Robust Methods of Synchronization in Amorphous Networks

Josh Grochow

December, 2002

Abstract

"Amorphous computing is the development of organizational principles and programming languages for obtaining coherent behavior from the cooperation of myriads of unreliable parts that are interconnected in unknown, irregular, and time-varying ways [Abelson, et. al. Amorphous Computer. White Paper, 1999.]." One of these principles is temporal control, and a very basic form of temporal control is synchronization. I present a method of synchronization in an ad-hoc network (such as an amorphous computer) which is similar to the Network Time Protocol (used to synchronize the internet) in its method of error estimation. The resulting error between any two processors in the network is worst case $\sqrt{2d}\sigma$, where d is the diameter of the network and σ is the standard deviation of error between two adjacent processors.

1 Introduction

"Amorphous computing is the development of organizational principles and programming languages for obtaining coherent behavior from the cooperation of myriads of unreliable parts that are interconnected in unknown, irregular, and time-varying ways [2]." One of these principles is temporal control, and a very basic form of temporal control is synchronization. I present a method of synchronization in an ad-hoc network (such as an amorphous computer) which is similar to the Network Time Protocol (used to synchronize the internet) in its method of error estimation. The resulting error between any two processors in the network is worst case $\sqrt{2d}\sigma$, where d is the diameter of the network and σ is the standard deviation of error between two adjacent processors.

Elson and Römer (2002) identify five application-specific variables which help to clarify the problem of synchronization:

- **Energy utilization.** How much energy do the processors need? Do they need, for example, energy-intensive GPS units? Do they have large external power supplies? How frequently do they have to communicate and how much energy does it take?
- **Precision.** To what degree must synchronization be achieved? If the network serves some artistic purpose for humans, then perhaps millisecond accuracy is acceptable, whereas if the system is a wireless sensor network it may require much better accuracy.
- **Lifetime.** How long can the system stay in synchrony before needing to be resynchronized?
- **Scope and Availability.** Does synchronization have to occur across the whole network, or only locally?
- **Physical cost and size.** How big are the machines and how much do they cost to produce?

To which I would add

- **Type of "synchronization."** Many behaviors fall under the term "synchronization:" phase-locking (phase differences between nodes are constant but not necessarily close to zero), frequency-locking (constant frequency differences), and constant average frequency (all nodes run at the same average frequency over time, but their relations at any given instant are unimportant), to name a few.

A distinction also needs to be made between *a priori* and *post facto* synchronization. *A priori* synchronization implies that the network is always synchronized, so any other processes running on the network can always assume such synchronization. *Post facto* synchronization implies that any data recorded can be compared in a synchronized timeframe after the fact, even if the data were not taken synchronously. Elson and Römer (2002) argue for *post facto* synchronization in wireless sensor networks (and other systems), as it allows for greater precision and saves power by only running when necessary. I will be focusing, however, on the *a priori* case.

The paper is organized as follows: in the next section I will enumerate some of the obstacles to synchronization at the hardware and software level. In Section 3 I will briefly review NTP and the Reference Broadcast System (RBS) [6], and why they are inappropriate in the amorphous, *a priori* setting. Additionally I will go over some algorithms I tried which failed, revealing some higher-level obstacles to synchronization. In Section 4 I will present my algorithm and prove two theorems about its running time and its precision.

2 Hardware and Software Obstacles

Nondeterministic delays are the main obstacles to synchronization. Compensation for deterministic delays can be hard coded, or can be achieved through the use of timestamps. There are four main sources of nondeterministic delays in an ad-hoc network:

- **Send time.** The time to construct the message and transfer it from the processor to it's network device. If using fixed-length packets, and if the network device doesn't need an interrupt request, then this should be a fixed delay.
- **Access time.** The time to get access to the network channel. This is dependent on the method used to compensate for message collisions, e.g. TDMA or exponential back-off.
- **Propagation time.** The time between when the first bit of the message leaves the sender and the last bit is received. If fixed-length packets are used, this is a fixed delay in a network with only local communications (no message routing).
- **Receive time.** The time for the receiving host to process the message before it has access to it. If incoming messages are timestamped early enough, this can be exactly determined and compensated for, even if it is a probabilistic delay.

