Distributed Systems (Events and Properties)

Some Definitions

1. **Safety:** When a safety property is violated it's not possible to enforce it again.
2. **Liveness**: If at time T the property does not hold, there can still be a time T' in the future in which the property is satisfied.

1. Communication links

Fair-Lossy Link

Events:

- < Send | q,m > Request to send message m to process q.
- < Deliver | p,m > Deliver message m sent by process p.

Properties:

1. **Fair-Loss**: If a correct process p infinitely often sends a message m to a correct process q, then q delivers m an infinite number of times.
2. **Finite-Duplication**: If a correct process p sends a message m a finite number of times to process q, then m cannot be delivered an infinite number of times by q.
3. **No Creation**: If some process q delivers a message m with sender p, then m was previously sent to q by process p.

Stubborn Link

Events:

- < Send | q,m > Request to send message m to process q.
- < Deliver | p,m > Deliver message m sent by process p.

Properties:

1. **Stubborn-Delivery**: If a correct process p sends a message m once to correct process q, then q delivers m an infinite number of times.
2. **No Creation**

P2P Link

Events:

- < Send | q,m > Request to send message m to process q.
- < Deliver | p,m > Deliver message m sent by process p.

Properties:

1. **Reliable Delivery**: if a correct process p sends m to q, and q is correct, then process q eventually delivers m.
2. **No duplication**
3. **No Creation**

FIFO P2P Link

Events:

- < Send | q,m > Request to send message m to process q.
- < Deliver | p,m > Deliver message m sent by process p.

Properties:

1. **Reliable Delivery (PL1)**
2. **No Duplication**
3. **No Creation**
4. **FIFO Delivery**: If some process sends message m_1 before it sends message m_2, then no correct process delivers m_2 unless it has already delivered m_1.

2. Relationship

Casual Relationship

Riguarda **l'Happened Before Relationship:**
Two events e and e' are related by Happened-Before relation (e→e') if:

1. Local Ordering: Exists p_i | e→e'
2. Snd-Rcv Ordering: For all m, send(m)→send(m')
3. Transitivity: Exists e'': (e→e'') and (e''→e') then e→e'

Logical Clock (Scalar Logical Clock)

Il logical clock è un software che incrementa un valore in un registro. $L_i(i)$ è il "logical timestamp" assegnato, usando il logical clock, da un processo pi all'evento e.

L'algoritmo **Scalar Logical Clock** utilizza il Logical Clock che ha questi 3 step:

1. Each process p_i initialized its logical clock L_i= 0;
2. Whenever an event occurs at process p_i the logical clock l_p is incremented by one unit
3. When p_i sends a message m:
    a. Creates an event send(m)
    b. Increases L_i
    c. Timestamps m with t=L_i
4. When p_i receives a message m with timestamp t:
    a. Updates its logical clock L_i=max (t, L_i)
    b. Produces an event receive(m)
    c. Increases L_i.

Vector Clock

Algoritmo per il Vector Clock:

1. Each process p_i initializes its clock V_i:
    a. Vi[j]=0 Per ogni j
2. P_i increases V_i[i] by 1 when its generates a new event: Vi[i]= Vi[i]+1
3. When p_i sends a message m:

    a. Creates an event send(m)
    b. Increases V_i
    c. Timestamps m with $t=V_i$
4. When p_i receives a message m containing timestamp V_t:
    a. Updates its logical clock $V_i[j]= \max(V_t[j], V_i[j])$ Per ogni j
    b. Generates an event receive(m)
    c. Increases V_i

Lamport's Mutual Exclusion

Events:

- < Request | m >: From the upper layer. Requests access to Critical Section (CS).
- < Grant | m >: To the upper layer. Grant the access to CS.
- < Release | m >: From the upper layer. Release the CS.

Properties:

1. **Mutual Exclusion**: Any time t, only one process is inside the CS.
2. **Liveness**: If a process p requests access, then it eventually enters the CS.
3. **Fairness**: If the request of process p happens before the request of process q, then q cannot access the CS before p.

Assume that every process is correct.

3. Failure Detectors

Perfect FD

Events:

- < Crash | p > Detects that process p has crashed.

Properties:

1. **Strong Completeness**: Eventually, every process that crashes is permanently detected by every correct process.
2. **Strong Accuracy**: If a process p is detected by any process, then p has crashed.

Eventually Perfect FD

Events:

- < Suspect | p > Notifies that process p is suspected to have crashed.
- < Restore | p > Notifies that process p is not suspected anymore.

Properties:

1. **Strong Completeness**: Eventually, every process that crashes is permanently detected by every correct process.

