Seminar Report: [Ordy]

Arnau Valls, Aniol Gàmiz, Nil Magnusson

April 3, 2019

Upload your report in PDF format.
Use this LaTeX template to format the report.
A compressed file (.tar.gz) containing all your source code files must be submitted together with this report[1].

1 Introduction

En aquesta pràctica haurem d'implementar 3 versions d'un codi que s'encarregui de gestinar l'ordre dels missatges que s'envien diferents *workers* entre ells a través d'uns gestors. En la primeras versió no hi ha cap mena d'ordre, en la segona versió tindrem un ordre causal i finalment l'última versió té ordre total.

2 Experiments

2.1 Basic multicast

i) **Modificació en el valor del *Jitter***

En aquest cas hem modificat el valor del Jitter. En les següents imatges podem veure que tot i que el Jitter és diferents, al no haver-hi cap mena d'ordre no es pot apreciar la diferencia entre les 2 versions.

[1] Describe in the report any design decision required to understand your code (if any)

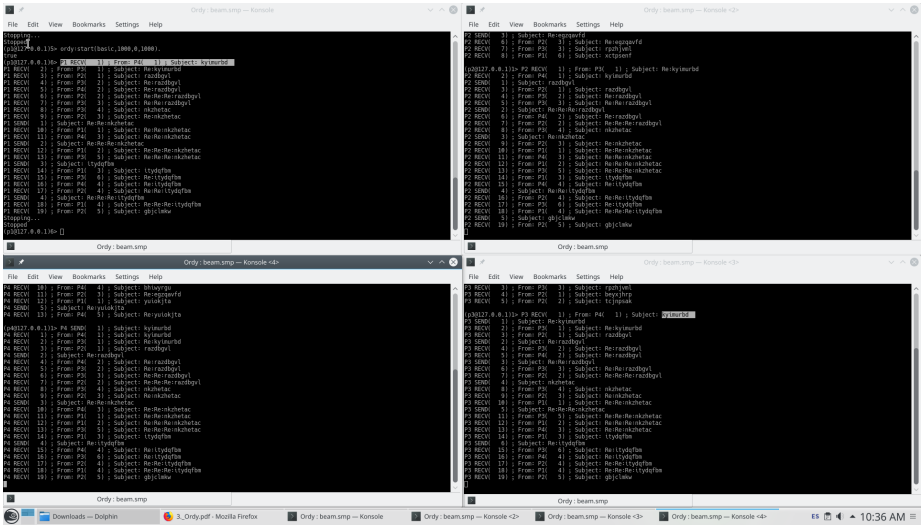

Figure 1: 0 ms Jitter

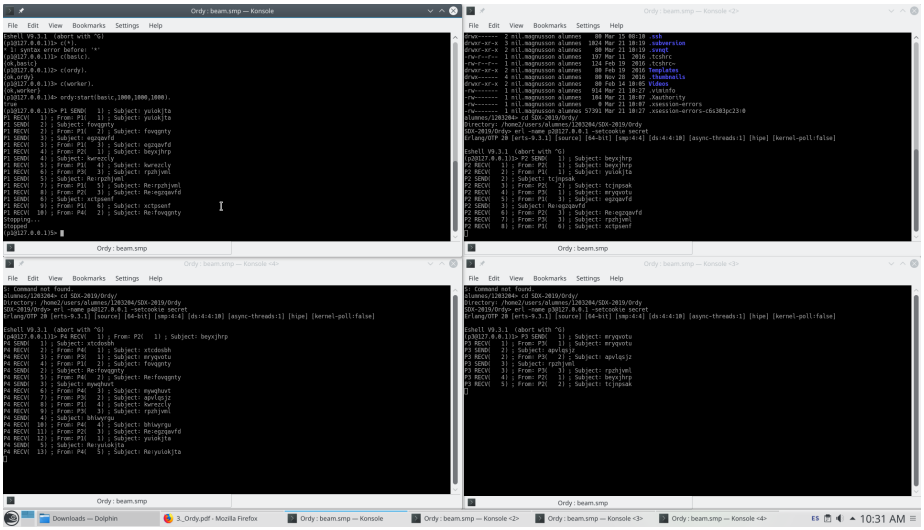

Figure 2: 1000 ms Jitter

ii) **Modificació en el valor de *Sleep***

Si reduim el temps que pot trigar un *worker* en enviar un missatge, s'envien més missatges (Figure 3). En canvi si augmentem aqust temps s'envien menys missatges (Figura 4).

