CS4616 Distributed Systems

Prof. M. Schellekens

General background on CS4616
Distributed Systems

Contact details

I prefer to be contacted by email:

m.schellekens@cs.ucc.ie

When you write:

Make sure you clearly state in the subject line the course number (CS4616) and a short indication of the purpose of your contact.

For instance: CS4616 assignment query, CS4616 question on distributed algorithms etc.

Make sure that your full name is included in the message.

You can also arrange for a meeting.

Office location

My office is located on the research floor of the Western Gate Building. CEOL, Office 2-55.

Module content

Techniques for the design and analysis of distributed algorithms for synchronous and asynchronous systems.

Formal issues regarding the course (see UCC book of Modules online for full information)

CS4616 is a 5 credit course.
Total marks: 100
End of year written examination: 80 marks.
Continuous assessment: 20 marks.
Passing requirement: 40 %.
Penalties for late submission of course work: Work which is submitted late shall be assigned a mark of zero.
End of year Written Examination Profile: One and a half hour exam.

Material used for this course

The course will be presented in part via lecture notes.

The course will rely on the following book:

Distributed Algorithms
Author: Nancy A. Lynch.
ISBN-13: 978-1-55860-348-6 and ISBN-10: 1-55860-348-4.

This book is required for the course.

What do I need to know for this course?

The exam will include exercises, similar in nature to the exercises we will see in class, as well as proofs (see below).

You need to be able to explain the fundamental principles of distributed systems.

You need to know and apply the algorithms we have seen in class and be able to design algorithms along the same principles as those given in class and be able to verify their correctness and performance.

You need to know all the main concepts and definitions we discuss in the lectures.

Some exercises will be more theoretical in nature, i.e. they will require the production of a formal argument to show that a certain property is true, or alternatively that a certain property is not true.

Distributed Algorithms relies heavily on mathematical proofs of correctness of the algorithms and of various complexity derivations. The proofs need to be known for this course and you need to be able to produce proofs for related problems.

Distributed Algorithms: introduction

Distributed algorithms are algorithms designed on hardware consisting of interconnected processors.

Pieces of a distributed algorithm run concurrently and independently, each with only a limited amount of information.

The algorithms should work correctly even if the individual processors and communication channels work at different speeds and even if some of the components fail.

Distributed algorithms arise in a wide range of applications: telecommunications, distributed information processing, scientific computing and real-time process control. Systems involving: telephone, airline reservation, banking, global information, weather prediction, airplane and nuclear plant control.

Distributed Algorithms: the simplified model

The settings in which the algorithms run are complicated, so the design can be extremely difficult. To reduce the complexity we will focus for the purpose of this course on the synchronous model. This is the simplest model to describe, to program and to reason about.

We assume that the components take steps simultaneously. So execution proceeds in synchronous rounds. The synchronous model is useful as an intermediate step to solve the problem in more realistic models and at times the techniques carry over to a more general case.

Distributed Algorithms: formal models

We will introduce a formal way to model distributed systems.

The reason for the formal approach is the complexity of the applications. To ensure that a program is correct one needs a formal model to enable verification.

The model we will introduce during this course is a model presented in Chapter 2 of the book on Distributed Algorithms.

Overview of course: chapter 2

We introduce a formal model for *synchronous algorithms*.

This model is based on *state machines*, often having an infinite number of states and usually having explicit names associated with their transitions.

A state machine can be used to model a component of a distributed system or an entire distributed system.

Each *state* of the machine represents an instantaneous snapshot of the component or system.

Overview of course: chapter 2 (continued)

Such a *snapshot* includes information such as the state of the memory of each processor, the program counter for each running program, and the messages that are in transit in the communication system.

The *transitions* describe changes that occur in the system, such as sending or receipt of a message, or the changes cause by a local computation.

The *use of a formal model* is a basis for: problem specification and algorithm correctness verification.

One useful method is that of *invariant assertions*, properties true of all reachable states of a system. Assertions are proved by induction on the number of steps in the system execution.

Overview of course: chapter 3

We present a simple example involving computation in ring networks. The problem is to elect a *unique leader* in a ring network, assuming that the processors at the nodes are identical except for *unique identifiers* (UID's).

The main uncertainty here is that the set of UID's actually possessed by the processors is unknown (although it is known that no two processors have the same UID's). The size of the network is also usually unknown.

Overview of course: chapter 3 (continued)

The main application of the problem is a local area ring network that operates as a token ring, in which there is always supposed to be a single token circulating, giving its current owner the sole right to initiate communication.

Sometimes the token gets lost and it becomes necessary for the processors to execute an algorithm that regenerates the missing token. This regeneration procedure amounts to *electing a leader*.

We will study the necessary time and the amount of communication required to elect a leader.

Overview of course: chapter 4

We give a brief survey of basic algorithms used in more general networks. In particular algorithms to:

- elect a leader
- conduct a breadth-first search
- find shortest paths
- find a minimum spanning tree
- find a maximal independent set of nodes

Overview of course: chapters 5 and 6 (as time permits)

We consider problems of reaching a consensus in a distributed network. Here a collection of distributed processors are required to reach a common decision, even if there are initial differences of opinion on what the decision ought to be.

For instance: processors could monitor separate altimeters on board of an aircraft and could be attempting to reach agreement about the altitude. Or the processors could carry out separate fault diagnosis procedures for some other system component and could attempt to combine their individual diagnoses into a common decision on whether or not to replace the component.

Overview of course: chapters 5 and 6 (continued)

In Chapter 5 we consider where links can fail by losing messages.

In Chapter 6 we consider two different types of processor failures:

1) stopping failures, where faulty processors can at some point just stop executing their local protocols.

2) Byzantine failures, where faulty processors can exhibit completely arbitrary behavior (subject to the condition that they can not corrupt portions of the system to which they do not have access).

Here the problem is to determine bounds on the number of tolerable faults, on the time and on the amount of communication.

Note: as time permits we will also cover the asynchronous model.

Formal model for synchronous system

A *synchronous network system* consists of a collection of computing elements located at nodes of a directed network graph.

We can consider the computing elements to be ”processors”. Or we can think of them as ”logical software processes” running on (but not identical to) the actual hardware processors. The results we will consider make sense in either case.

We call the computing elements ”processes” from now on.

In order to define a synchronous network system formally, we start with a directed graph (V,E).

n denotes the number of nodes in the digraph, also indicated by $|V|$.

Formal model: graph terminology

For each node i in the graph G we use the notation:

$in\text{-}nbrs_i$ denotes the incoming neighbors of i (nodes from which there edges to i).

