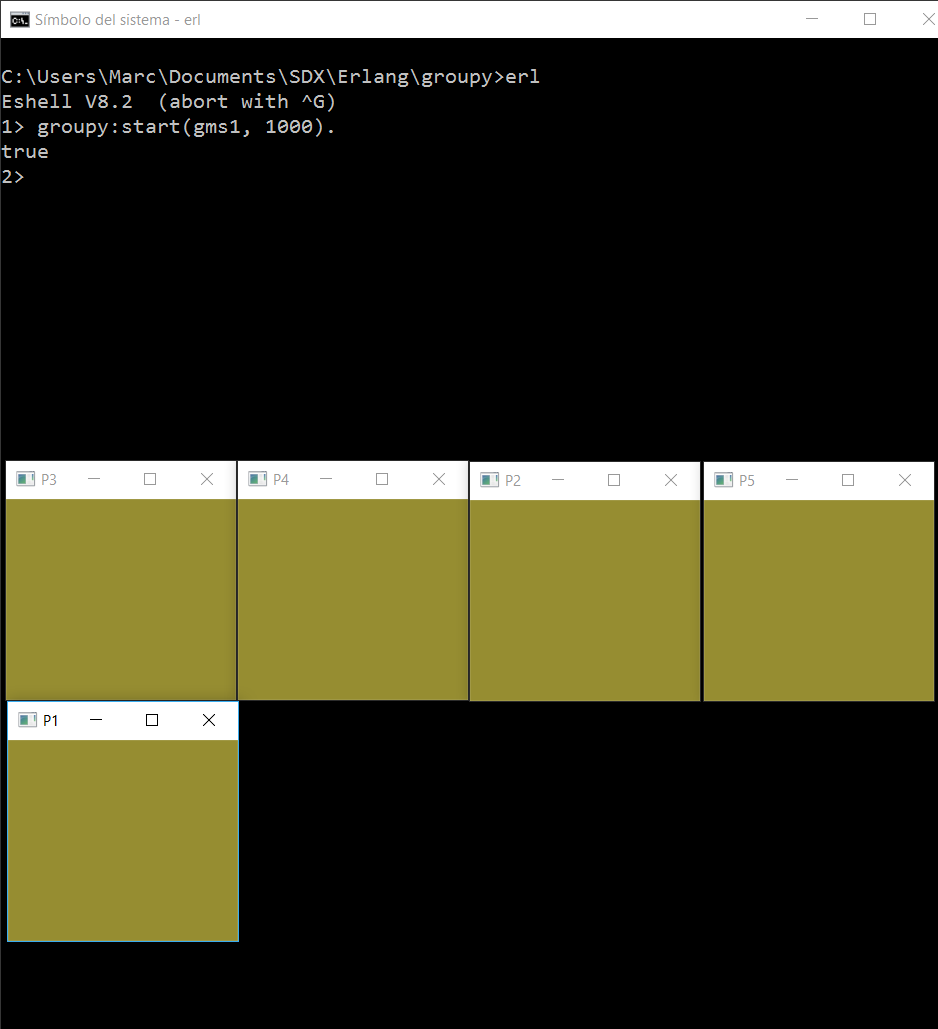
**2- The first implementation**

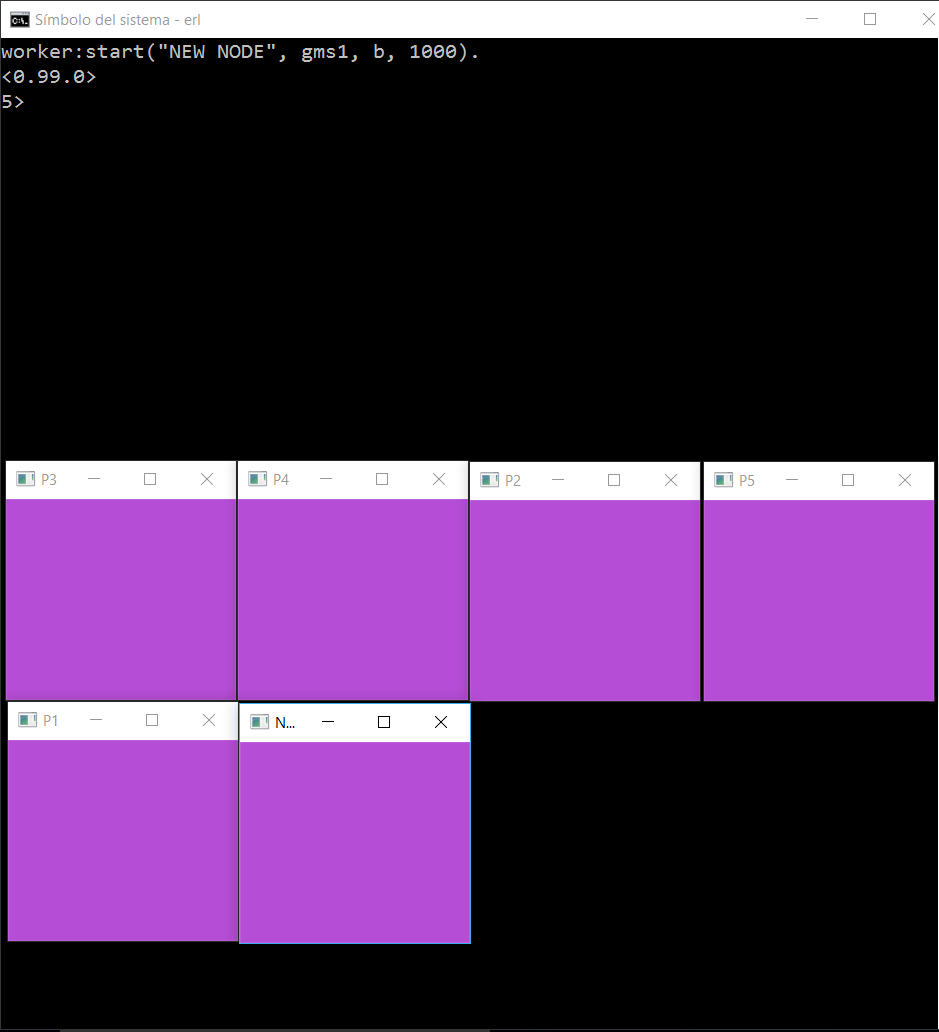
**Experiments:**  *Do some experiments to see that you can create a group, add some peers and keep their state coordinated.*

S’ha creat primer un grup de 4 esclaus i el líder, en total 5 processos instanciats sobre la mateixa màquina i que es communiquen mitjançant multicast FIFO + total order gràcies a la sequencialització del leader que fa que l’estat de grup (color) es preservi.



El nostre objectiu és també afegir un node al grup i mantenir-lo coordinat amb els altres. En aquest cas crearem un worker que pregunti per exemple al membre del grup “P2” per a accedir-hi, P2 farà forwarding (de la request de join) al leader i si no es perden missatges per el camí, el nou node podrà ser part del grup. El mòdul gms1 assegura també que el node afegit estarà sincronitzat gràcies a una request del color del grup.

Podem veure aquí el resultat:



**Experiments:**  *Adapt the groupy module to enable each worker to run in different machines.*

Hem adaptat el codi per a què els nous nodes puguin executar-se en diferents màquines. Es pot observar el codi en el fitxer (groupy\_remote.erl).

Execució d’atomic multicast amb possibilitat dels nodes per a ser executats des de diferents nodes:

Per simplicitat en el nostre exemple els nodes s’han executat en la mateixa màquina però modificant el paràmetre ‘Nodes’ permet l’execució dels processos en diferents màquines.

