Coordinación y Acuerdo

Ricardo Jiménez Peris, Marta Patiño Martínez

Lsd Distributed Systems Laboratory
Universidad Politécnica de Madrid (UPM)
http://lsd.ls.fi.upm.es/lsd/lsd.htm

La producción de este material ha sido financiada parcialmente por Microsoft Research Cambridge (Award MS-2004-393)

Índice

- Relojes y ordenación de eventos.
- Radiado (*multicast*).
- Consenso y problemas relacionados.
- Exclusión mutua distribuida.

Bibliografía

- Capítulo 11 del Colouris.
- Capítulo 5 del Mullender.
- Capítulos 2 y 7 del Veríssimo.
- M Raynal Distributed Algorithms and Protocols John Wiley 1988.
- Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System, Communications of the ACM, v. 21, n. 7, pp. 558-565, jul., 1978.
- R. Vitenberg, I. Keidar, G. V. Chockler, D. Dolev. Group Communication Specifications: A Comprehensive Study. ACM Computing Surveys. v33, n4, pp. 427-489. Dec. 2001.
- X. Defago, A. Schiper, P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys, v. 36, n. 4, 2004, pp. 372-421.

Bibliografía

- Birman, K. P. and R. Van Renesse, Reliable Distributed Computing with Isis Toolkit, IEEE Press, 1993.
- R. Friedman and R. van Renesse, Strong and Weak Virtual Synchrony in Horus, CS Dep., Cornell Univ., TR95-1537, 1995.
- Moser, L. E. and Y. Amir and Melliar-Smith, P. M. and Agarwal, D. A., Extended Virtual Synchrony, Proc. of the 14th IEEE Conf. on Distributed Computing Systems,pp. 56-65, june, 1994.
- Özalp Babaoğlu, Alberto Bartoli, Gianluca Dini. Enriched View Synchrony: A Programming Paradigm for Partitionable Asynchronous Distributed Systems. IEEE Transactions on Computers. V. 46, n. 6, June 1997. pp. 642 - 658. 1997.
- Jeremy Sussman Idit Keidar Keith Marzullo. Optimistic Virtual Synchrony. Proc. of the 19th IEEE Symp. on Reliable Distributed Systems (SRDS'00). 2000.
- A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *Proc. of IEEE Int. Conf. on Distributed Systems (ICDCS)*, pages 561–568, 1993.

Bibliografía

- Yair Amir, Claudiu Danilov, Michal Miskin-Amir, John Schultz, and Jonathan Stanton. The Spread Toolkit: Architecture and Performance. Johns Hopkins University, Center for Networking and Distributed Systems (CNDS) Technical report CNDS-2004-1.
- Hayden, M., The Ensemble System, TR-98-1662, Department of Computer Science. Cornell University, jan., 1998.
- Luís Rodrigues and Paulo Veríssimo. xAMp: a Multi-primitive Group Communications Service. October 1992. Proceedings of the 11th Symposium on Reliable Distributed Systems, Houston, Texas, USA.
- Ezhilchelvan, P.D., Macêdo, R.A. and Shrivastava, S.K., "Newtop: A Fault-Tolerant Group Communication Protocol". In Proc. of the 15th IEEE Int. Conf. on Distributed Computing Systems (ICDCS '95), 1995, pp. 296-306.
- Moser, L.E. et a., Totem: A Fault-Tolerant Multicast Group Communication System, Communications of the ACM, v. 39, n. 4, pp. 54-63, apr. 1996.
- D.A. Agarwal, L.E. Moser, P. M. Melliar-Smith, R.K. Budhia, The Totem Multiple-Ring Ordering and Topology Maintenance Protocol, Trans. on Computer Systems, v. 16, n. 2, pp. 93-132, 1998.
- D. Dolev and D. Malki, The Transis Approach to High Availability Cluster Communication, Communications of the ACM, v. 39, n. 4, pp. 64-70, 1996.
- *JGroups: A Toolkit for Reliable Multicast Communication*. http://www.jgroups.org.

Bibliografía

- Michael J. Fischer, Nancy A. Lynch and Michael S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," Journal of the ACM, April 1985, 32(2):374-382.
- Leslie Lamport, Robert Shostak, Marshall Pease. The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems (TOPLAS), n. 3, July 1982. pp. 382 - 401. 1982.
- R. Guerraoui. Revisiting the Relationship between Non-Blocking Atomic Commitment and Consensus. In *Proc. of Int. Workshop on Distributed Algorithms (WDAG)*, 1995.

Bibliografía

- AGRAWAL, D. AND ABBADI, A. E. 1992. The Generalized Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data. *ACM Transactions on Database Systems 17,* 4, 689–717.
- AMIR, Y. AND WOOL, A. 1998. Optimal Availability Quorums Systems: Theory and Practice. *Information Processing Letters 65,* 5, 223–228.
- CHEUNG, S. Y., AHAMAD, M., AND AMMAR, M. H. 1990. The grid protocol: a high performance scheme for maintaining replicated data. In *Proc. of Int. Conf. on Data Engineering (ICDE)*. Los Angeles, USA, 438–445.
- KUMAR, A. 1991. Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data. *IEEE Transactions on Computers 40,* 9 (Sept.), 996–1004.
- MAEKAWA, M. 1985. A sqrt(n) Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems 3,* 2, 145–159.
- NAOR, M. AND WOOL, A. 1998. The Load, Capacity, and Availability of Quorum Systems. *SIAM Journal of Computing 27,* 2 (Apr.), 423–447.
- PELEG, D. AND WOOL, A. 1994. Crumbling Walls: A Class of Practical and Efficient Quorum Systems. *Distributed Computing 10,* 2, 87–97.

Bibliografía

- THOMAS, R. H. 1979. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems 4,* 9 (June), 180–209.
- PELEG, D. AND WOOL, A. 1995. The Availability of Quorum Systems. *Information and Computation 123,* 2, pp. 210–223.
- GIFFORD, D. K. 1979. Weighted Voting for Replicated Data. *7th ACM Symp. on Operating Systems*, 150–162.
- ERDOS, P. AND LOVASZ, L. 1975. Problems and results on 3-chromatic hypergraphs and some related questions. In *Infinite and finite sets*. Colloq. Math. Soc. J´anos Bolyai 10, 609–627.
- LOVASZ, L. 1973. Coverings and colorings in hypergraphs. In *Proc. of 4th South-eastern Conf. Combinatorics, Graph Theory and Computing*. 3–12.
- R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. *Are Quorums an Alternative for Data Replication?* ACM Transactions on Database Systems, Vol. 28, N. 3, Sept. 2003, pp. 257-294, ACM Press.

Relojes Físicos y Ordenación de Eventos

- Una posibilidad para ordenar eventos en un sistema distribuido consiste en etiquetarlos con marcas de tiempo *(timestamps)* a partir del reloj local.
- La principal dificultad está en que los relojes no están perfectamente sincronizados y además pueden tener derivas diferentes.
- Un reloj de cuarzo ordinario tiene una deriva de 1 segundo cada 11-12 días. (10^{-6} secs/sec).
- Un reloj de cuarzo de alta precisión tiene una deriva de 10^{-7}-10^{-8} secs/sec.

Relojes Físicos y Ordenación de Eventos

- Hora universal coordinada (UTC):
  - La hora atómica internacional se basa en un reloj atómico muy preciso con derivas de 10^{-13} .
- UTC es el estándar universal para el mantenimiento de la hora.
- Se basa en tiempo atómico, pero se ajusta ocasionalmente con la hora astronómica.
- Se radia a través de emisoras de radio y de satélite (GPS).
- Un ordenador con un receptor adecuado puede sincronizar su reloj con estas señales.