I am ignoring collisions in this paper (except for a possible collision avoidance algorithm in §5.1); the delay I am most concerned with is in the receive time, as it is rare for a system to be able to timestamp early enough to fully account for this.

3 Related Work

3.1 The Network Time Protocol

The Network Time Protocol (NTP) is, in essence, a hierarchical system which minimizes error by pinging multiple sources. Each tier of the hierarchy is called a **stratum**, and the stratum-1 servers are connected a global source of time such as GPS or WWVB (NIST time). Stratum-2 servers synchronize to the stratum-1 servers, and so on down to stratum-16. To synchronize, a server will ping servers in its parent stratum and use timestamps to compute the roundtrip travel time of the packet, estimating the error of synchronization. This value can then be added to a received time in order to reduce the actual error. Analysis in [3] reveals that the three main sources of error are (in increasing order of magnitude): limited precision clocks and timestamps, clock skew, and network jitter (the difference in travel time of a packet going and coming between two servers). Limited precision will be a problem in any digital system; Bletsas (2001) presents linear programming methods to estimate and account for network jitter and clock skew.

The fact that NTP relies on a global clock makes it useless in any setting without such information. Also, while NTP has many redundant servers at each stratum, selecting and synchronizing such "leaders" in an amorphous network is not a simple task, and would also require a message routing system rather than only local communication.

3.2 The Reference Broadcast System

The Reference Broadcast System (RBS) [6] is designed to achieve good *post facto* synchronization. The basic idea is to have a node A transmit a reference packet to nodes 1 and 2 simultaneously (requiring either a physical medium or parallel network devices for each communication line). When 1 and 2 need to compare two datum, they exchange the receive times of the packet from A and calculate the offset. This can be improved to account for clock skew by keeping a record of multiple packets from A and doing linear regression. Note that there is nothing preventing 1 from sending A and 2 a reference packet, or 2 from doing similarly, allowing any two nodes to reference data from one another. Also, since most networks are greater than one hop, relative time scales can be established between two distant nodes so long as there is a path of intermediary nodes between them (see Figure 1).

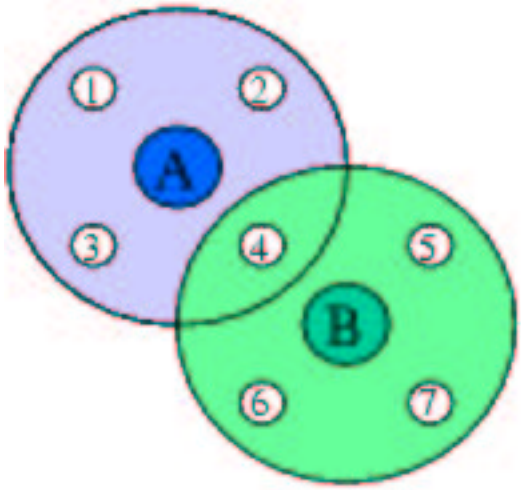

Figure 1: The intermediary node, 4, allows any two nodes to reference each other's data with appropriate time conversions. Image from [6].

While RBS is designed to achieve *post facto* synchronization, one might suspect that a similar scheme could be used to attempt synchronization *a priori*. This runs into a **symmetry breaking** problem: if we want 1 and 2 to actually think it is the same time, how do we choose which one should adopt the other's clock? If we choose arbitrarily, we may get a cycle: 1 adopts 2's clock, 2 then adopts A's clock, then A adopts 2's clock, and so on, with increasing error introduced each cycle. If instead we decide to average the two clocks, then later, when a new node gets added to the network, the entire network will go through a period of re-averaging during which it is asynchronous. This is an undesirable trait in a dynamic network.

3.3 Failed Algorithms

3.3.1 Basic Symmetry-Breaking

One very simple algorithm breaks symmetry by assigning each node a unique ID with high probability. This can be done either at manufacture time, or by selecting a sufficiently random number from a sufficiently large set of integers. Then each node can synchronize to the node with minimum ID by keeping track of, and broadcasting, the node's ID to which they are synching. (Note that node A synchronizing to node B only means that there is a path of nodes from A to B which are synchronizing to each other in order, not that A is routing messages directly to and from B.) Once the entire network is synchronized to the same node, however, all processors will listen to any of their neighbors, as they will all be broadcasting the minimum ID. Thus, there can still be cycles. Also, if a node with a new lowest ID is introduced, then the whole network must resynchronize, one of the problems associated with using RBS in this setting. One way to avoid this is to synchronize to the mode time value of the neighbors, and if there is no mode, *then* to the time value from the minimum ID of the neighbors. This way, if a group of neighbors is already synchronized, a node will stay with that group, even if a new node with lower ID is introduced. If there is ever a physical bottleneck in the network, however, we run into the symmetry breaking problem all over again (see Figure 2).