2. **Eventual Strong Accuracy**: Eventually, no correct process is suspected by any correct process.

4. Leader Election

Synchronous Leader Election

Events:

- < Leader | p > indicates that process p is elected as leader.

Properties:

1. **Eventual Detection**: Either there is no correct process, or some correct process is eventually elected as the leader.
2. **Accuracy**: If a process is a leader, then all previously elected leaders have crashed.

Eventual Leader Election

Events:

- < Trust | p > Indicates that process p is trusted to be leader.

Properties:

1. **Eventual Accuracy**: There is a time after which every correct process trusts some correct process.
2. **Eventual Agreement**: There is a time after which no two correct processes trust different correct processes.

5. Broadcast

BEB (Best Effort Broadcast)

Events:

- < beb,Broadcast | m > Broadcasts a message m to all processes.
- < beb, Deliver | p,m > Delivers a message m broadcast by process p.

Properties:

1. **Validity**: If a correct process broadcasts a message m, then every correct process eventually delivers m
2. **No Duplication**: No message is delivered more than once.
3. **No Creation**: If a process delivers a message m with sender s, then m was previously broadcast by process s.

Il problema del Best Effort Broadcast è che i messaggi inviati dai processi crashati non è sicuro che raggiungano i processi corretti.

Sistema Asincrono, P2P, con crash failure.

Probabilistic Broadcast

Events:

- < Broadcast | m > Broadcasts a message m to all processes.
- < Deliver | p,m > Delivers a message m broadcast by process p.

Properties:

1. **Probabilistic Validity**: There is a positive value ε such that when a correct process broadcasts a message m, the probability that every correct process eventually delivers m is at least 1 - ε.
2. **No Duplication**: No message is delivered more than once.
3. **No Creation**: If a process delivers a message m with sender s, then m was previously broadcast by process s.

REB (Regular Reliable Broadcast)

Events:

- < rb,Broadcast | m > Broadcasts a message m to all processes.
- < rb, Deliver | p,m > Delivers a message m broadcast by process p.

Properties:

1. **Validity**
2. **No Duplication**
3. **No Creation**
4. **Agreement**: If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

Lazy Reliable Broadcast (RB in Fail-Stop)

Costruito su: **Perfect FD** e **BEB**.
E' un'implementazione del REB in Fail-Stop, quindi segue le stesse proprietà del REB.
Il processo si tiene un array in cui salva i messaggi inviati dal processo p_i. Quando rileva un crash il processo invia una seconda volta in broadcast i messaggi inviati dal processo crashato.
Allo stesso modo quando riceve un messaggio da un processo crashato lo manda in broadcast.

Eager Reliable Broadcast (RB in Fail-Silent)

Costruito su: **Best Effort Broadcast, in Fail-Silent (asincrono con crash).**
L'idea utilizzata è:
un processo che vuole inviare un messaggio lo invia in broadcast (con BEB).
Quando i destinatari in broadcast ricevono il messaggio da un processo **inoltrano nuovamente questo messaggio in broadcast** in modo tale da essere sicuri che tutti i processi corretti lo ricevano.

Questo inoltro aggiuntivo è necessario perché è possibile che un processo riceva un messaggio prima che il processo mittente crashi quindi il messaggio non verrà ricevuto da tutti i processi nel sistema.
Quindi l'inoltro ci assicura che tutti i processi consegnino lo stesso set di messaggi (proprietà di Agreement).

URB (Uniform Reliable Broadcast)

Events:

- < Broadcast | m > Broadcasts a message m to all processes.
- < Deliver | p,m > Delivers a message m broadcast by process p.

Properties:

1. **URB1-3**: Same as regular reliable broadcast
4. **Uniform Agreement**: If a message m is delivered by some process (correct or faulty), then m is eventually delivered by every correct process.

URB in Fail-Stop (aka All Ack)

Same as URB but built with PFD.
Quando ricevo un messaggio metto nell'array di ack per quel messaggio il processo sender (ack[m] U {p}) e invio di nuovo in broadcast m. Un handler controlla continuamente se ho ricevuto gli ack da tutti i corretti (correct ⊆ ack[m]), in caso affermativo se non lo ho ancora consegnato lo consegno.

URB in Fail-Silent (aka Majority Ack)

Same as URB but built with Quorum (and no FD).
Come l'All Ack ma il candeliver è implementato con #(ack[m]) > N/2.

6. Ordered Communications

FIFO

Events:

- < frb, Broadcast | m >: Broadcasts a message m to all processes.
- < frb, Deliver | p,m >: Delivers a message m broadcast by process p.