Relojes Físicos y Ordenación de Eventos

- Las señales emitidas por las emisoras de radio tienen precisiones del orden de 0.1-10 ms.
- Las señales GPS tienen una precisión del orden de 1 μs.
- Existen protocolos para la sincronización de relojes con precisiones muy buenas:
  - Network Time Protocol (NTP) da precisiones de decenas de ms en Internet y de 1 ms en LANs.

Relojes Lógicos y Ordenación Causal de Eventos

- En sistemas distribuidos se distingue entre la recepción y entrega de un mensaje.
- La recepción es cuando el mensaje llega físicamente.
- La entrega es cuando el mensaje pasa a la capa de aplicación.
- Entre la recepción y entrega de un mensaje puede haber una diferencia no despreciable de tiempo.
- Estos eventos pueden emplearse para establecer una ordenación causal (lógica) de eventos.

Relojes Lógicos y Ordenación Causal de Eventos

- [Lamport cacm78] propuso la siguiente ordenación causal de eventos:
  - Dos eventos, e1 y e2, secuenciales en un proceso se preceden:

    $e1 \rightarrow e2$
  - La emisión de un mensaje precede a la entrega de éste:

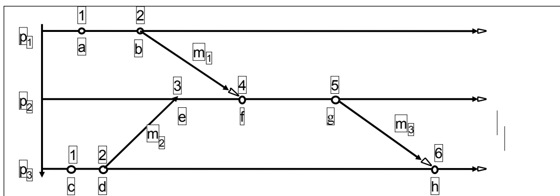
    $send(m) \rightarrow del(m)$
  - La relación causal entre eventos es transitiva:

    $e1 \rightarrow e2 \wedge e2 \rightarrow e3 \Rightarrow e1 \rightarrow e3$
- Se proponen relojes lógicos (en contraposición a los físicos) para etiquetar eventos con una hora lógica (*timestamp*).

Orden causal de eventos: Relojes lógicos

- Cada proceso *p* tiene asociado un *contador o reloj lógico, rl*.
- Inicialmente es 0 en todos los procesos:
  - *rl:= 0.*
- Cada vez que se envía un mensaje desde *p*, se incrementa su contador en 1 y se etiqueta el mensaje con éste:
  - *rl:= rl+1; ts:= rl.*
- Cuando se recibe un mensaje en *p* se obtiene como nuevo reloj lógico el resultante de elegir el máximo entre el reloj lógico local y el *timestamp* del mensaje incrementado en 1:
  - *rl:= max(rl,ts)+1.*

Orden causal de eventos: Relojes lógicos



• Si un evento precede a otro, *ev1* → *ev2*, entonces el primero tiene un reloj lógico menor que el segundo, *rl(ev1) < rl(ev2)*.

• Lo contrario no tiene por qué ser cierto. rl(b) < rl(e), pero no se cumple que b → e Los relojes lógicos no capturan la concurrencia.

• Directamente no permite implementar la entrega causal de mensajes.

Coordinación y Acuerdo

- Propiedades de los problemas de coordinación y acuerdo:
  - **Terminación (*termination*):** Propiedad de viveza, el acuerdo o coordinación, tarde temprano, tiene lugar.
  - **Validez (*validity*):** Las acciones tomadas por procesos correctos tarde o temprano acaban teniendo un efecto sobre el resto de los procesos (p. ej. un mensaje radiado por un proceso correcto acaba entregándose).
  - **Acuerdo (*agreement*):** La decisión tomada por los distintos procesos correctos cumple determinada propiedad (p.ej. todos los procesos toman la misma decisión).
  - **Integridad (*integrity*):** Propiedades para evitar soluciones ilógicas (p.ej. un proceso sólo puede tomar una decisión propuesta por algún otro proceso, sólo se entregan mensajes enviados por algún otro proceso).

Sistemas de comunicación a grupo (SCG)

- Bloque constructivo básico para el desarrollo de sistemas distribuidos tolerantes a fallos (procesos replicados, bases de datos distribuidas y replicadas, *load balancing, ...*).
- Los SCG proporcionan radiado a grupo con distintas calidades y un servicio de *membresía*.
- El servicio de membresía mantiene una lista de los procesos vivos que están conectados al grupo.
- Esta información se entrega a la aplicación siempre que se produce un cambio en la misma y se denomina *vista*.

Radiado (*multicast*)

- Multicast vs. Broadcast.
  - En un multicast se hace un envío a un grupo de procesos.
  - En un broadcast se hace un envío a todos los procesos del sistema.
- Ventajas:
  - Permite un uso eficiente del medio físico de comunicación tipo bus (p.ej. IP multicast sobre Ethernet).
  - El nodo emisor gasta menos recursos.
  - Si se suministran propiedades de fiabilidad y ordenación adecuadas se convierte en un bloque constructivo fundamental para los sistemas distribuidos fiables.

Radiado (*multicast*)

- Tipos de grupos:
  - Cerrados (*closed*): Sólo los miembros de un grupo pueden radiar mensajes al grupo.
  - Abiertos (*open*): Cualquier proceso puede enviar mensajes al grupo.
- Eventos en el radiado de un mensaje:
  - Envío (*sending*).
  - Recepción *(reception)*.
  - Entrega (*delivery*).

Radiado (*multicast*)

- Niveles de fiabilidad:
  - Radiado no fiable (e.g. ip-multicast).
  - Radiado fiable *(reliable)*.
  - Radiado uniforme *(uniform)*.

IP Multicast

- Basado en IP.
- Permite enviar un paquete IP a un conjunto de nodos que conforman un grupo de multicast.
- Membresía dinámica de los grupos.
- Grupos abiertos.
- El multicast se efectúa enviando un datagrama UDP a una dirección de multicast.
- Para recibir mensajes se hace que un socket se una al grupo *(s.joinGroup*(*group*)).
- No hay garantías de fiabilidad ni de ordenación (como con UDP tradicional).

Radiado fiable

- Propiedades:
  - **Integridad:** Un proceso correcto entrega un mensaje *m* como mucho una vez. Además sólo entrega mensajes que han sido radiados previamente.
  - **Validez:** Si un proceso correcto *p* radia un mensaje *m*, *m* será entregado por *p* tarde o temprano.
  - **Acuerdo:** Si un proceso correcto entrega un mensaje *m*, entonces todos los procesos correctos entregarán tarde o temprano *m*.

Radiado uniforme

- La definición de acuerdo anterior sólo hacía referencia a los procesos correctos.
- Si también estamos interesados en qué acciones realizan los procesos no correctos, entonces necesitamos uniformidad:
  - **Acuerdo uniforme:** Si un proceso, correcto o no, entrega un mensaje *m*, entonces todos los procesos correctos entregan, tarde o temprano, *m*.