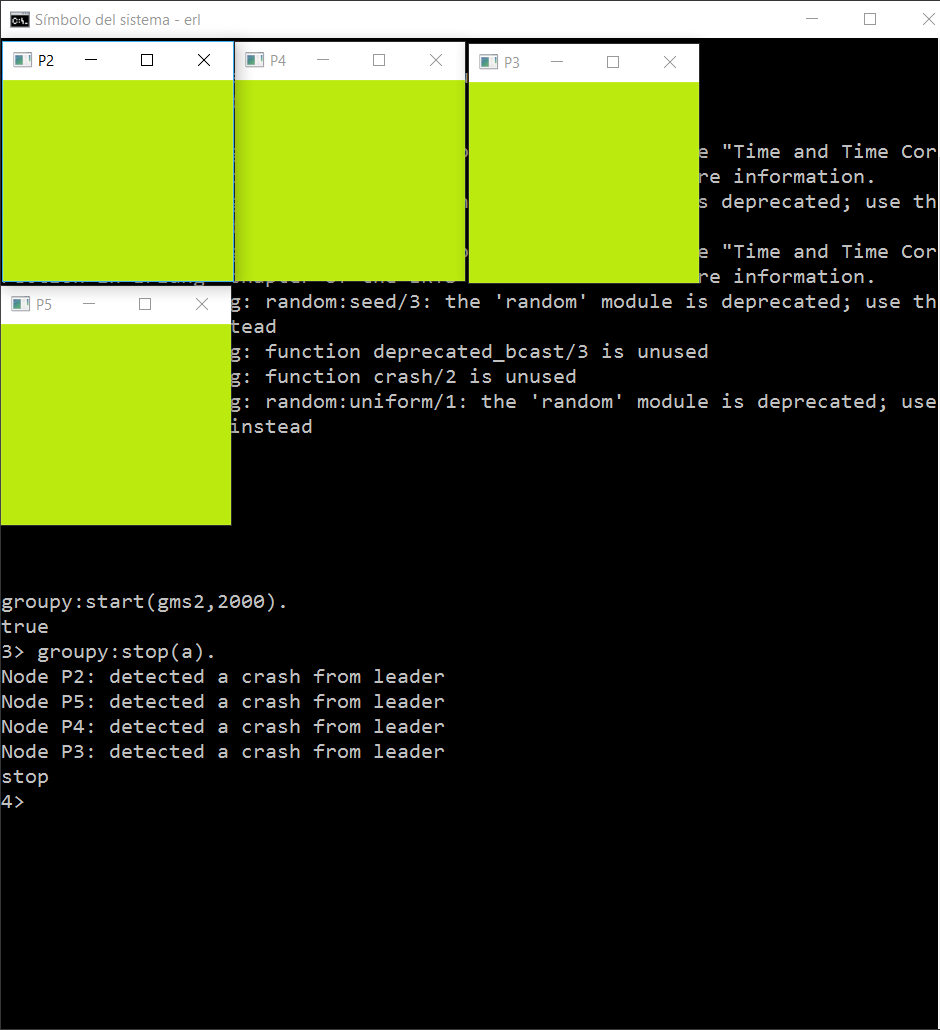
**3 - Handling failures**

**3.1- Failure detectors**

**Experiments:** *Do some experiments to see if the peers can keep their state coordinated even if nodes crash.*

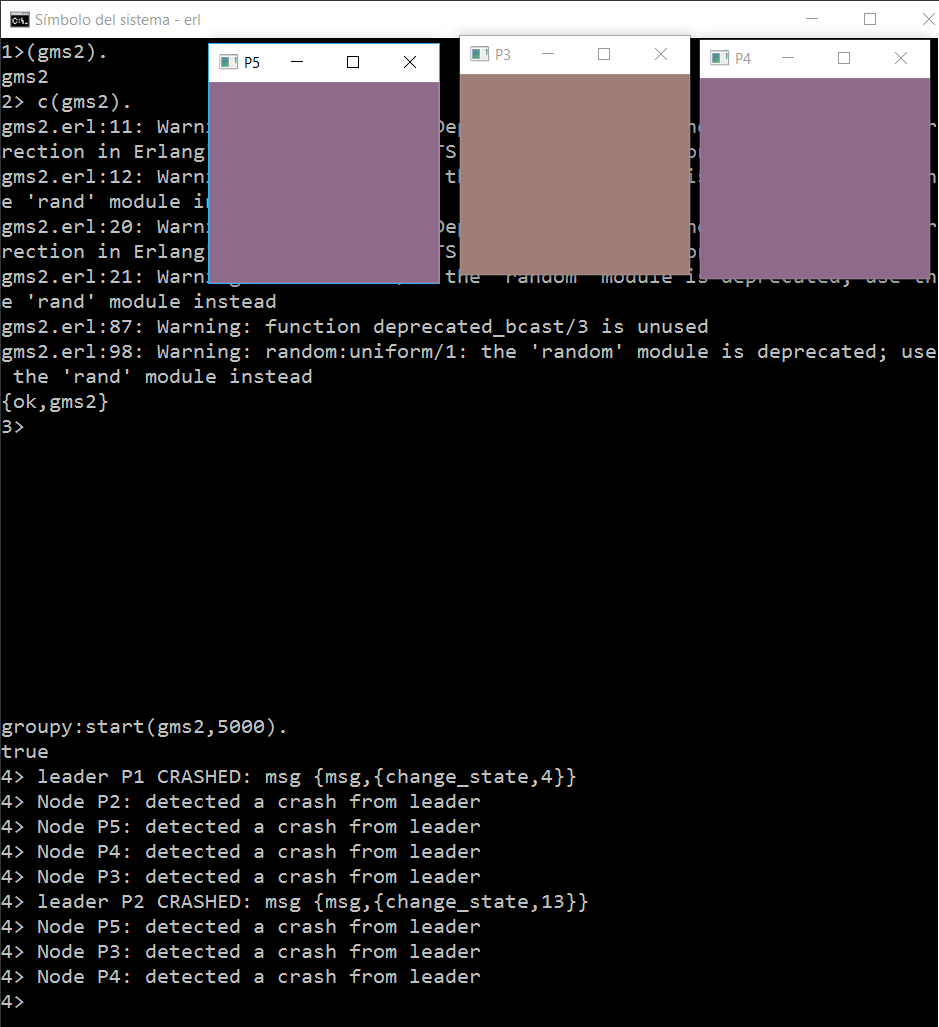
En aquest experiment es vol observar la capacitat de fallades del nostre sistema distribuit. Començem experiemant la part més crítica que és la **fallada del líder.** Gràcies a que Erlang ens permet monitoritzar processos, farem que els slaves monitoritzin al leader i si veuen que el leader ha caigut entraràn en un estat d’elecció on el primer node de la cua de slaves serà escollit. El nou leader per si de cas els slaves no s’enteren de que el leader ha fet crash, enviarà un multicast de la nova vista (nou leader i nous slaves).

Veiem doncs l’execució d’aquest mòdul (gms2) i el comportament quan el leader cau:



Hem programat també un output en el programa com a referència de quins nodes han detectat el crash del leader, com a curiositat.

Segons la nostra execució, el crash del leader sembla que no infueix en la descordinació del grup. Tot i així hem de reflexionar que en aquesta versió **si tenim la mala sort de que el leader fa crash mentres fa un broadcast dels missatges, els processos poden quedar descordinats** quan s’escolleixi un nou leader, i s’ha d’anar molt en compte amb aquesta execució.

Per veure-ho més clar ho mostrem amb un exemple:

Finalment volem veure l’efecte del crash d’un slave en el sistema. Degut a que volem mantenir la simplicitat, suposarem que si un slave fa crash, no cal modificar la vista i suposarem que fa crash per sempre.

El crash del client per tant no hauria de ser de gran preocupació, simplement no rebrà els missatges que se li envíen ni tampoc enviarà missatges no influint així en la coordinació del grup.