Properties:

1. **FRB1 - FRB4:** Same as properties **RB1-RB4** in (regular) reliable broadcast.
5. **FIFO Delivery**: If some process broadcasts message m_1 before it broadcasts message m_2, then no correct process delivers m_2 unless it has already delivered m_1.

In **Uniform FIFO**:

1. **FIFO Delivery** = If some process p broadcasts a message m before m', then no ~~correct~~ process delivers m' unless it has previously delivered m.

Causal Order Broadcast

Events:

- < crb, Broadcast | m >: Broadcasts a message m to all processes.
- < crb, Deliver | p,m >: Delivers a message m broadcast by process p.

Properties:

1. **CRB1 - CRB4:** Same as properties **RB1-RB4** in (regular) reliable broadcast.
5 **Causal Delivery**: For any message m_1 that potentially caused a message m_2, i.e. $m_1 \rightarrow m_2$ no process delivers m_2 unless it has already delivered m_1.

Causal Order → FIFO Order

In Uniform CO:

1. **UCRB1 - UCRB4:** Same as properties **URB1-URB4** in uniform reliable broadcast (Validity, No Duplication, No Creation, Uniform Agreement).
5 **Causal Delivery**: Come sopra

No-Wait Causal Reliable Broadcast (Fail SIlent)

Qui ogni processo ogni volta che fa il broadcast invia tutta la propria storia a tutti i processi attraverso un array di messaggi (past). Ogni volta che riceve un messaggio non deliverato controlla che tutti i messaggi in past sono stati consegnati, se non consegna prima quelli. Il problema però è che l'array past potrebbe essere troppo grande.

Improved No-Wait CRB (Fail Stop)

Viene preso un messaggio dall'array di Past e viene chiesto a tutti i processi se hanno quel messaggio nell'array di PAST.
Quando il processo riceve gli ack da tutti i processi allora cancella il messaggio dal proprio array di past.

Waiting Causal Reliable Broadcast (Fail Silent)

E' un'idea diversa dalle precedenti. Utilizza il Vector Clock.
Quando riceviamo un messaggio, al messaggio è attaccato un vector clock e se il vector clock che riceviamo è più grande di quello che ci aspettiamo di ricevere allora significa che dobbiamo aspettare altri messaggi prima di consegnare questo messaggio, quindi il messaggio verrà inserito nel set di pending.

Total Order Broadcast

Events:

- < Broadcast | m > Broadcasts a message to all processes.
- < Deliver | p,m > Delivers a message m broadcast by process p.

Properties:

1. **TOB1 - TOB4:** Same as properties **RB1-RB4** in (regular) reliable broadcast.
5. **Total Order**: Let m_1 and m_2 be any two messages and suppose p and q are any two correct processes that deliver m_1 and m_2. If p delivers m_1 before m_2, then q delivers m_1 before m_2.

In Uniform TO:

1. **URB1 - URB4:** Same as properties **URB1-URB4** in uniform reliable broadcast.
5. **Uniform Total Order:** Let m1 and m2 be any two messages and suppose p and q are any two processes that deliver m1 and m2 (whether correct or faulty). If p delivers m1 before m2, then q delivers m1 before m2.

7. Distributed Registers

Regular Register

Events:

- < Read >: Invokes a read operation on the register.
- < Write | v >: Invokes a write operation with value v on the register.
- < ReadReturn | v >: Completes a read operation on the register with return value v.
- < WriteReturn | v >: Completes a write operation on the register.

Properties:

1. **Termination:** If a correct process invokes an operation, then the operation eventually completes.
2. **Validity:** A read that is not concurrent with a write returns the last value written; a read that is concurrent with a write returns the last value written or the value concurrently written.

Read-One-Write-All: Regular Register

Same as Regular Register but built with PFD (Fail-Stop).
Tutti i processi che ricevono il messaggio di scrittura da parte di un writer devono rispondere con un ack dopo aver modificato il proprio valore del registro con il nuovo valore ricevuto. L'operazione di write termina dopo che il writer ha ricevuto gli ack da tutti i processi corretti.

Majority Voting: Regular Register

Same as Regular Register but built with Quorum (and no FD, Fail Silent).
Il writer si tiene un timestamp logico (wts) che aggiorna ad ogni write. Gli altri quando ricevono una write controllano se il wts > proprio ts e mandano gli ack. Ad ogni ack per la write corrente (ts' = wts) il writer aggiorna il count degli ack e controlla se c'è un quorum di ack, se si consegna e svuota l'ack.
Ad ogni read il reader aggiorna il counter dei read (rid+1) e istanzia una readlist vuota. Ad ogni richiesta di read ognuno invia il proprio valore e il rid della richiesta. Ad ogni risposta read si controlla se la readlist ha raggiunto un quorum, se si l'handler prende il valore con il ts più alto, si svuota la readlist e si fa il readreturn.

