In distributed systems we have nodes that can communicate through loosely coupled messages. Nodes have independent (concurrent) activity. They are unreliable, meaning that they can lie, fail etc. In this type of systems there are some problems that cannot be solved.

For keeping a system wide counter that is important for many reasons, simplest solution is to have manager, server, leader or “coordinator”. It is not relevant which node is the server. It is important that in case of server crash, we can pick another node to be the server. In case of network splitting in two or more parts we need to have new server for every part that is created. We need leader election algorithm to decide who the server is. Nodes themselves need to decide who the server is.

**Leader election**

All nodes have the same functionality. There is no specific type that has server function. They have status that can be set to elected or not. If the node is elected, it stays elected until it crashes. At the end of algorithm, all nodes will have status defined and only one will have status elected. Election is not possible if nodes do not have IDs.

*Le Lann, 1977:*

-send value of own ID to the left

- when receiving an id j (from the right):

- if j > ID then forward to the left (we lose)

- if j = ID then elect self (we win)

- if j < ID then do nothing

Leader Election (as shown above) is simpliﬁed, meaning that there is no message loss, all processors start at the same time, no processor fails, no processor cheats, special ring topology.

Nice thing about it is that it is intuitive (filtering the max value), but complexity is not optimal. It has complexity O(n^2), and the best optimized is n\*log(n).

**MANET (ad hock-wireless)**

Intro: Leader election algorithms are still research topic for the papers. It is just theoretical, but people are still trying to improve it. One such example is MANET.

It is designed for wireless networks. When it connects, every node starts campaign for leader with a broadcast message. More than one such message from different nodes can exist. Node that has lowest timestamp or the highest priority wins. Two cheat-proof algorithms are provided. It is applicable for synchronous (exist only in theory, but are easy to handle) and asynchronous distributed systems. All the nodes have the same code running. Algorithm is weakly self-stabilizing.

For this topology we have next setting. Each node has some internal state. Each node has input and output buffer for sending and receiving messages. Channel” is empty (all messages are in queues.

System is described with global state that records every buffer in form of the vector. This kind of snapshot of whole system is called god view and it is not possible in reality, but it is very useful for research.

We have asynchronous message passing where we cannot control when nodes in the network are sending the messages. Node sends the message when it is ready. This approach is model of real thing. On the other hand we have synchronous message passing that is only a theoretical concept. It is platform on which we can much easier develop some innovations.

We have different failure models. Simple crashes where we assume that node doesn’t have any more influence on the system after in fails. In reality it is newer the case. After crash node can stay active and send data that it shouldn’t. This is type of failure is called Byzantine. There is a special case when system can recover from node failures with some algorithm. It is called self –stabilization.

Dykstra said that algorithm is self-stabilizing, if it terminates in a correct state no matter what starting state it initially had. That means that such system would be able to recover from different types of failures and that is really important for autonomic systems. This ability can be translated in self-healing in IBMs approach to autonomic systems.

**Token ring** is an example of simple self-stabilizing system. It is special case of mutual exclusion. Mutual exclusion can guarantee that only one process operates at the same time. In the ring nodes do not have IDs. There is one special node ’0’. Only one node is has the token, meaning that it can send. Token is passed on clockwise to the next node in the ring after the send. This setup allows that there is no need for all nodes to know who has the token (LE). Problem is when token is lost or we have two tokens in the system. There are measures embedded in algorithm that solves such problems.

Node 0: wins if s(0) == s(N-1) Node i: wins if s(i) != s(i-1)

Release token: Release token:

s(0) := (s(0) + 1) mod m s(i) := s(i-1)

This algorithm is self-stabilizing. For any starting state it will always after some time get in correct state of the system if message passing is asynchronous. If all nodes send the messages at the same time that cannot happen. Errors can also be added at any time, and system will again after some time get to correct state. Negative aspect here is that there is no Token Ring algorithm for an anonymous ring without special node 0.

Self-stabilization is different form fault tolerance where we try to mask faults. Here we try to recover from them, not mask and ignore.

**Logical clocks**

Without GPS there is no way to synchronize nodes in distributed system with exact global time. Instead of that we simulate time with message passing events (send or receive). Each node has a local counter that counts events called logical clock. In 1989 Mattern proposed that every node keeps a vector of all logical clocks it learned. In that way we can avoid that if L(a)>L(b) we get a->b. This approach leads to the problems only when we have concurrent communication between 2 sets of nodes. Then we have only partial order in the system. //see slides 11, 12 for explanation.

**Distributed snapshot**

If we cannot have true global state of the system, we can have weaker form of it where global state is combination of local recordings at some time which could have occurred. It doesn’t have to be composed of true local recordings, but it needs to be consistent over the system. We need this global state to be able to recover from crashes of nodes, lost tokens, etc. It is assumed that network is strongly connected and that we don’t have message loss.

To make useful global state, snapshot needs to have some properties that are called consistent cut. Either all events before the cut have to “happened-before” the events after the cut, or the events are unrelated (“concurrent”). This can be checked with vector clocks. Nodes periodically write their current state to disk and in case of a node crash, after restart, the node knows its latest “cut”. Other nodes in the system have to roll back to a maximally consistent cut.

There is one algorithm from Chandy and Lamport that creates one (possible) consistent cut of the system. Assumptions for their algorithm are that we have reliable message exchange, order of messages is FIFO, messages are sent only in one direction over the links, there is no failures (of nodes, links), there is strongly connected graph. Snapshot is initiated by sending a marker M. When M is received, node stops processing, records own state, sends out marker on all links. After that, node record all subsequently received messages (except where M arrived) until a next marker is received on each link. In this way we have made consistent cut together with incoming queue.

// look at the slide 22 for explanation

**Byzantine agreement**

Here we have all types of message failures including malicious messages. In 1982 Fischer, Lynch and Peterson, 1982 have proved that it is not possible to solve the Byzantine problem for asynchronous settings.

If message passing is not reliable two nodes can never be completely sure that information is exchanged correctly. If we have reliable messaging, and only Byzantine problem, we still have a problem. There is no protocol for byzantine agreement among 3 nodes if at least one node fails.

// I think that slide 37 explains it the best

Theoretical results show that protocols for solving Byzantine problem exist, if we have 3f+1 nodes for f failures, but still not for asynchronous model. Oral Message Protocol proposed by Lamport, has an idea that vote is repeated recursively in smaller and smaller groups. This way for a group of 4 nodes there will be two groups of 3 nodes that have right solution and one that has the wrong one. Another idea is that lieutenants exchange their bits and what they heard from whom so that they can at the end discover who is lying. This idea can be applied in many ways. Main thing is to find some way to detect who not to trust.