Basic Asynchronous Network Algorithms

Carlos Baquero
Distributed Systems Group
Universidade do Minho

MAPI 2007

Plan

Study a selection of algorithms tailored for synchronous or asynchronous networks:

- Leader Election in an Arbitrary Network (no longer a ring)
- Construction of a Spanning Tree.

First we start with the synchronous versions.
Then we introduce a particular asynchronous network model and adapt the algorithms for the new (harder) setting.
This should highlight the important diferences between the two models.

Leader Election in an Arbitrary Network
Synchronous Algorithm

Carlos Baquero
Distributed Systems Group
Universidade do Minho

Problem

- Eventually one process should change its *status* to *leader* and all other processes to *nonleader*.

Assumptions

- Processes have unique UIDs.
- Network Diameter d is known by all processes. *Diameter as the largest of the shortest paths across all nodes*

Leader Election in an Arbitrary Network
Synchronous *FloodMax* Algorithm

Carlos Baquero
Distributed Systems Group
Universidade do Minho

Initial state in node i

- uid_i // Globally unique
- $maxuid := uid_i$
- $status \in \{nil, leader, nonleader\} := nil$
- $rounds := 0$ // Integer

Informal Algorithm

In each round:

- Increment *rounds*
- $maxuid := max(maxuid, maxuid_{n1}, \ldots, maxuid_{nk})$
- send *maxuid* to all neighbours $\{n1, \ldots .nk\}$.

When $rounds = d$ do **if** $maxuid = uid$ **then** $status := leader$ **else** $status := nonleader$

Leader Election in an Arbitrary Network

Synchronous *FloodMax* Algorithm

Let i_{max} be the process with maximum *uid* and uid_{max} that *uid*.

Rounds are consistent

For every r and j, after r rounds, $rounds_j = r$.

u_{max} is an upper bound

For every r and j, after r rounds, $maxuid_j \leq u_{max}$.

u_{max} makes its way to each *maxuid*

For $0 \leq r \leq diam$ and j, after r rounds, if the distance from i_{max} to j is at most r, then $maxuid_j = u_{max}$.

Leading to Theorem

In *FloodMax* process i_{max} outputs *leader* and each other process *nonleader* within *diam* rounds.

Leader Election in an Arbitrary Network
Synchronous *FloodMax* Algorithm

Graph $G = (V, E)$ with *diam* diameter.

Complexity

- Time: Election (and termination) in *diam* rounds.
- Message: $diam \times |E|$. In the first *diam* rounds messages are sent across all edges E.

Upper Bound on Diameter

If only an upper bound d on diameter, $d \geq diam$ is known the algorithm also works. Account for added Time and Message complexity.

Leader Election in an Arbitrary Network
Synchronous *OptFloodMax* Algorithm

Reducing communication complexity

FloodMax modification into *OptFloodMax*

In round r only send *maxuid* to neighbours if *maxuid* changed. Bookkeeping is done in a new boolean state variable *newinfo*.

Proof is based on equivalence between relevant state components after a given number of rounds.

Simulation Relation proof

For any r, $0 \leq r \leq diam$, after r rounds, the values of the u, *maxuid*, *status* and *rounds* components are the same in the states of both algorithms.

Notice that round counting can be done even is messages are not received. Messages not received upon timeout are perceived as *null* messages.

Spanning Trees
Synchronous *SyncBFS* Algorithm

Definitions:

- A directed graph is called strongly connected if for every pair of vertices u and v there is a path from u to v and a path from v to u.
- Distance from u to v is the lenght of the shortest path from u to v.
- A directed spanning tree with root node i is breadth first provided that each node at distance d from i in the graph appears at depth d in the tree.
- Every strongly connected graph has a breadth-first directed spanning tree.

Spanning Trees
Synchronous *SyncBFS* Algorithm

Processes communicate over directed edges. Unique UIDs are available, but network diameter and size is unknown.