Tipos de ordenación

- **Sin ordenación**.
- **FIFO:** Mensajes del mismo emisor se entregan en el orden de emisión:

$$send(p,m) \to send(p,m') \Rightarrow del(q,m) \to del(q,m')$$

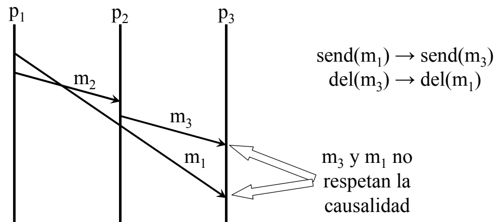
- **Causal:**
  - FIFO
  - Si dos mensajes están relacionados causalmente se entregan respetando el orden causal.

$$send(p,m) \to send(p',m') \Rightarrow del(q,m) \to del(q,m')$$

- **Total:** Los mensajes son entregados en el mismo orden en todos los miembros del grupo.

$$del(p,m) \to del(p,m') \Rightarrow del(q,m) \to del(q,m')$$

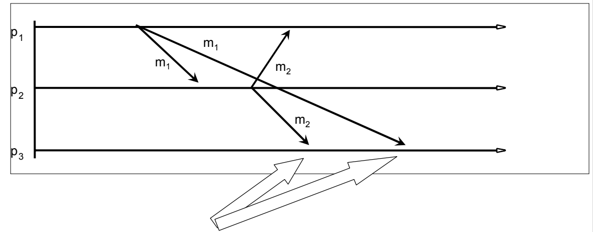
Orden Causal



$send(m_1) \rightarrow send(m_3)$
$del(m_3) \rightarrow del(m_1)$

m_3 y m_1 no respetan la causalidad

Orden Causal



Las entregas de m_1 y m_2 no respetan el orden causal en p_3
$send(m_1) \rightarrow del(m_1) \rightarrow send(m_2)$ y $del(m_2) \rightarrow del(m_1)$

Sistemas de comunicación a grupo: vistas

- Propiedades de seguridad:
  - Si un proceso instala una vista, es miembro de esa vista.
  - Los identificadores de vista son crecientes.
- El servicio de membresía puede ser de *componente primaria* (Isis) o *particionable* (Transis).
- Con componente primaria las vistas instaladas por los procesos están ordenadas totalmente.
- Con una membresía particionable las vistas están parcialmente ordenadas, coexisten varias vistas disjuntas concurrentemente.

Sistemas de comunicación a grupo: Vistas

- En un servicio con componente primaria siempre hay al menos un proceso que transita de una vista a la siguiente.
- Algunos sistemas particionables generalmente informan a los procesos de si están o no en una vista primaria (Horus).
- La propiedad de *sending view delivery* garantiza que los mensajes son entregados en la vista en la que fueron enviados (Isis, Totem).
- Evita que se tenga que enviar con cada mensaje información sobre la vista en la que fue enviado éste.

SCG : Entrega de mensajes

- Para implementar la propiedad de *sending view delivery* , los procesos deben de dejar de enviar mensajes cuando se está realizando un cambio de vista.
- El sistema de comunicación envía a la aplicación un evento de bloqueo.
- Y la aplicación responde con un *flush*, indicando que no enviará más mensajes hasta que se entregue la nueva vista (Horus).

SCG : Entrega de mensajes

- La propiedad *same view delivery* indica que si dos procesos entregan un mensaje, ambos lo hacen en la misma vista (Transis).
- A diferencia de la *sending view delivery,* con la entrega en la misma vista no hay que bloquear a la aplicación durante el cambio de vista .

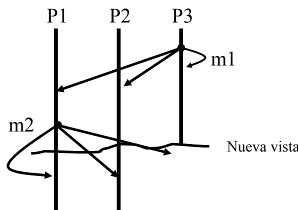
Sincronía Virtual

- La propiedad de sincronía virtual en un SCG garantiza que si dos procesos transitan de una vista, V_k, a la vista siguiente, V_l, ambos procesos han entregado los mismos mensajes en la vista V_k.
- Esta propiedad fue definida en Isis. Todos los SCG proporcionan esta propiedad.
- **Sincronía virtual fuerte (*strong virtual synchrony*):** Aparte de la sincronía virtual los mensajes enviados en una vista, *v,* deben entregarse en dicha vista, *v* (sincronía virtual+*sending view delivery*).

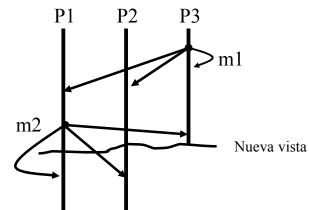
Sincronía Virtual

- La sincronía virtual es especialmente importante por establecer un punto en el que puede transferirse el estado de forma consistente.
- De esta forma se sabe qué mensajes se han entregado en el estado transferido y se garantiza que los mensajes posteriores serán recibidos por los miembros nuevos.
- Existen otras definiciones de sincronía virtual: débil, optimista, extendida, enriquecida, …

Sincronía virtual sin "*sending view delivery*"



Sincronía virtual sin "*same view delivery*"



Sincronía virtual débil

- Se distingue entre vistas regulares y vistas sugeridas.
- Cada vista regular va precedida de al menos una vista sugerida.
- Las vistas sugeridas son un superconjunto ordenado de la siguiente vista regular.
- Los mensajes enviados durante una vista sugerida se entregan en la siguiente vista regular.
- Si algún proceso intenta incorporarse a la vista durante una vista sugerida, no podrá hacerlo hasta después de entregarse la vista regular correspondiente.

Sincronía virtual optimista

- Se entrega una vista optimista antes de cada vista regular.
- La vista optimista coincidirá con la vista regular si no hay más cambios de membresía durante el cambio de vista.
- Si la vista regular no coincide con la optimista, el efecto de los mensajes entregados de forma optimista dependiendo de la aplicación puede conservarse (aunque no todos los procesos de la nueva vista los hayan aplicado) o deshacerse (en cuyo caso habría que reenviar los mensajes).
- Otros tipos de sincronía virtual son la sincronía virtual extendida y la enriquecida.

Algoritmos de Radiado

- Radiado fiable:
  - El emisor radia el mensaje al grupo receptor.
  - El emisor espera la recepción de acks por parte de los miembros del grupo receptor.
  - Si no recibe todos los acks, reenvía periódicamente el mensaje hasta que obtenga todos los acks de los procesos correctos y sospeche de los procesos no correctos.
  - Si el emisor se cae, alguno de los receptores del mensaje, cuando detecte la caída, reenviará el mensaje hasta que sea recibido por todos los receptores correctos.

Algoritmos de Radiado

- Esperar por el ack de un mensaje antes de enviar el siguiente genera latencias muy elevadas.
- Una alternativa es emplear un protocolo de ventana deslizante (*sliding window*).
  - Se envían hasta un número *n* de mensajes sin obtener acks.
  - Cuando se recibe el ack del primer mensaje sin ack se puede desplazar la ventana.
  - La ventana deslizante también sirve de mecanismo de control de flujo.
- El envío de acks en radiado implica el envío de muchos mensajes *(ack implosion)*.
- Se puede evitar empleando *piggybacking, adjuntando los acks* en otros mensajes que se envíen.

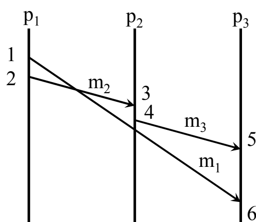
Algoritmos de Radiado

- Orden FIFO:
  - Cada proceso tiene asociado un contador.
  - Los mensajes se etiquetan con el identificador del emisor y el valor actual del contador.
  - Cada vez que se envía un mensaje se incrementa el contador local.
  - Un mensaje es entregado sólo cuando todos los anteriores han sido entregados.

Algoritmos de Radiado: FIFO

- Acks negativos, nacks (*negative acknowledgements*):
  - Cada emisor mantiene un contador de mensajes enviados, que asocia a cada mensaje enviado.
  - Cada receptor mantiene para cada emisor el último contador recibido.
  - Cuando se envía un mensaje se envía el contador del proceso.
  - Cuando se recibe un mensaje, si tiene un contador que es consecutivo con el último recibido de ese proceso, se puede entregar.
  - Si el contador recibido es superior al que tiene ese proceso del emisor, faltan mensajes. Se solicitan al emisor los mensajes perdidos con **nack**.
  - Si el contador es inferior al último recibido, el mensaje está duplicado y se descarta.
  - Si un nodo no envía mensajes, envía un *hearbeat* periódicamente para indicar cuál fue el último mensaje enviado.

Algoritmos de Radiado: Orden Causal



$send(m_1) \rightarrow send(m_3)$
$del(m_3) \rightarrow del(m_1)$

m_3 y m_1 no respetan la causalidad. Los relojes lógicos no proporcionan suficiente información.