$out\text{-}nbrs_i$ denotes the outgoing neighbors of i (nodes to which there are edges from i).

$distance(i, j)$ denotes the length of the shortest path from i to j if any path exists and ∞ otherwise.

$diam$ denotes the "diameter", the maximum distance $distance(i, j)$ taken over all (i, j).

Formal model:
Processes, States and Transitions

We assume that we have some fixed message alphabet M and let $null$ be a placeholder indicating the absence of a message.

With each node i in V we associate a process.

A process formally consists of four components:

- $states_i$, a (possibly infinite) set of *states*
- $start_i$, a nonempty subset of $states_i$ known as the *start states* or *initial states*
- $msgs_i$, a $message\text{-}generating\ function$ mapping $states_i \times out\text{-}nbrs_i$ to elements from $M \cup \{null\}$
- $trans_i$, a $state\text{-}transition function$ mapping $states_i$ and vectors (indexed by $in\text{-}nbrs_i$ of $M \cup \{null\}$ to $states_i$

Formal model: Processes, States and Transitions (continued)

Each process has a *set of states*, among which a subset of *start states*.

There can be infinitely many states (to model unbounded processes such as counters).

The *message-generating function* specifies for each state and outgoing neighbor the message (if any) that process i sends to the indicated neighbor, starting from the given state.

The *state-transition function* specifies for each state and collection of messages from all the incoming neighbors, the new state to which process i moves.

Formal model:
Channels and Execution

Associate with each edge (i, j) in G a *channel*, also known as a *link*, a location that, at any time, holds at most a single message in M.

Execution of the entire system begins with all the processes in arbitrary start states and all channels empty.

Then the process, in lock-step, repeatedly performs the following two steps:

1) Apply message-generating function to the current state to generate messages to be sent to all outgoing neighbors. Put these messages in the appropriate channels.

2) Apply state-transition function to the current state and the incoming messages to obtain a new state. Remove messages from channels.

Formal model:
Rounds

The combination of these two steps is called a *round*.

We do not place restrictions on the amount of computation a process does to compute the values of these two functions.

The model presented here is deterministic. The two functions are single-valued functions. In other words, given a particular collection of start states, the computation unfolds in a unique way.

Formal model:
Halting, Inputs, Outputs

Halting: We can distinguish some of the states as *halting states*, and specify that no further activity can occur from these states. That is, no messages are generated and the only transition is a self-loop.

Inputs and Outputs: Inputs and outputs are encoded in states. Inputs are placed in designated input-variables in the start states. A process can have multiple start-states, so we can accommodate different possible inputs. In fact, normally we assume that the only source of multiplicity of start states is the possibility of different input values in the input variables. Outputs appear in designated output variables; each of these records the result of only the first write operation that is performed (i.e., it is a write-once variable). Output variables can be read any number of times however.

Formal model: Executions

In order to reason about the behavior of a synchronous network system, we need a formal notion of a system "execution".
A *state assignment* of a system is defined to be an assignment of a state to each process in the system. Also, a *message assignment* is an assignment of a (possibly $null$) message to each channel. An *execution* of the system is defined to be an infinite sequence

$$C_0, M_1, N_1, C_1, M_2, N_2, C_2, \ldots,$$

where each C_r is a state assignment and each M_r and N_r represent the messages that are sent and received at round r respectively. (These may be different because channels may lose messages. We will return to this in "faults").

Formal model:
Executions (continued)

We refer to C_r as the state assignment that occurs at *time r*. That is, time r refers to the point just after r rounds have occurred.

If α and α' are two executions of a system, we say that α is indistinguishable from α' with respect to a process i, if i has the same sequence of states, the same sequence of outgoing messages and the same sequence of incoming messages in α and α'.

α and α' are indistinguishable to process i through r rounds if i has the same sequence of states, the same sequence of outgoing messages and the same sequence of incoming messages up to the end of round r, in α and α'.

Formal model:
Proof method: invariant assertions

An invariant assertion is a property of the system state (in particular, of the states of all the processes) that is true in every execution, after every round.

We allow the number of completed rounds to be mentioned in assertions, so that we can make claims about the state after each particular number r of rounds.

Invariant assertions are generally proved by induction on r, the number of completed rounds, starting with $r = 0$.

Formal model:
Complexity measures

The *time complexity* of a synchronous system is measured in terms of the number of rounds until all the required outputs are produced, or until the processes all halt.

The *communication complexity* is measured in terms of the total number of non-$null$ messages that are sent. Occasionally one takes into account the number of bits in a message.

The time measure is the most important measure. The communication complexity is only significant if it causes enough congestion to slow down processing.

Leader election in synchronous ring

- In a local area ring network a token circulates around
- The token gives the current owner the sole right to initiate communication
- This avoids simultaneous communication interference
- Sometimes the token gets lost
- A procedure is needed to regenerate the token
- This amounts to electing a leader

The problem

- Network graph
- n nodes, 1 to n clockwise
- symmetry and local knowledge only
- nodes do not know their number or neighbor numbers
- nodes can distinguish clockwise and anticlockwise neighbors
- notation: operations mod n to facilitate matters
- Requirement: eventually, exactly one process outputs the decision leader:
  change status component of state to value: *leader*

Possible versions of the problem

- The other non-leader processes must also output non-leader
- The ring can be unidirectional (e.g. clockwise message passing only)
- The ring can be bidirectional
- Number of processes n can be: known; unknown. (In the known case: the value n can be used in programs)
- Processes can be: identical or have totally ordered unique identifiers (UID)
  (UID's different but no constraint on which UID's appear)
- UID's restricted to comparisons or admit unrestricted operations

Identical processes: impossibility to break symmetry

If all the processes are identical then the problem cannot be solved. This is so even if the ring is bidirectional or the ring size is known to processes.

Theorem 1 *Let A be a system of n processes,* $n > 1$*, arranged in a bidirectional ring. If all the processes in A are identical then A does not solve the leader election problem.* □

Proof: Assume WLOG that we have one starting state. (A solution admiting several starting states would have to work for any of those). We have, therefore, a unique execution. By a trivial induction on r , the rounds executed, we can see that all processes have identical state after any number of rounds. Therefore, if any process outputs leader, so must the others, contradicting the uniqueness requirement.