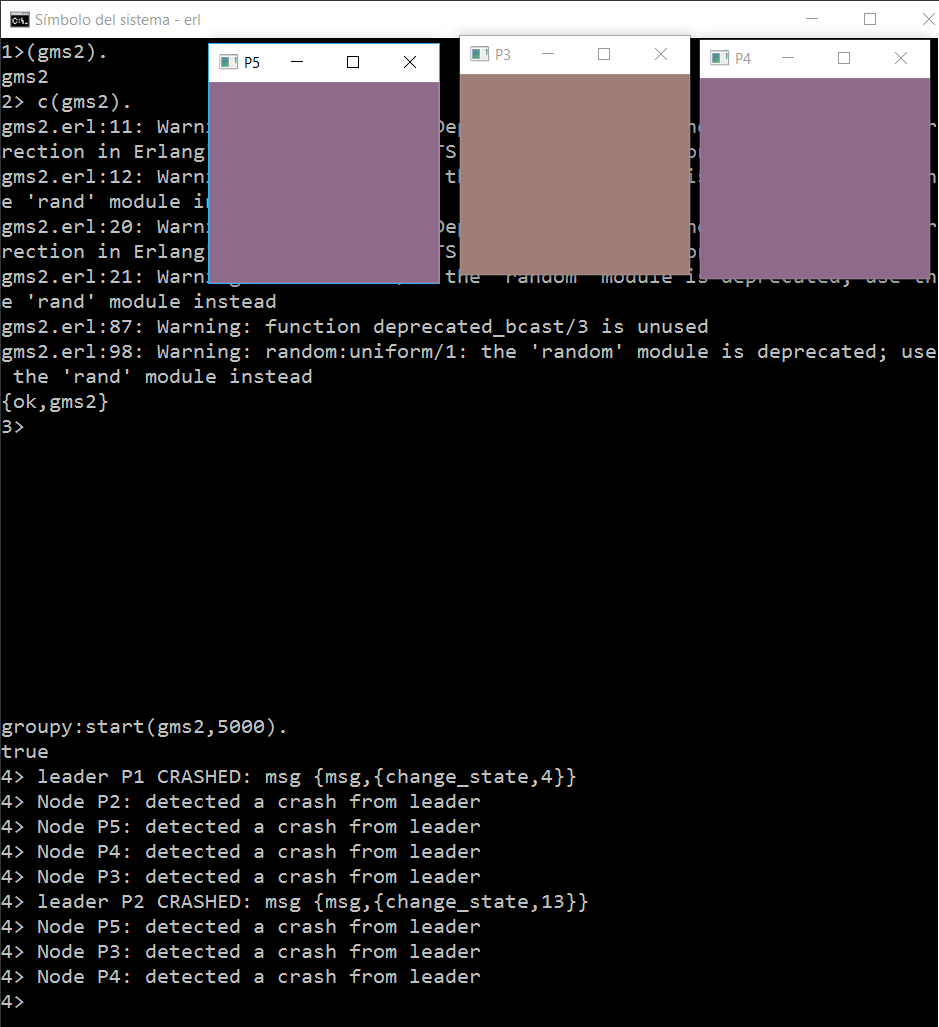


Obtenim com a conclusió que si volem un sistema coordinat en tot moment, el nostre sistema actualment no és tolerant a fallades.

**3.2 - Missing messages**

**Experiments:** *Repeat the experiments and see if you can have the state of the workers become out of synch.*

Excecució amb probabilitat de crash cada missatge:



Amb aquest exemple podem concloure que sí, si que es pot produïr una descordinació en el grup.

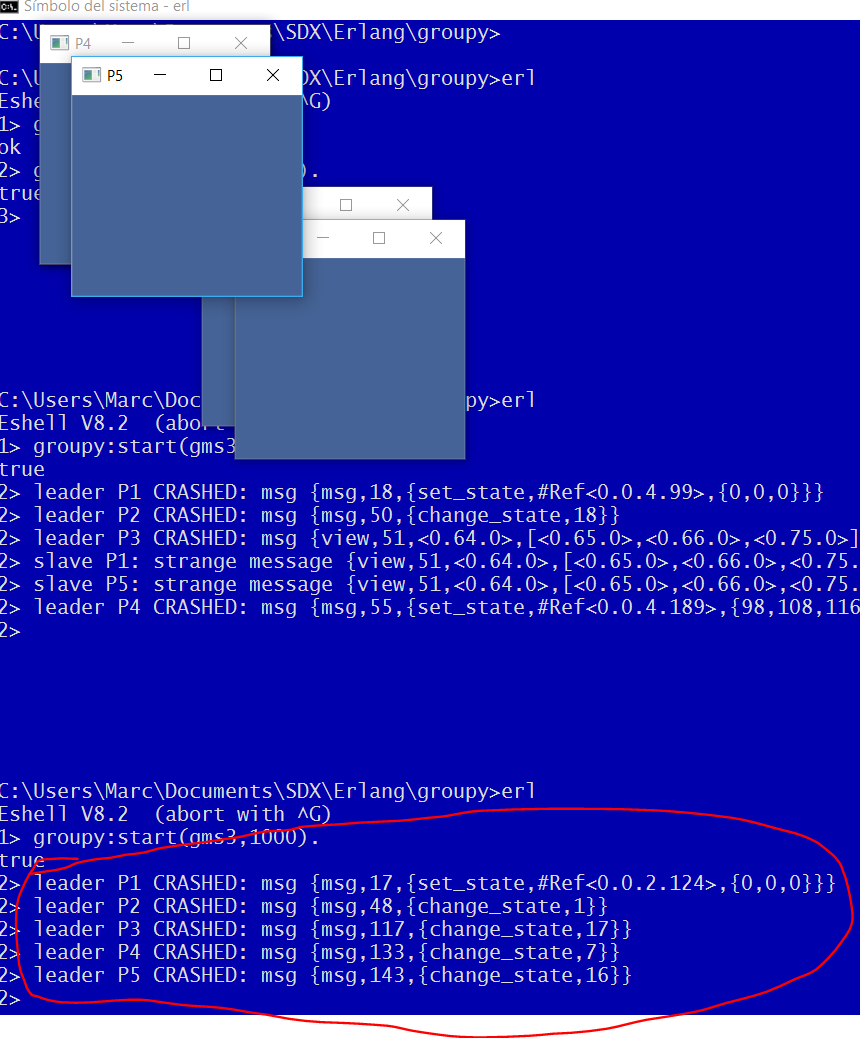
**Open questions** *Why is this happening?*