Algoritmos de Radiado: Orden Causal

- *reloj vectorial, vc_i*, de un proceso *i* es un vector con tantas componentes como procesos hay en el grupo.
- Los relojes vectoriales se comparan componente a componente. $vc_k \leq vc_j$, si para todo i, $vc_k[i] \leq vc_j[i]$
- $vc_k < vc_j$, si $vc_k \leq vc_j$ y existe i, $vc_k[i] < vc_j[i]$
- Los relojes vectoriales representan la relación de causalidad $m_i \rightarrow m_j$, si $vc(m_i) < vc(m_j)$. Al contrario también es cierto.
- Dos eventos, m_i m_j ,son concurrentes si no se cumple $vc(m_i) \leq vc(m_j)$, ni $vc(m_j) \leq vc(m_i)$

Algoritmos de Radiado: Orden Causal

- Orden causal (relojes vectoriales, *vector clocks*):
  - Cada proceso p_i tiene asociado un *reloj vectorial* vc_i.
  - Inicialmente es vc_i=(0,…,0) en todos los procesos.
  - Cada vez que hay un envío desde p_i, se incrementa la componente *i* de su *reloj vectorial* y se etiqueta el mensaje con dicho reloj vectorial:
    - $vc_i[i]=vc_i[i]+1;\ ts= vc_i$

Algoritmos de Radiado: Orden Causal

- Un mensaje de p_k marcado con *ts* entrega en p_i si

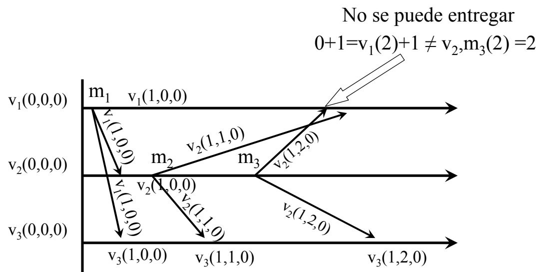
$$ts[k] = vc_i[k] \wedge \forall j \neq k . ts[j] \leq vc_i[j]$$

- *Esto es, si* p_i ha entregado los mensajes anteriores de p_k y los mensajes que p_k hubiese entregado cuando envió este mensaje.
- Cuando se entrega un mensaje en p_k se obtiene como nuevo *timestamp* el resultante de elegir para cada componente el máximo entre vc_k y el timestamp del mensaje, *ts*.

$$\forall i \in \{1..n\} . i \neq k . vc_k[i] := \max(vc_k[i], ts[i])$$

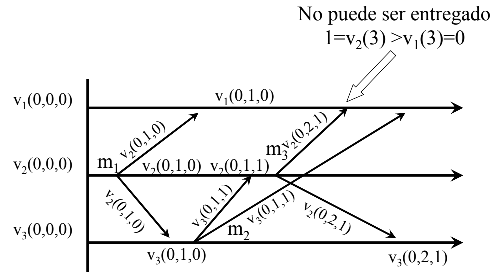
Orden Causal

No se puede entregar
$0+1=v_1(2)+1 \neq v_2,m_3(2) =2$

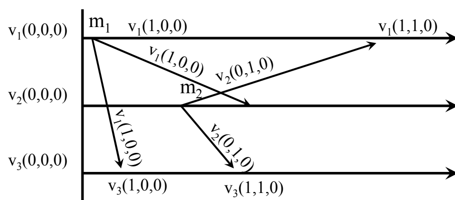


Orden Causal

No puede ser entregado
$1=v_2(3) > v_1(3)=0$



Orden Causal (mensajes concurrentes)



Algoritmos de Radiado: Orden Total

- ¿Dónde se ordenan los mensajes?
  - Emisor:
    - Basado en la historia
    - Basado en privilegios
  - Se emplea un secuenciador (no es necesariamente ni el emisor ni receptor):
    - Fijo
    - Móvil
  - Destino:
    - Acuerdo en el destino

Orden Total: basado en la historia

- Emplea la relación "ocurrió antes" de Lamport para ordenar los mensajes. Se usan relojes lógicos. Dados dos mensajes m_1 y m_2, si $rl(m_1) < rl(m_2)$, m_1 se entregará antes que m_2.
- La relación "ocurrió antes" es un orden parcial. Los emisores tienen un orden. Si los mensajes son concurrentes, se usa el orden de los emisores. En este caso, un mensaje m_1 se entrega antes que m_2, si $sender(m_1) < sender(m_2)$.
- Cuando se envía un mensaje, *m*, se envía su reloj lógico (rl).
- Un receptor entrega *m* cuando sabe que no va a recibir un mensaje con un r*l* menor que el de *m*.

Orden total: basado en la historia

Initialization:

received$_p$:= Ø; { *Messages received by process* p }

delivered$_p$:= Ø ; { *Messages delivered by process* p }

LC$_p$[p$_1$ . . . p$_n$] := {0 , . . . , 0 }

{LCp[q]*: logical clock of process* q*, as seen by process* p }

procedure *TO-multicast*(m) { *To TO-multicast a message* m }

LC$_p$[p] := LC$_p$[p] + 1

ts(m) := LC$_p$[p]

send FIFO (m, ts(m)) to all

Orden total: basado en la historia

when receive (m, ts(m))

LC$_p$[p] := max(ts(m), LC$_p$[p]) + 1

LC$_p$[sender(m)] := ts(m)

received$_p$:= received$_p$ U {m}

deliverable := Ø;

for each message m´ in received$_p$ \ delivered$_p$ **do**

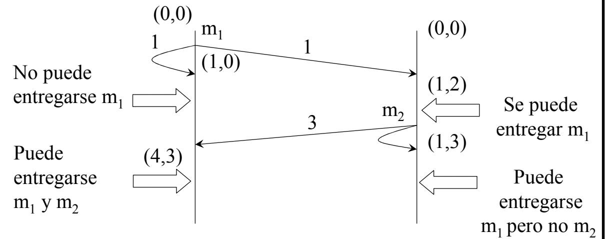
if ts(m´) ≤ min$_{q \in \Pi}$LC$_p$[q] **then**

deliverable := deliverable ∪ {m´}

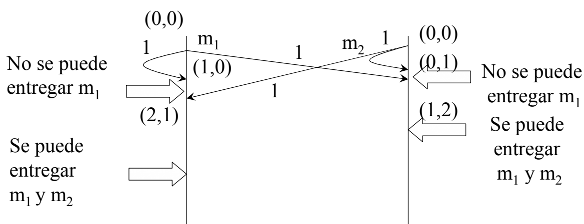
deliver all messages in deliverable, in increasing order of (ts(m), sender(m))

delivered$_p$:= delivered$_p$ ∪ deliverable

Orden total: basado en la historia



Orden total: basado en la historia



Orden Total: basado en la historia

- No hay viveza si algún proceso deja de enviar mensajes. Los algoritmos fuerzan al envío de mensajes.
- NewTop (Universidad de Newcastle. Ezhilchelvan, Shrivastava)

Orden total basado en privilegios

- El emisor sólo puede enviar mensajes cuando se le concede ese privilegio (testigo).
- El testigo circula entre los emisores.
- Hay que asegurar FIFO multicast y que el paso del testigo no viola el orden.
- El testigo lleva el número del siguiente mensaje a enviar.
- Cuando un proceso recibe el testigo usa el número para asignarlo a los mensajes que va a enviar.