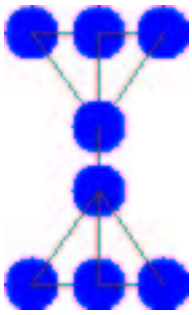

Figure 2: The two nodes in the middle cannot choose between the phase of their three neighbors (the mode) and that of the other node in the middle.

3.3.2 Integrate-and-Fire

Another set of algorithms is based on the model of the integrate-and-fire neuron discussed in [8]. This attempts to synchronize phase over the span of a few periods, rather than synchronizing to a "global" time (for example, say a

wireless sensor network takes a reading every 5 seconds and all we care is that the readings are taken synchronously, even if the timestamps are off, then two nodes whose clocks are off by exactly 5 seconds are in synchrony). The ideas behind integrate-and-fire are a bit messy for our purposes, but the overall effects are rather simple. All nodes share an intrinsic frequency at which they send messages. When a node hears a message, it advances its phase, more so if the message is heard near the end of its period than near the beginning. The end result of this weighting is that nodes slowly average their clocks (it is not an exact average: the final time just ends up somewhere between the original two). In situations without delays, this works perfectly, but that's not saying much. This is actually how I got started on this problem, but turns out to be much more complicated than necessary. A possible application to collision avoidance is mentioned in §5.1.

4 The Algorithm

4.1 Basics

So far we have run into five major obstacles to synchronization: probabilistic delays, clock skew, symmetry, cycles, and the dynamics of the network (new nodes being introduced, to which we can add nodes dying and also general changes in network topology). Once symmetry is no longer an issue, it might seem that cycles are no longer an issue. This is not the case, however: there can still by cycles which propagate an increasing error. The basic idea of this algorithm is to remove symmetry through the use of unique IDs, remove cycles and add robustness to network dynamics through the use of **active gradient trees** (see Appendix A), and remove probabilistic delays through successive estimates based on pinging.

The heart of the algorithm lies in a node's response to a received message. In a LISP-like pseudocode, it will be:

```
(define (receive msg)
  (timestamp msg my-time)
  (maintain-nbrs msg)           ;; Explained below
  (maintain-gradient msg)       ;; See Appendix A
  (estimate-delay msg)          ;; Explained below
  (if (= my-count 1)
    (detect-root-error msg))    ;; See Section 4.2
  (if (= my-count 0)
    (root-code msg)))           ;; See Section 4.2
```

The two most relevant data structures I will be using in my pseudocode, `message` and `statistic`, are included in Appendix B.

- **Probabilistic delays.** Similar to NTP, calculate the roundtrip time of a packet. Subtract off any known delays and any delays calculated via timestamps. Keep a running average of these probabilistic delays, which will eventually approach the mean.

  ```
  (define (estimate-delay msg)
    (let ((round-trip-time (- (end-time msg) (init-time msg)))
          (travel-time (- round-trip-time
                          constant-delay                ;; Known delays
                          (- (send-time msg) (rcv-time msg)))))
      (include-in-stat travel-time delay)))
  ```

  This is not affected by network jitter, one of the largest sources of error in NTP [3], because packets are only exchanged locally: there is no message routing, and thus no network to delay the propagation of a packet. Over long distances in a physical medium, there may be physical jitter, but I assume short enough distances that this is insignificant.

