T-79.4001 Seminar on Theoretical Computer Science
Spring 2007 – Distributed Computation

Constructing a spanning tree

Toni Kylmälä
toni.kylmala@tkk.fi

Spanning tree

To construct a spanning tree we must get every node in the graph *G* to choose a subset of its neighbours to be in the acyclic graph T that spans the entire graph *G*. which we then call the “spanning tree”.

Constraints

We first look at a Unique Initiator case and limit ourselves to a fully connected network of bidirectional links where no failures occur. Denote **RI**.

Shout

Shout is a simple spanning tree construction protocol in which entities “ask” their neighbours whether they are neighbours in the spanning tree as well.

1. An entity will answer “yes” the first time it is asked.

2. Otherwise it will answer “no”. The initiator s will always reply “no”.

3. Each entity terminates independently when it has received a reply from all its neighbours.

Summarising this kind of protocol is a flood with an acnowledgement for every information message. This type of structure will be called *Flood + Reply*.

PROTOCOL Shout
Status: S = {INITIATOR, IDLE, ACTIVE, DONE}
S_{INIT} = {INITIATOR, IDLE}
S_{TERM} = {DONE}
Restrictions: **R**;UI;

INITIATOR
Spontaneuosly
begin
root := **true**
Tree-neighbours := $\emptyset$
send(Q)*toN*(x)
counter := 0
become*ACTIVE*
end

IDLE
Receiving (Q)
begin
root := **false**
parent := **sender**
$Tree-neighbours := \{$**sender**$\}$
send$(Yes)to\{$**sender**$\}$
counter := 1
if*counter* $= |N(x)|$**then**
become*DONE*
else
send(Q)*toN(x)* $- \{Sender\}$
become*ACTIVE*
endif
end

ACTIVE
Receiving (Q)
begin
send$(No)to\{$**sender**$\}$
end

Receiving (Yes)
begin
$Tree-neighbours := Tree-neighbours \bigcup \{$**sender**$\}$
counter := counter + 1
if counter=|N(x)| **then**
become DONE
endif
end

Receiving (No)
begin
counter := counter + 1
if counter=|N(x)| **then**
become DONE
endif
end

Proof of correctness

Because of the correctness of flooding every entity will receive Q. And by construction reply either “yes” or “no” to each Q. Since every entity only replies “yes” the first time it receives Q there can be no cycles. And because every $x \neq s$ is connected to s through a path of “parents” where on every link a “yes” was sent the tree spans the entire graph G.

Important

Shout ends in local termination. No entity is aware of global termination. This is fairly common in distributed computing. However Shout can easily be modified to allow global termination knowledge.

Costs

The message costs of *Flood + Reply*, and thus of Shout, are simple to analyze. *Flood + Reply* consists of executing *Flooding(Q)* with a reply for every *Q*:

$$\mathbf{M}[\textit{Flood + Reply}]=2\mathbf{M}[\textit{Flooding}].$$
$$\mathbf{T}[\textit{Flood + Reply}]=\mathbf{T}[\textit{Flooding}] + 1.$$

Thus:

$$\mathbf{M}[\textit{Shout}]=4m - 2n + 2$$
$$\mathbf{T}[\textit{Shout}]=r(s_*)+1 \leq d + 1$$

Theorem 2.5.2

$$M(\textbf{SPT}/\textbf{RI}) \geq \text{m}.$$

Proof

Similarly to the broadcast problem. If fewer than m messages are sent then there can be an entity that neither receives *Q* nor sends replies.

Theorem 2.5.3

$$T(\textbf{SPT}/\textbf{RI}) \geq \text{d}.$$

These imply that Shout is both time and message optimal.

Property 2.5.1

The message complexity of spanning tree construction under **RI** is *O(m)*.

Property 2.5.2

The ideal time complexity of spanning tree construction under **RI** is *O(d)*.

Hacking

Do we have to send “no” messages?