Orden total: basado en privilegios

Senders (code of processs$_i$):
Initialization:
  tosend$_{si}$:= Ø ;
  if $s_i = s_1$ **then**
    token.seqnum := 1
    send token to s_1

procedure *TO-broadcast*(m) { *To TO-broadcast a message* m }
  tosend$_{si}$:= tosend$_{si}$ U {m}

when receive token
  for each m´ in tosend$_{si}$ **do**
    send (m´, token.seqnum) to destinations
    token.seqnum := token.seqnum + 1
  tosend$_{si}$:= Ø;
  send token to $s_{i+1 \text{ (mod n)}}$

Orden total: basado en privilegios

Destinations (code of process$_{pi}$):
Initialization:
  nextdeliver$_{pi}$:= 1
  pending$_{pi}$:= Ø;

when receive (m, seqnum)
  pending$_{pi}$:= pending$_{pi}$ U {(m, seqnum)}
  while exists (m´, seqnum´) **in** pending$_{pi}$ s.t.
      seqnum´ = nextdeliver$_{pi}$ **do**
    deliver (m´)
    nextdeliver$_{pi}$:= nextdeliver$_{pi}$ + 1

Orden total basado en privilegios

- Los procesos necesitan conocerse para que circule el testigo. No hay equidad, si un proceso no deja de enviar mensajes mientras tiene el tesigo.
- Totem (Universidad de California en Santa Bárbara, Amir, Moser, Melliar-Smith)

Orden Total: Secuenciador fijo

- Un proceso se elige como secuenciador y es el encargado de ordenar los mensajes.
- Para hacer un multicast de un mensaje, éste se envía al secuenciador.
- El secuenciador le asigna un número de secuencia y lo envía a todos los procesos del grupo.
- Los procesos entregan los mensajes según el número de secuencia.

Orden total: secuenciador fijo

Sender:
  procedure *TO-broadcast*(m) { To *TO-broadcast* a message m }
    send (m) to sequencer

Sequencer:
  Initialization:
    seqnum := 1

  when receive (m)
    sn(m) := seqnum
    send (m, sn(m)) to all
    seqnum := seqnum + 1

Orden total: secuenciador fijo

Destinations (code of process p_i):

Initialization:

$nextdeliver_{pi}$:= 1

$pending_{pi}$:= Ø;

when receive (m, seqnum)

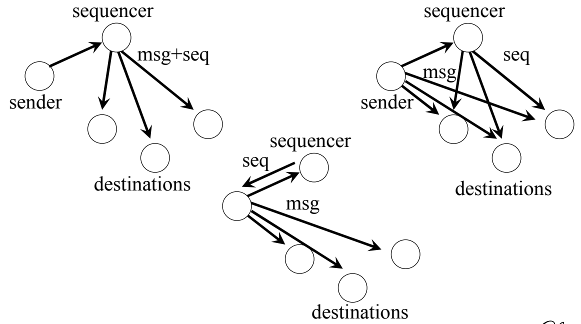
$pending_{pi}$:= $pending_{pi}$ U {(m, seqnum)}

while exists (m´, seqnum´) € $pending_{pi}$: seqnum´ = $nextdeliver_{pi}$ **do**

deliver (m´)

$nextdeliver_{pi}$:= $nextdeliver_{pi}$ + 1

Orden total: Secuenciador Fijo (variantes)



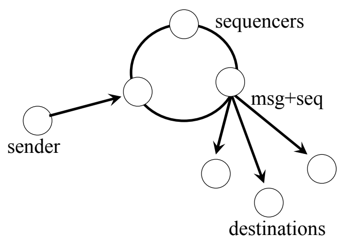
Orden Total: Secuenciador fijo

- El secuenciador es un cuello de botella y punto único de fallo.
- Isis, Phoenix

Orden Total: Secuenciador cambiante

- El secuenciador es un grupo de procesos
- El proceso secuenciador es uno de los miembros de ese grupo. Se hace equilibrado de carga.
- Para radiar un mensaje, el emisor lo radia a los secuenciadores.
- Hay un testigo que circula entre los secuenciadores. El testigo lleva un número de secuencia y la lista de los mensajes que ya tienen número de secuencia.
- Cuando un proceso recibe el testigo, asigna número de secuencia a los mensajes que no lo tienen y los envía.

Orden total: secuenciador cambiante



Orden total: secuenciador cambiante

Sender:

procedure *TO-broadcast*(m) { *To TO-broadcast a message* m }

send (m) to all sequencers

Sequencers (code of process s_i):

Initialization:

$received_{si}$:= Ø;

if $s_i = s_1$ **then**

token.seqnum := 1

token.sequenced := Ø;

send token to s_1

when receive m

$received_{si}$:= $received_{si}$ U {m}

Orden total: secuenciador cambiante

when receive token from s_{i-1}
  for each m´ **in** $received_{si}$ \ token.sequenced **do**
    send (m´, token.seqnum) to destinations
    token.seqnum := token.seqnum + 1
    token.sequenced := token.sequenced U {m0}
  send token to $s_{i+1 \text{ (mod n)}}$

Destinations (code of $process_{pi}$):
 Initialization:
  $nextdeliver_{pi}$:= 1
  $pending_{pi}$:= Ø;

when receive (m, seqnum)
 $pending_{pi}$:= $pending_{pi}$ U {(m, seqnum)}
 while exists (m´, seqnum´) **in** $pending_{pi}$ s.t. seqnum´ = $nextdeliver_{pi}$ **do**
  deliver (m´)
  $nextdeliver_{pi}$:= $nextdeliver_{pi}$ + 1

Orden Total: Secuenciador cambiante

- Estos algoritmos se pueden usar con grupos abiertos.
- El orden total es construido por el secuenciador, a diferencia de los algoritmos basados en privilegios, donde lo construye el emisor.
- En los algoritmos basados en privilegios, el paso de testigo es necesario para asegurar la viveza del algoritmo.

Orden total: acuerdo en el destino

- Los destinatarios reciben los mensajes sin indicación de orden, intercambian información para definir el orden (consenso).
- Cuando un proceso recibe un mensaje le asigna una marca de tiempo y la envía al resto de los procesos.
- Cuando un proceso ha recibido las marcas de tiempo de todos los procesos, le asigna una marca de tiempo global, la máxima marca de tiempo.
- El mensaje se puede entregar cuando tiene una marca de tiempo global y ningún otro mensaje sin entregar puede recibir una marca global inferior.

Orden total: acuerdo en destino

Sender:
 procedure *TO-broadcast*(m) { *To TO-broadcast a message* m }
  send (m) to destinations

Destinations (code of $process_{pi}$):
Initialization:
 $stamped_{pi}$:= Ø;
 $received_{pi}$:= Ø;
 LC_{pi} := 0 { LC_{pi} *: logical clock of process* pi }

when receive m
 ts_i (m) := LC_{pi}
 $received_{pi}$:= $received_{pi}$ U {(m, ts_i (m))}
 send (m, ts_i (m)) to destinations
 LC_{pi} := LC_{pi} + 1

Orden total: acuerdo en destino

when receive (m, ts_j (m)) from p_j

 LC_{pi} := max(ts_j , LC_{pi})+ 1
 if received (m, ts(m)) from all destinations **then**
  sn(m) := $\max_{k=1\cdots n} ts_k$ (m)
  $stamped_{pi}$:= $stamped_{pi}$ U {(m, sn(m))}
  $received_{pi}$:= $received_{pi}$ \ {m}
  deliverable := Ø;
  for each (m´, sn(m´)) **in** $stamped_{pi}$ s.t.
    forall m´´ **in** $received_{pi}$: sn(m´) < ts_i (m´´) **do**
   deliverable := deliverable U {(m´, sn(m´))}
  deliver all messages in deliverable in increasing order of (sn(m), sender(m))
  $stamped_{pi}$:= $stamped_{pi}$ \ deliverable

Algoritmos de Radiado

- Radiado uniforme:
  - El principio es el mismo que en el radiado fiable.
  - La entrega se retrasa hasta que se sabe que el mensaje ha sido recibido por todos los receptores correctos.
  - Esto último se puede conseguir:
    - Enviando los *acks* al emisor y cuando éste los reúne todos radia un mensaje indicando que el mensaje es **estable**.
    - Radiando los *acks* a todos los miembros del grupo.
    - Se crea un anillo lógico en los receptores en el que circula un testigo. En la primera vuelta se envía el mensaje y en la segunda se indica la estabilidad de éste.