Breaking symmetry

- Impossibility follows from symmetry
- We must break symmetry; e.g. with unique UIDs
- Symmetry breaking is an important part of many problems in distributed systems

A basic algorithm: LCR

- LCR algorithm (Le Lann, Chang, Roberts)
- Uses comparisons on UIDs
- Assumes only unidirectional ring
- Does not rely on knowing ring size
- Only the leader produces output, announcing it is the leader

Intuitive description of LCR

- Each process sends its UID to next
- If a received UID is greater than self UID, it is relayed on
- If it is smaller, it is discarded
- If it is equal, the process ouputs ”leader”

Formal model and LCR

Message alphabet M = set of UID's
For each i, the states in $states_i$ consist of:

u: a UID	[initially i's UID]
$send$: a UID or $null$	[initially i's UID]
$status$: $unknown$ or $leader$	[initially $unknown$]

The set of start states $start_i$ consists of the single state defined by the given initializations.

For each i, the message-generating function $msgs_i$:

send the current value of $send$ to process $i+1$

Note: process i would use a relative name for process $i+1$, for example "$clockwise\ neighbor$". We write $i + 1$ since it is simpler.

Formal model and LCR (continued)

Note: $null$ is used to indicated the absence of a message. If the value of $send$ is null, the msg_i function does not send any message.

For each i, the transition function $trans_i$ is defined by the following pseudo-code:

$send$:= $null$
if the incoming message is v, a UID, then
    cases:
        $v > u : send := v$
        $v = u : status := leader$
        $v < u : do\,nothing$

The first line of the transition cleans up the state from the effects of the preceding message delivery (if any).

Correctness of LCR

Let i_{max} be the index of process with maximum UID. Denote the UID be u_{max}.

Show two lemmas:

Lemma: Process i_{max} ouputs leader by the end of round n.
Lemma: Processes $i \neq m$ never ouput leader.
Theorem: LCR solves the leader election.

Correctness of LCR: first lemma

"Process i_{max} ouputs leader by end of round n."

Proof: u_{max} = initial value of the variable $u_{i_{max}}$ (the variable u at process i_{max}) by initialization.

Values of u variables never change (see code)
These values are all distinct (by assumption)
i_{max} has largest u value (by definition of i_{max})

Show the *invariant assertion:*

After n rounds, $status_{i_{max}} = leader$.

By induction on the number of rounds.

To do this we need another invariant that says something about the situation after a smaller number of rounds.

Correctness of LCR: first lemma (continued)

For $0 \leq r \leq n-1$:
after r rounds, $send_{i_{max}+r} = u_{max}$.

(Recall: in our ring addition is modulo n).

The assertion is show by induction over r.

$r = 0$: by initialization, $send_{i_{max}} = u_{max}$.

For $r > 0$: assume that after $r-1$ rounds:

$$send_{i_{max}+(r-1)} = u_{max}.$$

Then: on the next round: $send_{i_{max}+r} = u_{max}$.

(Nodes $\neq i_{max}$ place u_{max} in the $send$ component since u_{max} is greater than all other values.)

"After n rounds, $status_{i_{max}} = leader$" follows for $r = n-1$ and carrying out extra round.

Correctness of LCR: second lemma

No process other than i_{max} outputs $leader$.

Proof: We show that other processes always have $status$ = $unknown$.

i, j processes, $i \neq j$, let:

$[i, j) = \{i, i{+}1, \ldots, j{-}1\}$ (addition modulo n).

Invariant:

After r rounds:

if $i \neq i_{max}$ and $j \in [i_{max}, i)$ then $send_j \neq u_i$.

(Such i can not receive u_i and $status_i \neq leader$.)

Key in proof: a non-maximum value can't get past i_{max} ($u_{max} >$ all other UID's).

Only process i_{max} can receive its own UID in a message. Only process i_{max} outputs $leader$.

LCR solves the leader election problem.

Halting and non-leader outputs for LCR

Modify the algorithm by allowing elected leader to send special *report* message around the ring.

Any process receiving the *report* message halts after passing it on.

One can have the process creating $non\text{-}leader$ as output before halting.

By attaching the leader's index to the $report$ message, one can also allow processes to output identity of leader.

One can let non-leader node output $non\text{-}leader$ after receiving UID greater than its own. Does not tell nodes when to halt.

Complexity Analysis of LCR

Time complexity of LCR: n rounds.
Communication complexity: $O(n^2)$ (worst case).

Complexity Analysis of halting LCR

Time complexity: $2n$.
Communication complexity: $O(n^2)$.

The extra time needed for halting and for non-leader announcements is n rounds. The extra communication is only n messages (since processes halt).

An $O(nlogn)$ algorithm: HS

- HS algorithm (Hirshberg, Sinclair)
- Uses comparisons on UIDs
- Assumes bidirectional ring
- Doesn't rely on knowing the size of the ring
- Only the leader performs output

Intuitive description of HS

- Processes operate in phases l = 0, 1, 2, ...
- In each phase, processes send token with UID in *both* directions
- Tokens in phase l travel 2l and turn back to sender
- If a received UID is greater than self UID, it is relayed on
- If it is smaller, it is discarded
- If it is equal, the process ouputs leader
- Tokens on their way back are simply relayed on
- Tokens may not make it back (due to discarding/leader electing)
- Next phase kicks in when *both* tokens make it back safely

Formal model and HS

Bookkeeping (carried with token):
Flags: inbound or outbound.
Hop counts: distance travelled outbound
(decides reversal of direction).

Message alphabet M is a set of triples:

UID
Flag value: $\{out, in\}$
Hop-count: positive integer

For each i, the states $states_i$:

u: a UID [initially i's UID]
$send^+$: element of M or $null$
[initially triple (i's UID,$out, 1$)]
$send^-$: element of M or $null$
[initially triple (i's UID,$out, 1$)]
$status$: $unknown$ or $leader$ [initially $unknown$]
$phase$: a nonnegative integer [initially 0]

Formal model and HS (continued)

The set of start states $start_i$ with a single state consisting of initializations.

For each i, the transition function $trans_i$:

$send^+ := null$
$send^- := null$

Message from $i-1$ is (v, out, h) then cases:
  $v > u$ and $h > 1$: $send^+ := (v, out, h-1)$
  $v > u$ and $h = 1$: $send^- := (v, in, 1)$
  $v = u$: $status := leader$
Message from $i+1$ is (v, out, h) then cases:
  $v > u$ and $h > 1$: $send^- := (v, out, h-1)$
  $v > u$ and $h = 1$: $send^+ := (v, in, 1)$
  $v = u$: $status := leader$
Message from $i-1$ is $(v, in, 1)$ and $v \neq u$:
  $send^+ := (v, in, 1)$
Message from $i+1$ is $(v, in, 1)$ and $v \neq u$:
  $send^- := (v, in, 1)$

Messages from $i-1$ and $i+1$ are both $(u, in, 1)$:

$phase$:= $phase + 1$

$send^+$:= $(u, out, 2^{phase})$

$send^-$:= $(u, out, 2^{phase})$

As before, first two lines clean up state

Next two pieces of code: outbound tokens

Next two pieces of code: inbound tokens

Process i receives both tokens back: next phase

Communication complexity of HS

Phase 0: every process sends token: at most $4n$ messages (out and back).

