**Synchronous Distributed System**

A synchronous distributed system comes with **strong guarantees** about properties and nature of the system. Because the system makes strong guarantees, it usually comes with **strong assumptions** and certain **constraints.**

Synchronous nature by itself is multi-faceted, and the following points will elaborate more on this:

***1. Upper Bound on Message Delivery***

There is a **known upper bound** on message transmission delay from one process to another process OR one machine/node to another machine/node. Messages are not expected to be delayed for arbitrary time periods between any given set of participating nodes.

***2. Ordered Message Delivery***

The communication channels between two machines are expected to deliver the messages in FIFO order. It means that the **network will never deliver messages in an arbitrary or random order**that can’t be predicted by the participating processes.

***3. Notion of Globally Synchronized Clocks***

Each node has a local clock, and the clocks of all nodes are always in sync with each other. This makes it trivial to establish a global real time ordering of events not only on a particular node, but also across the nodes.

***4. Lock Step Based Execution***

The participating processes execute in lock-step. An example will make it more clear. Consider a distributed system having a coordinator node that dispatches a message to other follower nodes, and each follower node is expected to process the message once the message is received. It cannot be the case that different follower nodes process the input message independently at different times and thus generate output state at different times. This is why we say processes execute in lock step synchrony a la lock step marching.

The main thing to remember about synchronous systems is that they allow us to make assumptions about time and order of events in a distributed system. This comes from the fact that clocks are in sync and there is a hard upper bound on message transmission delay between nodes.

The problem with synchronous distributed systems is that they are not really practical. Any software system based on strong assumptions tends to be less robust in real world settings and begins to break in practical/common workloads. For example, relying on the network that it is definitely going to deliver the message in a fixed amount of time is not really a practical assumption. In real world, software system is subjected to multiple kinds of failure.

**Asynchronous Distributed System**

The most important thing about an asynchronous distributed system is that it is more suitable for real world scenarios since it does not make any strong assumptions about time and order of events in a distributed system.

***1. Clock may not be accurate, clocks can be out of sync***

Clocks of different nodes in a distributed system can drift apart. Thus it is not at all trivial to reason about the global real time ordering of events across all the machines in the cluster. Machine local timestamps will no longer help here since the clocks are no longer assumed to be always in sync.

***2. Messages can be delayed for arbitrary period of times***

Unlike synchronous distributed system, there is no known upper limit on message transmission delay between nodes.

Asynchronous distributed system is tough to understand since it is not based on strong assumptions and does not really impose any constraints on time and ordering of events. It is also tough to design and implement such a system since the algorithms should tolerate different kinds of failures.

Our algorithms can no longer be designed to handle only a subset of failure conditions by ruling out some failure scenarios using strong assumptions. The onus and challenge of developing robust distributed algorithms is more in asynchronous distributed system.

**Arbitrary Networks**

Assume that we are given an arbitrary processor network. A packet routing problem of size N on this network is defined by a set of N packets each of which has a source and a destination node. The goal is to route each packet from its source to its destination. A routing problem in which every node is the source of h packets and the destination of h packets is called an h-to-h-routing problem, and a routing problem in which every node sends h packets to random destinations chosen independently and uniformly from the set of nodes is called a random h-routing problem.

**Election Algorithms**

**Definition**

* Each process executes the same algorithm
* The algorithm is decentralized (may be > 1 initiator)
* Always terminates in a configuration with a single leader

**Assumptions**

* System is fully asynchronous
* Each process has a unique id (from totally ordered set) of size *w* bits
* Each msg has *O(w)* bits for fair comparisons

The algorithms presented here always elect the initiator with smallest id.

## Korach-Kutten-Moran

**Purpose:** construction of election algorithm given a traversal algorithm *(a traversal algorithm is a centralized algorithm with only one token in motion)*.

**General Idea:** when 2 traversals intersect, one should replace the other.

**Practical Issues:** how to inform one another? how to make this choice consistently? how to avoid mass suicide?

* *when two fronts meet:* i.e. the 2 tokens of 2 traversals meet: simply abandon both and start a new traversal which takes precedence over the 2 previous partial traversals ==> use higher level.
* *when one front meets another traversal in the middle,* there 3 options:
  1. die
  2. wait for another front
  3. chase after the front of this traversal

options 2 and 3 are both about 2 fronts of the same level meeting because that is the only time when we can effectively kill both and replace them with a single new one (of higher level).

**General Description of Algorithm**

* a front always dies when it meets a traversal of higher level (but not when same level, to avoid mass suicide).
* if not yet chased then chase it:
  + either create a new traversal of higher order
  + or die because of traversal of higher order
* if chased then wait:
  + either for traversal of higher order
  + or for other front of same level ==> create higher traversal
  + or for chasing token (if the waiting token has encoutered its own wave again)

**Algorithm:** A token <level,id> can be *annexing* or *chasing*.

**annex <q,l> arrives at p <catp,levp>**

* traversal algorithm terminates (q elected)
* l>levp ==> overwrite and keep traversing
* l<levp ==> killed
* token waiting ==> start new traversal <l+1,p>
* q<catp ==> token waits
* front chased ==> token waits
* q>catp ==> chase front
* else (q=catp) ==> traverse as usual

**chase <q,l> arrives at p <catp,levp>**

* levp>l ==> killed
* token waiting ==> start new traversal <l+1,p>
* front chased ==> token waits
* else keep chasing

In geometry, a hypercube is an n-dimensional analogue of a square (n = 2) and a cube (n = 3). It is a closed, compact, convex figure whose 1-skeleton consists of groups of opposite parallel line segments aligned in each of the space's dimensions, perpendicular to each other and of the same length. A unit hypercube's longest diagonal in n-dimensions is equal to {\sqrt {n}}

An n-dimensional hypercube is also called an n-cube or an n-dimensional cube. The term "measure polytope" is also used, notably in the work of H. S. M. Coxeter (originally from Elte, 1912), but it has now been superseded.

The hypercube is the special case of a hyperrectangle (also called an n-orthotope).