Grupos no Disjuntos

- Las garantías de orden afectan a los miembros de un grupo. Si un proceso pertenece a dos o más grupos hay que redefinir las garantías de orden de los mensajes.
- El *radiado atómico* garantiza que si dos procesos, p y q, pertenecen a dos grupos, g1 y g2, y un mensaje m1 se envía a g1 y otro mensaje m2 se envía a g2, y p entrega m1 y después m2, entonces q entregará los mensajes en ese orden [Guerraoui&Schiper97-wdag].
- El radiado atómico es caro, hay que enviar mensajes a procesos que no pertenecen al grupo al que se envió el mensaje.
- El orden fifo y el causal también pueden definirse para grupos no disjuntos. Birman99 demuestra que el orden causal también es caro.

Grupos no Disjuntos

- Isis no garantiza este tipo de radiado atómico.
- En Transis y Totem los mensajes se envían a demonios (grupo pesado) para ordenarlos totalmente. Los demonios envían estos mensajes a los miembros del grupo (ligero) en el que se envió el mensaje.
- Horus y Ensemble pueden suministrar radiado atómico usando distintas pilas.
  - Si se suministra radiado atómico se coloca una capa de grupos ligeros encima de la capa de orden total para multiplexar los mensajes a distintos grupos.
  - Si no hay radiado atómico, la capa de grupos ligeros se coloca debajo de la capa de orden total y los mensajes son ordenados en los grupos destinatarios.

Sistemas de Comunicación a Grupo

- Transis (Technion). Soporta radiado uniforme con orden total y causal.
- Tótem (Santa Barbara). Soporta radiado uniforme con orden total y causal (algoritmos basados en paso de token).
- Ensemble (evolución de Horus, que a su vez evoluciona de Isis, Cornell). Pila de protocolos configurable.
- JavaGroups (Cornell). Pila de protocolos configurable.
- Spread (John Hopkins). Biblioteca con primitivas de radiado con soporte para redes de área local y extendida.
- SpinGlass (Cornell). Radiado probabilista para sistemas *peer-to-peer*.

Problemas Relacionados

- **Consenso:** Todos los procesos proponen valores y tiene que decidirse uno de los valores propuestos.
- **Compromiso atómico:** Cada proceso vota sí o no. Si todos votan sí, se decide comprometer, en otro caso se decide abortar.
- **Compromiso atómico no bloqueante:** Ídem, pero con la propiedad de viveza, de que tarde o temprano se decide comprometer o abortar.
- **Elección de líder:** Se tiene que elegir un líder que sea el mismo para todos los procesos.

Consenso

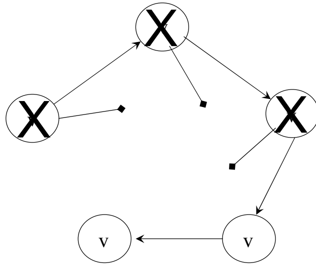
- **Terminación:** Todo proceso correcto decide tarde o temprano algún valor.
- **Validez:** Si todos los procesos proponen el mismo valor *v*, todos los procesos correctos deciden tarde o temprano *v*.
- **Acuerdo:** Si un proceso correcto decide *v*, entonces todos los procesos correctos deciden tarde o temprano *v*.
- **Integridad:** Todo proceso correcto decide como mucho un valor, y el valor decidido debe haber sido propuesto por algún proceso.

Consenso

- En un sistema síncrono con N procesos, donde como mucho hay f fallos de tipo *crash*.
- Hacen falta $f+1$ rondas para implementar el consenso [Dolev&Strong83]. Una ronda dura el tiempo necesario para radiar un mensaje.
- Cada proceso radia su voto sin ningún tipo de garantía.

$valores_1 = \{v_i\}$ $valores_0 = \{\}$
En la ronda r, 1 <=r <=f+1
  multicast($valores_r - valores_{r-1}$)
  $valores_{r+1} = valores_r$
  mientras en la ronda r
    cuando se entregue V_j
      $valores_{r+1} = valores_{r+1} \cup V_j$
Después de f + 1 rondas
  decisión = mínimo($valores_{f+1}$)

Consenso: ¿Por qué hacen falta f+1 rondas?



Consenso

- Ningún algoritmo determinista resuelve el consenso en un sistema asíncrono en la presencia de uno o más fallos [FLP85].
- Formas de superar la imposibilidad:
  - Algoritmos probabilistas (*randomized algorithms*): Permiten alcanzar el consenso en un tiempo máximo.
  - Sistemas parcialmente síncronos: El tiempo de transmisión de mensajes está acotado, pero no se conoce la cota.
  - Detectores de fallos [Chandra&Toueg96].

Consenso. Detectores de fallos.

- El resultado debe satisfacer dos propiedades: completitud (*completness*) y precisión (*accuracy*).

- Completitud: sospechas correctas.
- Precisión: sospechas erróneas.

Consenso. Detectores de fallos.

- Completitud:
  - Fuerte: todos los procesos caídos son sospechados por permanentemente por **todos** los procesos correctos.
  - Débil: todos los procesos caídos son sospechados por permanentemente por **algún** proceso correcto.
- Precisión:
  - Fuerte: No se sospecha de un proceso antes de que se caiga.
  - Débil: de algún proceso correcto no se sospecha nunca.
  - Tarde o temprano fuerte: hay un tiempo a partir del cual los procesos correctos no son sospechados por ningún proceso correcto.
  - Tarde o temprano débil: hay un tiempo a partir del cual **algún** proceso correcto no es sospechado por ningún proceso correcto.

Consenso. Detectores de fallos.

Accuracy / *Completeness*	*Strong*	*Weak*	*Eventually Strong*	*Eventually Weak*
Strong	*Perfect* P	*Strong* S	*Eventually Perfect* ◊ P	*Eventually Strong* ◊ S
Weak	Q	*Weak* W	◊ Q	*Eventually Weak* ◊ W

Consenso

- Con un detector no fiable se puede resolver el consenso si se produce el fallo de menos de N/2 procesos y la comunicación es fiable.
- El detector de fallos más débil es *eventually weak*. Al menos un proceso correcto no es sospechado por nadie (precisión), cada proceso incorrecto es sospechado por algún proceso correcto (completitud).
- Chandra y Toueg demuestran que este tipo de detector de fallos no puede ser implementado en un sistema asíncrono basándose únicamente en paso de mensajes.

El problema de los generales bizantinos

- Tres o más generales tienen que decidir atacar o no.
- Uno de los generales (comandante) y da la orden al resto de los generales (lugartenientes).
- Éste puede engañar, decir una cosa a un general y otra a otro. Si el general es correcto, todos los procesos deciden el valor propuesto por éste.
- Un general puede decir a otro que el general le ordenó atacar y a otro decirle que no.
- A diferencia del consenso, un proceso propone un valor y los demás deben decidir si están de acuerdo o no en ese valor.

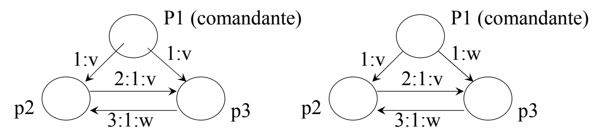
El problema de los generales Bizantinos

- Terminación: Todos los procesos correctos deciden tarde o temprano un valor.
- Acuerdo: Todos los procesos correctos deciden el mismo valor.
- Integridad: Si el proceso que propone el valor es correcto, todos los procesos deciden el valor propuesto por éste.