Initial state in *SyncBFS*

- $parent = nil$
- $marked = False$ ($True$ in root node i_0)

SyncBFS algorithm

- Process i_0 sends a *search* message in round 1.
- Unmarked processes receiving a *search* message from x do $marked = True$ and set $parent = x$, in the next round *search* messages are sent from these processes.

Spanning Trees
Synchronous *SyncBFS* Algorithm

Complexity

- Time: At most *diam* rounds (depending on i_0 eccentricity).
- Message: $|E|$. Messages are sent across all edges E.

Child Pointers

If parents need to know their offspring, processes must reply to *search* messages with either *parent* or *nonparent*. This is only easy if the graph is undirected, but is achievable in general strongly connected graphs.

Termination: Making i_0 know that the tree is constructed

All processes respond with *parent* or *nonparent*. Parent terminates when all children terminate. Responses are collected from leaves to tree root.

Applications of Breath First Spanning Trees.

Aggregation (Global Computation)

Input values in each process can be aggregated towards a sync node. Each value only contributes once, many functions can be used: Sums, Averages, Max, Voting.

Leader Election

Largest *UID* wins. All process become root of their own tree and aggregate a $Max(UID)$. Each decide by comparing their own *UID* with $Max(UID)$.

Spanning Trees
Synchronous *SyncBFS* Algorithm

Applications of Breath First Spanning Trees.

Broadcast

Message payload m can be attached to *SyncBFS* construction ($m|E|$ message load) or broadcasted once tree is formed ($m|V|$ message load).

Computing Diameter

Each process constructs a *SyncBFS*. Then determines *maxdist*, longest tree path. Afterwards, all processes use their trees to aggregate $max(maxdist)$ from all roots/nodes.
Complexity: Time $O(diam)$ and messages $O(diam \times |E|)$.

Asynchronous setup

Carlos Baquero
Distributed Systems Group
Universidade do Minho

We will now work on a "bare asynchronous network model, avoiding, for now, usefull abstractions like logical time and global snapshots.

System Model

The lack of helping tools is compensated by a "generous" system model:

Faults

No faults.

Channels

Reliable FIFO send/receive channels.

Reliable FIFO send/receive channels
Automaton

Carlos Baquero
Distributed Systems Group
Universidade do Minho

Signature:

Input: $send(m)_{i,j}, m \in M$

Output: $receive(m)_{i,j}, m \in M$

States:

$queue = \langle\rangle$

Transitions:

$send(m)_{i,j}$

Effect:

$queue := queue + \langle m \rangle$

$receive(m)_{i,j}$

Precondition:

$queue.head = m$

Effect:

$queue.pophead()$

Reliable FIFO send/receive channels
Allowed trace behaviour

Let β be a sequence of actions, and *cause*() a function mapping each receive event $e \in \beta\,|_{receive}$ to a preceeding send event $s \in \beta\,|_{send}$ such that:

1. $\forall receive(x) \in \beta\,|_{receive}$: $cause(receive(x)) = send(y) \Rightarrow x = y$. Messages dont come out of the blue.
2. *cause*() is surjective. For every *send* there is a mapped *receive*. Messages are not lost.
3. *cause*() is injective. For every *receive* there is a distinct *send*. Messages are not duplicated.
4. $receive <_\beta receive' \Rightarrow cause(receive) <_\beta cause(receive')$. Order is preserved.

Leader Election in an Arbitrary Network
Asynchronous *FloodMax* Algorithm

Assumptions

- Undirected graph.
- Unique UIDs.
- Known *diam*.

Round simulation for *FloodMax*

Each process sends a round r message with a r tag. Recipients only execute a round when all neighbour messages are received from the previous round.

Q&A

Q: Do we really need a r tag in messages?
A: Nope. It suffices to wait for all neighbour messages and the network operates in lock step. But this is as slow as the slowest link.

Leader Election in an Arbitrary Network

Asynchronous *OptFloodMax* Algorithm

In *OptFloodMax* messages are only sent if *maxuid* changes. Even if rounds are tagged how to decide if one can proceed to the next round?

If we keep waiting for all neighbours then *dummy* messages need to be always exchanged and the optimization is lost.