- **Clock skew.** This is accounted for by frequent synchronization, an unfortunately necessary step in the *a priori* case.
- **Symmetry between nodes.** Assign each node a unique ID, with high probability, as discussed in the basic symmetry-breaking algorithm (§3.3).
- **Cycles.** Once the network has unique IDs, we may select our leader (analogous to the stratum-1 servers of NTP) as that node with the lowest ID, again as in §3.3. The key difference between this and the basic symmetry breaking method is that "strata" are then defined by hops from that leader, through the use of an active gradient tree. This ensures that each node is only synchronizing to its (unique) "predecessor," and thus no cycles can occur. (See Appendix A for pseudocode: this is done automatically by the active gradient.)
- **Changes in network topology.** There are three possible types of changes: node death, node birth, and connection changes (e.g. brought about by node movement, if the network uses a physical broadcast layer). Once a portion of the network has synchronized, node birth is accounted for by recognizing the new node as new, and synchronizing it to the rest (by assigning it a higher ID than the current leader, for example), as shown here. (Note that I explicitly check here whether this node is the pinger or responder: see Appendix B for discussion.)

```
(define (maintain-nbrs msg)
  (let ((nbr-d (if (= (pinger-id? msg) my-id)  ;; Choose whichever
                   (responder-id msg)          ;; ID is not my-id
                   (pinger-id msg))))
  (if (not (recognize-nbr? nbr)) (begin
    (add-nbr nbr-id)
    ;; Ask the new neighbor if it is new, or if it should become
    ;; this node's new leader
    (if (and (not (= my-leader-id my-id)) ;; This node has a leader
             (< nbr-id my-leader-id)      ;; New node could be leader
      (transmit (new-message nbr-id 'new)))))))
```

  Note that whenever a new node is encountered, it will receive `'new` messages from all its neighbors who have selected other leaders. If several of its neighbors agree on another leader, it will set its ID to be higher than that leader's ID. This has the effect that if several nodes with very low IDs are all within one or two hops of one another, one of them will become the leader, though not necessarily the on which initially had the lowest ID. By the time this is done, though, the leader is still guaranteed to have the lowest ID.

  Node death is accounted for through the use of an active gradient tree, including death of the leader. The active gradient tree will continue to prevent cycles as long as connection changes happen slowly enough for the tree to rebuild as the changes happen. (Again, see Appendix A for pseudocode.)

4.2 Malfunctions and error detection

In removing cycles with an active gradient tree, however, we have introduced a leader. Were the leader to malfunction (for example, by resetting its clock), the entire network would have to resynchronize. To avoid this, we could have each node synchronize first to the mode time of its neighbors, and then to its predecessor's time. There is no bottleneck situation here because the symmetry within the bottleneck is broken by the predecessor relationship. But this does re-introduce cycles. Instead, we introduce a somewhat similar method of back-checking, but only on the root:

- **Error detection (root malfunction).** Say a node, "A," is adjacent to the root and has already synchronized with the root. Later, A receives a time from the root, corrects for delays, and this corrected time differs from A's time by a certain threshold (which we will return to in a minute). A then sends a message to the root containing A's time, and notifying it of this discrepancy. (Note that since the network is constantly synchronizing and thus accounting for clock skew, clock skew should not contribute significantly to this error.) The root then listens to all its neighbors' times, and if the probability that they are distributed around A's time is higher than the probability that they are distributed around the root's time, the root resynchronizes to the mean. If the root cannot determine which is more likely (perhaps no other nodes have synchronized to the root), then the root send a message to A which forces A to resynchronize to it.

```
(define (detect-root-error msg)
  (if (= (id msg) my-pid)           ;; Message is from the root
    (if (> (abs (- my-time (+ (send-time msg) (mean delay)))) threshold)
      (transmit (new-message my-pid 'error-detected)))))

(define (root-code msg)
  (if (eq? (comment msg) 'error-detected)
    (start-querying-nbrs))          ;; Starts a loop to ask neighbors
                                    ;; for their times, and stores
                                    ;; them in the nbr-time statistic
  (if (> nbrs-queried (* .5 num-nbrs))  ;; Heard from half the neighbors
    (if (> (abs (- my-time (mean nbr-time)))
           (abs (- error-time (mean nbr-time)))) \
      (begin
        (set! my-time (+ error-time (mean delay)))
        (transmit (new-message error-detecting-nbr 'error-corrected)))
      (transmit (new-message error-detecting-nbr 'no-error)))))
```

As for error the threshold, this can be either hard coded as a multiple of the standard deviation of delays (if known at manufacture time), or the nodes around the root can use the standard deviation of the ping times they have received so far.

