1. **Intro to Distributed Systems (DS)**

* A distributed system is a collection of independent computers which give the user the impression that he is working with one computer.
  + A number of independent computers each consisting of a CPU... I/O
  + Some of the I/O ports are used for communication between computers → communication network. If the computers cannot communicate with each other, this distributed system is not very interesting.
  + The computers share a common state. If state is thought of a range of global invariants, the maintenance of these invariants requires the common/coordinated cooperation of the computers.
  + There is no common physical clock
  + There is no shared memory
  + Geographic separation
  + Autonomy/heterogeneity
* Used for:
  + Load distribution → performance boost (coarsely grained programs)
  + Bridge distances → resource access
  + Increased reliability
  + Scalability
* **Parallel Systems**:
  + Multiprocessor systems (Uniform Memory Access [UMA])
  + Multicomputer parallel system (Non-Uniform Memory Access [NUMA])
    - Normally use private memory, do not share clock
  + Array processors
* **Classification of primitives of DS**:
  + Synchronous (send/receive)
    - Handshake between sender and receiver
  + Asynchronous (send)
    - Control returns after the data got copied to the buffer
  + Blocking (send/receive)
    - Control returns after processing of primitive
  + Non-blocking
    - Returns to process immediately
    - Can check using handle
    - Can check using wait (handle)
* **Challenges** **of creating a DS:**
  + API for communications
  + Data storage and access
  + Syncing
  + Consistency and replication
  + Fault tolerance
  + Transparency – hiding from the user
    - Access, location → Network transparency
    - Migration
    - Replication
    - Concurrency
    - Failure
    - Performance
    - Mobility
    - Scaling
* Classes
  + Socket
  + ServerSocket

**II A Model of Distributed Executions**

A process consists of actions

Actions are atomic and indivisiblt

Actions modeled as three types of events

inverted events

send message event

receive message event

A Model of Distributed Computations

exi → event x on process pi (for messages send(m) and receive(m))

events change the state of a distributed system

internal event changes state of process

send event → process sending message, channel the message is getting through

receive event → process receiving message, channel

executions of processes pi → e1i, e2i, … , eni

history of a process → Hi = (hi, → I)

|->the set of events produced by pi

|-> binary relationship that orders events

send/receive events → flow of info between processes

creates causal dependency between sender and receiver: “→msg” is the binary relationship

space-time diagram (on website)

Causal Precedence

Distributed application → results in a set of distributed events

H=Uihi “binary relationship on History”

picture online

1st case internal event: x preceded y

2nd case event x is sent to event y

3rd case there is a process between x and y that relates x with y

For any two events ei & ej, ei -/> ej means no causal dependency between them

For any two events ei & ej, if ei -/> ej != ej -/> ei

concurrent eventsz

If ei -/> ej and ej -/> ei => ei || ej (concurrently)

Models of Communication Networks

FIFO, CO, and non-FIFO

|-> two messages mi,j & mk,i iff: send(mi,j) → send(mk,i) => rec(mi,j) → rec(mk,i)

CO < FIFO < non-FIFO

1. **Time**

alpha \* Hi(t) + beta = Ci(t)x

drift, drift rate – how far off your clock gets and the rate thereof

we have drift, we need to have some correctness => make assumptions concerning some bound D

monotonicity requirement

time(ticks) = a ticks + b

a = (tx – tscew)/N tx = tr\*(h+N)/h

b = ts – h\*a

2 ways of sync clocks

internal – clocks agree within a bound D

external – clock are accurate within bounds D

**Christian's Method** - external

-algorithm:

| Time| <----| | → asks for time time + Tround/2

|server| ---->|client| → gives time delta = Tround – 2min

best case: +/- Tround/2 – min

**Berkeley Algorithm –** internal

choose a coordinator

polling all the clients (uses Christian's Method)

**Network Time Protocol** – external

hierarchy of servers:

Primary servers (Connected directly to a time source)

Secondary servers (Synchronized with the primaries)

Workstations

three different modes of service

Multicast – Telling others what time it is

Proceedure Call – Using Round Travel Time (RTT), see Christian's Method

Symmetric Mode:

Server A ti-3. ti . Tb = Ta + D

/ \ / \

Client B./ \ ./ \

ti-2 ti-1

Logical Clocks

Something to Physical clocks

Happened before relationship

order of events on a process as observed

sending happens before receiving

happen before relationship is transistional: if e1 -> e2 and e2 -> e3 then e1 -> e3

Lamport's Logical Clocks

counter, only incrementing

processes have their own clocks

timestamp events

Rules: (L() -> Logical of ())

e->e' L(e) < L(e')

the reverse is not true!

Vector Time Stamps

overcome the shortcomings of Lamport's timestamp (L(e) < L(e') =/> e -> e')

vector clock Vi at process Pi ->

array of integers, dim N=num of processes

VC1: initially Vi[j] = 0 for j 1...N

VC2: before Pi timestamps event it sets Vi[i] = Vi[i]++

VC3: Pi piggybacks t=Vi

VC4: when Pi receives a message (m,t) it sets Vi[j] = max(Vi[j],t(j)) then before we set something, add 1 to own element Vi[i]

V = V' <=> Vj:V[j] = V'[j]

V <= V' <=> Vj: V[j] <= V'[j]

V < V' <=> V,=V' and V != V'

V || V' <=> !(V<V') and !(V>V')

**IV. Global States**

Examples and Apps

garbage collection - 1 process does not know when another process has a reference to a local object

deadlock – 2 processes waiting for each other

termination

History

Prefix of History

global history

state

global state

cut c= hc11 union hc22 union hc33 union ... hcnn

**Chandy and Lamport Snapshot Algorithm**

markers receiving rule (on channel c):

if (process recorded state)

(NO)record state of channel c as empty

start observing state on all other channels

execute marker sending rule

(YES)record the state of channels

marker sending rule:

record your state

send out marker message on all outgoing channels

start: any process can start the algorithm by pretending that a marker message was rec'd

Transactions

A: Atomicity

C: Consistency

I: Isolation

D:

For two transaction to be serially equivalent, it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access.

Operation -Conflict- Why:

Read read -No- Because the effect of a pair of read operations does not depend on the order in which they are executed

read write -Yes- Because the effect of a read and a write operation depends on the order of their execution

write write -Yes- Because the effect of a pair of write operations depends on the order of their execution

Recoverability from Absorb

Dirty Read => “Strict Executions”: transactions need to both delay reads and writes on an object until all transactions that previously read or wrote have committed or aborted