Clearly the way seems to be propagating a *maxuid* each times it changes, in a purelly asynchronous way that ignores rounds in lock step.

But now, how do we know when to stop?

Carlos Baquero
Distributed Systems Group
Universidade do Minho

Termination of an asynchronous *OptFloodMax* can be achived by a consistent global snapshot algorithm (covered later). All processes must be terminated and no messages in transit that can activate them.

Another option for Leader Election is to build a Spanning Tree for Broadcasting and Convergecasting.

Spanning Trees

AsynchSpanningTree automaton for node i

Signature:

Input: $receive(\text{"}search\text{"})_{i,j}$ $\quad j \in nbrs$

Output: $send(\text{"}search\text{"})_{i,j}$ $\quad j \in nbrs$

Output: $parent(j)_i$ $\quad j \in nbrs$

States:

$parent := null$ $\quad parent \in nbrs \cup \{null\}$

$reported := false$ $\quad reported \in \{false, true\}$

for all $j \in nbrs$:

$$sendto(j) := \begin{cases} yes & i = i_0 \\ no & \text{otherwise} \end{cases} \qquad sendto(j) \in \{yes, no\}$$

Transitions:

$parent(j)_i$

Precondition:

$parent = j$

$reported = false$

Effect:

$reported := true$

Spanning Trees

AsynchSpanningTree automaton for node i

Signature:

Input: $receive(\text{"}search\text{"})_{i,j}$ $\quad j \in nbrs$

Output: $send(\text{"}search\text{"})_{i,j}$ $\quad j \in nbrs$

Output: $parent(j)_i$ $\quad j \in nbrs$

Transitions:

$send(\text{"}search\text{"})_{i,j}$

Precondition: $sendto(j) = yes$

Effect: $sendto(j) := no$

$receive(\text{"}search\text{"})_{j,i}$

Effect:

if $i \neq i_0$ and $parent = null$ then

$parent := j$

for all $k \in nbrs \setminus \{j\}$ do

$sendto(k) := yes$

Channel automaton

Consumes $send(\text{"}search\text{"})_{f,t}$ and produces $receive(\text{"}search\text{"})_{f,t}$ in reliable FIFO order.

Spanning Trees

AsynchSpanningTree vs *SynchBFS*

While *AsynchSpanningTree* looks like an asynchronous translation of *SynchBFS*, the former does not necessarely produce a breadth first spanning tree.

Faster longer paths will win over a slower direct path when setting up *parent*.

One can however show that a spanning tree is constructed.

Carlos Baquero
Distributed Systems Group
Universidade do Minho

Invariant: A tree is gradually formed

In any reachable stat, the edges defined by all *parent* variables form a spanning tree of a subgraph of G, containing i_0; moreover, if there is a message in any channel $C_{i,j}$ the i is in this spanning tree.

Invariant: All contacts are searched

In any reachable state, if $i = i_0$ or $parent \neq null$, and if $j \in nbrs_i \setminus \{i_0\}$, then either $parent_k \neq null$ or $C_{i,j}$ contains a search message or $sendto(j)_i$ is *yes*.

Leading to:

Theorem

The *AsynchSpanningTree* algorithm constructs a spanning tree in the undirected graph G.

Spanning Trees

AsynchSpanningTree

In asynchronous systems altough time is unbounded its is pratical to assume a upper bound on time taken to execute a process effect, time l, and time taken to deliver a message in channel, time d.

Complexity

- Messages are $O(|E|)$.
- Time is $O(diam(l + d))$.

Altough a tree with height h, such that $h > diam$, can occur it only occurs if it does not take more time than a tree with $h = diam$.
Faster long paths must be faster!

Carlos Baquero
Distributed Systems Group
Universidade do Minho

Child pointers and Broadcast

If nodes report *parent* or *nonparente* one can build a tree that broadcasts.
Is the time complexity of this tree still $O(diam(l + d))$?
No, because a fast path is not always fast. Complexity is $O(h(l + d))$, at most $O(n(l + d))$, where $n = |V|$.