4.3 Results

Before going onto the results, it is important to note that bidirectional communications are absolutely necessary here: since there is no message routing, pinging must take place directly between adjacent nodes. Another important assumption is that the changes in network topology happen slowly enough for the active gradient tree to settle. While analysis to determine the threshold frequency of changes in network topology may be difficult, numerical results are forthcoming.

4.3.1 Precision

Although the network jitter discussed in [3] is effectively white Gaussian noise, network jitter does not affect networks with strictly local communications. It is reasonable to assume that the receiver delay we are dealing with is normally distributed, based on hardware tests in [6]. We are now ready to state and prove one of the main theorem's of this paper:

Theorem 4.1 *In an amorphous network with bidirectional communications, normally distributed communication delays between adjacent nodes, and network changes slow enough to accommodate an active gradient tree, estimating synchronization error by successive pinging results in worst case error of $\sqrt{2d}\sigma$ between any two nodes, where σ is the standard deviation of error between two adjacent nodes, and d is the diameter of the network.*

Proof In pinging, we are calculating an average of round-trip delays. Since these round-trip delays are normally distributed (by assumption), this average is also normally distributed. Note that the average is of the *magnitude* of error, while the error itself (from the receiver's viewpoint) is distributed around a negative mean. We are thus adding two normally distributed variables with opposite means. So the means cancel to 0, and the variances add to $2\sigma^2$, yielding a standard deviation of $\sqrt{2}\sigma$. By induction, we see that a node n hops from the root of the gradient will have error distributed around 0 with standard deviation $\sqrt{n}\sigma$. Looking at the difference between two nodes, each possibly d hops from the root, we see that their variances also add, giving us a maximum error of $\sqrt{2d}\sigma$. ■

One might suspect that the longer this runs for, the better the synchronization, because the average delays calculated become closer to the true average. On a small scale, this may be true, but on the larger scale of the entire network, it is reasonable to look at the distributions, which are not affected by how long we synchronize for.

Note that this result is comparable to the degree of *post facto* synchronization achieved by RBS [6], which was $\sqrt{d}\sigma$. The factor of $\sqrt{2}$ is not a very large price to pay for *a priori* synchronization: the largest price is the energy cost of frequent communications.

Additionally, we intuitively expect that synchronization better than $\Theta(\sqrt{d})$ cannot be achieved if error between adjacent nodes is normally distributed: even if it is distributed around 0 to begin with, information must get from one end of the network to the other in order to have synchrony, and each hop adds to the variance.

4.3.2 Time to Synchronize

Theorem 4.2 *Under the assumptions of Theorem 4.1, the above described method will achieve $\sqrt{2d}\sigma$ synchronization in worst-case $O(d)$ time.*

Proof The precision of $\sqrt{2d}\sigma$ is the result of Theorem 4.1. The $O(d)$ running time follows directly from the time for the active gradient tree to settle, which is the same as a breadth-first search on a distributed system, $O(d)$. Once the tree has settled, all parent nodes have transmitted synchronization information to the child nodes. The child nodes still need to have at least one estimate of the delay before synchronization can be achieved. The nodes one hop from the root can do this as soon as they ping the root, which takes $O(1)$. The nodes two hops from the root will not be accurately synchronized until their predecessors have done so *and* they have pinged their predecessors. While the pinging itself takes $O(1)$, it must do so after the predecessors have pinged the root. This must happen d times before the entire network is properly synchronized. Adding this new $O(d)$ to the $O(d)$ required for the gradient tree does not affect the asymptotic running time, though we do expect it take roughly twice as long to synchronize as it does to set up the gradient tree. ■

The constant in $O(d)$ depends mostly on frequency of successful communications (which depends in turn on frequency of collisions and packet loss), but also depends on propagation time and processing time, which are usually relatively small.

4.3.3 Error Propagation

As mentioned above, using this method of error detection throughout the network would re-introduce cycles, so we don't. This means that an error not at the root could propagate down its subtree in a very localized fashion: each time the error propagates one hop, the correct time will be propagating right behind it. (Since we do not account for collisions, the correct time may actually be a few hops behind the incorrect time, depending on the frequency of transmission and the packet loss rate.) If the network is arranged geometrically (and is dense enough so that it approximates Euclidean geometry reasonably well), then the localized errors will travel on along a well-defined vector away from the root, making them easy to spot by an outside observer.

