivers. Although our work does not investigate the usage of adaptive announce timers, adaptive timers are complementary in spirit to the goal of proper parameterization of the AL algorithm and could be incorporated into the model proposed in this chapter.

4.8 Summary of Results

In this Chapter, we discussed the usage of the Announce-Listen algorithm as a means to disseminate state information among groups of processes and to create a replicated distributed registry. We highlighted that AL is part of the protocol spectrum that does not rely on any feedback mechanisms for communication among processes.

We identified several metrics to gauge the performance of such an algorithm: consistency, convergence time, network overhead and memory usage. We focused primarily on the metric of consistency and pinpointed three different types of errors that arise and work against it: false negatives, false positives and inconsistent state. We derived a model of the probability of these types of errors occurring and calculated the likelihood of any given registry entry being in error, $Pr[\textit{one entry is inconsistent}]$. From this, we were able to deduce the overall consistency in a given registry, $E[\%\ \textit{inconsistency}]$, and the overall consistency of a collection of distributed registries participating in the Announce-Listen algorithm.

We cast the other metrics in terms of how they meet consistency requirements. We presented an analysis of the convergence time, $E[\textit{convergence time}]$, as a function of the number of attempts it will take to reach N participating registries, $E[\#\ \textit{attempts}]$, noting that parameters for r and T_A may need adjustment to attain a given level of consistency. Based on the update periodicity, r, the network overhead metric, $E[\#\ \textit{wasted}]$, can be expressed in terms of $E[\#\ \textit{attempts}]$ as well. In our discussion of memory overhead, we described a method to parameterize the cache so that entries were not prematurely expired: ensure that $kT_L > E[\textit{convergence time}]$.

Our main contribution with regard to these metrics is establishing a working model that can be used to parameterize them. Although other researchers have studied some of these metrics in the past, they have done so only with simulation and without the accompanying analysis to support or predict their findings. In addition, our new analytic model for AL exposed several key parameters not accounted for in previous models: timers both for announcement periodicity and cache expiration, transmission delay, and group size. With a sound model in hand, algorithms based on AL can be fine tuned to operate comfortably within the range for which they were designed.

4.9 Future Work

Extensions to the Model: Sporadic Listening, Proxies and Hierarchies. The model thus far has assumed that announcements are periodic and that listening is continuous. What happens if the listening interval is bounded? An example of such a system is a sensor network where sensors turn listening on briefly, then off again in order to conserve power. By giving up persistent listening, a process may need to rely on other processes more capable of listening full-time. When the intermittent process awakens, it can request updated information from the proxy cache.

Such a process may also require that a proxy make announcements on its behalf. This leads to the idea of proxy processes that not only collect, but also re-distribute information on behalf of multiple processes. A proxy announcer becomes a secondary (versus primary) source of information. However, in a distributed registry that includes proxy services, multiple announcers may announce the same information, and an algorithm for conflict resolution will need to be devised.

We are interested in investigating the tradeoffs between the various conflict resolution approaches: to embed additional information in announcements such as timestamps, versioning of data, or the "distance" from the actual information source. Distance might measure how many levels of indirection exist between the node that owns the information and the one propagating the information, or supply a topology measurement of delay time or number of hops. Are there substantial differences in the impact on the consistency in the registry? Furthermore, we want to understand the effect of redundant announcement messages on consistency, i.e., the likelihood that process p is in $q.registry$, when it should or should not be.

Finally, to what extent can we build a hierarchical multicast registry, and parameterize it based on the metrics we have derived in the non-heirarchical case? Are the metrics additive?

Approximate Knowledge. In the future, we would like to explore alternate caching strategies. The idea is to store more state information than a single key-value pair. A simple approach is to extend the cache to store more than one data value per entry. Cache policies might include:

- Cache j values per announcer in general, where j is small or is a function of memory constraints.
- Store all observed data values within the aggregate expiration timeout period.

By storing multiple key values, the registry can track the percentage of time a given announcer spends in each state. When the registry is queried for information pertaining to a given process, these statistics might dictate the order in which it returns the possible options and that the application subsequently tries each of the entries. The registry could also track the degree to which state values change, oscillate or go unreported. These statistics would help to estimate the proper registry response when a query for information follows the recent expiration of data.

Although there is no way to distribute updates more quickly than T (thus we assume $r \geq T$), we can use the knowledge of past history to predict the "direction" in which the data is going (e.g., akin to estimating the trajectory of an object) based on where it has been in the past. For example, if the state information for a process cycles through a small set of values, past history may allow us to predict the next value, even in the face of expired data.

Estimation of Transmission Delay. We have made the assumption that transmission delay, Δ, is easy to determine and that there are out-of-band means to calculate it. However, this assumption may not be true, nor is a fixed Δ value necessarily reliable.

We propose to explore the use of announcements themselves to derive estimates for one-way transmission delays between processes. This, in turn, could be used to parameterize the Announce-Listen algorithm, as well as other algorithms when AL is coupled to them (e.g., as in the case of Leader Election, Chapter 5).