Quan el leader fa crash al mig d’un broadcast, només una fracció dels slaves reben l’últim missatge. Com els canvis de vista/color en els workers són acumulatius (addició d’un color sobre l’actual), si un slave no ha rebut un missatge, al seu worker li falta una part del color actual. Per tant es desincronitza.

**3.3 Reliable multicast**

Amb la implementació de reliable multicast, ara sí que tot i les múltiples fallades dels leaders com es pot veure en els resultats de l’experiment en la foto a continuació, els processos poden seguir coordinats.

A més cada cop que un leader fa crash, el nou leader crea un nou procés slave que demana ser afegit al grup, intentant així mantenir el mateix nombre de processos durant el temps.



**3.4 What could possibly go wrong**

*a) How would we have to change the implementation to handle the possibly lost messages?*

Marc’s take:

Veiem quins efectes té perdre missatges en diferents etapes, si perdem un missatge d’un procés slave cap al leader, això no té cap implicació en el nostre entorn, suposarem que aquell missatge mai s’ha creat. (quitaria esto, los slaves no mandan más que acks creo)

Els problemes poden esdevenir si perdem un missatge dels que el leader entrega als slaves amb el multicast, que actualment la pèrdua d’aquest suposa una descordinació del nostre sistema. Ens podem basar en la implementacio del scalable multicast, de manera que quan un node rep un missatge i el seu numero de sequencia es major, esperara un tmeps random i es fara un NACK al leader i un multicast d’aquest NACK cap a altres nodes, quan els altres nodes rebin el NACK i ells tambe vulguin fer un NACK quedaran parats (per nomes enviar un NACK al leader).

Finalment el leader amb un missatge de NACK enlloc de retransmetre tota la sequencia de missatges i tenir una cua per guardar-los, simplement enviarem un multicast amb change state (amb l’estat del lider unicament) cap a tots els nodes per simplicitat. Els altres nodes quan vegin un missatge de change\_state estaran obligats a sincronitzar-se evitant-nos malts de cap.

Quan el leader falla, tampoc podem assegurar que (com abans) el primer procès tingui l’últim missatge degut a possibles pèrdues de missatge. El que podem fer és un algoritme d’elecció més complex, que seleccióni el líder basant-se en el número de seqüència més gran lliurat. Un cop seleccionat el nou leader, com fins ara feiem, el nou lider envia un missatge de change\_state a tots els slaves, els nodes es coordinen i continua la comunicacio.

(IMATGE EXPLICACIO) ?

Victor’s take:

Considerarem només la implementació de la part del líder, assumint que les peticions unicast en direcció al leader presenten molta menys complexitat. Per solucionar el problema dels missatges perduts s’ha d’implementar alguna versió de reliable multicast.

La versió escalable oferiria millor performance, però podria perjudicar la qualitat de la sincronització. Això és així perquè un slave que no rep un missatge enviarà el NACK quan el leader li envïi el següent. D’aquesta manera, el node endarrerit estarà un temps desincronitzat. Si el temps entre estats és prou gran, aquesta descoordinació serà observable.

A més, podria donar-se la situació de que un slave perdés més d’un missatge, i el leader només ho sabria a partir del primer missatge que arribi. Més enllà de l’efecte que això tindria sobre la sincronització, el leader hauria de guardar tota la sequüencia de canvis d’estat, no només l’ultim com fa ara. I per prevenir una fallada del leader, tots els procesos haurien de tenir tota la seqüencia de canvis d’estat.