5 Conclusions and Future Work

Through the application of an adapted NTP algorithm and active gradients, we can achieve $\sqrt{2d}\sigma$ synchronization in an ad-hoc dynamic network such as an amorphous computer. One major cost of this algorithm, and indeed any *a priori* synchronization algorithm, is the frequency of communication necessary to account for clock skew. One shortcoming is the way in which malfunctions occurring at any node other than the root are handled: they create a localized error which moves up the active gradient tree, away from the root. Ideally, we would like to detect and correct errors on the spot, but "pushing" them to the edge of the network is still better than forcing the entire network to resynchronize. Also, better delay estimation algorithms, such as those proposed and mentioned in [3], or perhaps a robust adaptation of RBS, might provide better precision, though perhaps at the cost of speed of synchronization and processing of information.

Finally, it has been suggested that rather than using a tree, we allow the active gradient to simply build a directed acyclic graph, and have a node synchronize to the average of all of its predecessors. Based on some back-of-the-envelope calculations, this looks like it should work significantly better, but I have not had time to test this theory. One foreseeable problem with this is that a synchronization packet from one processor could interfere with the ping timestamps being sent to another processor.

5.1 Integrate-and-fire for Collision Avoidance

One possible application of the integrate-and-fire algorithm (§3.3.2) is collision avoidance. If the phase is retarded by messages rather than advanced, the network tends to *de*synchronize. If this can be modified to ensure that every subnetwork of hop count 2 is desynchronized, it would work extremely well for collision avoidance. [1], [4], [5], and [9] are about the emergence and stability of desynchronization in networks of integrate-and-fire neurons, and would be a good starting point towards this end.

A Active Gradient Trees

An **active gradient tree** spans some portion of the entire network (possibly all of it), and has the following properties:

- Once the tree has settled, there is only one root node.
- Once the tree has settled, every node in the the tree knows exactly how many hops it is from the root node.
- Once the tree has settled, every node in the tree (except the root) has exactly one parent, or "predecessor," node, which is exactly one hop closer to the root.
- When a portion of the tree is destroyed or disrupted, it rebuilds such that the above three properties again hold. An inactive gradient tree does not have this property - it is only created once.

While leader (root) selection by minimum ID is not absolutely necessary, some form of symmetry breaking is. The method of building and maintaining an active gradient tree described here uses the minimum ID symmetry breaking.

Every node stores the following data:

- Its own ID, `id`.
- Its current leader's ID, `lid`, which starts as `id`.
- Its predecessors ID, `pid`.
- Its current hop count from its current leader, `count`.

(The adjective "current" is used to emphasize that these values may change before the tree settles, and may change again if the tree is disrupted.) Every node transmits an `(id,lid,count+1)` triplet. Upon receiving a message, it runs the following pseudocode (explained below):

```
0 (define (maintain-gradient msg)
1   (let ((r-id (id msg))
2         (r-lid (leader-id msg))
3         (r-count (count msg)))
4     (if (< r-lid my-lid) (begin             ;; Change leaders?
5       (set! my-lid r-lid)
6       (set! my-count r-count)
7       (set! my-pid r-id)))

8     (if (and (= r-lid my-lid)
9              (< r-count my-count)) (begin    ;; Update hop count?
10      (set! my-count r-count)
11      (set! my-pid r-id)))))
```

When a node receives a message, it must decide whether to change its own `lid`: if the received `r-lid` is less than its own `my-lid`, then it does so (line 4). It must then decide whether to update `my-count` or not: if it accepted the new `r-lid`, then it takes the received `r-count` as its own (line 6). Otherwise, it only updates `my-count` if `r-count` $<$ `my-count`, that is, if it is actually closer to its leader than it thought it was (line 10). Whenever `my-count` is updated, it sets its predecessor to the transmitting node, `(set! my-pid r-id)` (lines 7,11). Since

it only does so when it actually updates its count, its predecessor ID will not oscillate between two possible predecessor nodes with the same hop count.