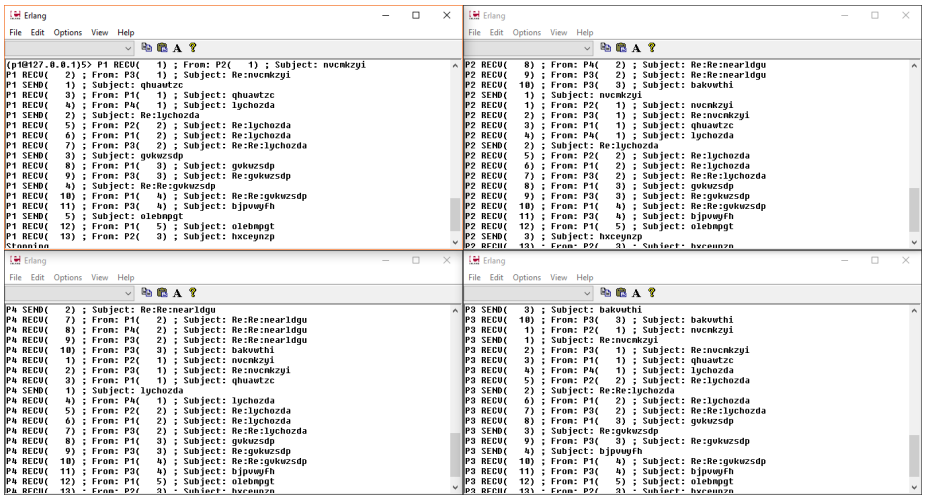

Figure 3: 800 ms Sleep

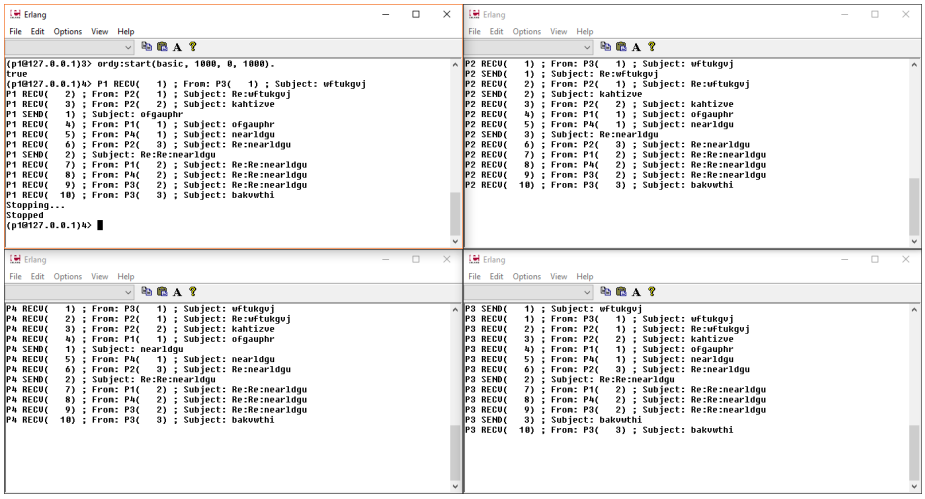

Figure 4: 1000 ms Sleep

iii) **Modificació en el valor de *Duration***

En aquest últim cas modifiquem el temps de duració del programa, el que fa que com a conseqëncia s'enviin menys missatges. Com en aquesta versió no hi ha cap mena d'ordenació no podem assegurar que tots els missatges que s'han enviat han arribat o no a la restas de nodes.