Broadcast with Acks

Its is possible to build a *AsynchBcastAck* algorithm that collects acknowledgmenets as the tree is constructed. Upon incoming broadcast messages each node Acks if they already know the broadcast and Acks to parent once when all neighbours Ack to them.

Leader Election with *AsynchBcastAck*

This algorithm includes termination and if all nodes initiate it and report their UIDs it can be used for Leader Election with unknown diameter and number of nodes.

Spanning Trees
AsynchBFS

It would be nice to have a breadth first spanning tree. In particular if each edge/link can be expected to have similar delays in the long run. Once constructed, broadcasts could expect time complexity of $O(diam(l + d))$ and no longer depend on the particular timming of the constructing run.
The key to this is to modify *AsynchSpanningTree* in order to keep updating *parent* designations.

Spanning Trees

AsynchBFS

Signature:

Input: $receive(m)_{i,j}$ $\quad m \in \mathbb{N}, j \in nbrs$

Output: $send(m)_{i,j}$ $\quad m \in \mathbb{N}, j \in nbrs$

States:

$parent := null$ $\quad parent \in nbrs \cup \{null\}$

$$dist := \begin{cases} 0 & i = i_0 \\ \infty & \text{otherwise} \end{cases} \qquad dist \in \mathbb{N} \cup \{\infty\}$$

for all $j \in nbrs$:

$$sendto(j) := \begin{cases} \langle 0 \rangle & i = i_0 \\ \langle\rangle & \text{otherwise} \end{cases} \qquad sendto(j) \in \mathbb{N}^*(FIFO)$$

Spanning Trees

AsynchBFS

Signature:

Input: $receive(m)_{i,j}$ $m \in \mathbb{N}, j \in nbrs$

Output: $send(m)_{i,j}$ $m \in \mathbb{N}, j \in nbrs$

Transitions:

$send(m)_{i,j}$

Precondition: m first in $sendto(j)$

Effect: remove first in $sendto(j)$

$receive(m)_{j,i}$

Effect:

if $m + 1 < dist$ then

$dist := m + 1$

$parent := j$

for all $k \in nbrs \setminus \{j\}$ do

add $dist$ to $sendto(k)$

Spanning Trees
AsynchBFS

Assertion: Each process has info on some path

In any reachable state:

- For every $i \neq i_0$, if $dist_i \neq \infty$ then $dist_i$ is the lenght of some path p from i_0 to i in G where predecessor of i is $parent_i$.
- For every message m in channel $C_{i,j}$, m is the lenght of some path p from i_0 to i.

Assertion: Information on shortest paths is eventually propagated

In any reachable state. For every pair of neighbours i and j, either $dist_j \leq dist_i + 1$, or else either $sento(j)_i$ or $C_{i,j}$ contains the value $dist_i$.

Ultimately this will lead to:

Theorem

In any fair execution of *AsyncBFS* the system eventually stabilizes to a state in which parent variables form a breadth first spanning tree.

But, there is a caveat to AsynchBFS ...

Termination

Notice that *AsynchBFS* did not produce the output report *parent*. It is not trivial to detect when the system stabilizes in the desired solution. It is however possible to devise a solution by accounting Acks for all messages, in the style of *AsynchBcastAck*.

If one needs termination its best to use a *LayeredBFS* algorithm (and pay a price in time complexity).

Spanning Trees
LayeredBFS

Synopsis:

- i_0 coordinates layer construction.
- Layer 1 will include level 1 nodes of the tree. Thus, 1 hop neighbours of i_0.
- Layer $k+1$ only starts construction after layer k is constructed and reported by convergecast back to i_0.
- When layer k is constructed, i_0 asks level k nodes to try a $k+1$ layer with their neighbours.
- Algorithm stops when no new nodes are found for a potential $k+1$ layer.

Bibliography

For details on the algorithms and access to the references of the original research that lead to them, consult the relevant Bibliographic Notes in *Distributed Algorithms* book by Nancy Lynch.