Chapter 3

Broadcast and convergecast

Here we'll describe protocols for propagating information throughout a network from some central initiator and gathering information back to that same initiator. We do this both because the algorithms are actually useful and because they illustrate some of the issues that come up with keeping time complexity down in an asynchronous message-passing system.

3.1 Flooding

Flooding is about the simplest of all distributed algorithms. It's dumb and expensive, but easy to implement, and gives you both a broadcast mechanism and a way to build rooted spanning trees.

We'll give a fairly simple presentation of flooding roughly following Chapter 2 of [AW04].

3.1.1 Basic algorithm

The basic flooding algorithm is shown in Algorithm 3.1. The idea is that when a process receives a message M, it forwards it to all of its neighbors unless it has seen it before, which it tracks using a single bit seen-message.

Theorem 3.1.1. *Every process receives M after at most D time and at most $|E|$ messages, where D is the diameter of the network and E is the set of (directed) edges in the network.*

Proof. Message complexity: Each process only sends M to its neighbors once, so each edge carries at most one copy of M.

Time complexity: By induction on $d(\mathsf{root}, v)$, we'll show that each v receives M for the first time no later than time $d(\mathsf{root}, v) \leq D$. The base

```
initially do
    if pid = root then
        seen-message ← true
        send $M$ to all neighbors
    else
        seen-message ← false

upon receiving $M$ do
    if seen-message = false then
        seen-message ← true
        send $M$ to all neighbors
```

Algorithm 3.1: Basic flooding algorithm

case is when $v = \mathsf{root}$, $d(\mathsf{root}, v) = 0$; here root receives message at time 0. For the induction step, Let $d(\mathsf{root}, v) = k > 0$. Then v has a neighbor u such that $d(\mathsf{root}, u) = k - 1$. By the induction hypothesis, u receives M for the first time no later than time $k - 1$. From the code, u then sends M to all of its neighbors, including v; M arrives at v no later than time $(k - 1) + 1 = k$. □

Note that the time complexity proof also demonstrates correctness: every process receives M at least once.

As written, this is a one-shot algorithm: you can't broadcast a second message even if you wanted to. The obvious fix is for each process to remember which messages it has seen and only forward the new ones (which costs memory) and/or to add a **time-to-live** (TTL) field on each message that drops by one each time it is forwarded (which may cost extra messages and possibly prevents complete broadcast if the initial TTL is too small). The latter method is what was used for searching in `http://en.wikipedia.org/wiki/Gnutella`, an early peer-to-peer system. An interesting property of Gnutella was that since the application of flooding was to search for huge (multiple MiB) files using tiny (100 byte) query messages, the actual bit complexity of the flooding algorithm was not especially large relative to the bit complexity of sending any file that was found.

We can optimize the algorithm slightly by not sending M back to the node it came from; this will slightly reduce the message complexity in many cases but makes the proof a sentence or two longer. (It's all a question of what you want to optimize.)

3.1.2 Adding parent pointers

To build a spanning tree, modify Algorithm 3.1 by having each process remember who it first received M from. The revised code is given as Algorithm 3.2

```
initially do
    if pid = root then
        parent ← root
        send $M$ to all neighbors
    else
        parent ← $\perp$
upon receiving $M$ from $p$ do
    if parent = $\perp$ then
        parent ← $p$

        send $M$ to all neighbors
```

Algorithm 3.2: Flooding with parent pointers

We can easily prove that Algorithm 3.2 has the same termination properties as Algorithm 3.1 by observing that if we map parent to seen-message by the rule $\perp \to$ **false**, anything else $\to$ **true**, then we have the same algorithm. We would like one additional property, which is that when the algorithm **quiesces** (has no outstanding messages), the set of parent pointers form a rooted spanning tree. For this we use induction on time:

Lemma 3.1.2. *At any time during the execution of Algorithm 3.2, the following invariant holds:*

1. *If u.parent $\neq \perp$, then u.parent.parent $\neq \perp$ and following parent pointers gives a path from u to* root*.*
2. *If there is a message M in transit from u to v, then u.parent $\neq \perp$.*

Proof. We have to show that any event preserves the invariant.

Delivery event M used to be in u.outbuf, now it's in v.inbuf, but it's still in transit and u.parent is still not $\perp$.[1]

[1]This sort of extraneous special case is why I personally don't like the split between outbuf and inbuf used in [AW04], even though it makes defining the synchronous model easier.

Computation event Let v receive M from u. There are two cases: if v.parent is already non-null, the only state change is that M is no longer in transit, so we don't care about u.parent any more. If v.parent is null, then

1. v.parent is set to u. This triggers the first case of the invariant. From the induction hypothesis we have that u.parent $\neq \bot$ and that there exists a path from u to the root. Then v.parent.parent $=$ u.parent $\neq \bot$ and the path from $v \to u \to$ root gives the path from v.
2. Message M is sent to all of v's neighbors. Because M is now in transit from v, we need $v.parent \neq \bot$; but we just set it to u, so we are happy.