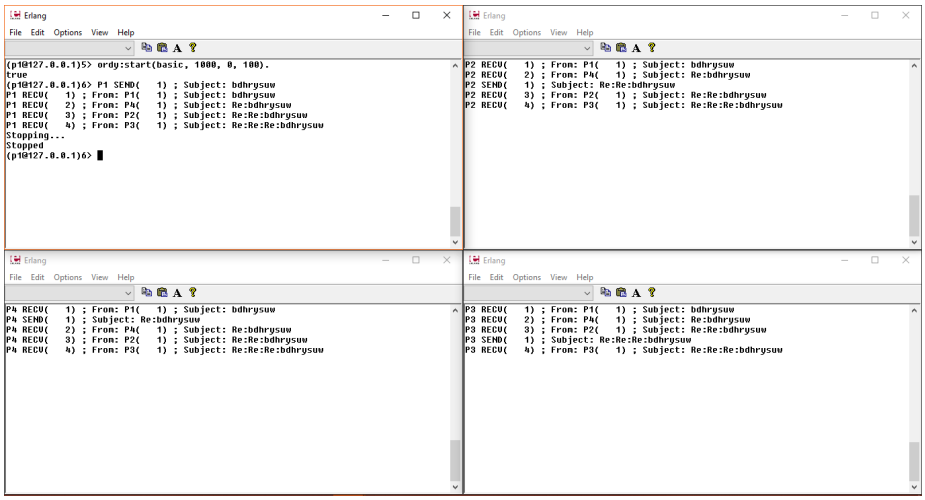

Figure 5: 100 ms Duration

2.2 Causal order multicast

En el multicast amb ordre causal hauríem de veure com es respecte l'ordre de programa de cada instància per separat i a més l'ordre de causalitat entre els missatges relacionats (un post i les seves respostes). L'única cosa que no ens assegura aquesta ordenació però que si ho fa la Total és que totes les instancies vegin el mateix ordre final, és a dir, cada missatge té un número assignat que és comú per totes les instancies corrent.

i) **Modificació del valor de *Jitter***
En aquest primer experiment variarem el valor del Jitter a 0 i a 1000, en un temps d'execució de 2 segons i un Sleep de 500.

Podem apreciar com tot i que la ordenació causal es manté com s'esperaria, i tot i que el número assignat no és igual perquè el protocol no ho procura, com que el jitter 0 la probabilitat que es desordenin és pràcticament nul·la i per això no veiem cap missatges fora d'ordre.

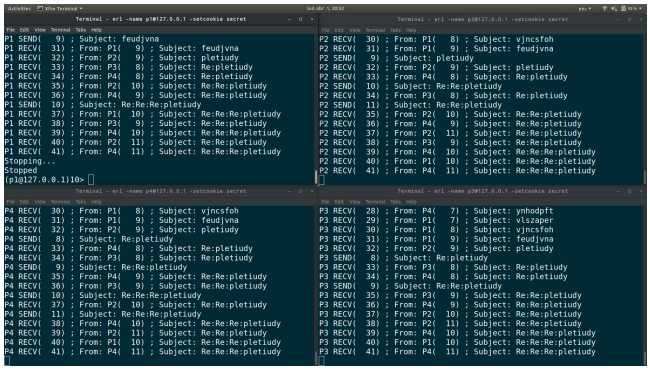

Figure 6: Ordenació causal amb Jitter = 0 i Sleep = 500

Amb el jitter a 1000 veiem que si bé l'ordre causal es manté(mai reps una resposta abans que la pregunta), no es manté l'ordre general del programa.

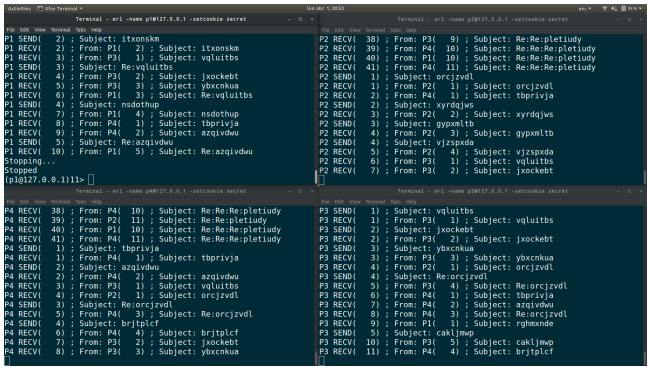

Figure 7: Ordenació causal amb Jitter = 1000 i Sleep = 500

ii) **Modificació del valor de *Sleep***

Quan el valor del Sleep = 0 observem que s'envia una quantitat ingent de missatge però es segueix mantenint l'ordre de programa.