We would only send “no” to entities that send us *Q* after we have received *Q* for the first time. By this time we have already sent those same entities *Q*. Thus we send *Q* and “no” to exactly the same entities. “No” messages can therefore be left out. The resulting Shout+ has message complexity:

$$\mathbf{T}[Shout+]=r(s_*)+1 \leq \mathrm{d}+1$$
$$\mathbf{M}[Shout+]=2\mathrm{m}$$

Note that time complexity does not change.

Other SPT constructions with Single Initiator

SPT construction by traversal. A depth first traversal actually constructs a spanning tree (df-tree) of G. We obtain the tree by removing back-edges. Simple changes in local bookkeeping will allow entities to correctly determine tree neighbours. Thus costs are unchanged:

$$\mathbf{M}[df\text{-}SPT]=4m - 2n + f_* + 1.$$
$$\mathbf{T}[df\text{-}SPT]=2n - 2.$$

We can now better analyze the variable f_*. It is exactly the number of *leaves* in the df-tree.

SPT construction by broadcasting. With simple modifications flooding and df-traversal can be used to construct a spanning tree if a unique initiator is present. This is part of an interesting phenomenon: under **RI**.

> The execution of any broadcast protocol constructs a spanning tree.

Any broadcast execution will result in all entities receiving the information. Every entity will call the entity it first received it from a parent. Every $x \neq s$ has one parent while s has none.

Theorem 2.5.4

The parent relationship x $\succ$ y defines a spanning tree rooted in the initiator s.

Relation between broadcast, traversal, spanning tree and global protocols

As found previously in section 2.3.4 every traversal protocol performs broadcast. Thus under **RI** any traversal protocol constructs a spanning tree.

$$\textbf{Bcast} \equiv \textbf{SPT}(\text{UI}). \quad (2.20)$$

Actually we can make a much stronger statement. Call a problem *global* if every entity must participate in its solution. Clearly all three above are global. Now every single-initiator protocol that solves a *global problem* **P** solves also **Bcast**; thus from equation 2.20 under **RI**

> The execution of any solution to any global problem **P** constructs a spanning tree.

Consider the constructed tree

Using the spanning tree we can perform broadcast and traversal with optimal *O(n)* messages instead of *O(m)* messages which could be as bad $O(n^2)$

Important

Trees constructed with different methods are different. In fact because of unpredictable communication delays every possible spanning tree can be constructed with shout. This has implications for time costs. Broadcast time will be *d(T)* instead of *d(G)*. But *d(T)* can be much greater. In a fully connected graph n-1 as opposed to 1.

Thus we want to construct a tree that has minimal *d(T)*, a broadcast tree. For this we need to find the center c of the graph for the initiator node and use a breadth-first algorithm BFT(c, G). Center is the node x with minimum radius.

$$r_G(x) = \text{Max}\{d_G(x,y) : y \in v\}.$$

The radius of G is the minimum radius of all nodes x.

$$r(G) = \text{Min}\{r_G(x) : x \in V\}.$$

Relationship between radius and diameter of G in every graph G

$$r(G) \leq d(G) \leq 2r(G).$$

Unfortunately center finding and Breadth-First SPT construction are not simple problems to solve as will be seen in later chapters.

Theorem 2.5.5

BFT(c, G) is a broadcast tree of G.

Application: better traversal

Use Shout+ to construct a spanning tree T and perform traversal of T using DF_Traversal.

$$\mathbf{M}[\textit{SmartTraversal}]=2(m + n - 1)$$
$$\mathbf{T}[\textit{SmartTraversal}]=\leq 2n + d - 1$$

SmartTraversal is simple as well as time and message optimal.

Spanning tree with multiple initiators

Unfortunately a single initiator is a strong assumption as well as difficult and expensive to guarantee. In Shout if we have more than one initiator the result is a spanning tree forest since initiators have no parents and all other entities have only one parent. Removing the single initiator restriction brings out the true nature of the problem.

Theorem 2.5.6

The **SPT** problem is *deterministically* unsolvable under **R**.

Impossibility result

This means that there is no deterministic protocol that always correctly terminates within finite time.