Sequential Consistency
There exists a global ordering that respects the local ordering seen by each process.

Atomic Register

Events:
- < Read >: Invokes a read operation on the register.
- < Write | v >: Invokes a write operation with value v on the register.
- < ReadReturn | v >: Completes a read operation on the register with return value v.
- < WriteReturn | v >: Completes a write operation on the register.

Properties:
1. **ONAR1-ONAR2:** Same as properties of **Termination** and **Validity** of Regular Register
2. **Ordering:** if a read returns a value v and a subsequent read returns a value w, then the write of w does not precede the write of v.

Read-Impose Write-All (Fail Stop)
Built on: **BEB, PFD, PP2P.**
Ad ogni read si legge in locale e si invia la write in broadcast con il timestamp e il valore. Se il writer decide di scrivere aggiorna il timestamp e fa broadcast. Quando ricevo una read aggiorno il valore solo se il ts è maggiore e poi mando gli ack. Ad ogni consegna degli ack il writer aggiunge il processo a writeset (writeset U {p}). Quando ho ricevuto ack da tutti i corretti faccio il return dell'operazione in corso (if reading).

Read-Impose Write-Majority (Fail Silent)
Built on: **BEB, PP2P.**
È lo stesso algoritmo del Read-Impose Write-All solo che il writer e la fase di Impose aspettano un quorum di ack (> N/2).

8. Consensus

Consensus in Asynchronous System

Possiamo risolvere il consensus in un sistema asincrono solo se il numero di fallimenti nel sistema è 0.

Regular Consensus in Synchronous Systems

Regular Consensus **Non-Uniform:**
Events:

- < Propose | v >: Propose value v for consensus.
- < Decide | v >: Outputs a decided value v of consensus.

Properties:

1. **Termination:** Every correct process eventually decides some value
2. **Validity:** If a process decides v, then v was proposed by some process.
3. **Integrity:** No process decides twice.
4. **Agreement:** No two correct processes decide differently.

Regular Consensus **Uniform:**
Properties:

1. **UC1-UC3:** Same three properties of (regular) consensus.
2. **Uniform Agreement:** No two processes decide differently.

Leader Based Strategy

Stesse proprietà del Regular Consensus in Synchronous Systems.
E' un algoritmo utilizzato per implementare il Consensus.

Utilizza un leader il quale deciderà il valore per tutti i processi e invierà a broadcast il valore deciso a tutti.
Quando il leader crasha, verrà rilevato grazie al PFD, e verrà eletto un nuovo leader.

Il **problema** con questo algoritmo è che se il leader crasha dopo aver inviato in broadcast dei messaggi di "decide" allora è possibile che un processo corretto riceva questo messaggio e decida quel valore. Questo è un problema perchè in questo caso il processo corretto non potrà più decidere nessun altro valore, quindi non tutti i processi corretti avranno lo stesso valore deciso (violata la "Termination").

Per risolvere questo problema si utilizza l'algoritmo di Hierarchical Consensus.

Hierarchical Consensus

Stesse proprietà del Regular Consensus in Synchronous Systems.
Due tipi di algoritmo Hierarchical Consensus:

1. **NON-Uniform:** Questo algoritmo è differente dalla Leader Based Strategy perchè qua ad ogni round viene eletto un nuovo leader dove in quel round

SOLAMENTE il leader deciderà il valore. Quindi a N round tutti i processi del sistema saranno una volta leader e decideranno il valore.

2. **Uniform:** In questo caso anche i processi faulty devono consegnare lo stesso valore. L'idea è uguale allo Hierarchical Consensus "Non-Uniform" solamente che qua quando il leader invia a broadcast il messaggio di "Decide" a tutti i processi, prima di poter consegnare quel valore, deve aspettare l'ack da tutti i processi corretti (detectedranks U ackranks = {1…N}).

Consensus in Eventually Synchronous: Paxos

Events:

- < Prepare | m >: A proposer chooses a new proposal round number n, and sends a 'prepare' request to a majority of acceptors.
- < Accept | n, v >: If the proposer receives responses from a majority of the acceptors, then it can issue an accept request with number n and value v.
- < Decide | v >: A learner that receives (ACCEPT, n, v) from a majority of acceptors, decides v, and sends (DECIDE, v) to all other learners.