Phase $l > 0$: every process sends a token only if it receives both its phase $l - 1$ tokens back.

This happens if it is not "defeated" by another process in distance 2^{l-1} in either direction. Within any group of $2^{l-1} + 1$ consecutive processes, at most one goes on to initiate tokens at phase l.

So at most $\lfloor \frac{n}{2^{l-1}+1} \rfloor$ processes initiate tokens at phase l.

Total number of messages sent at phase l at most $4(2^l \times \lfloor \frac{n}{2^{l-1}+1} \rfloor) \leq 8n$.

Communication complexity of HS (continued)

Total number of phases executed before leader election and all communication stops is at most $1 + \lceil logn \rceil$ (including phase 0).

Hence total number of messages is at most $8n(1+\lceil logn \rceil)$ and thus $O(nlogn)$.

Time complexity of HS

The time complexity for HS is $O(n)$.

Note that the time for each phase l is $2 \times 2^l = 2^{l+1}$ (tokens go out and return).

Final phase takes time n, an incomplete phase where tokens only travel outbound.

Next-to-last phase is phase $l = \lceil logn \rceil - 1$ and its time complexity is at least the total time complexity of all preceding phases.

Total time complexity of all phases (except for last): $2 \times 2^{\lceil logn \rceil}$.

Hence total time at most: $3n$ if n is a power of 2 and $5n$ otherwise. Conclusion: $O(n)$.

Algorithms in General Synchronous Networks

We considered algorithms for leader election in unidirectional and bidirectional rings. We proceed to more general networks based on strongly connected digraphs.

A directed graph is strongly connected if there is a path from each vertex in the graph to every other vertex. In particular, this means paths in each direction; a path from a to b and also a path from b to a. The diameter of such a graph is finite.

To support efficient communication we discuss *electing a leader* to ”take charge” of network computation but also *breadth-first search (BFS)*, finding *shortest paths*, finding a *minimum spanning tree (MST)* and finding a *maximal independent set (MIS)* in such networks.

The context of the more general networks

We consider arbitrary strongly connected digraphs $G = (V, E)$ with n nodes.

Processes only communicate over directed edges of the digraph.

Nodes are named via indices $1, \ldots, n$ but, unlike the ring's indices, these have no special connection to the nodes' positions in the graph.

The processes do not know their indices, nor those of neighbors, and refer to neighbors by local names.

If a process i has the same process j as incoming and outgoing neighbor, then we assume that i knows the two processes are the same.

Leader Election in a General Network: The Problem

Processes have unique identifiers. No two identifiers are the same.

Eventually one process needs to elect itself the leader, by changing its status component to the value *leader*.

It might be required that all non-leader processes output the fact that they are not the leader, change status to: *non-leader*.

The number of nodes n and the diameter, *diam*, can either be known or unknown to the processes. Or an upper bound on these quantities might be known.

A Simple Flooding Algorithm

The algorithm requires that processes know *diam*. The algorithm floods the maximum UID throughout the network. It is called *FloodMax*.

FloodMax algorithm (informal): Every process maintains a record of the maximum UID (initially its own). At each round, each process propagates this maximum on all of its outgoing edges. After *diam* rounds, if the maximum value seen is the process's own UID, the process elects itself the leader; otherwise, it is a non-leader.

FloodMax algorithm (formal I)

Message alphabet = set of UID's

$states_i$ consists of components:

u, a UID
(initially i's UID)

$max - uid$, a UID
(initially i's UID)

$status \in \{unknown, leader, non - leader\}$
(initially $unknown$)

$rounds$, an integer
(initially 0)

FloodMax algorithm (formal II)

$msgs_i$:

If rounds $< diam$ then
send $max\text{-}uid$ to all $j \in out - nbrs$.

$trans_i$:

$rounds := rounds + 1$
let U be the set of UID's that arrive from processes in in-nbrs
$max\text{-}uid := max(\{max\text{-}uid\} \cup U)$
if $rounds = diam$ then
if $max\text{-}uid = u$ then $status := leader$
else $status := non - leader$

FloodMax algorithm: correctness

Floodmax elects the leader: the process with maximum UID. As before, define i_{max} to be the index of the process with maximum UID and u_{max} that UID.

Theorem 4.1 *In the FloodMax algorithm, process i_{max} outputs leader and each of the other processes outputs non-leader, within diam rounds.*

We show the following:

Assertion 4.1.1 *After diam rounds, $status_{i_{max}}$ = leader and $status_j$ = non-leader for every $j \neq i_{max}$.*

FloodMax algorithm: key to proof

The key to the proof is that after r rounds, the maximum UID has reached every process that is within distance r of i_{max}, as measured along directed paths in G. This is captured by the invariant:

Assertion 4.1.2 *For $0 \leq r \leq diam$, and for every j, after r rounds, if the distance from i_{max} to j is at most r, then $max - uid_j = u_{max}$.*

In view of the definition of the diameter, Assertion 4.1.2 implies that every process has the maximum UID by the end of diam rounds. To prove Assertion 4.1.2, we prove the additional invariants:

FloodMax algorithm: more invariants

Assertion 4.1.3: For every r and j, after r rounds, $rounds_{s_j} = r$.

Assertion 4.1.4: For every r and j, after r rounds, $max - uid_j \leq u_{max}$.

These three assertions, specialized to $r = diam - 1$, plus an argument what happens at round $diam$, imply Assertion 4.1.1 and therefor the result.

Complexity Analysis

The time until the leader is elected (and all other processes know that they are not the leader) is *diam* rounds. The number of messages is $diam \times |E|$, where $|E|$ is the number of directed edges, since a messages is sent on every directed edge for each of the first $diam$ rounds.

The algorithm also works correctly if the processes all know an upper bound d on the diameter rather than the diameter itself. The complexity measures then increase so that they depend on d instead of $diam$.