Proof

Consider a simple three entity system where all three are connected. In the beginning all are in an identical state. If a solution A exists it must work here as well. Consider a *synchronous schedule* (i.e. an execution where communication delays are unitary) and let all three start the execution of A simultaneously. Since the states are identical they will all execute the same steps and end in possibly new identical states. However for there to be a spanning tree one of the tree edges must be removed. Thus the entities will have to end in *distinct* states with different values. But since the states are always identical no solution A exists.

SPT with Initial Distinct values

Despite this very negative result we can solve the problem with additional constraints. ID:s are an often used solution. These can be used to distinguish between initiators. A simple solution is to have many overlapping spanning trees. This however requires much local bookkeeping. The costs depend solely on the number k_* of initiators. In the case of Shout+ there will be $2mk_*$ messages which could be as bad as $O(n^3)$.

Selective construction

A better solution is to allow only one of these to complete and “kill” all the others. This can be done by letting the minimum (or maximum) valued initiators SPT be completed. Thus every time an entity receives Q with a lower initiator than the one it currently has it discards all previously collected information and joins the new tree. This reduces local bookkeeping and ensures that all entities have a single shared spanning tree.

Important

An entity might have terminated when it must start again when it receives Q with lower initiator. Thus we must have some way to ensure effective local termination. This is accomplished by having the initiator become aware that the tree is finished and all other trees have been killed and notifying all other entities using the newly constructed spanning tree. This protocol is called MultiShout.

Theorem 2.5.7

Protocol MultiShout constructs a spanning tree rooted in the initiator with the smallest initial value.

Costs of MultiShout

It's clearly better than the first one although the worst case can be as bad as $O(n^3)$. Consider an n-node graph with a fully connected subgraph with n - k nodes. The k nodes are initiators all with a single connection to the subgraph whose messages arrive in the subgraph in valued order so that when the subgraph has completed the construction a message Q arrives from the next k with a smaller ID. This is possible because we have unpredictable message delays. Thus the SPT will be constructed k times. with n - k nodes in each iteration. $O((n - k)^2)$ messages in the subgraph. Thus a total of $O(k(n - k)^2)$ messages will be used. If k is a linear fraction of n (e.g. k=$n/2$), then the cost will be $O(n^3)$.

This does not imply that the selective construction is not efficient. Merely that the solution here is not. This will be further examined in *leader election*.

PROTOCOL MultiShout
Status: S = {IDLE, ACTIVE, DONE}
    S_{INIT} = {IDLE}
    S_{TERM} = {DONE}
Restrictions: **R**;ID;

IDLE
    Spontaneously
    begin
        root := **true**
        root_id := v(x)
        Tree-neighbours := $\emptyset$
        send$(Q, root\_id)toN(x)$
        counter := 0
        check_counter := 0
        become*ACTIVE*
    end

    Receiving(Q, id)
    begin
        CONSTRUCT
    end

```
ACTIVE
Receiving(Q,id)
begin
    if root_id = id then
        counter := counter + 1
        if counter=|N(x)| then
            done := true
            CHECK
        endif
    else
        if root_id > id then
            CONSTRUCT
    endif
end

Receiving(Yes, id)
begin
    if root_id = id then Tree-neighbours := Tree-neighbours⋃{sender}
        counter := counter + 1
        if counter=|N(x)| then
            done := true
            CHECK
        endif
    endif
end

Receiving(Check, id)
begin
    if root_id = id then
        check_counter := check_counter + 1
        if (done ∧ check_counter=|Children|) then
            TERM
        endif
    endif
end
Receiving(Terminate)
begin
    send(Terminate) to Children
    become DONE
end
```

```
Procedure CONSTRUCT
begin
    root := false
    root_id := id
    Tree-neighbours := {sender}
    parent := sender
    send (Yes,root_id) to {sender}
    counter := 1
    check_counter = 0
    if counter = |N(x)| then
        done := true
        CHECK
    else
        send(Q,root_id) to N(x) - {sender}
    endif
    become ACTIVE
end

Procedure CHECK
begin
    Children := Tree-neighbours - {parent}
    if Children = ∅ then
        send (Check,root_id) to parent
    endif
end

Procedure TERM
begin
    if root then
        send(Terminate) to Tree-neighbours
        become DONE
    else
        send(Check,root_id) to parent
    endif
end
```