Acuerdo con fallos Bizantinos en sistemas asíncronos

- Los procesos tienen fallos arbitrarios: pueden enviar mensajes con cualquier valor, pueden no enviar mensajes, pueden enviar mensajes en cualquier momento.
- Suponemos que los canales entre dos procesos son privados (otro proceso no ve lo que se envía por él).
- Para tener acuerdo en este tipo de sistema el número de procesos, N, tiene que ser $N \geq 3f+1$, siendo f el número de fallos [Lamport82].
- El número de rondas mínimo de cualquier algoritmo es f+1.

Acuerdo con fallos Bizantinos en sistemas asíncronos



- Si hay firmas digitales, puede haber acuerdo con tres procesos y un fallo.

Compromiso Atómico

- **Validez Uniforme:** Una transacción es comprometida sólo si todos los procesos votaron *sí*.
- **Acuerdo Uniforme**: No hay un par de participantes que tomen una decisión diferente.
- **No trivialidad:** Si todos los procesos votaron *sí* y no hay fallos ni sospechas falsas la decisión tiene que ser *commit*.

Elección de Líder

- Algoritmo en el que se elige un único proceso para adoptar un papel particular dentro de un protocolo (p.ej. secuenciador para el radiado ordenado totalmente).
- La elección de líder es iniciada a petición de algún proceso.
- Si se inician concurrentemente elecciones por parte de distintos procesos, sólo un líder debe ser elegido.
- Se elige el proceso con mayor identificador.
- Seguridad: un proceso participante tiene como valor de líder el identificador de un proceso vivo con el mayor identificador o un valor indefinido
- Viveza: todos los procesos eligen un líder o se caen.

Elección de Líder: basado en anillo

- Sistema asíncrono [Chang&Roberts79]. Cada proceso tiene un canal con el siguiente proceso en el anillo. Los mensajes circulan en sentido de las agujas del reloj.
- El proceso que inicia el algoritmo se marca como participante y envía su identificador en un mensaje de *elección* a su vecino.
- Cuando un proceso recibe un mensaje de elección compara el identificador recibido con el suyo.
  - Si es menor el recibido y el proceso no es un participante, sustituye el identificador en el mensaje por el suyo y lo reenvía al vecino y se marca como participante.

Elección de Líder: basado en anillo

- Si es mayor el recibido, reenvía el mensaje y se marca como participante.
- Si es menor el recibido y el proceso es un participante, no hace nada (no envía ningún mensaje).
- Si el identificador coincide con el del proceso, ese proceso es el líder.

- El líder se marca como no participante y envía un mensaje *elegido* al siguiente proceso.
- Cuando un proceso distinto al líder recibe este mensaje, anota qué proceso es el líder y reenvía el mensaje.

Elección de Líder: basado en anillo

- El estado participante no participante sirve para eliminar elecciones concurrentes lo antes posible.
- El peor caso es cuando el proceso con mayor identificador es el anterior al que inicia la elección de líder. 3n-1 mensajes.
  - Hacen falta n-1 mensajes para alcanzar a ese proceso.
  - n mensajes para transmitir el identificador de ese proceso.
  - n mensajes de elegido.
- No tolera fallos de ningún proceso.

Elección de Líder: algoritmo del dictador

- Permite caídas de los procesos, supone canales fiables. Sistema síncrono [García-Molina 82].
- Supone que cada proceso conoce a los procesos con identificadores mayores y que se puede comunicar con esos procesos.
- Hay tres tipos de mensajes:
  - *Elección*, para anunciar una elección.
  - *Respuesta*, enviado en respuesta a un mensaje de elección.
  - *Coordinador*, anuncia la identidad del proceso líder.

Elección de Líder: algoritmo del dictador

- Un proceso inicia el algoritmo cuando sospecha (por medio de *timeouts*) que el líder se ha caído.
- El proceso que sabe que tiene el mayor identificador puede enviar un mensaje de coordinador a todos los procesos con identificador menor para indicar que es el nuevo líder.
- Los procesos con menor identificador inician el algoritmo enviando un mensaje de elección a los procesos con mayor identificador y esperan un mensaje de respuesta.

Elección de Líder: algoritmo del dictador

- Cuando a un proceso le llega un mensaje de elección envía una respuesta e inicia una elección, a no ser que ya la haya iniciado.
- Si a un proceso no le llega ningún mensaje de respuesta, se considera el nuevo coordinador y envía un mensaje de coordinador a los procesos con menor identificador.
- Si le llegan las respuestas, espera a que le llegue un mensaje de coordinador. Si éste no llega, inicia el algoritmo de nuevo.
- Cuando un proceso recibe un mensaje de coordinador anota la identidad del nuevo líder.

Elección de Líder: algoritmo del dictador

- El algoritmo se llama así porque cuando un proceso reemplaza a un proceso caído, si tiene el mayor identificador, se impondrá como líder aunque el líder esté vivo.
- La propiedad de seguridad no se cumple si los procesos caídos se reemplazan con procesos con la misma identidad.
- El nuevo proceso y uno vivo enviarán mensajes de coordinador concurrentemente. Lo mismo ocurre si los timeouts no son precisos.
- En el peor de los casos hacen falta $O(n^2)$ mensajes, el proceso con menor identificador detecta el fallo del líder y n-1 procesos inician la elección.

Exclusión Mutua Distribuida

- Cada proceso ejecuta dos bloques de instrucciones, la sección crítica y la sección no crítica.
- Dentro de la sección crítica sólo puede estar como mucho un proceso.
- Propiedades:
  - Seguridad *(safety)*: Como mucho un proceso está dentro de la sección crítica.
  - Viveza *(liveness)*:
    - Un proceso que desea entrar o salir de la sección crítica tarde o temprano lo consigue.

Exclusión Mutua Distribuida: Servidor central

- Un proceso es el encargado de conceder permiso para entrar en la sección crítica.
- Para tolerar fallos es necesario implementar también un algoritmo de elección de líder.
- El servidor se convierte en un cuello de botella.
- Número de mensajes:
  - Uno para la petición.
  - Uno para la concesión.
  - Uno para liberar la exclusión mutua.

Exclusión Mutua Distribuida: Anillo

- Los procesos se organizan en un anillo lógico. Un testigo circula a través de éste.
- El proceso en posesión del testigo puede entrar en la sección crítica.
- Para tolerar fallos es necesario implementar un algoritmo para detectar pérdidas del testigo y regenerarlo.
- Latencia: entre 0 y n (acaba de pasar el testigo)
- Los procesos están enviando mensajes aunque no quieran entrar en la sección crítica.

Exclusión Mutua Distribuida: Algoritmo de Ricart y Agrawala

- Cada proceso mantiene un *timestamp* (reloj lógico).
- Los mensajes de petición de entrada van etiquetados con el *timestamp* y el identificador de proceso.
- Cada proceso tiene como estado:
  - *Released*, si está fuera de la región crítica.
  - *Wanted* si ha pedido acceso.
  - *Held* si está dentro de la región crítica.
- Un proceso para entrar en la región crítica actualiza su estado a *wanted* y radia a los demás un mensaje de petición de entrada.
- Cuando el proceso reciba permiso de todos los demás procesos (el estado es *released* en todos los procesos) podrá entrar en la región crítica.