There are really two types of Δ values of interest to each process: pair-wise Δ, the pair-wise transmission delay between a given process and every other process, and group-wise Δ, an average transmission delay that is calculated from all the pair-wise values a given process has collected. In order to make these estimations, a process must be both an announcer and a listener. We can imagine at least three uses for Δ calculations:

- **Parameter Adaptation.** Because, realistically speaking, Δ is not rigidly fixed, we can use Δ estimates to adapt the parameters of the AL algorithm itself. For example, if the registry convergence time must be below a given threshold, and p and Δ_{max} are given, then adjusting T_A may assist with that. In the future, we would like to understand the utility of incorporating this type of feedback into the operation of the algorithm. Although there exist many other adaptive multicast algorithms, we have not seen this technique incorporated into AL for the express purpose of consistency or convergence times. Sharma's work [55] focuses on adaptive timers to optimize bandwidth metrics (to keep usage below a given threshold).

- **Process Selection.** The idea is for each process to derive pair-wise Δ values from announcements (we explain the process below), then to calculate a group-wise Δ estimate, and subsequently to share the group-wise Δ information within announcements with other processes.

On receiving other process' information, a listener can compare its group-wise Δ with other processes' calculations. A process will be able to assess its relative "distance" from the rest of the group, i.e., if it is an outlier or topologically close to most of the processes. Outliers are less likely to receive announcements in time, and are therefore less likely to be consistent and more likely to remain inconsistent longer. Under the Suppression algorithm (Chapters 2 and 3), they generate more extra messages, whereas under Leader Election (Chapter 5) they are more prone to falsely elect a leader and less likely to re-elect in a timely fashion.

A process can use the group-wise calculations to determine if it makes sense for it to become an early awakener in the Suppression algorithm or a leader in the Leader Election algorithm. In short, if a process's group-wise Δ lies above other processes' group-wise estimate, then it refrains from responding quickly to SUP or LE, i.e., to try to suppress other announcements, or to become leader. Likewise, if a process is positioned well within the group of processes, its Suppression timer is shortened for SUP and LE.

In [53], a similar technique was used to calculate group-wise ttl's in order to scope multicast sessions properly.

- **Subgroup Establishment.** We could also use the group-wise Δ calculation to encourage listeners with nearly identical values to become associated with the same "class," where all the processes within a "class" share network characteristics and belong to a separate subaddress. From our Suppression analysis, we know how fixed-Δ systems behave and could predict and parameterize these groups well.

  In addition, the diameter of the processes within a "class" is now well-known and could be used to establish at which layer in the multicast registry hierarchy a subgroup belongs. The expectation is that, now that the groups are homogeneous in some network parameters, metrics pertaining to consistency are additive/subtractive as we go up/down the registry hierarchy.

The process for calculating pair-wise Δ values is as follows. Listeners are already caching information from specific announcers. If announcers were to include the local time that the message was sent, as well as the sequence number of the announcement (i.e., the number announcement since time $t = 0$), then a listener could echo this information back to the announcer when it becomes time for the listener to announce its own information. In addition, the listener must include how much time passes since the receipt of the announcement and when the listener issues its next announcement. Because announcements are multicast, the calculations targeted for different listener processes will be included in the same messages.

In Figure 4.14, we show that even though the exchange of announcements between two processes are asynchronous, it is now possible to ascertain round-trip times, and subsequently approximate Δ from that. Node 0 sends an announcement at time s_0, which is received at listener process 1 at time

r_0; the announcement includes time s_0. When node 1 subsequently makes its own announcement, it echoes back the time s_0 as well as r_0 to node 0, timestamping its own message as being sent at time s_1. On receipt of the announcement from node 1, node 0 is able to approximate the one-way Δ between the two processes:

$$\Delta_{01} \quad \approx \quad ((r_1 - s_0) - (s_1 - r_0))/2$$

Likewise, when node 0 issues its next announcement, it piggybacks the necessary information for node 1 to obtain Δ_{10}. Variants of this technique are used in such protocols as RTCP and SRM [54][27].

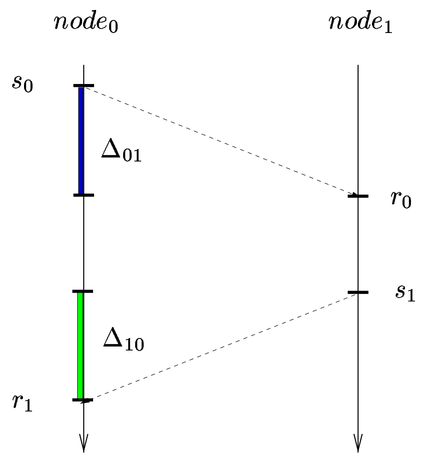

Figure 4.14: **Roundtrip Time Estimate: Calculating Pair-wise Δ.**

ACK- and NACK-based Schemes. We would like to identify the conditions under which multicast AL outperforms ACK/NACK messaging schemes, or vice versa. More specifically, we want to find the r for which ACKed messaging is a better choice than AL. By "better," we mean to evaluate the performance of different metrics, such as consistency, convergence time, and overhead (network and memory).