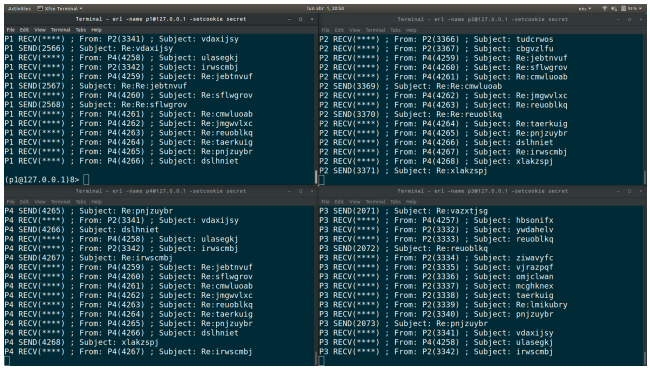

Figure 8: Ordenació causal amb Jitter = 0 i Sleep = 0

Aquesta captura del programa amb Sleep = 1000 ens permet veure el clar exemple de perquè la ordenació total pot no ser suficient. Just al final de l'execució veiem com P4 envia el missatge **euavfong** i P1 i P2 responen aquest missatge. Si ens fixem en les 4 instàncies veurem que tot i que certament ningú ha vist la resposta abans que la pregunta, cada instància veu les respostes en un ordre diferent perquè les respostes al missatge original no tenen relació causal entre elles, sinó només amb el missatge al qual responen.

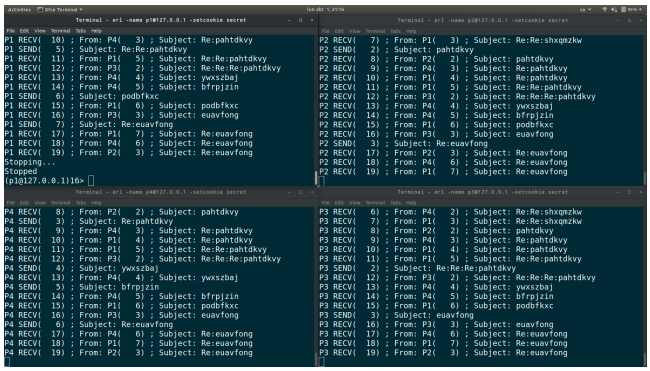

Figure 9: Ordenació causal amb Jitter = 0 i Sleep = 1000

2.3 Total order multicast

i) **Modificació del valor de *Jitter***

Aquesta última versió del codi funciona amb ordenament total. Les seqüències de missatges són les mateixes i en el mateix ordre per totes les instàncies d'Erlang.

En aquest experiment hem provat amb diferents valors de Jitter, primer amb 0 ms i després amb un valor més gran com *1000 ms*. En cap dels dos casos s'hauria de desordenar.

Un cop hem fet l'experiment veiem que no s'ha desordenat en cap dels casos però han sortit pocs missatges degut a que el valor de *Sleep* és força alt.