The tree described above already repairs itself if any node other than the root is disrupted, so long as the network continues to run this algorithm. If the root is disrupted, then some nodes will not hear from their predecessors, and assume them dead. These nodes then send out their own, new gradients, which vie for stability based on minimum ID again. Note that if the network is already synchronized, rebuilding a disrupted tree ensures that no cycles are present, but does not affect the

B Data Structures Used in Pseudocode

B.1 Message data structure

The `message` data structure stores the information that will be communicated from one node to another. For each datum stored in a message, there is an accessor by the same name which takes a single method as its argument. There is also a single constructor `(new-message to-id comment)`, where both arguments are optional: `to-id` is the ID of the desired receiving node (all other nodes will ignore this message), and `comment` is used to denote particular types of messages. The constructor adds the appropriate information (initial timestamps, node ID, leader ID, hop count, etc.).

Note that a single `message` stores all information from a round trip ping, in addition to any other necessary communications. All messages (except the very first) are the start of one ping and the end of another: when a node A pings a node B, and B returns the ping, this returned ping from B is the beginning of a ping from B to A. So we actually store two of each of the variables below, whose roles reverse each time the role of pinger reverses, but this is a subtlety I ignore in the pseudocode, which is written from the pinger's point of view (with one noted exception). (The pseudocode from the responder's point of view simply receives, timestamps, and returns the ping, and is not included.)

- `pinger-id`: The ID of the node which initiated the ping. Since communications are radially symmetric, we need the `pinger-id` to make sure that multiple pings to the same node do not interfere.
- `responder-id`: The ID of the node which is responding to the ping.
- `count`, `leader-id`: The necessary information to maintain the active gradient tree (Appendix A), in conjunction with the two previous IDs.
- `init-time`, `end-time`: The times at which the pinger sent out the ping and received the response, respectively.
- `rcv-time`, `send-time`: The times at which the responder received the ping and sent the response, respectively. Note that `send-time` is the time which the responder thinks it is, so if the responder is a node's parent, that node will synchronize to `send-time`.

B.2 Statistic data structure

The `statistic` data structure is used to calculate the mean and standard deviation of a set of numbers input over time, and has the following methods:

```
(include-in-stat val stat-name) ;; Includes val in the statistic referenced
                                ;; and recalculates the mean and variance;
                                ;; creates the statistic if it does not exist.
(mean stat-name)                ;; Retrieves the mean
(stddev stat-name)              ;; Retrieves the standard deviation
(num-vals stat-name)            ;; Returns the number of values included
(reset stat-name)               ;; Removes all values previously included
```

References

[1] L. Abbott and C. van Vreeswijk. Asynchronous states in networks of pulse-coupled oscillators. *Physical Review E,* Vol. 48, p. 1483, 1993.

[2] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, R. Weiss. Amorphous computing. White Paper, 1999. Available at http://www.swiss.ai.mit.edu/projects/amorphous/papers/aim1665.pdf.

[3] A. Bletsas. Time keeping in myriad networks: theories, applications, and solutions. S.M., MIT Media Lab. May 2001.

[4] P. Bressloff and S. Coombes. Dynamics of strongly coupled spiking neurons. *Neural Computation,* Vol. 12, p. 91, 2000.

[5] S. Coombes and G. Lord. Desynchronization of pulse-coupled integrate-and-fire neurons. *Physical Review E,* Vol. 55, No. 3, pp. 2104-2017, March 1997.

[6] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation* (OSDI 2002), Boston, MA. December 2002. Available at http://lecs.cs.ucla.edu/Publications/papers/broadcast-osdi.pdf.

[7] J. Elson and K. Römer. Wireless sensor networks: a new regime for time synchronization. *Proceedings of the First Workshop on Hot Topics in Networks,* pp. 28-29, October 2002. Available at http://www.cs.washington.edu/hotnets/papers/elson.ps.

[8] R. Mirollo and S. Strogatz. Synchronization of pulse-coupled biological oscillators. *SIAM Journal on Applied Mathematics,* Vol. 50, No. 6, pp. 1645-1662, December, 1990. Available at http://tam.cornell.edu/SStrogatz_bio_oscillators_sync.pdf.

[9] C. van Vreeswijk. Analysis of the asynchronous state in networks of strongly coupled oscillators. *Physical Review Letters,* Vol. 84, p. 5110, 2000.