Reducing the Communication Complexity

We can improve the communication complexity somewhat (but not by an order of magnitude) when processes send their max-uid values only when they first learn about them, not at every round. This algorithm is $OptFloodMax$. The modification to the code of $FloodMax$ is as follows:

$OptFloodMax$ algorithm:

$states_i$ has an additional component:

$new\text{-}info$, a Boolean
(initially true)

$OptFloodMax$ **continued**

$msgs_i$

if $rounds < diam$ and $new{-}info = true$
then send $max\text{-}uid$ to all $j \in out\text{-}nbrs$

$trans_i$

$rounds := rounds + 1$
Let U be the set of UID's that arrive from processes in in-nbrs
if $max(U) > max\text{-}uid$
    then $new\text{-}info := true$
    else $new\text{-}info = false$
$max\text{-}uid := max(\{max\text{-}uid\} \cup U)$
if $rounds = diam$ then
    if $max\text{-}uid = u$ then $status := leader$
    else $status := non\text{-}leader$

$OptFloodMax$: correctness

This is a first example of the simulation method to make the correctness proof shorter, by relying on the correctness proof for $FloodMax$.

Theorem 4.2: *In the OptFloodMax algorithm, process i_{max} outputs leader and the other processes output non-leader, within diam rounds.*

Assertion 4.1.5: *After diam ronds, $status_{i_{max}}$ = leader and $status_j$ = non-leader for ever $j \neq i_{max}$.*

Invariant:
Assertion 4.1.6: *For any $r, 0 \leq r \leq diam$, and any i, j where $j \in out\text{-}nbrs_i$, the following holds: after r rounds, if $max\text{-}uid_j < max\text{-}uid_i$ then $new\text{-}info_i = true$.*

By induction on r...

To show $OptFloodMax$ is correct, run it side by side with $FloodMax$, starting with the same UID assignment. We show the invariant assertion that shows the states of the algorithms after the same number of rounds.

Assertion 4.1.7: *For any $r, 0 \leq r \leq diam$, after r rounds, the values of the u, max-uid, status and rounds components are the same in the states of both algorithms.*

By induction on r. Interesting step: show max-uid values remain identical.

Assertions 4.1.7 and 4.1.1 imply Assertion 4.1.5.

Directed spanning tree

A digraph is *strongly connected* if for every pair of vertices u and v there is a directed path from u to v.

A digraph G=(VE) is said to have *a root r* if every vertex v is reachable from r, i.e. if there is a directed path from r to v .

A digraph is called a *directed* tree if it has a root and if the underlying (undirected) graph is a tree. That is, it must look like a tree, and the structure imposed by the directional arrows must flow “away” from the root.

If H is a subgraph of a digraph G , then H is said to be a *directed spanning tree* of G if H is a directed tree and H contains all vertices of G.

If r is the root of H , then r is a root of G.

Directed spanning tree: construction

If r is any root of G , it is possible to construct a directed spanning tree H of G with root r as follows:

1) Construct H edge by edge starting from r.

2) At each vertex, add any edge from G whose terminus is a vertex not yet in H.

Note that since r is a root of G , this process is guaranteed to include each vertex in G; since we are choosing at each step only vertices not yet visited, we are guaranteed to end up with a tree. The tree satisfies:

Each node at distance i from the root in G occurs at depth i in the tree.

This is a *breadth first* spanning tree.

Breadth First Search problem (BFS)

For the BFS problem we assume that:

- the network is strongly connected (so every two nodes always can communicate).

- there is a distinguished source node i_0.

- processes only communicate over directed edges and have unique UID's

- processes have no knowledge of the size or diameter of the network.

The algorithm needs to output the structure of a breadth-first spanning tree of the network graph, rooted at i_0. This output appears in a distributed fashion: each process (other than i_0) has a parent component to indicate the node that is its parent in the tree.

A Basic Breadth First Search Algorithm

SynchBFS algorithm
At any point during the execution there is a set of processes that is "marked", initially only i_0.

Process i_0 sends out a *search* message at round 1, to all of its outgoing neighbors.

At any round, if an unmarked process receives a *search* message, it marks itself and chooses one of the processes from which the *search* arrived as its parent.

At the first round after a process gets marked, it sends a *search* message to all of its outgoing neighbors.

The SynchBFS algorithm produces a BFS tree.

Complexity Analysis

Time complexity: number of rounds is at most the maximum distance from the root node i_0 to any other node.

Communication complexity: number of messages is the number of edges $|E|$. Each *search* message is sent once on each directed edge.

Reducing communication complexity: a newly marked process need not send a *search* message in the direction of any process from which it already has received such a message.

Adaptations of the algorithm

Message broadcast: The algorithm can be adapted to implement message broadcast. Piggyback a message on the search message sent in round 1. Other processes piggyback this message on their search messages. Since the tree spans all nodes, the message is delivered to all processes.

Child pointers Each process should learn its parent in the tree *and* its chlidren. Each process receiving a *search* message responds with a parent or non-parent message, telling the sender whether or not it has been chosen by the recipient as parent. The chosen parent then marks the sender as child.

If bi-directional communication is allowed between all pairs of neighbors (undirected graph), then this can be achieved at little extra cost.

Child pointers for unidirectional networks

If the network is unidirectional, some *parent* and *non-parent* messages may have to be sent via indirect routes.

The message could be sent via a new call to SynchBFS (piggybacking on a search message).

For recognition by the recipient: the message should carry the UID of the recipient (plus a local name by which the recipient knows the sender).

Executions of these SynchBFS calls can go on in parallel. (Many messages can be fitted into one to accommodate single message sending in our model).

Complexity Analysis for the case of child pointers

For an undirected graph:

Time complexity: O(*diam*)

Communication complexity: O($|E|$)

For directed graph:

Time complexity: O(*diam*)
(BFS executions in parallel)

Communication complexity: O(*diam*$|E|$)
(at most $|E|$ messages sent at each round)

Tutorial: analyze number of bits per message.

Termination (Convergecast)

The source process i_0 determines whether the construction of the tree is completed:

Each search message is answered with *parent* or *non-parent*. So after a process receives response from all its *search* messages: it knows its children and that they are marked.

Starting from the leaves of the tree: completion notification is fanned in to the source:

Each process sends notification of completion to its parent as soon as it received:

a) response from all its *search* messages

b) completion notification from all children.

This procedure is called *convergecast*.

Complexity Analysis for convergecast

For an undirected graph:

Time complexity: O($diam$)

Communication complexity: O($|E|$)

For directed graph:

Time complexity: O($diam^2$)
(Notification proceeds sequentially,
level per level in tree)

Communication complexity: O($diam^2|E|$)
(at most $|E|$ messages sent at each round)

Tutorial: analyze number of bits per message.

Applications of Breadth-First-Search and SynchBFS

Broadcast Use piggybacking as above. Or reuse BFS tree (in case it has been set up with child pointers): message only needs to be propagated along edges from parents to children. Additional time to broadcast: $O(n)$.

Global computation Collection of information throughout the network (computing a function based on distributed inputs).