Properties:

1. TODO

Phase 1: Con tutti round numbers differenti i proposer inviano le loro proposte tramite le prepare request. Gli acceptor accettano solo richieste con round > attuale e restituiscono info al proposer con una promise response.
Phase 2: I proposer tramite le richieste propose inviano valori agli acceptor che a loro volta inviano ai learner. Se i learner si accorgono che c'è una maggioranza degli acceptor corretti scelgono definitivamente quel valore.

9. Software Replication

Linearization Property

There are three properties that implies the Linearization Property:

1. **Atomicity:** Given an invocation [x op(arg) pi], if one replica of the object x handles this invocation, then every correct replica of x also handles the invocation [x op(arg) pi].
2. **Ordering:** Given two invocations [x op(arg) pi] and x op(arg) pi] if two replicas handle both the invocations, they handle them in the same order.
3. **Blocking operation:** The client does not see the result of an operation until at least one (correct) replica handles it.

Primary Backup

Client sends an operation to an only primary process and this primary process sends the operation to all other processes ("backups").

Synchronous Primary Backup

Implementation of a Primary Backup System with a PFD.

Idea:
Il client decide di effettuare un'operazione e l'attacca ad un oggetto, dove l'operazione ha un timer (il timer server per rendere la richiesta univoca). Invia la richiesta al Primary il quale invia la richiesta a tutti i backup (replica) in modo tale che ogni backup lo esegua.
Il primary invia un'operazione di "Update" a tutti i backup, questa operazione significa che i backup devono appendere alla loro "lista" di operazioni anche l'operazione inviata dal primary.
Poi il primary aspetterà gli ack da tutte le replica per assicurarsi che tutte le replica siano aggiornate. E solo quando tutte le repliche sono aggiornate allora il primary invierà la risposta dal client.

L'unico problema che si può verificare è quando fallisce il processo "Primary".
Il "Primary" può crashare in tre scenari:

1. Scenario 1 (**Fallisce dopo aver inviato i messaggi di update**):
    a. *Il primary fallisce e non è abbastanza veloce per inviare la risposta al server:* in questo caso il client non vedrà la risposta dell'operazione, quindi scaduto il timeout invierà di nuovo la risposta. Ma per evitare che il primary esegua nuovamente l'operazione controlla nella lista delle operazioni se è già stata effettuata.
    b. *Il primary è abbastanza veloce da inviare la risposta*: l'operazione è completata e viene scelto un nuovo primary.
2. Scenario 2 (**Fallisce prima di inviare il messaggio di update a tutti i backup**): il client aspetta che termini il timeout e il client invia nuovamente l'operazione.
3. Scenario 3 (**Fallisce nel mezzo dell'invio degli update a tutti i backup**): è l'unico caso in cui il sistema non soddisfa le proprietà perchè i backup non hanno tutti la stessa operazione.
    a. Per risolvere questo problema si utilizza la **Leader Election** dove verrà scelto come processo Primary il backup che ha la lista delle operazioni più lunga e l'id più piccolo. Una volta che il leader viene eletto si deve assicurare che tutti i backup abbiano le stesse operazioni nella lista.

Raft (Ev. Synch. System)

Implementation of a Primary Backup in a Eventually Synchronous System and with the assumption of majority correct process.

TODO

Active Replication

TODO

10. Byzantine Faults

Authenticated Perfect Link

Events:

- < Send | q, m > Requests to send message m to process q.
- < Deliver | p,m > Delivers message m sent by process q.

Properties:

1. **Reliable delivery**: If a correct process sends a message m to a correct process q, then q eventually delivers m.
2. **No duplication**: No message is delivered by a correct process more than once.
3. **Authenticity**: If some correct process q delivers a message m with sender p and process p is correct, then m was previously sent to q by p.

Byzantine Consistent Broadcast

Events:

- < Broadcast | m > Broadcasts a message m to all processes. Executed only by process s.
- < Deliver | p,m > Delivers a message m broadcast by process p.

Properties:

1. **Validity**
2. **No Duplication**
3. **Integrity**: If some correct process delivers a message m with sender p and process p is correct, then m was previously broadcast by p.
4. **Consistency**: If some correct process delivers a message m and another correct process delivers a message m', then m = m'.

Byzantine Reliable Broadcast

Events:

- < Broadcast | m > Broadcasts a message m to all processes. Executed only by process s.
- < Deliver | p,m > Delivers a message m broadcast by process p.

Properties:

1. **BRB1-BRB4**: Same properties as BCB1-BCB4 (as above)
5. **Totality**: If some message is delivered by any correct process, every correct process eventually delivers a message.