Implementar basic reliable multicast ens donaria més control sobre la sincronització. En particular, podriem forçar que el leader no acceptés un canvi d’estat fins que tots els slaves hagin enviat l’ACK del canvi d’estat anterior. Així acotariem la desincronització a un sol canvi d’estat (tant pel que fa al color com pel que fa al temps que els workers han estat descoordinats.)

A efectes pràctics, si assumim que els nodes slave no poden fer crash, aquesta segona solució és atomic multicast. Si els slaves poguessin fallar hauriem de cambiar l’actual implementació de les vistes.

En les dues solucións no hem parlat de la situació de canvi de líder. En un sistema amb pérdua de missatges, no podem estar segurs de que el següent de la llista haurà rebut l’últim missatge. Una possible solució seria implementar un procés d’elecció que prioritzés nodes que disposin del missatge amb el màxim número de seqüencia.

Per finalitzar, afegir que la primera solució sería factible amb un parell de canvis. Primer, limitar el nombre de missatges que es guarden. És improbable que un node slave perdi més d’un cert nombre de missatges de manera consecutiva, és més eficient que considerar que no passarà.

Amb això, si un node s’endarrerís més missatges dels que caben en aquesta “caché”, llavors el sistema no es podria recuperar. Però en aquest cas improbable, simplement fem un reset del node. De la mateixa manera que en l’operació de join, transmetre l’estat actual. Això és factible perquè el nostre estat, color, és poc complex.

b)

Marc’s take:

EXPLICAR-HO MILLOR LA PERFORMANCE

Un dels canvis més significatius introduïts és un nou algoritme d’elecció. Si fem servir l’algoritme Enhanced ring enviaríem 2N missatges com a màxim cada eleció (on N són Slaves). En comparació amb l’anterior algorisme d’elecció que afegeix 0 missatges. Tenim per començar un overhead en cost de 2N missatges i temporal de 2N latències per a escollir el leader a cada crash.

El segon canvi que hem introduït és el de si un procès rep un missatge amb nombre de sequència més gran, es sincronitza envian un missatge de NACK al server i S NACK (on S son slaves) per prevenir mes NACK. El lider al detectar un NACK fa S broadcast de missatges change\_state (amb l’estat del lider) per als slaves i aixi tots els slaves queden sincronitzats amb el leader.

Un escenari dolent per exemple es quan cada multicast N/2 processos reben un nombre de sequencia mes gran, es a dir, estan dessincronitzats, i per tant fan N/2 NACK (pitjor dels casos) degut a que la espera random abans del NACK no ha sigut efectiva, i el servidor envia N/2 missatges de change\_state, i tot aixo per a cada ronda (on ronda = enviament de cada missatge). En la seguent ronda els altres N/2 processos faran el mateix i aixi succesivament. En total N missatges 2 latencies d’overhead per ronda.

Victor’s take:

L’impacte sobre el rendiment de la primera solució (millorada) seria negligible. El cost d’un NACK (si es que és necessari) per multicast és mínim respecte al nombre de missatges del multicast, un per a cada node. En cas d’haver de recuperar missatges, el cost depén del tamany de la caché. El temps de recuperació sería mínim si s’enviesin tots els missatges de cop. Fer reset de l’estat només costa un missatge.

La segona solució, basic reliable multicast, implica doblar el nombre de missatges. Un ACK per a cada missatge del multicast. A més, esperar els ACKS implica que el líder s’ha d’esperar max(RTT) abans d’enviar el següent canvi d’estat.

**Añadir aquí lo del proceso de elección que ha puesto Marc (adaptarlo si acaso)**

**c)**

Amb la implementació actual, el sistema ja tindrà un nou líder. Els esclaus enviaran les seves peticions de multicast al nou líder. Però el vell líder encara gestionarà les peticions de de canvi d’estat del seu worker, per les que sí farà multicast.

La resta de nodes, tant el nou lider com els slaves, processaràn aquests canvis d’estat foranis perquè no controlem que el missatge de canvi d’estat provingui del líder. Com tots els workers rebràn els mateixos missatges, estaràn sincronitzats. Excepte el worker associat al vell líder, que ja no és a la llista dels demés i només respondrà a les seves propies peticions.