Example: Each process has non-negative value integer input value. Goal: find sum of inputs.

Use a BFS tree, start at leaves, "fan in" results via convergecast. Each leaf sends its value to its parent; each parent waits for value from all its children, adds them to its input value and sends sum to its parent. Sum at root = answer.

Applications of Breadth-First-Search (continued I)

Undirected case: $O(diam)$ time and $O(n)$ messages (assuming BFS constructed earlier).

Similar algorithm to find maximum or minimum of inputs.

Electing a leader Using SynchBFS, initiate breadth-first searches in parallel. Each process i uses its tree and global computation to determine maximum UID of any process in the network. The process with the maximum UID then declares itself the leader and the other announce non-leader.

Undirected case: $O(diam)$ time and $O(diam|E|)$ messages (take into account parallelism).

Applications of Breadth-First-Search (continued II)

All processes initiate breadth first searches in parallel. Each process uses its tree to determine $max - dist_i$ (maximum distance to any other process in the network). Each process i then reuses Breadth-First-Search Tree for global computation to determine the maximum of the max-dist values.

Undirected case: $O(diam)$ time and $O(diam|E|)$ messages.

Tutorial: analyze number of bits per message.

Shortest paths

Aim: find ”cheapest” (”lightest”) path with respect to a ”cost” (”weight”) on edges.

Each edge has non-negative real-valued weight. Weights represent costs of edge traversal (communication delay, monetary charge,...)

Path weight = sum of the weights on its edges.

Problem = find ”shortest” (”lightest”) path from a source node i_0 to each node in the graph, i.e. the path with minimum weight.

The collection of shortest paths from i_0 to all other nodes in the digraph is a subtree of the graph, the *shortest-paths-tree*, all of whose edges are oriented from parent to child.

Assumptions

The weight of an edge appears in special weight variables at both its endpoint processes.

Each process knows the number n of nodes in the digraph.

Each process determines its parent in a particular shortest path tree and its distance (total weight of shortest path) from i_0.

If all edges have equal weight then BFS is a shortest path tree. In this case, the SynchBFS construction can be adapted to yield distance as well as parent pointers.

Otherwise, the Bellman Ford algorithm can be adapted to the distributed context.

BellmanFord algorithm for shortest paths

Each process i keeps track of:

$dist$ = shortest distance from i_0 known so far.

parent = incoming neighbor that precedes i in a path with weight $dist$.

Initially: $dist_{i_0} = 0$, $dist_i = \infty$ for $i \neq i_0$ and *parent* components are undefined.

On each round, each process sends $dist$ to outgoing neighbors.

Then each process i updates $dist$ to be minimum of $dist$ and values $dist_j + weight_{j,i}$, where j is an incoming neighbor. When *dist* changes, *parent* is updated.

BellmanFord algorithm: correctness and complexity

After $n-1$ rounds, *dist* contains shortest distance and *parent* is parent in shortest path tree.

By induction on r: show that every process i has its *dist* and *parent* components correspond to a shortest path among the paths from i_0 to i consisting of at most r edges. (If there are not such paths then *dist* = ∞, *parent* = undefined).

Time complexity: $n-1$, *Communication complexity*: $(n-1)|E|$.

Time complexity is <u>not</u> $diam$. Consider triangular network with source node i_0 and bidirectional communication. All weights are 1, except the weight of an edge from i_0 to one of its out-bound neighbors, which is 100. The diameter is 1, but 2 rounds to find shortest path.

Minimum spanning tree

Problem: find minimum-weight spanning tree (MST) in undirected graph network with weighted edges.

Broadcast communication: minimizing communication cost from source to any other process.

A *forest* = acyclic graph that is not necessarily connected.

A *spanning forest* = a forest that contains every vertex of G.

A *spanning tree* = a connected spanning forest.

Consider a graph with weighted edges. The *weight of a subgraph* (spanning tree or spanning forest) = sum of weight of its edges.

Assumptions

Undirected graph, i.e. ”bidirectional edges”.

Weight on directed edge = non-negative real number.

For any two nodes i, j: weight$_{i,j}$ = weight$_{j,i}$.

Weight stored in variables at both endpoints of processes.

Unique UID's, size n is known.

Problem: find minimum-weight (undirected) spanning tree. Each process to decide which of its incident edges are part of the tree.

Basic Sequential Strategy of Construction

Start with trivial spanning forest consisting of n single nodes and no edges.

Repeatedly ”merge” components along edges.

To merge: select an arbitrary component C in the forest and arbitrary outgoing edge of C with minimum weight.

Combine C with the component at the other end of e, including edge e in the new combined component.

Stop when the forest has a single component (and hence is a tree).

What About The Distributed Case?

Goal: extend forest with several edges determined concurrently to adapt to distributed setting.

Each of several components determines minimum weight of its outgoing edges independently.

These minimum weight edges are added to forest.

Several components are merged at once.

Unfortunately this strategy for the distributed case is not correct in general.

The problem

Consider an undirected graph with three nodes.

These three nodes initially form the spanning forest.

Assume that the weights on the edges are all equal to 1 and that these three edges are the only outgoing edges and they form a cycle.

If we add the minimum weight outgoing edge in parallel by choosing these three edges, we form a cycle and hence we do not construct a tree.

If *all* edges have distinct weight this problem does not occur.

The Distinct Weights Case

Lemma: if all edges of a graph have distinct weights, then there is exactly one Minimum Spanning Tree for G.

By contradiction: (sketch) suppose there are two MST's, say T and T'. Let e be the minimum weight edge that appears in only one of these trees (this edge exists since all weights are distinct). Say (without loss of generality) that this edge occurs in T. Then the graph T" obtained by adding this edge e to T' contains a cycle (T' was already spanning, thus adding a new edge creates a cycle.) At least one other edge e' on the cycle is not in T (else T contains a cycle. This is impossible since T is a tree). Since the edge weights are all distinct and by choice of e, we have: weight(e') $>$ weight(e). Removing e' from T" yields a spanning tree with smaller weight than T'. This is a contradiction.

The Distinct Weights Case

Strategy: if all edges have distinct weights, then by the previous Lemma, there is a unique MST.

In this case, at any stage of the construction, a component in the forest has exactly one minimum-weight outgoing edge.

By the Lemma: if we begin the stage with a forest, all of whose edges are in the unique MST, then all of the minimum weight outgoing edges, for all components, are also in the unique MST.

So we can add them all at once without any danger of creating a cycle.

The Algorithm SynchGHS

SynchGHS is named after Gallagher, Humblet and Spira. The algorithm combines components in ”levels”.