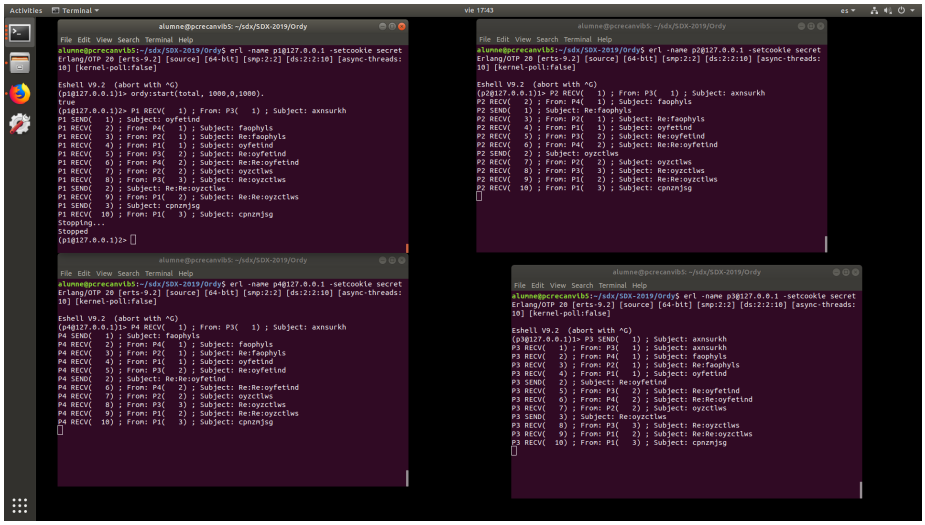

Figure 10: Jitter 0 ms

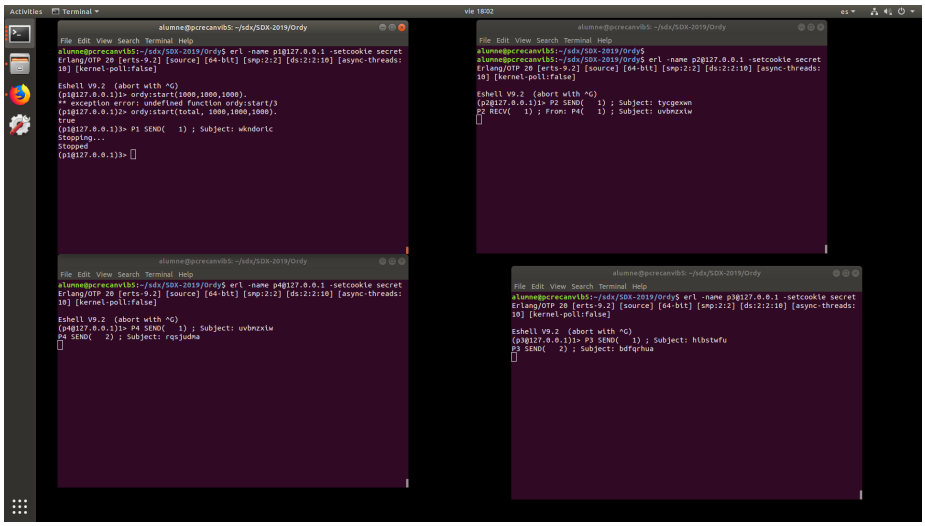

Figure 11: Jitter 1000 ms

ii) **Modificació del valor de *Sleep***
En aquest segon experiment hem modificat el valor de *Sleep*. Començant des d'un valor petit 500 i acabant amb un valor de 1000. Mantindrem el jitter a 500. No hauria d'haver cap canvi respecte a l'ordenació dels missatges i hauria de sortir tot igual. L'únic que pot canviar és la quantitat de missatges que ens surten per la pantalla: si posem un valor de *Sleep* més alt veurem menys missatges que si el posem baix, degut a que triguem més/menys temps per enviar cada missatge.
Un cop hem fet l'experiment veiem el que ens suposavem, amb el temps de *Sleep* de 500 ms enviem més missatges que amb el paràmetre de 1000 ms.

