Complexity measures

Message complexity Number of messages sent (in total) during an execution of the algorithm

Time complexity Two ways to define depending on the system:

- for synchronous systems can just count number of steps taken
- for asynchronous systems:
  1. measure number of steps each process has to take
  2. measure number of rounds
     (Round is a minimal section of execution where every process takes a step)
     (Rounds may be of different length)

Space complexity Size of data structures in shared memory

Other measures of quality of solution:

- Number of failures tolerated by the algorithm
- Probability of success (for randomized algorithm or for deterministic algorithm that is executed in system where there is a known probability distribution on executions)

Most of these quantities are measured in terms of the size of input.

Types of Problems

There are some problems that are inherently distributed.

1. Resource sharing problems - problems must communicate to divide up resources (ex: mutual exclusions)
2. Problems where input is distributed along the system
3. Fundumental communication problems (like broadcasting a message)
4. How to distribute info across a system for fault tolerance (ex: replicating a database in case of a disk crash; how to update such a database?)

Formal models of distributed systems

Distributed systems can often be modelled formally by I/O (Input/Output) automata. See Lynch's book for details. Here we show how to model an asynchronous message passing system with reliable FIFO channels.

An I/O automaton is defined by

1. A set of states (not necessarily finite): the state represents contents of component's memory
2. Set of initial states

3. A set of actions
   There are 3 types of actions:
   - internal actions (not visible to other components of the system)
   - input actions (get info from other components of the system)
   - output actions (give info to other components of the system)
4. Transition relation R - a subset of $Q \times ACT \times Q$. If $(q, act, q) \in R$ then if the automaton is in state q, it can perform the action act and go into state q'

A process can be modelled as an automaton. Local variables (+ programme counter) are part of the state of the automaton and the programme the process is executing are encoded in the transition relation. Messages sent to other processes are modelled as output actions, messages received from the other processes are input actions.

Message channels can also be modelled as I/O automata.

FIFO, reliable channel from process i to process j:

State of channel - contents of 2 buffers, which both act as FIFO queues.

Input action $send_{i,j}(m)$ New message m gets added to the end of outbuffer

Internal action $move_{i,j}(m)$ Moves message m from front of outbuffer to end of inbuffer.

Output action $receive_{i,j}(m)$ When a message m gets removed from the front of inbuffer queue

These actions are only allowed to happen (by the transition relation) if the channel is in an appropriate state. For example,

$$(([m_1, ..., m_k], [m_{k+1}, ..., m_l]), receive_{i,j}(m_l), ([m_1, ..., m_k], [m_{k+1}, ..., m_{l-1}])) \in R$$

$$(([m_1, ..., m_k], [m_{k+1}, ..., m_l]), move_{i,j}(m_k), ([m_1, ..., m_{k-1}], [m_k, ..., m_l])) \in R$$

We can easily modify specification of I/O automaton to describe possible errors in the channel. (Eg. message losses, message reorderings, etc.)

The *move* action is used to model the fact that message delivery can take arbitrarily long in an asynchronous system. Process i *send*s the message. At some later point, a *move* action occurs, and only then is the message available for process j to receive it.

Combining automata into a system.

The action can happen both in the process and in the channel simultaneously. For example, when the message is sent by the process, $send_{i,j}(m)$ occurs in the process and $receive_{i,j}(m)$ occurs in the channel.

Generalizing this example, if several components of the system have same action, that action can occur in the system iff it can happen in every component (according to that component's transition relation). When the action takes place, all of the components involved in the action undergo a state transition simultaneously.

A system would then consist of a collection of process automata and a collection of channel automata to represent the communication channels between the processes.

Broadcasting

Reference: Attiya and Welch, Sections 2.1-2.3.

Model:

- Msg passing system (connected network),
- asynchronous,
- reliable, FIFO biderectional channels,
- each proc initially knows nothing about network topology (all the process knows in the beginning is that it has some number of outgoing channels)
- Process halting failures, but assume failures never cause a disconnected network

Problem: Some process P gets (as an input action) a message. Eventually every other (non-faulty) process in the system must receive a copy of the message (if P is non-faulty)

Main idea: "Flood" network: When you receive a message, forward it to all your neighbours (except the process that sent it to you). Don't bother forwarding message again if you get it more than once. To detect duplicates, assign a distinct serial number to each broadcast, and attach the serial number to the message as a header. (The serial number can consist of the id of the process originating the broadcast together with the number of broadcasts that have been originated by that process already.)

Every non-faulty process will eventually get the message.

Proof: Suppose some non-faulty process Q did not receive the message. There is a path of non-faulty processes from P to Q (since the process failures do not disconnect the network graph). Then for every process in that path it is true that if that process did not get a message, its predecessor did not get it as well (since the predecessor would have forwarded the message if it had received the message). So we have a contradiction, because the path starts at P, which did get the message.

If the network graph has n nodes (processes), m edges, the message complexity of the broadcast is at most $2m$ (the message can be sent across every edge at most once in each direction)

Suppose P needs to know when the broadcast is complete. To satisfy this need, we can augument our system with acknowledgements (assuming there are no faulty processes): when you receive a message from X, send it to your other neighbours and wait until you get ACKs from all of them, then send ACK1 to X. When P receives all ACKs - done. If you get the same message a second time, send back ACK2 (different type of ACK). Edges where ACK1's were sent form a spanning tree (ACK1 basicly means "you are my parent in the spanning tree"). (Exercise: Prove this.)

In subsequent broadcasts from P, we can use just the tree edges instead of flooding the network (again, assuming no faults). Then the message complexity of each broadcast except the first is reduced to $O(n)$.