At each level k, the components constitute a spanning forest.

Each of these components is a tree that is a subgraph of MST and a spanning tree of the component.

Each level k component has at least 2^k nodes.

Every component at every level has a leader node.

The processes allow for a fixed number of rounds to complete a level. This is $O(n)$.

The Algorithm SynchGHS: property of levels

Level 0 components = all individual nodes, no edges.

Assume inductively that:

- the level k components have been determined (and their leaders).

- each of the processes knows the UID of the leader. (This is the UID used to identify the entire component.)

- each process knows which of its incident edges are in the component's tree.

The Algorithm SynchGHS: constructing the next level

To construct the level $k+1$ components:

Each level k component conducts a search along its spanning tree edges for the minimum-weight outgoing edge of the component.

The leader broadcasts search requests along tree edges, using the message broadcast strategy described earlier.

Each process finds (among its incident edges) the minimum weight outgoing edge (if it exists).

The Algorithm SynchGHS: finding the minimum weight outgoing edge

The process sends a test message along all non-tree edges, asking whether the node at the other end is in the same component.

This is checked by comparing component identifiers.

The processes convergecast this local minimum-weight edge info towards the leader, taking minima on the way.

The minimum obtained by the leader is the minimal weight outgoing edge of the component.

The Algorithm SynchGHS: completing the next level

After all level k components found their minimum weight outgoing edge:

the leader of each component communicates with the component adjacent to the minimum weight outgoing edge to make sure the edge is marked as being in the new tree. The process at the other end also does this.

a new leader is chosen for each of the level $k+1$ components.

The Algorithm SynchGHS: choice of new leader

Each group of components combined into a $k+1$ component has a unique edge e that is the common minimum weight outgoing edge of *two* components in the group. (This is shown later).

The new leader = endpoint of e with larger UID.

Broadcast UID of new leader through the new component.

Eventually, the spanning forest consists of a single component containing all network nodes.

Finding a minimum weight outgoing edge fails.

The leader learns this and broadcasts a completion message.

The Algorithm SynchGHS: Remaining observation

Among each group of level k components that get combined on the next level:

There is a unique (undirected) edge that is the common minimum weight outgoing edge of both endpoint components.

consider the component digraph G' whose nodes are the level k components that get combined and whose edges are the minimum weight outgoing edges. Every node of G' has exactly one outgoing edge. G' (when we consider its undirected version) is a connected graph. Such a graph must contain exactly one cycle of length 2 (exercise, tutorials). Copy the figure we produce on the whiteboard as an example.

The Algorithm SynchGHS: Synchronisation

Synchronisation is essential to ensure that when a process i tries to determine whether an endpoint j of a candidate edge is in the same component or not, both i and j have up to date component UID's. If the UID at j is observed to be different from that at i, we need to be certain that i and j really are in different components, not just that hey haven't yet received their component UID's from their leaders. In order to execute the levels synchronously, processes allow for a predetermined number of rounds for each level. To be certain that all the computation for the round has completed, the number will be O(n). The need to count this number of rounds is the reason the nodes need to know n.

The Algorithm SynchGHS: Complexity

Time complexity: the number of nodes in each level k is at least 2^k. At each level, each component is combined with at least one other component at same level. So number of levels is at most log n. Since each level takes O(n) time, the time complexity is O(nlogn). The communication complexity is $O((n + |E|)logn)$ since at each level, O(n) messages are sent in total along all the tree edges and $O(|E|)$ additional messages are required for finding the local minimum weight edges.

The Algorithm SynchGHS: Non distinct weights revisited

Non unique weight edges: how can we handle them? ... Use (weight$_{i,j}$,v,w) where v is smallest UID and w largest UID of i and j. Then use lexicographical order and run the prior algorithm.

Maximal Independent Set

We consider the problem of finding a *maximal independent set* (MIS) of the nodes of an undirected graph.

A set of nodes is called an independent set if it contains no neighboring nodes.

An independent set is maximal if it cannot be increased without losing its independence property.

Motivation: allocating shared resources to processes in a network. Neighbors represent processes that cannot simultaneously preform some activity involving shared resources (for example, data base access or radio broadcast). If we wish to select processes that can act simultaneously, we need to select an independent set. For performance, we try to make it maximal.

The problem

Let $G = (V, E)$ be an undirected graph. A set $I \subseteq V$ is independent if for all nodes $i, j \in I. (i, j) \notin E$. It is maximal if any set I' that strictly contains I is not independent.

To compute a maximal independent set, each process whose index is in I should output winner and the others should output loser.
We assume that n the number of nodes, is known to all processes. We do not assume the existence of unique UID's.

A Randomized Algorithm

The MIS problem can not be solved in some graphs in case the processes are required to be deterministic.

To solve this, randomized versions are used. The difference is that in a small number of cases, the algorithm may not terminate. This collection of cases will have probability zero (it is very sparse). The algorithm is called *LubyMIS* after its discoverer Luby.

The algorithm uses an iterative scheme in which an arbitrary non empty set is selected from the given graph G.

Nodes of this set and all neighbors are removed from the graph and the process is repeated.

Iterative scheme

If W is a subset of the nodes of a graph, the we use *nbrs(W)* to denote the set of neighbors of nodes in W.

Let I be a set of nodes, initially empty.

while grahp.nodes $\neq$ do
  choose a nonempty set $I' \subseteq graph.nodes$
  that is independent in graph
  $I := I \cup I'$
  graph := induced subgraph on
  graph.nodes - I' - graph.nbrs(I')

We recall that the induced subgraph of a graph G on a subset W is the defined to be the subgraph (W, E') where E' is the set of edges of G that connect nodes in W.

Correctness of Iterative scheme

The iterative scheme produces a maximal independent set.

Indendent: at each stage, the selected set I' is independent. And we discard from the remaining graph all neighbors of vertices that are put in I'.

Maximal: the only nodes removed from consideration are neighbors of nodes in I'.

Question: How does one choose an independent set?

Choice of independent set

We use randomization to choose the independent set I' at each stage.

In each stage, each process i chooses an integer val_i in the range $\{1, \ldots, n^4\}$ at random, using the uniform distribution.

The choice of n^4 guarantees that with high probability all processes in the graph choose distinct values. (Luby's research paper discusses the reason why this works).

Once the processes have chosen these values we define I' to consist of all the nodes i that are local winners. That is those nodes i such that $val_i > val_j$ for all neighbors j of i. This yields an independent set since two neighbors cannot simultaneously defeat each other.