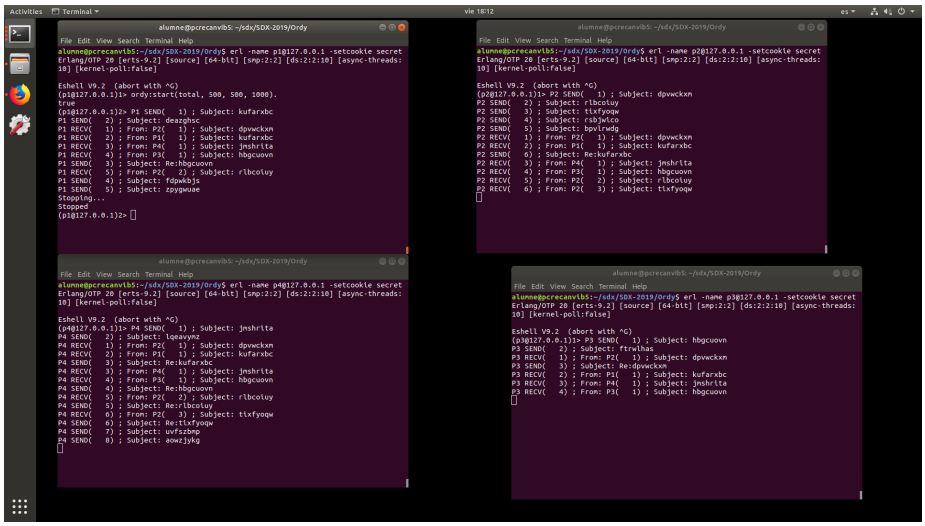

Figure 12: Sleep 500 ms

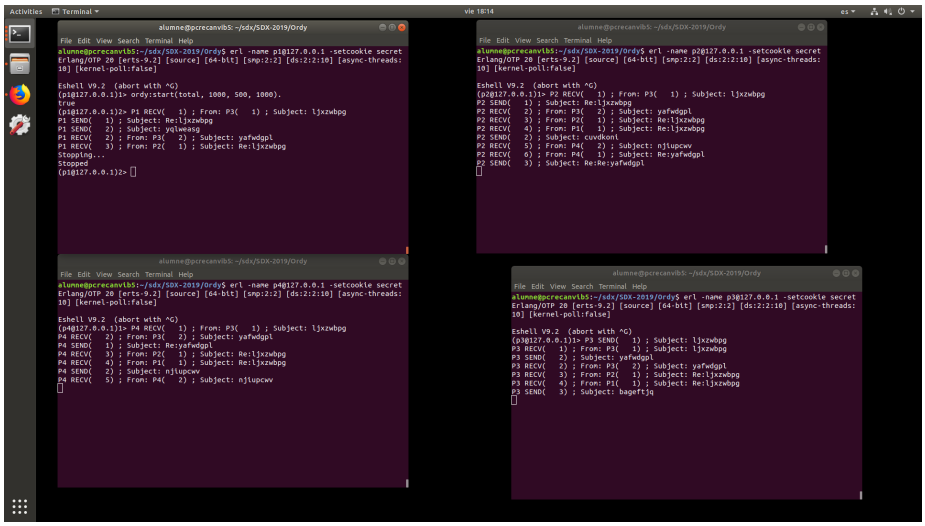

Figure 13: Sleep 1000 ms

iii) **Modificació del valor de *Duration***
En el tercer experiment realitzat hem modificat el valor del *Duration*. L'únic que pot passar és que s'enviin més/menys missatges perquè el temps d'execució varia segons el paràmetre que hi posem. Provarem amb valors de 500 ms i 1000 ms.