Exclusión Mutua Distribuida: Algoritmo de Ricart y Agrawala

- Si un proceso, *p,* solicita entrar en la región crítica y ningún otro la está ocupando, todos los procesos le darán permiso para que entre.
- Si un proceso, *q,* está dentro de la exclusión mutua, no contestará. Cuando la libere concederá permiso a *p* y éste podrá entrar en la región crítica.
- Si dos o más procesos solicitan la exclusión mutua concurrentemente, el que tenga menor *timestamp* podrá entrar en la región crítica, no contesta al otro solicitante.
- A igualdad de *timestamps*, se le concederá la entrada al de menor identificador de proceso. El otro proceso solicitante concederá permiso al comparar los *timestamps*.
- Para entrar hacen falta 2(n-1) mensajes si no hay radiado y n mensajes, si hay radiado disponible.

Sistemas de Quórums

- La entrada en la región crítica en el algoritmo anterior requiere obtener permiso de todos los demás procesos, lo que disminuye su escalabilidad.
- Una alternativa es emplear sistemas basados en quórums.
- Un sistema de quórums, *sq*, se define sobre el conjunto de nodos del sistema n*s={1,...,n}* (universal) como:

$$sq \subset 2^{ns} \mid \forall q_1, q_2 \in sq . q_1 \neq q_2 \wedge q_1 \cap q_2 \neq \varnothing$$

Sistemas de Quórums

- También se pueden extender los sistemas de quórums para exclusión mutua lectores/escritores, esto es, lecturas compartidas y escrituras exclusivas.
- Un sistema de quórums de lectores/escritores, *sq*, se define como:

$$sq = (sq_r, sq_w) \mid sq_r, sq_w \subset 2^{ns}$$
$$\wedge \forall q_{w1}, q_{w2} \in sq_w . q_{w1} \neq q_{w2} \wedge q_{w1} \cap q_{w2} \neq \varnothing$$
$$\wedge \forall q_r \in sq_r . \forall q_w \in sq_w . q_r \cap q_w \neq \varnothing$$

Sistemas de Quórums

- Se definen las siguientes propiedades:
  - **Grado:**
    $$i \in ns$$
    $$grado(i) = \mid \{q \in sq \mid i \in q\} \mid$$
  - **Uniformidad:**
    $$\forall q \in sq . \mid q \mid = s \Rightarrow sq \text{ es } s-uniforme$$
  - **Equidad:**
    $$\forall i \in ns . grado(i) = g \wedge sq \text{ es } s-uniforme$$
    $$\Rightarrow sq \text{ es } (s,g)-equitativo$$

Exclusión Mutua Distribuida: Quórums

- El algoritmo de Maekawa [Maekawa85] se basa en un sistema de quórums que es (s, g)-equitativo y que tiene tantos quórums como procesos.
- Cada proceso está asociado a un quorum que lo contiene.
- El sistema de quórums es √n-uniforme.
- Para entrar en la exclusión mutua el proceso radia a su quórum la petición de entrada.
- Cuando obtiene permiso de todos los procesos en su quórum puede entrar en la región crítica.

Exclusión Mutua Distribuida: Algoritmo de Maekawa

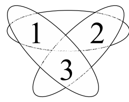
- Las peticiones tienen una prioridad que está determinada por un número de secuencia y el identificador del nodo.
- Cuando un proceso recibe una *petición* (*request*) de un nodo *i*, la almacena en una cola ordenada por prioridad.
- Si no ha concedido la exclusión a otro, envía un mensaje para *conceder* (*locked*) la exclusión mutua.
- Si ya la ha concedido, comprueba si *i* tiene menos prioridad que el proceso al que se la ha concedido, si es así, contesta a *i* con un mensaje de *fallo* (*failed*).
- Si *i* tiene más prioridad, se envía un mensaje al proceso al que se le dio permiso *preguntando* (*inquire*) si ha obtenido permiso del resto de los procesos.

Exclusión Mutua Distribuida: Algoritmo de Maekawa

- Cuando un proceso recibe un mensaje preguntando contesta con un mensaje *liberar* (*relinquish*), si ha recibido algún mensaje de fallo.
- Cuando un proceso recibe un mensaje liberar (*relinquish*), almacena en la cola la petición a la que había concedido la exclusión mutua y se la concede al proceso de mayor prioridad de la cola (envía un mensaje conceder)
- Para entrar en la región crítica hacen falta 2√n mensajes. √n para solicitar la exclusión mutua y el mismo número de respuestas.
- Para salir de la región crítica hacen falta √n mensajes.

Sistemas de Quórums

- **Mayoría (majority o quorum consensus).**
  - Este sistema de quórums tiene como quórums todos los subconjuntos del universal de tamaño n div 2 + 1.
  - Existe también la mayoría ponderada (weighted voting).

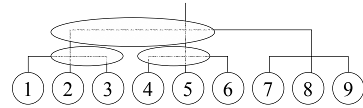


- **1 2**
- **2 3**
- **1 3**

Es (n div 2 + 1, n div 2 + 1)-equitativo

Sistemas de Quórums

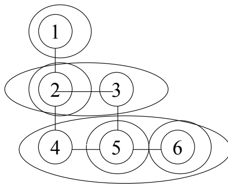
- **Mayoría Jerárquica (hierarchical quorum consensus)**.
  - Se toma un número de nodos de tamaño h^3. Estos nodos se dividen en tres grupos de igual tamaño.
  - Esto se hace recursivamente hasta realizar *h* subdivisiones.
  - Un quórum se forma recursivamente escogiendo una mayoría grupos de primer nivel, una mayoría de grupos de segundo nivel de cada uno de estos grupos y así.



2 3 4 5

Sistemas de Quórums

- **Triángulo *(triangle).***
  - Los elementos del universal se organizan en filas de tamaño 1, 2, 3, …
  - Un quórum está formado por una fila completa y un representante de cada una de las filas inferiores restantes.



- 2 3 5
- 1 2 6
- 4 5 6

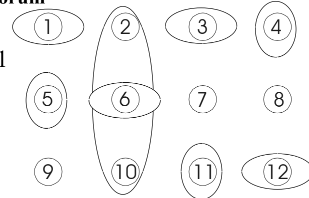
¿Qué tamaño tienen los quórums? f

Sistemas de Quorums

- **Rejilla rectangular *(rectangular grid)*.**
  - Los elementos del universal se organizan en una rejilla rectangular de *f* filas y *c* columnas.
  - Un quórum está formado por una fila completa y un representante de cada una de las filas restantes.

Write Quorum

2 4 5 6 10 11



Read Quorum

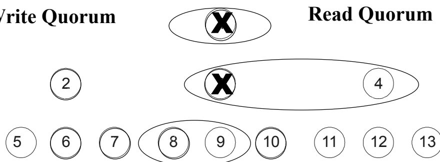
1 3 6 12

Sistemas de Quórums

- **Arborescente.**
  - Los elementos del universal se organizan en un árbol de grado impar (>= 3).
  - Los quórums se definen recursivamente del siguiente modo:
    - La raíz forma parte del quórum.
    - Una mayoría de sus descendientes pertenece al quórum y así sucesivamente hasta las hojas.

Write Quorum

Read Quorum



Sistemas de Quórums

- Existe un compromiso entre el grado y la uniformidad:
  - A menor grado, mayor uniformidad y viceversa.
  - La equidad es óptima cuando el grado y la uniformidad son raíz de *n*.
- También existe un compromiso entre la fiabilidad del sistema de quórums y la uniformidad.
  - Cuanto menor sea el tamaño del quórum, menor fiabilidad *($p > 0.5$)*.
- Si la fiabilidad de los nodos es $p < 0.5$, resulta más fiable un sistema centralizado (monarquía).
- Si la fiabilidad de los nodos es $p > 0.5$, el sistema más fiable es el de mayoría.

Fiabilidad de los Quórums

$p = 0.9$