$\square$

At the end of the algorithm, the invariant shows that every process has a path to the root, i.e., that the graph represented by the parent pointers is connected. Since this graph has exactly $|V| - 1$ edges (if we don't count the self-loop at the root), it's a tree.

Though we get a spanning tree at the end, we may not get a very good spanning tree. For example, suppose our friend the adversary picks some Hamiltonian path through the network and delivers messages along this path very quickly while delaying all other messages for the full allowed 1 time unit. Then the resulting spanning tree will have depth $|V| - 1$, which might be much worse than D. If we want the shallowest possible spanning tree, we need to do something more sophisticated: see the discussion of **distributed breadth-first search** in Chapter 4. However, we may be happy with the tree we get from simple flooding: if the message delay on each link is consistent, then it's not hard to prove that we in fact get a shortest-path tree. As a special case, flooding always produces a BFS tree in the synchronous model.

Note also that while the algorithm works in a directed graph, the parent pointers may not be very useful if links aren't two-way.

3.1.3 Termination

See [AW04, Chapter 2] for further modifications that allow the processes to detect termination. In a sense, each process can terminate as soon as it is done sending M to all of its neighbors, but this still requires some mechanism for clearing out the inbuf; by adding acknowledgments as described in [AW04], we can terminate with the assurance that no further messages will be received.

3.2 Convergecast

A **convergecast** is the inverse of broadcast: instead of a message propagating down from a single root to all nodes, data is collected from outlying nodes to the root. Typically some function is applied to the incoming data at each node to summarize it, with the goal being that eventually the root obtains this function of all the data in the entire system. (Examples would be counting all the nodes or taking an average of input values at all the nodes.)

A basic convergecast algorithm is given in Algorithm 3.3; it propagates information up through a previously-computed spanning tree.

```
initially do
    if I am a leaf then
        send input to parent
upon receiving $M$ from $c$ do
    append $(c, M)$ to buffer
    if buffer contains messages from all my children then
        $v \leftarrow f$(buffer, input)
        if pid = root then
            return $v$
        else
            send $v$ to parent
```

Algorithm 3.3: Convergecast

The details of what is being computed depend on the choice of f:

- If input $= 1$ for all nodes and f is sum, then we count the number of nodes in the system.
- If input is arbitrary and f is sum, then we get a total of all the input values.
- Combining the above lets us compute averages, by dividing the total of all the inputs by the node count.

- If f just concatenates its arguments, the root ends up with a vector of all the input values.

Running time is bounded by the depth of the tree: we can prove by induction that any node at height h (height is length of the longest path from this node to some leaf) sends a message by time h at the latest. Message complexity is exactly $n - 1$, where n is the number of nodes; this is easily shown by observing that each node except the root sends exactly one message.

Proving that convergecast returns the correct value is similarly done by induction on depth: if each child of some node computes a correct value, then that node will compute f applied to these values and its own input. What the result of this computation is will, of course, depend on f; it generally makes the most sense when f represents some associative operation (as in the examples above).

3.3 Flooding and convergecast together

A natural way to build the spanning tree used by convergecast is to run flooding first. This also provides a mechanism for letting the leaves know that they are leaves and initiating the protocol. The combined algorithm is shown as Algorithm 3.4.

However, this may lead to very bad time complexity for the convergecast stage. Consider a wheel-shaped network consisting of one central node p_0 connected to nodes $p_1, p_2, \ldots, p_{n-1}$, where each p_i is also connected to p_{i+1}. By carefully arranging for the $p_i p_{i+1}$ links to run much faster than the $p_0 p_i$ links, the adversary can make flooding build a tree that consists of a single path $p_0 p_1 p_2 \ldots p_{n-1}$, even though the diameter of the network is only 2. While it only takes 2 time units to build this tree (because every node is only one hop away from the initiator), when we run convergecast we suddenly find that the previously-speedy links are now running only at the guaranteed ≤ 1 time unit per hop rate, meaning that convergecast takes $n - 1$ time.

This may be less of an issue in real networks, where the latency of links may be more uniform over time, meaning that a deep tree of fast links is still likely to be fast when we reach the convergecast step. But in the worst case we will need to be more clever about building the tree. We show how to do this in Chapter 4.

```
initially do
    children $\leftarrow \emptyset$
    nonChildren $\leftarrow \emptyset$
    if pid = root then
        parent $\leftarrow$ root
        send init to all neighbors
    else
        parent $\leftarrow \bot$

upon receiving init from $p$ do
    if parent $= \bot$ then
        parent $\leftarrow p$
        send init to all neighbors
    else
        send nack to $p$

upon receiving nack from $p$ do
    nonChildren $\leftarrow$ nonChildren $\cup \{p\}$

as soon as children $\cup$ nonChildren includes all my neighbors do
    $v \leftarrow f$(buffer, input)
    if pid = root then
        return $v$
    else
        send ack($v$) to parent

upon receiving ack($v$) from $k$ do
    add $(k, v)$ to buffer
    add $k$ to children
```

Algorithm 3.4: Flooding and convergecast combined