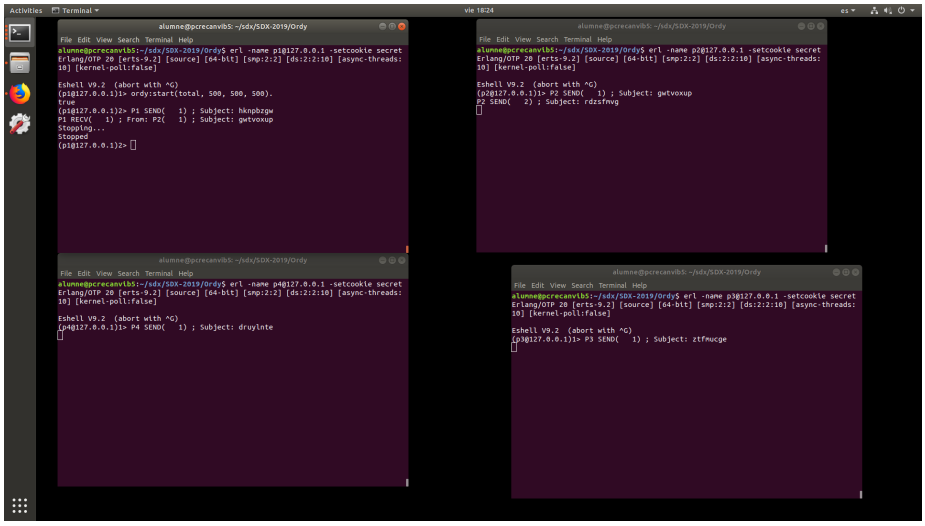

Figure 14: Duration 500 ms

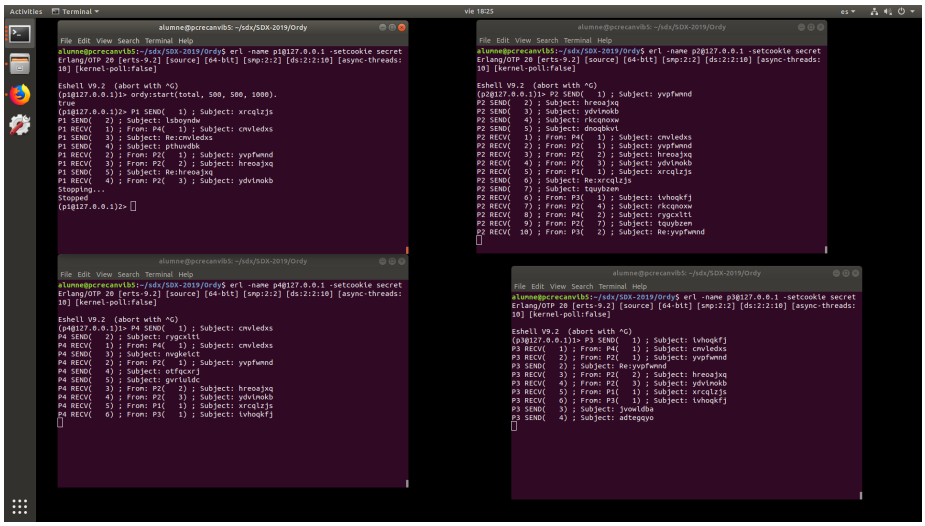

Figure 15: Duration 1000 ms

iv) **We have a lot of messages in the system. Derive a theoretical quantification of the number of messages needed to deliver a multicast message as a function of the number of workers and check experimentally that your formulation is correct (you can modify the source code as needed to count the messages).**
Total = 1 x send + n x request + n x proposal + n x agreed + n x deliver = 4n + 1

v) **Compare with the basic multicast implementation regarding the number of messages needed.**
Basic = 1 x send + n x multicast + n x deliver = 2n + 1

3 Open questions

3.1 Basic multicast

i) **Are the posts displayed in FIFO, causal, and total order? Justify why.**
En aquest cas hi ha ordenació FIFO ja que els nodes lliuren els missatges no en l'ordre en el que els arriben, no es preocupen de cap ordenació entre els missatges.

3.2 Total order multicast

1. **Are the posts displayed in FIFO, causal, and total order? Justify why.**
En el cas del total order els missatges són lliurats en ordre total ja que es té en compte l'ordre de programa, també es mira que el missatge que es va lliurar sigui el missatge que s'espera rebre, és a dir, posterior a la resta de missatges ja lliurats i per últim els missatges d'un *worker* es veu en la mateixa posició de lectura per la resta de *workers*.

4 Personal opinion

Considerem que aquesta pràctica és molt interessant ja que podem veure de forma pràctica el funcionament del diferents tipus d'ordenacions. Tot i això és una pràctica complicada ja que ens hem hagut de fixar molt en els missatges per veure si realment la nostra implementació funcionava. També és una pràctica molt útil per veure com funciona la recursió en erlang, tot i que aquest no sigui l'objectiu principal d'aquesta.