Non termination

In this implementation, it is possible that the set I' might be empty at some stages (if the random choices are unlucky). Assuming the algorithm does not keep performing useless stages, accomplishing nothing, we can ignore the useless stages and assert that the algorithm correctly follows the general scheme.

Informal description of the algorithm

The algorithm works in stages, each consisting of three rounds.

Round 1: processes choose their respective values and send them to their neighbors. By the end of this round, when all the values have been received, the winners - the processes in I' - know who they are.

Round 2: the winners notify their neighbors. By the end of this round, the losers - that is the processes having neighbors in I' - know who they are.

Round 3: each loser notifies its neighbors. Then all the involved processes - the winners, the losers and the losers' neighbors - remove the appropriate nodes and edges from the graph. More precisely: winners and losers discontinue par-

ticipation after this stage, and the losers' neighbors remove all the edges that are incident on the newly removed nodes.

Randomized Algorithms

The technique of randomization can break symmetry. The leader-election problem cannot be solved without UID's. Randomization can help when UID's are available, it may even help to break symmetry faster.

Randomized algorithms may only be correct or have a certain performance with a high probability, not with certainty.

Crucial properties need to hold with certainty, not probabilistically. For instance, the LubyMIS algorithm is guaranteed to produce an independent set, regardless of the outcomes of the random choices, when it terminates. The performance (and termination) depends on the luckiness of the random choices. Whether or not this is a serious drawback depends on the application for which it is used.

Distributed Concensus with Link Failures

Examples of Distributed Concensus:

Processors need to reach agreement on whether to commit or abort results of a distributed data base transaction.
Estimate an airplane's altitude based on readings of altimeters.
Classify a system component as faulty, given results of separate diagnostic tests carried out by separate processes.

We study one problem: The Coordinated Attack Problem, useful for the case where messages can get lost.

Battlefield interpretation

Several generals plan an attack from different directions against common objective. The only chance of success is if they all attack. If only some attack, the armies will be destroyed.

Each general has an initial opinion on whether his army is ready to attack.

The generals are located in different places and communicate to nearby generals via messengers. The messengers could get shot and not make it.

The generals must manage to agree on whether or not to attack and they should attack if possible.

Distributed Databases

The battlefield interpretation corresponds to a real computer science problem. A collection of processes participated in the processing of a transaction. After this processing, each process arrives at an initial ”opinion” about whether the transaction ought to be *committed* (its results made permanent for the use of other transactions) or *aborted* (the results discarded).

A process will generally favor committing the transaction if all local computations on behalf of that transaction have been completed, and will favor aborting otherwise. The processes need to communicate and eventually agree to commit or abort. If possible, the outcome should commit.

Impossibility result

In case no messages get lost, the problem is solvable.

Otherwise it is not solvable (but we will see a version that is solvable).

We formalize first: Consider n processes indexed $1, \ldots, n$, in an undirected graph network. Each process knows the entire graph, including the process indices. Each process starts with an input in $\{0, 1\}$ in a state component. 1 denotes "attack" (commit), 0 denotes "don't attack" (abort). We use our usual synchronous model, but allow any number of messages to be lost during the execution. The goal is for all the processes to eventually output decisions in $\{0, 1\}$, setting the decision state component.

Imposed conditions

There are three conditions imposed on the decisions:

1) Agreement:

No two processes decide on different values.

2) Validity:

If all processes start with 0
then 0 is the only possible decision value.

If all processes start with 1 *and*
all messages are delivered
then 1 is the only possible decision value.

3) Termination:
All processes eventually decide.

Impossibility result

Theorem *If G is a graph consisting of two nodes, 1 and 2, connected by a single edge then there is no algorithm that solves the coordinated attack problem on G.*

See proof on whiteboard and tutorial.

Randomization

We weaken the problem statement, allowing for some probability of error.

Same validity condition, but weaken the agreement condition to allow a small probability ϵ of disagreement.

We assume the algorithm terminates after $r \geq 1$ rounds, where each process has to output a decision in $\{0, 1\}$.

A message is sent around each edge at each round $k, 1 \leq k \leq r$ and any number of messages may be lost.

We represent the lost messages by the idea of an *adversary*.

Adversaries

A communication pattern is a subset of the set:

$\{(i, j, k) |$ (i, j) is an edge in the graph and $1 \leq k\}$.

A communication pattern γ is *good* in case: $k \leq r$ for every $(i, j, k) \in \gamma$.

A good communication pattern represents the set of messages delivered in an execution: if (i, j, k) is in the communication pattern, it means a message sent by i to j in round k succeeds in getting delivered.

This also determines which messages are not delivered (the edges that have been left out per round). Hence a good communication pattern also determines an "adversary" (some opponent that kills off messages, as in the battlefield example).

Adversaries and probabilities

An adversary is a choice of:

a) Assignment of input values to all processes.
b) A good communication pattern.

We can now state our weakened agreement condition:

Agreement: for every adversary B:

Prob^B[some process decides 0 and some process decides 1] $\leq \epsilon$.

If we can achieve ϵ to be small, then the chance that processes take a different decision at the end is small. We will present an algorithm for which $\epsilon = \frac{1}{r}$.

Information Order

For any communication pattern γ, we define an ordering $\leq_\gamma$ on pairs (i, k), where i is a process index and k is a time (the moment right after k rounds occurred). The ordering represents information flow between processes at various times.

Information flow at the same process:

1) $(i, k) \leq_\gamma (i, k')$ when $k \leq k'$.

Information flow from sender to receiver:

2) If $(i, j, k) \in \gamma$ then $(i, k-1) \leq_\gamma (j, k)$.

The order is transitive:

3) If $(i, k) \leq_\gamma (i', k')$ and $(i', k') \leq_\gamma (i", k")$ then $(i, k) \leq_\gamma (i", k")$.

Information Levels

Each process starts at level 0. When it hears from all the other processes, it advances to level 1. When it hears that all the other processes reached level 1, it moves to level 2 and so on.

1) $k = 0$ then $level_\gamma(i, k) = 0$.

2) $k > 0$ and there is some $i \neq j$ such that $(j, 0) \not\leq_\gamma (i, k)$: $level_\gamma(i, k) = 0$.

3) $k > 0$ and for every $j \neq i$: $(j, 0) \leq_\gamma (i, k)$: for every $j \neq i$, let

$$l_j = max\{level_\gamma(j, k') : (j, k') \leq_\gamma (i, k)\}.$$

(The largest level that i knows that j has reached.)

$$level_\gamma(i, k) = 1 + min\{l_j : j \neq i\}.$$

Computing the Information Levels

Copy the example given in class from the white-board.

The complete randomized algorithm based on levels has been given in class where notes needed to be taken from the whiteboard.